Introduction to Algorithms

Jon Kleinberg Éva Tardos

Cornell University
Spring 2003

Contents

1 Introduction **7**
- 1.1 Introduction: The Stable Matching Problem 7
- 1.2 Computational Tractability 16
- 1.3 Interlude: Two Definitions 19
- 1.4 Five Representative Problems 22
- 1.5 Exercises 28

2 Algorithmic Primitives for Graphs **35**
- 2.1 Representing Graphs 35
- 2.2 Paths, Cycles, and Trees 37
- 2.3 Graph Connectivity and Graph Traversal 39
- 2.4 Two Applications of Graph Traversal 45
- 2.5 Extensions to Directed Graphs 50
- 2.6 Directed Acyclic Graphs and Topological Ordering 51
- 2.7 Exercises 54

3 Greedy Algorithms **57**
- 3.1 The Greedy Algorithm Stays Ahead 58
- 3.2 Exchange Arguments 64
- 3.3 Shortest Paths in a Graph 73
- 3.4 The Minimum Spanning Tree Problem 76
- 3.5 Minimum-Cost Arborescences: A Multi-Phase Greedy Algorithm 84
- 3.6 Exercises 89

4 Divide and Conquer **105**
- 4.1 A Useful Recurrence Relation 105
- 4.2 Counting Inversions 107
- 4.3 Finding the Closest Pair of Points 109
- 4.4 Exercises 115

5 Dynamic Programming **117**
- 5.1 Weighted Interval Scheduling: The Basic Set-up 117
- 5.2 Segmented Least Squares: Multi-way Choices 123
- 5.3 Subset Sums and Knapsacks: Adding a Variable 128
- 5.4 RNA Secondary Structure: Dynamic Programming Over Intervals 133

5.5 Sequence Alignment . . . 138
5.6 Sequence Alignment in Linear Space . . . 144
5.7 Shortest Paths in a Graph . . . 148
5.8 Negative Cycles in a Graph . . . 156
5.9 Exercises . . . 161

6 Network Flow **179**
6.1 The Maximum Flow Problem . . . 180
6.2 Computing Maximum Flows . . . 182
6.3 Cuts in a Flow Network . . . 187
6.4 Max-Flow Equals Min-Cut . . . 189
6.5 Choosing Good Augmenting Paths . . . 192
6.6 The Preflow-Push Maximum Flow Algorithm . . . 196
6.7 Applications: Disjoint Paths and Bipartite Matchings . . . 204
6.8 Extensions to the Maximum Flow Problem . . . 210
6.9 Applications of Maximum Flows and Minimum Cuts . . . 214
6.10 Minimum Cost Perfect Matchings . . . 226
6.11 Exercises . . . 230

7 NP and Computational Intractability **253**
7.1 Computationally Hard Problems . . . 254
7.2 Polynomial-time Reductions . . . 259
7.3 Efficient Certification and the Definition of NP . . . 266
7.4 NP-complete Problems . . . 269
7.5 Sequencing and Partitioning Problems . . . 271
7.6 The Hardness of Numerical Problems . . . 280
7.7 co-NP and the Asymmetry of NP. . . . 283
7.8 Exercises . . . 285

8 PSPACE **301**
8.1 PSPACE . . . 301
8.2 Some Hard Problems in PSPACE . . . 302
8.3 Solving Problems in Polynomial Space . . . 305
8.4 Proving Problems PSPACE-complete . . . 311
8.5 Exercises . . . 317

9 Extending the Limits of Tractability **319**
9.1 Finding Small Vertex Covers . . . 320
9.2 Solving NP-hard Problem on Trees . . . 323
9.3 Tree Decompositions of Graphs . . . 327
9.4 Exercises . . . 343

10 Approximation Algorithms **347**
10.1 Load Balancing Problems: Bounding the Optimum 347
10.2 The Center Selection Problem 351
10.3 Set Cover: A General Greedy Heuristic 354
10.4 Vertex Cover: An Application of Linear Programming 358
10.5 Arbitrarily Good Approximations for the Knapsack Problem 364
10.6 Exercises 369

11 Local Search **377**
11.1 The Landscape of an Optimization Problem 377
11.2 The Metropolis Algorithm and Simulated Annealing 383
11.3 Application: Hopfield Neural Networks 386
11.4 Choosing a Neighbor Relation 390
11.5 Exercises 394

12 Randomized Algorithms **397**
12.1 A First Application: Contention Resolution 398
12.2 Finding the global minimum cut 403
12.3 Random Variables and their Expectations 407
12.4 A Randomized Approximation Algorithm for MAX-3-SAT 410
12.5 Computing the Median: Randomized Divide-and-Conquer 413
12.6 Randomized Caching 416
12.7 Chernoff Bounds 420
12.8 Load Balancing 421
12.9 Packet Routing 423
12.10Constructing an Expander Graph 429
12.11Appendix: Some Probability Definitions 434
12.12Exercises 440

13 Epilogue: Algorithms that Run Forever **451**

Chapter 1

Introduction

1.1 Introduction: The Stable Matching Problem

As a beginning for the course, we look at an algorithmic problem that nicely illustrates many of the themes we will be emphasizing. It it motivated by some very natural and practical concerns, and from these we formulate a clean and simple statement of a problem. The algorithm to solve the problem is very clean as well, and most of our work will be spent in proving that it is correct and giving an acceptable bound on the amount of time it takes to terminate with an answer. The problem itself — the *Stable Matching Problem* — has several origins.

One of its origins is in 1962, when David Gale and Lloyd Shapley, two mathematical economists, asked the question: "Could one design a college admissions process, or a job recruiting process, that was *self-enforcing*?" What did they mean by this?

To set up the question, let's first think informally about the kind of situation that might arise as a group of your friends, all juniors in college majoring in computer science, begin applying to companies for summer internships. The crux of the application process is the interplay between two different types of parties: companies (the employers) and students (the applicants). Each applicant has a preference ordering on companies and each company — once the applications come in — forms a preference ordering on its applicants. Based on these preferences, companies extend extend offers to some of their applicants, applicants choose which of their offers to accept, and people begin heading off to their summer internships.

Gale and Shapley considered the sorts of things that could start going wrong with this process, in the absence of any mechanism to enforce the status quo. Suppose, for example, that your friend Raj has just accepted a summer job at the large telecommunications company CluNet. A few days later, the small start-up company WebExodus, which had been dragging its feet on making a few final decisions, calls up Raj and offers him a summer job as well. Now, Raj actually prefers WebExodus to CluNet — won over perhaps by the laid-back, anything-can-happen atmosphere — and so this new development may well cause him

to retract his acceptance of the CluNet offer, and go to WebExodus instead. Suddenly down one summer intern, CluNet offers a job to one of its wait-listed applicants, who promptly retracts his previous acceptance of an offer from the software giant Babelsoft, and the situation begins to spiral out of control.

Things look just as bad, if not worse, from the other direction. Suppose your friend Chelsea, destined to go Babelsoft but having just heard Raj's story, calls up the people at WebExodus and says, "You know, I'd really rather spend the summer with you guys than at Babelsoft." They find this very easy to believe; and furthermore, on looking at Chelsea's application, they realize that they would have rather hired her than some other student who actually *is* scheduled to spend the summer at WebExodus. In this case, if WebExodus were a slightly less scrupulous start-up company, it might well find some way to retract its offer to this other student and hire Chelsea instead.

Situations like this can rapidly generate a lot of chaos, and many people — both applicants and employers — can end up unhappy with both the process and the outcome. What has gone wrong? One basic problem is that the process is not *self-enforcing* — if people are allowed to act in their self-interest, then it risks breaking down.

We might well prefer the following, more stable, situation, in which self-interest itself prevents offers from being retracted and re-directed. Consider another one of your friends, who has arranged to spend the summer at CluNet but calls up WebExodus and reveals that he, too, would rather work for them. But in this case, based on the offers already accepted, they are able to reply, "No, it turns out that we prefer each of the students we've accepted to you, so we're afraid there's nothing we can do." Or consider an employer, earnestly following up with its top applicants who went elsewhere, being told by each of them, "No, I'm happy where I am." In such a case, all the outcomes are stable — there are no further outside deals that can be made.

So this is the question Gale and Shapley asked: Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, one of the following two things is the case? —

(i) E prefers every one of its accepted applicants to A; or

(ii) A prefers her current situation to the situation in which she is working for employer E.

If this holds, the outcome is stable: individual self-interest will prevent any applicant/employer deal from being made behind the scenes.

Gale and Shapley proceeded to develop a striking algorithmic solution to this problem, which we will discuss presently. Before doing this, let's comment that this is not the only origin of the Stable Matching Problem. It turns out that for a decade before the work of Gale

and Shapley — unbeknownst to them — the National Resident Matching Program had been using a very similar procedure, with the same underlying motivation, to match residents to hospitals. Indeed, this system is still in use today.

This is one testament to the problem's fundamental appeal. And from the point of view of this course, it provides us with a nice first domain in which to reason about some basic combinatorial definitions, and the algorithms that build on them.

Defining the Problem

To try best understanding this concept, it helps to make the problem as clean as possible. The world of companies and applicants contains some distracting asymmetries. Each applicant is looking for a single company, but each company is looking for many applicants; moreover, there may be more (or, as is sometimes the case, fewer) applicants than there are available slots for summer jobs. Finally, each applicant does not typically apply to every company.

Following Gale and Shapley, we can eliminate these complications and arrive at a more "bare-bones" version of the problem: each of n applicants applies to each of n companies, and each company wants to accept a *single* applicant. We will see that doing this preserves the fundamental issue inherent in the problem; in particular, our solution to this simplified version will extend to the more general case as well.

Let's make one more modification, and this is purely to provide a change of scenery. Instead of pairing off applicants and companies, we consider the equivalent problem of devising a system by which each of n men and n women can end up happily married.

So consider a set $M = \{m_1, \ldots, m_n\}$ of n men, and a set $W = \{w_1, \ldots, w_n\}$ of n women. Let $M \times W$ denote the set of all possible ordered pairs of the form (m, w), where $m \in M$ and $w \in W$. A *matching* S is a *set* of ordered pairs, each from $M \times W$, with the property that each member of M and each member of W appears in at most one pair in S. A *perfect matching* S' is a matching with the property that each member of M and each member of W appears in *exactly* one pair in S'.

Matchings and perfect matchings are objects that will recur frequently during the course; they arise naturally in modeling a wide range of algorithmic problems. In the present situation, a perfect matching corresponds simply to a way of pairing off each man with each woman, in such a way that everyone ends up married to somebody, and nobody is married to more than one person — there is neither singlehood nor polygamy.

Now we can add the notion of *preferences* to this setting. Each man $m \in M$ *ranks* all the women; we will say that m *prefers* w *to* w' if m ranks w higher than w'. We will refer to the ordered ranking of m as his *preference list*. We will not allow ties in the ranking. Each woman, analogously, ranks all the men.

Given a perfect matching S, what can go wrong? Guided by our initial motivation in terms of employers and applicants, we should be worried about the following situation: There

are two pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. In this case, there's nothing to stop m and w' from abandoning their current partners and heading off into the sunset together; the set of marriages is not *self-enforcing*. We'll say that such a pair (m, w') is an *instability* with respect to S: (m, w') does not belong to S, but each of m and w' prefers the other to their partner in S.

Our goal, then, is a set of marriages with no instabilities. We'll say that a matching S is *stable* if (i) it is perfect, and (ii) there is no instability with respect to S. Two questions spring immediately to mind:

(†) Does there exist a stable matching for every set of preference lists?

(†) Given a set of preference lists, can we efficiently construct a stable matching if there is one?

Constructing a Stable Matching

We now show that there exists a stable matching for every set of preference lists among the men and women. Moreover, our means of showing this will answer the second question as well: we will give an efficient algorithm that takes the preference lists and constructs a stable matching.

Let us consider some of the basic ideas that motivate the algorithm.

- Initially, everyone is unmarried. Suppose an unmarried man m chooses the woman w who ranks highest on his preference list and *proposes* to her. Can we declare immediately that (m, w) will be one of the pairs in our final stable matching? Not necessarily — at some point in the future, a man m' whom w prefers may propose to her. On the other hand, it would be dangerous for w to reject m right away; she may never receive a proposal from someone she ranks as highly as m. So a natural idea would be to have the pair (m, w) enter an intermediate state — *engagement*.

- Suppose we are now at a state in which some men and women are *free* — not engaged — and some are engaged. The next step could look like this. An arbitrary free man m chooses the highest-ranked woman w to whom he has not yet proposed, and he proposes to her. If w is also free, then m and w become engaged. Otherwise, w is already engaged to some other man m'. In this case, she determines which of m or m' ranks higher on her preference list; this man becomes engaged to w and the other becomes free.

- Finally, the algorithm will terminate when no one is free; at this moment, all engagements are declared final, and the resulting perfect matching is returned.

Here is a concrete description of the *Gale-Shapley algorithm*. (We will refer to it more briefly as the *G-S algorithm*.)

```
Initially all m ∈ M and w ∈ W are free
While there is a man m who is free and hasn't proposed to every woman
   Choose such a man m
   Let w be the highest-ranked woman in m's preference list
      to which m has not yet proposed
   If w is free then
      (m, w) become engaged
   Else w is currently engaged to m'
      If w prefers m' to m then
         m remains free
      Else w prefers m to m'
         (m, w) become engaged
         m' becomes free
      Endif
   Endif
Endwhile
Return the set S of engaged pairs
```

An intriguing thing is that, although the G-S algorithm is quite simple to state, it is not immediately obvious that it returns a stable matching, or even a perfect matching. We proceed to prove this now, through a sequence of intermediate facts.

First consider the view of a woman w during the execution of the algorithm. For a while, no one has proposed to her, and she is free. Then a man m may propose to her, and she becomes engaged. As time goes on, she may receive additional proposals, accepting those that increase the rank of her partner. So we discover the following.

(1.1) *w remains engaged from the point at which she receives her first proposal; and the sequence of partners to which she is engaged gets better and better (in terms of her preference list).*

The view of a man m during the execution of the algorithm is rather different. He is free until he proposes to the highest-ranked woman on his list; at this point he may or may not become engaged. As time goes on, he may alternate between being free and being engaged; however, the following property does hold.

(1.2) *The sequence of women to whom m proposes gets worse and worse (in terms of his preference list).*

Now we show that the algorithm terminates, and give a bound on the maximum number of iterations needed for termination.

(1.3) *The G-S algorithm terminates after at most n^2 iterations of the* `While` *loop.*

Proof. A useful strategy for upper-bounding the running time of an algorithm, as we are trying to do here, is to find a measure of *progress*. Namely, we seek some precise way of saying that each step taken by the algorithm brings it closer to termination.

In the case of the present algorithm, each iteration consists of some man proposing (for the only time) to a woman he has never proposed to before. So if we let $\mathcal{P}(t)$ denote the set of pairs (m, w) such that m has proposed to w by the end of iteration t, we see that for all t, the size of $\mathcal{P}(t+1)$ is strictly greater than the size of $\mathcal{P}(t)$. But there are only n^2 possible pairs of men and women in total, so the value of $\mathcal{P}(\cdot)$ can increase at most n^2 times over the course of the algorithm. It follows that there can be at most n^2 iterations. ■

Two points are worth noting about the previous fact and its proof. First, there are executions of the algorithm (with certain preference lists) that can involve close to n^2 iterations, so this analysis is not far from the best possible. Second, there are many quantities that would not have worked well as a *progress measure* for the algorithm, since they need not strictly increase in each iteration. For example, the number of free individuals could remain constant from one iteration to the next, as could the number of engaged pairs. Thus, these quantities could not be used directly in giving an upper bound on the maximum possible number of iterations, in the style of the previous paragraph.

Let us now establish that the set S returned at the termination of the algorithm is in fact a perfect matching. Why is this not immediately obvious? Essentially, we have to show that no man can "fall off" the end of his preference list; the only way for the `While` loop to exit is for there to be no free man. In this case, the set of engaged couples would indeed be a perfect matching.

So the main thing we need to show is the following.

(1.4) *If m is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.*

Proof. Suppose there comes a point when m is free but has already proposed to every woman. Then by (1.1), each of the n women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must also be n engaged men at this point in time. But there are only n men total, and m is not engaged, so this is a contradiction. ■

(1.5) *The set S returned at termination is a perfect matching.*

Proof. At no time is anyone engaged to more than one person, and so the set of engaged pairs always forms a matching. Let us suppose that the algorithm terminates with a free man m. At termination, it must be the case that m had already proposed to every woman,

for otherwise the `While` loop would not have exited. But this contradicts (1.4), which says that there cannot be a free man who has proposed to every woman. ■

Finally, we prove the main property of the algorithm — namely, that it results in a stable matching.

(1.6) *Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.*

Proof. We have already seen, in (1.5), that S is a perfect matching. Thus, to prove S is a stable matching, we will assume that there is an instability with respect to S and obtain a contradiction. As defined above, such an instability would involve two pairs (m, w) and (m', w') in S with the properties that

- m prefers w' to w, and
- w' prefers m to m'.

In the execution of the algorithm that produced S, m's last proposal was, by definition, to w. Now we ask: did m propose to w' at some earlier point in this execution? If he didn't, then w must occur higher on m's preference list than w', contradicting our assumption that m prefers w' to w. If he did, then he was rejected by w' in favor of some other man m'', whom w' prefers to m. m' is the final partner of w', so either $m'' = m'$ or, by (1.1), w' prefers her final partner m' to m''; either way this contradicts our assumption that w' prefers m to m'.

It follows that S is a stable matching. ■

All Executions Yield the Man-Optimal Matching

If we think about it, the G-S algorithm is actually under-specified: as long as there is a free man, we are allowed to choose *any* free man to make the next proposal. Different choices specify different executions of the algorithm; this is why, to be careful, we stated (1.6) as "Consider an execution of the G-S algorithm that returns a set of pairs S," instead of "Consider the set S returned by the G-S algorithm."

Thus, we encounter another very natural question:

(†) Do all executions of the G-S algorithm yield the same matching?

This is a genre of question that arises in many settings in computer science: we have an algorithm that runs *asynchronously*, with different independent components performing actions that can be inter-leaved in complex ways, and we want to know how much variability this asynchrony causes in the final outcome. If the independent components are, for example,

engines on different wings of an airplane, the effect of asynchrony on their behavior can be a big deal.

In the present context, we will see that the answer to our question is surprisingly clean: all executions yield the same matching.

There are a number of possible ways to prove a statement such as this, many of which would result in quite complicated arguments. It turns out that the easiest and most informative approach for us will be to uniquely *characterize* the matching that is obtained, and then show that all executions result in the matching with this characterization.

What is the characterization? First, we will say that a woman w is a *valid partner* of a man m if there is a stable matching that contains the pair (m, w). We will say that w is the *best valid partner* of m if w is a valid partner of m, and no woman whom m ranks higher than w is a valid partner of his. We will use $b(m)$ to denote the best valid partner of m.

Now, let S^* denote the set of pairs $\{(m, b(m)) : m \in M\}$. We will prove the following fact.

(1.7) *Every execution of the G-S algorithm results in the set S^*.*

This statement is surprising at a number of levels. First of all, as defined, there is no reason to believe that S^* is matching at all, let alone a stable matching. After all, why couldn't it happen that two men have the same best valid partner? Secondly, the result shows that the G-S algorithm gives the best possible outcome for every man simultaneously; there is no stable matching in which any of the men could have hoped to do better. And finally, it answers our question above by showing that the order of proposals in the G-S algorithm has absolutely no effect on the final outcome.

Despite all this, the proof is not so difficult.

Proof of (1.7). Let us suppose, by way of contradiction, that some execution $\mathcal{E}$ of the G-S algorithm results in a matching S in which some man is paired with a woman who is not his best valid partner. Since men propose in decreasing order of preference, this means that some man is rejected by a valid partner during the execution $\mathcal{E}$ of the algorithm. So consider the first moment during the execution $\mathcal{E}$ in which some man, say m, is rejected by a valid partner w. Again, since men propose in decreasing order of preference, and since this is the first time such a rejection has occurred, it must be that w is m's best valid partner $b(m)$.

The rejection of m by w may have happened either because m proposed and was turned down in favor of w's existing engagement, or because w broke her engagement to m in favor of a better proposal. But either way, at this moment w forms an engagement with a man m' whom she prefers to m.

Since w is a valid partner of m, there exists a stable matching S' containing the pair (m, w). Now we ask: who is m' paired with in this matching? Suppose it is a woman $w' \neq w$.

Since the rejection of m by w was the first rejection of a man by a valid partner in the execution $\mathcal{E}$, it must be that m' had not been rejected by any valid partner at the point in $\mathcal{E}$ when he became engaged to w. Since he proposed in decreasing order of preference, and since w' is clearly a valid partner of m', it must be that m' prefers w to w'. But we have already seen that w prefers m' to m, for in execution $\mathcal{E}$ she rejected m in favor of m'. Since $(m', w) \notin S'$, (m', w) is an instability in S'.

This contradicts our claim that S' is stable, and hence contradicts our initial assumption. ■

So for the men, the G-S algorithm is ideal. Unfortunately, the same cannot be said for the women. For a women w, we say that m is a *valid partner* if there is a stable matching that contains the pair (m, w). We say that m is the *worst valid partner* of w if m is a valid partner of w, and no man whom w ranks lower than m is a valid partner of hers.

(1.8) *In the stable matching S^*, each woman is paired with her worst valid partner.*

Proof. Suppose there were a pair (m, w) in S^* so that m is not the worst valid partner of w. Then there is a stable matching S' in which w is paired with a man m' whom she likes less than m. In S', m is paired with a woman $w' \neq w$; since w is the best valid partner of m, and w' is a valid partner of m, we see that m prefers w to w'.

But from this it follows that (m, w) is an instability in S', contradicting the claim that S' is stable, and hence contradicting our initial assumption. ■

Example: Multiple Stable Matchings

We began by defining the notion of a stable matching; we have now proven that the G-S algorithm actually constructs one, and that it constructs the same one, S^*, in all executions. Here's an important point to notice, however. This matching S^* is not necessarily the *only* stable matching for a set of preference lists; we now discuss a simple example in which there are multiple stable matchings.

Suppose we have a set of two men, $\{m, m'\}$, and a set of two women $\{w, w'\}$. The preference lists are as follows:

m prefers w to w'.
m' prefers w' to w.
w prefers m' to m.
w' prefers m to m'.

In any execution of the Gale-Shapley algorithm, m will become engaged to w, m' will become engaged to w' (perhaps in the other order), and things will stop there. Indeed, this matching

is as good as possible for the men, and — as we see by inspecting the preference lists — as bad as possible for the women.

But there is another stable matching, consisting of the pairs (m', w) and (m, w'). We can check that there is no instability, and the set of pairs is now as good as possible for the women (and as bad as possible for the men). This second stable matching could never be reached in an execution of the Gale-Shapley algorithm in which the men propose, but it would be reached if we ran a version of the algorithm in which the women propose.

So this simple set of preference lists compactly summarizes a world in which *someone* is destined to end up unhappy: the men's preferences mesh perfectly; but they clash completely with the women's preferences. And in larger examples, with more than two people on each side, we can have an even larger collection of possible stable matchings, many of them not achievable by any natural algorithm.

1.2 Computational Tractability

The big focus of this course will be on finding efficient algorithms for computational problems. At this level of generality, our topic seems to encompass the whole of computer science; so what is specific to our approach here?

First, we will be trying to identify broad themes and design principles in the development of algorithms. We will look for paradigmatic problems and approaches that illustrate, with a minimum of irrelevant detail, the basic approaches to designing efficient algorithms. At the same time, it would be pointless to pursue these design principles in a vacuum — the problems and approaches we consider are drawn from fundamental issues that arise throughout computer science, and a general study of algorithms turns out to serve as a nice survey of computational ideas that arise in many areas.

Finally, many of the problems we study will fundamentally have a *discrete* nature. That is, like the Stable Matching Problem, they will involve an implicit search over a large set of combinatorial possibilities; and the goal will be to efficiently find a solution that satisfies certain clearly delineated conditions.

The first major question we need to answer is the following: How should we turn the fuzzy notion of an "efficient" algorithm into something more concrete?

One way would be to use the following working definition:

> An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Let's spend a little time considering this definition. At a certain level, it's hard to argue with; one of the goals at the bedrock of our study of algorithms is that of solving real

problems quickly. And indeed, there is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.

But there are some crucial things missing from the definition above, *even if our main goal is to solve real problem instances quickly on real computers.* The first is that it is not really a complete definition. It depends on *where*, and *how well*, we implement the algorithm. Even bad algorithms can run quickly when applied to small test cases on extremely fast processors; even good algorithms can run slowly when they are coded sloppily. Certain "real" input instances are much harder than others, and it's very hard to model the full range of problem instances that may arise in practice. And the definition above does not consider how well, or badly, an algorithm may *scale* as problem sizes grow to unexpected levels. A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly while the other consumes a huge amount of time.

So what we could ask for is a concrete definition of efficiency that is platform-independent, instance-independent, and of predictive value with respect to increasing input sizes. Before focusing on any specific consequences of this claim, we can at least explore its implicit, high-level suggestion: that we need to take a more mathematical view of the situation.

Let's use the Stable Matching Problem as an example to guide us. The input has a natural "size" parameter N; we could take this to be the total size of the representation of all preference lists, since this is what any algorithm for the problem will receive as input. N is closely related to the other natural parameter in this problem: n, the number of men and the number of women. Since there are $2n$ preference lists, each of length n, we can view $N = 2n^2$, suppressing more fine-grained details of how the data is represented. In considering the problem, we seek to describe an algorithm at a high level, and then analyze its running time mathematically as a function of the input size.

Now, even when the input size to the Stable Matching Problem is relatively small, the *search space* it defines is enormous: there are $n!$ possible perfect matchings between n men and n women, and we need to find one that is stable. The natural "brute-force" algorithm for this problem would plow through all perfect matching by enumeration, checking each to see if it is stable. The surprising punch-line, in a sense, to our solution of the Stable Matching Problem is that we needed to spend time proportional only to N in finding a stable matching from among this stupendously large space of possibilities. This was a conclusion we reached at an *analytical level*. We did not implement the algorithm and try it out on sample preference lists; we reasoned about it mathematically. Yet, at the same time, our analysis indicated how the algorithm could be implemented in practice, and gave fairly conclusive evidence that it would be a big improvement over exhaustive enumeration.

This will be a common theme in most of the problems we study: a compact representation, implicitly specifying a giant search space. For most of these problems, there will be an

obvious "brute-force" solution: try all possibilities, and see if any of them works. Not only is this approach almost always too slow to be useful, it is an intellectual cop-out; it provides us with absolutely no insight into the structure of the problem we are studying. And so if there is a common thread in the algorithms we emphasize in this course, it would be the following:

> An algorithm is efficient if it achieves qualitatively better performance, at an analytical level, than brute-force search.

This will turn out to be a very useful working definition of "efficiency" for us. Algorithms that improve substantially on brute-force search nearly always contain a valuable heuristic idea that makes them work; and they tell us something about the intrinsic computational tractability of the underlying problem itself.

If there is a problem with our second working definition, it is *vagueness*. What do we mean by "qualitatively better performance?" When people first began analyzing discrete algorithms mathematically — a thread of research that began gathering momentum through the 1960's — a consensus began to emerge on how to quantify this notion. Search spaces for natural combinatorial problems tend to grow exponentially in the size N of the input; if the input size increases by one, the number of possibilities increases multiplicatively. We'd like a good algorithm for such a problem to have a better scaling property: when the input size increases by a constant factor — say a factor 2 — the algorithm should only slow down by some constant factor C.

Arithmetically, we can formulate this scaling behavior as follows. Suppose an algorithm has the following property: There are absolute constants $c > 0$ and $k > 0$ so that on every input instance of size N, its running time is bounded by cN^k primitive computational steps. (In other words, its running time is at most proportional to N^k.) For now, we will remain deliberately vague on what we mean by the notion of a "primitive computational step" — but everything we say can be easily formalized in a model where each step corresponds to a single assembly-language instruction on a standard processor, or one line of a standard programming language such as C or Java. In any case, if this running time bound holds, for some c and k, then we say that the algorithm has a *polynomial running time*, or that it is a *polynomial-time algorithm*. Note that any polynomial-time bound has the scaling property we're looking for. If the input size increases from N to $2N$, the bound on the running time increases from cN^k to $c(2N)^k = c \cdot 2^k N^k$, which is a slow-down by a factor of 2^k. Since k is a constant, so is 2^k; of course, as one might expect, lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials.

From this notion, and the intuition expressed above, emerges our third attempt at a working definition of "efficiency."

> An algorithm is efficient if it has a polynomial running time.

Where our previous definition seemed overly vague, this one seems much too prescriptive. Wouldn't an algorithm with running time proportional to n^{100} — and hence polynomial — be hopelessly inefficient? Wouldn't we be relatively pleased with a non-polynomial running time of $n^{1+.02(\log n)}$? The answers are, of course, "yes" and "yes." And indeed, however much one may try to abstractly motivate the definition of efficiency in terms of polynomial time, a primary justification for it is this: *It really works.* Problems for which polynomial-time algorithms exist almost always turn out to have algorithms with running times proportional to very moderately growing polynomials like n, $n \log n$, n^2, or n^3. Conversely, problems for which no polynomial-time algorithm is known tend to be very difficult in practice. There are certainly a few exceptions to this principle — cases, for example, in which an algorithm with exponential worst-case behavior generally runs well on the kinds of instances that arise in practice — and this serves to reinforce the point that our emphasis on worst-case polynomial time bounds is only an abstraction of practical situations. But overwhelmingly, our concrete mathematical definition of polynomial time has turned out to correspond empirically to what we observe about the efficiency of algorithms, and the tractability of problems, in real life.

There is another fundamental benefit to making our definition of efficiency so specific: It becomes negatable. It becomes possible to express the notion that *there is no efficient algorithm for a particular problem.* In a sense, being able to do this is a pre-requisite for turning our study of algorithms into good science, for it allows us to ask about the existence or non-existence of efficient algorithms as a well-defined question. In contrast, both of our previous definitions were completely subjective, and hence limited the extent to which we could discuss certain issues in concrete terms.

In particular, the first of our definitions, which was tied to the specific implementation of an algorithm, turned efficiency into a moving target — as processor speeds increase, more and more algorithms fall under this notion of efficiency. Our definition in terms of polynomial-time is much more an absolute notion; it is closely connected with the idea that each problem has an intrinsic level of computational tractability — some admit efficient solutions, and others do not.

With this in mind, we survey five representative problems that vary widely in their computational difficulty. Before doing this, however, we briefly digress to introduce two fundamental definitions that will be used throughout the course.

1.3 Interlude: Two Definitions

Order of Growth Notation

In our definition of polynomial time, we used the notion of an algorithm's running time being at most *proportional to* N^k. Asymptotic order of growth notation is a simple but useful notational device for expressing this.

Given a function $T(N)$ (say, the maximum running time of a certain algorithm on any input of size N), and another function $f(n)$, we say that $T(N)$ *is* $O(f(N))$ (read, somewhat awkwardly, as "$T(N)$ is order $f(N)$") if there exist constants $c > 0$ and $N_0 \geq 0$ so that for all $N \geq N_0$, we have $T(N) \leq c \cdot f(N)$. In other words, for sufficiently large N, the function $T(N)$ is bounded above by a constant multiple of $f(N)$.

Note that $O(\cdot)$ expresses only an upper bound. There are cases where an algorithm has been proved to have running time $O(N \log N)$; some years pass, people analyze the same algorithm more carefully, and they show that in fact its running time is $O(N)$. There was nothing wrong with the first result; it was a correct upper bound. It's simply that it wasn't the "tightest" possible running time.

There is a complementary notation for lower bounds. Often when we analyze an algorithm — say we have just proven that its running time $T(N)$ is $O(N \log N)$ — we want to show that this upper bound is the best one possible. To do this, we want to express the notion that for arbitrarily large input sizes N, $T(N)$ is *at least* a constant multiple of $N \log N$. Thus, we say that $T(N)$ is $\Omega(N \log N)$ (read as "$T(N)$ is omega $N \log N$") if there exists an absolute constant ϵ so that for infinitely many N, we have $T(N) \geq \epsilon N \log N$. (More generally, we can say that $T(N)$ is $\Omega(f(N))$ for arbitrary functions.)

Finally, if a function $T(N)$ is both $O(f(N))$ and $\Omega(f(N))$, we say that $T(N)$ is $\Theta(f(N))$.

Graphs

As we discussed earlier, our focus in this course will be on problems with a discrete flavor. Just as continuous mathematics is concerned with certain basic structures such as real numbers, vectors, matrices, and polynomials, discrete mathematics has developed basic combinatorial structures that lie at the heart of the subject. One of the most fundamental and expressive of these is the *graph*.

A graph G is simply a way of encoding pairwise relationships among a set of objects. Thus, G consists of a pair of sets (V, E) — a collection V of abstract *nodes*; and a collection E of *edges*, each of which "joins" two of the nodes. We thus represent an edge $e \in E$ as a two-element subset of V: $e = \{u, v\}$ for some $u, v \in V$, where we call u and v the *ends* of e.

We typically draw graphs as in Figure 1.1, with each node as a small circle, and each edge as a line segment joining its two ends.

Edges in a graph indicate a symmetric relationship between their ends. Often we want to encode asymmetric relationships, and for this we use the closely related notion of a *directed graph*. A directed graph G' consists of a set of nodes V and a set of *directed edges* E'. Each $e' \in E'$ is an *ordered pair* (u, v); in other words, the roles of u and v are not interchangeable, and we call u the *tail* of the edge and v the *head*. We will also say that edge e' *leaves node* u and *enters node* v. The notion of leaving and entering extends naturally to sets of vertices: we say that edge e' *leaves a set* $S \subseteq V$ if $u \in S$ and $v \notin S$, and e' *enters* S if $v \in S$ and

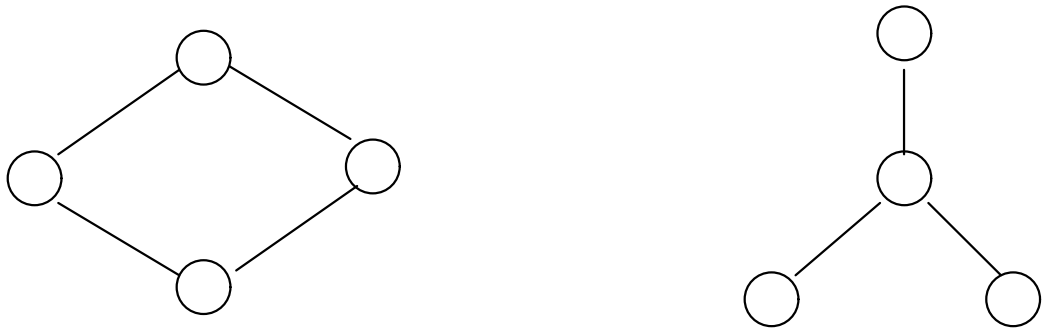


Figure 1.1: Two graphs, each on four nodes.

$u \notin S$.

The more one thinks about graphs, the more one tends to see them everywhere. Here are some examples of graphs:

The collection of all computers on the Internet, with an edge joining any two that have a direct network connection.
The collection of all cities in the world, with an edge joining any two that are within a hundred miles of each other.
The collection of all students at Cornell, with an edge joining any two that know each other.
The collection of all atoms in a cholesterol molecule, with an edge joining any two that have a covalent bond.
The collection of all natural numbers up to $1,000,000$, with an edge joining any two that are relatively prime.

Here are some examples of directed graphs:

The collection of all World Wide Web pages, with an edge (u, v) if u has a hyperlink to v.
The collection of all cities in the world, with an edge (u, v) if there is a non-stop airline flight from u to v.
The collection of all college football teams, with an edge (u, v) if u defeated v in the 1999 season.
The collection of all genes in a human cell, with an edge (u, v) if gene u produces a protein that regulates the action of gene v.
The collection of all courses at Cornell, with an edge (u, v) if course u is an official pre-requisite of course v.
The collection of all natural numbers up to $1,000,000$, with an edge (u, v) if u is a divisor of v.

We pause to mention two warnings in our use of graph terminology. First, although an edge e in an undirected graph should properly be written as a *set* of vertices $\{u, v\}$, one will more often see it written (even in this course) in the notation used for ordered pairs: $e = (u, v)$. Second, *nodes* in a graph are also frequently called *vertices*; in this context, the two words have exactly the same meaning. For both of these things, we apologize in advance.

Bipartite Graphs. We now mention a specific type of (undirected) graph that will be particularly useful in our study of algorithms. We say that a graph $G = (V, E)$ is *bipartite* if its node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y. A bipartite graph is pictured in Figure 1.2; often, when we want to emphasize a graph's "bipartiteness," we will draw it this way, with the nodes in X and Y in two parallel columns. But notice, for example, that the two graphs in Figure 1.1 are also bipartite.

Bipartite graphs are very useful for expressing relationships that arise between two *distinct* sets of objects. For example, here are some bipartite graphs:

The set X of all people who have used amazon.com, the set Y of all books in Amazon's catalog, and an edge from each person to all the books that he or she has purchased through Amazon.

The set X of all houses in New York State, the set Y of all fire stations, and an edge from each house to all the fire stations that are at most a twenty minute drive away.

The set X of all Cornell CS professors, the set Y of all CS courses offered at Cornell, and an edge from each professor to all the courses he or she might be assigned to teach.

The set X of all molecules that Merck Pharmaceuticals knows how to make, the set Y of all enzymes in a human cell, and an edge from each molecule to all the enzymes that it has been observed to inhibit.

1.4 Five Representative Problems

We now discuss five problems that are representative of different themes that will come up in our study of algorithms. We will encounter them in order as the course progresses, since they are associated with a wide range of levels of computational complexity.

Interval Scheduling. Consider the following very simple scheduling problem. You have a resource — it may be a lecture room, or a supercomputer, or an electron microscope — and many people request to use the resource for periods of time. A *request* takes the form: "*Can*

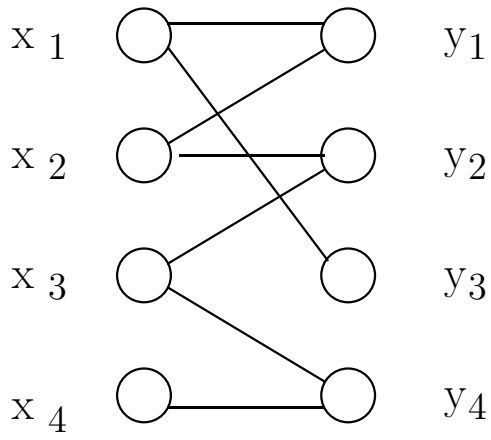


Figure 1.2: A bipartite graph.

I reserve the resource starting at time s, until time f?" We will assume that the resource can be used by at most one person at a time. A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time. The goal is to maximize the number of requests accepted.

More formally, there will be n requests labeled $1, \ldots, n$, with each request i specifying a start time s_i and a finish time f_i. Naturally, we have $s_i < f_i$ for all i. Two requests i and j are *compatible* if the requested intervals do not overlap: that is, either request i is for an earlier time interval than request j ($f_i \leq s_j$), or request i is for a later time than request j ($f_j \leq s_i$). We'll say more generally that a *subset* A of requests is *compatible* if all pairs of requests $i, j \in A$, $i \neq j$ are compatible. The goal is to select a compatible subset of requests of maximum possible size.

We illustrate an instance of this *Interval Scheduling Problem* in Figure 1.3. Note that there is a single compatible set of size four, and this is the largest compatible set.

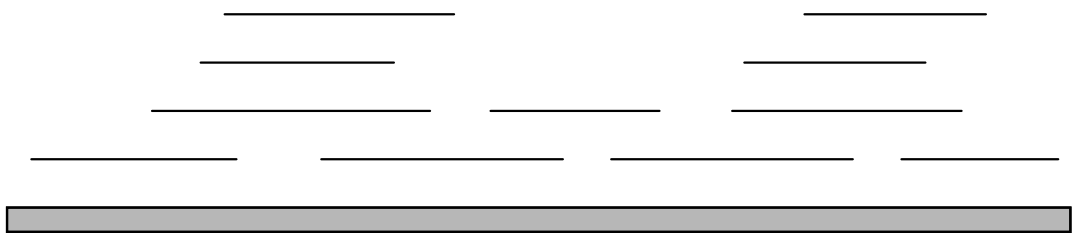


Figure 1.3: An instance of the interval scheduling problem.

We will see shortly that this problem can be solved by a very natural algorithm that orders the set of requests according to a certain heuristic, and then "greedily" processes them in one pass, selecting as large a compatible subset as it can. This will be typical of a class of

greedy algorithms that we will consider for various problems — myopic rules that process the input one piece at a time with no apparent look-ahead. When a greedy algorithm can be shown to find an optimal solution for all instances of a problem, it's often fairly surprising. We typically learn something about the structure of the underlying problem from the fact that such a simple approach can be optimal.

Weighted Interval Scheduling. In the Interval Scheduling Problem, we sought to maximize the *number* of requests that could be accommodated simultaneously. Now, suppose more generally that each request interval i has an associated *value*, or *weight*, $w_i > 0$; we could picture this as the amount of money we will make from the i^{th} individual if we schedule his or her request. Our goal will be to find a compatible subset of intervals of maximum total weight.

The case in which $w_i = 1$ for each i is simply the basic Interval Scheduling Problem; but the appearance of arbitrary weights changes the nature of the maximization problem quite a bit. Consider, for example, that if w_1 exceeds the sum of all other w_i, then the optimal solution must include interval 1 regardless of the configuration of the full set of intervals. So any algorithm for this problem must be very sensitive to the values of the weights, and yet degenerate to a method for solving (unweighted) interval scheduling when all the weights are equal to 1.

There appears to be no simple "greedy" rule that walks through the intervals one at a time, making the correct decision in the presence of arbitrary weights. Instead we employ a technique, *dynamic programming*, that builds up the optimal value over all possible solutions in a compact, tabular way that requires only polynomial time.

Bipartite Matching. When we considered stable marriages, we defined a *matching* to be a set of ordered pairs of men and women with the property that each man and each woman belong to at most one of the ordered pairs. We then defined a *perfect matching* to be a matching in which every man and every woman belong to some pair.

We can express these concepts more generally in terms of bipartite graphs, and it leads to a very rich class of problems. In the case of bipartite graphs, the edges are pairs of nodes, so we say that a *matching* in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M. M is a *perfect matching* if every node appears in *exactly* one edge of M.

To see that this does capture the same notion we encountered in the stable matching problem, consider a bipartite graph G' with a set X of n men, a set Y of n women, and an edge from every node in X to every node in Y. Then the matchings and perfect matchings in G' are precisely the matching and perfect matchings among the set of men and women.

In the stable matching problem, we added preferences to this picture. Here, we do not

consider preferences; but the nature of the problem in arbitrary bipartite graphs adds a different source of complexity: there is not necessarily an edge from every $x \in X$ to every $y \in Y$, so the set of possible matchings has quite a complicated structure. Consider, for example the bipartite graph G in Figure 1.2; there are many matchings in G, but there is only one *perfect matching*. (Do you see it?)

Matchings in bipartite graphs can model situations in which objects are being *assigned* to other objects. Thus, the nodes in X can represent jobs, the nodes in Y can represent machines, and an edge (x_i, y_j) can indicate that machine y_j is capable of processing job x_i. A perfect matching is then a way of assigning each job to a machine that can process it, with the property that each machine is assigned exactly one job. In the spring, the computer science faculty are often seen pondering one of the bipartite graphs discussed earlier, in which X is the set of professors and Y is the set of courses; a perfect matching in this graph consists of an assignment of each professor to a course that he or she can teach, in such a way that every course is covered.

Thus, the *Bipartite Matching Problem* is the following: given an arbitrary bipartite graph G, find a matching of maximum size. If $|X| = |Y| = n$, then there is a perfect matching if and only if the maximum matching has size n. We will find that the algorithmic techniques discussed above do not seem adequate for providing an efficient algorithm for this problem. There is, however, a very elegant polynomial-time algorithm to find the maximum matching; it inductively builds up larger and larger matchings, selectively backtracking along the way. This process is called *augmentation*, and it forms the central component in a large class of efficiently solvable problems called *network flow problems*.

Independent Set. Now let's talk about an extremely general problem, which includes most of these earlier problems as special cases. Given a graph $G = (V, E)$, we say a set of nodes $S \subseteq V$ is *independent* if no two nodes in S are joined by an edge. The *Independent Set Problem* is then the following: given G, find an independent set that is as large as possible.

The *Independent Set* Problem encodes any situation in which you are trying to choose from among a collection of objects, and there are pairwise *conflicts* among some of the objects. Say you have n friends, and some pairs of them don't get along. How large a group of your friends can you invite to dinner, if you don't want there to be any inter-personal tensions? This is simply the largest independent set in the graph whose nodes are your friends, with an edge between each conflicting pair.

Interval Scheduling and Bipartite Matching can both be encoded as special cases of the *Independent Set* Problem. For Interval Scheduling, define a graph $G = (V, E)$ in which the nodes are the intervals, and there is an edge between each pair of them that overlap; the independent sets in G are then just the compatible subsets of intervals. Encoding Bipartite Matching as a special case of *Independent Set* is a little trickier to see. Given a bipartite

graph $G' = (V', E')$, the objects being chosen are *edges*, and the *conflicts* arise between two edges that share an end. (These, indeed, are the pairs of edges that cannot belong to a common matching.) So we define a graph $G = (V, E)$ in which the *node set* V is equal to the *edge set* E' of G'. We define an edge between each pair of elements in V that correspond to edges of G' with a common end. We can now check that the independent sets of G are precisely the matchings of G'. While it is not complicated to check this, it takes a little concentration to deal with this type of "edges-to-nodes, nodes-to-edges" transformation.[1]

Given the generality of the *Independent Set* Problem, an efficient algorithm to solve it would be quite impressive. It would have to implicitly contain algorithms for Interval Scheduling, Bipartite Matching, and a host of other natural optimization problems.

The current status of *Independent Set* is this: no polynomial-time algorithm is known for the problem, and it is conjectured that no such algorithm exists. The obvious brute-force algorithm would try all subsets of the nodes, checking each to see if it is independent, and then recording the largest one encountered. It is possible that this is close to the best we can do on this problem. We will see later in the course that *Independent Set* is one of a large class of problems that are termed *NP-complete*. No polynomial-time algorithm is known for any of them; but they are all *equivalent* in the sense that a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for all of them.

Here's a natural question: Is there anything *good* we can say about the complexity of the *Independent Set* Problem? One positive thing is the following: If we have a graph G on 1000 nodes, and we want to convince you that it contains an independent set S of size 100, then it's quite easy. We simply show you the graph G, circle the nodes of S in red, and let you check that no two of them are joined by an edge. So there really seems to be a great difference in difficulty between *checking* that something is a large independent set and actually *finding* a large independent set. This may look like a very basic observation — and it is — but it turns out to be crucial in understanding this class of problems. Furthermore, as we'll see next, it's possible for a problem to so hard that there isn't even an easy way to "check" solutions in this sense.

Competitive Facility Location. Finally, we come to our fifth problem, which is based on the following two-player game. Consider McDonald's and Burger King (our two players), competing for market share in a geographic area. First McDonald's open a franchise; then Burger King opens a franchise; then McDonald's; then Burger King; and so on ... Suppose they must deal with zoning regulations that require no two franchises to be located too close

[1] For those who are curious, we note that not every instance of the *Independent Set* Problem can arise in this way from Interval Scheduling or from Bipartite Matching; the full *Independent Set* Problem really is more general. The first graph in Figure 1.1 cannot arise as the "conflict graph" in an instance of Interval Scheduling, and the second graph in Figure 1.1 cannot arise as the "conflict graph" in an instance of Bipartite Matching.

together, and each is trying to make its locations as convenient as possible. Who will win?

Let's make the rules of this "game" more concrete. The geographic region in question is divided into n *zones*, labeled $1, 2, \ldots, n$. Each zone has a *value* b_i, which is the revenue obtained by either of the companies if it opens a franchise there. Finally, certain pairs of zones (i, j) are *adjacent*, and local zoning laws prevent two adjacent zones from each having a fast-food franchise in them, regardless of which company owns them. (They also prevent two franchises from being opened in the same zone.) We model these conflicts via a graph $G = (V, E)$, where V is the set of zones, and (i, j) is an edge of E if the zones i and j are adjacent. The zoning requirement then says that the full set of franchises opened must form an independent set in G.

Thus our game consists of two players, P_1 and P_2, alternately selecting nodes in G, with P_1 moving first. At all times, the set of all selected nodes must form an independent set in G. Suppose that player P_2 has a target bound B, and we want to know: is there a strategy for P_2 so that no matter how P_1 plays, P_2 will be able to select a set of nodes of total value at least B? We will call this an instance of the *Competitive Facility Location Problem*.

Consider, for example, the instance pictured below, and suppose that P_2's target bound is $B = 20$. Then P_2 does have a winning strategy. On the other hand, if $B = 25$, then P_2 does not.

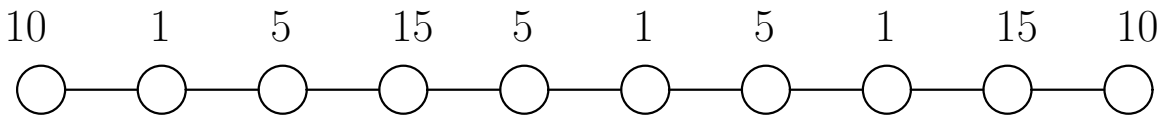


Figure 1.4: An instance of the competitive facility location problem.

One can work this out by looking at the figure for a while; but it requires some amount of case-checking of the form, "If P_1 goes here, then P_2 will go there; but if P_1 goes over there, then P_2 will go here ..." And this appears to be intrinsic to the problem: not only is it computationally difficult to determine whether P_2 has a winning strategy; on a reasonably-sized graph, it would even be hard for us to *convince* you that P_2 has a winning strategy. There does not seem to be a short proof we could present; rather, we'd have to lead you on a lengthy case-by-case analysis of the set of possible moves.

This is in contrast to the *Independent Set* Problem, where we believe that finding a large solution is hard but checking a proposed large solution is easy. This contrast can be formalized in the class of *PSPACE-complete problems*, of which *Competitive Facility Location* is an example. PSPACE-complete problems are believed to be strictly harder than NP-complete problems, and this conjectured lack of short "proofs" for their solutions is one indication of this greater hardness. The notion of PSPACE-completeness turns out to capture a large collection of problems involving game-playing and planning; many of these

are fundamental issues in the area of artificial intelligence.

1.5 Exercises

Note: Exercises denoted with an asterisk ($*$) tend to be more difficult, or to rely on some of the more advanced material.

1. Gale and Shapley published their paper on the stable marriage problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

   Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

   The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

   We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

   - First type of instability: There are students s and s', and a hospital h, so that
     - s is assigned to h, and
     - s' is assigned to no hospital, and
     - h prefers s' to s.
   - Second type of instability: There are students s and s', and hospitals h and h', so that
     - s is assigned to h, and
     - s' is assigned to h', and
     - h prefers s' to s, and
     - s' prefers h to h'.

   So we basically have the stable marriage problem from class, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an efficient algorithm to find one. The input size is $\Theta(mn)$; ideally, you would like to find an algorithm with this running time.

2. We can think about a different generalization of the stable matching problem, in which certain man-woman pairs are explicitly *forbidden*. In the case of employers and applicants, picture that certain applicants simply lack the necessary qualifications or degree; and so they cannot be employed at certain companies, however desirable they may seem. Concretely, we have a set M of n men, a set W of n women, and a set $F \subseteq M \times W$ of pairs who are simply *not allowed* to get married. Each man m ranks all the women w for which $(m, w) \notin F$, and each woman w' ranks all the men m' for which $(m', w') \notin F$.

   In this more general setting, we say that a matching S is *stable* if it does not exhibit any of the following types of instability.

   (i) There are two pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. *(The usual kind of instability.)*

   (ii) There is a pair $(m, w) \in S$, and a man m', so that m' is not part of any pair in the matching, $(m', w) \notin F$, and w prefers m' to m. *(A single man is more desirable and not forbidden.)*

   (ii′) There is a pair $(m, w) \in S$, and a woman w', so that w' is not part of any pair in the matching, $(m, w') \notin F$, and m prefers w' to w. *(A single woman is more desirable and not forbidden.)*

   (iii) There is a man m and a woman w, neither of which is part of any pair in the matching, so that $(m, w) \notin F$. *(There are two single people with nothing preventing them from getting married to each other.)*

   Note that under these more general definitions, a stable matching need not be a perfect matching.

   Now we can ask: for every set of preference lists and every set of forbidden pairs, is there always a stable matching? Resolve this question by doing one of the following two things: (a) Giving an algorithm that, for any set of preference lists and forbidden pairs, produces a stable matching; or (b) Giving an example of a set of preference lists and forbidden pairs for which there is no stable matching.

3. Consider a town with n men and n women seeking to get married to one another. Each man has a preference list that ranks all the women, and each woman has a preference list that ranks all the men.

The set of all $2n$ people is divided into two categories: *good* people and *bad* people. Suppose that for some number k, $1 \leq k \leq n-1$, there are k good men and k good women; thus there are $n-k$ bad men and $n-k$ bad women.

Everyone would rather marry any good person than any bad person. Formally, each preference list has the property that it ranks each good person of the opposite gender higher than each bad person of the opposite gender: its first k entries are the good people (of the opposite gender) in some order, and its next $n-k$ are the bad people (of the opposite gender) in some order.

(a) Show that there **exists** a stable matching in which every good man is married to a good woman.

(b) Show that in **every** stable matching, every good man is married to a good woman.

4. $(*)$ For this problem, we will explore the issue of *truthfulness* in the stable matching problem, and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman w. Suppose w prefers man m to m', but both m and m' are low on her list of preferences. Can it be the case that by switching the order of m and m' on her list of preferences (i.e., by falsely claiming that she prefers m' to m) and running the algorithm with this false preference list, w will end up with a man m'' that she truly prefers to both m and m'? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

   Resolve this questions by doing one of the following two things:

   (a) Giving a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman's partner in the Gale-Shapley algorithm; or

   (b) Giving an example of a set of preference lists for which there is a switch that would improve the partner of a woman who switched preferences.

5. There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.

   Suppose we have two television networks; let's call them AOL-Time-Warner-CNN and Disney-ABC-ESPN, or $\mathcal{A}$ and $\mathcal{D}$ for short. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a *schedule* — an assignment of each show to a distinct slot — so as to attract as much market share as possible.

   Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed *Nielsen rating*, which is based on

the number of people who watched it last year; we'll assume that no two shows have exactly the same rating. A network *wins* a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to *win* as many time slots as possible.

Suppose in the opening week of the fall season, Network $\mathcal{A}$ reveals a schedule S and Network $\mathcal{D}$ reveals a schedule T. On the basis of this pair of schedules, each network wins certain of the time slots, according to the rule above. We'll say that the pair of schedules (S, T) is *stable* if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' so that Network $\mathcal{A}$ wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' so that Network $\mathcal{D}$ wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapley's question for this kind of stability is: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things: (a) Giving an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or (b) Giving an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships, and provides service to n ports. Each of its ships has a *schedule* which says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

   (†) No two ships can be in the same port on the same day.

   The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

   Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an efficient algorithm to find them.

Example: Suppose we have two ships and two ports, and the "month" has four days. Suppose the first ship's schedule is

port P_1; at sea; port P_2; at sea

and the second ship's schedule is

at sea; port P_1; at sea; port P_2

Then the (only) way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.

7. Some of your friends are working for CluNet, a builder of large communication networks, and they are looking at algorithms for switching in a particular type of input/output crossbar.

   Here is the set-up. There are n *input wires* and n *output wires*, each directed from a *source* to a *terminus*. Each input wire meets each output wire in exactly one distinct point, at a special piece of hardware called a *junction box*. Points on the wire are naturally ordered in the direction from source to terminus; for two distinct points x and y on the same wire, we say that x is *upstream* from y if x is closer to the source than y, and otherwise we say x is *downstream* from y. The order in which one input wire meets the output wires is not necessarily the same as the order in which another input wire meets the output wires. (And similarly for the orders in which output wires meet input wires.)

   Now, here's the switching component of this situation. Each input wire is carrying a distinct data stream, and this data stream must be *switched* onto one of the output wires. If the stream of Input i is switched onto Output j, at junction box B, then this stream passes through all junction boxes *upstream* from B on Input i, then through B, then through all junction boxes *downstream* from B on Output j. It does not matter which input data stream gets switched onto which output wire, but each input data stream must be switched onto a *different* output wire. Furthermore — and this is the tricky constraint — no two data streams can pass through the same junction box following the switching operation.

   Finally, here's the question. Show that for any specified pattern in which the input wires and output wires meet each other (each pair meeting exactly once), a valid switching of the data streams can always be found — one in which each input data

stream is switched onto a different output, and no two of the resulting streams pass through the same junction box. Additionally, give an efficient algorithm to find such a valid switching. (The accompanying figure gives an example with its solution.)

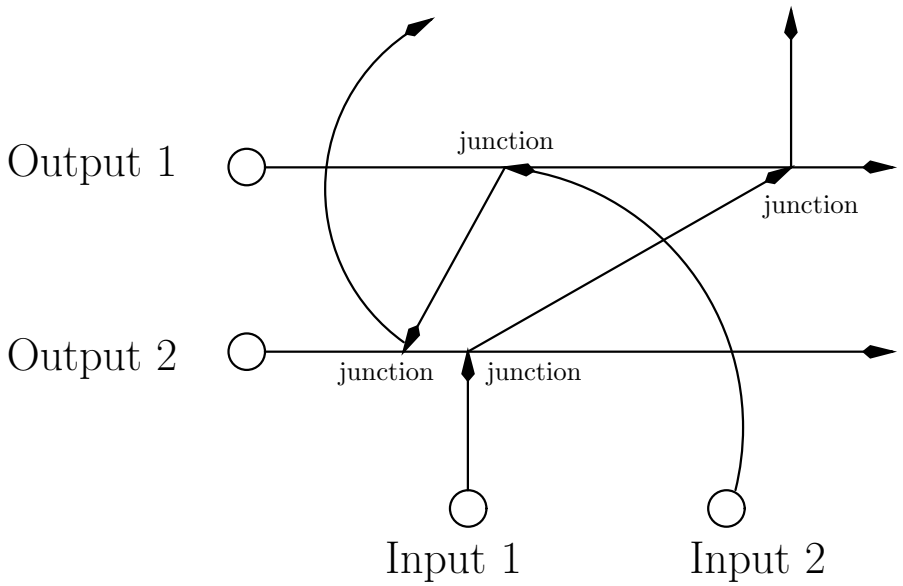


Figure 1.5: An example with two input wires and two output wires. Input 1 has its junction with Output 2 upstream from its junction with Output 1; Input 2 has its junction with Output 1 upstream from its junction with Output 2. A valid solution is to switch the data stream of Input 1 onto Output 2, and the data stream of Input 2 onto Output 1. On the other hand, if the stream of Input 1 were switched onto Output 1, and the stream of Input 2 were switched onto Output 2, then both streams would pass through the junction box at the meeting of Input 1 and Output 2 — and this is not allowed.

Chapter 2

Algorithmic Primitives for Graphs

Much of the course is concerned with techniques for designing algorithms, and graphs will be a ubiquitous modeling tool in this process. As such, it is important to lay the foundation by developing some algorithmic primitives for graphs.

We begin by considering algorithms that operate on undirected graphs, and then move on to the case of directed graphs. Many of the basic concepts for undirected graphs carry over to directed graphs in a more complex form; and new issues arise with directed graphs that would not have made sense in the undirected context. In keeping with our view of undirected graphs as the more basic object, we'll use the word "graph" (with no modifier) to mean an undirected graph by default.

2.1 Representing Graphs

In order to consider algorithms that operate on graphs, we need to understand how a graph $G = (V, E)$ will be presented as input; such a representation should allow us quick access to the nodes and edges of G, and also allow the algorithm to modify the structure of G as necessary.

There are two standard approaches to this representation problem. In describing both, we will assume that the G has n nodes, and that they are labeled $\{1, 2, \ldots, n\}$.

Adjancency matrices. An *adjacency matrix* is simply a two-dimensional array A, with n rows and n columns, where $n = |V|$. The entry $A[i, j]$ is equal to 1 if there is an edge joining i and j, and it is equal to 0 otherwise. Thus, row i of A "corresponds" to the node i, in that the sequence of 0's and 1's in row i indicate precisely the nodes to which i has edges. Column i of A performs the same function; notice that since G is undirected, A is symmetric: $A[i, j] = A[j, i]$.

Adjacency matrices have nice properties. We can determine whether i and j are joined by an edge in constant time, by simply querying the array entry $A[i, j]$. We can modify G

by inserting or deleting the edge (i, j), also in constant time, simply by switching the value of $A[i, j]$.

Given this, why would we want any other representation for G? There are two reasons. First, the adjacency matrix of G is an enormous object; it has size $\Theta(n^2)$, even when G has many fewer than n^2 edges. As a thought experiment, consider the graph G that has one node for each person in the world, and an edge (i, j) whenever i and j know each other on a first name basis. If we assume there are about six billion people in the world; and each person knows about 1000 other people (this turns out to be a reasonable estimate); and each person is represented by a 5-byte unique identifer (with room to spare); then it would be possible to write down a complete description of G in about 30 trillion bytes by simply listing, for each person, all the other people they know. (Here we're just considering the space required to store G, not the effort involved in collecting the data for constructing G.) Thirty trillion bytes is very large, to be sure, but it's something manageable — it's comparable to the size of a large crawl of the Web, and much less than the amount of customer transaction data stored in Wal-Mart's databases. On the other hand, the adjacency matrix of G would be something utterly beyond our power to store: it would require more than 10^{19} entries, an amount of space significantly greater than the capacity of all the hard disks sold world-wide last year.

Here's a second, related, problem with the adjacency matrix. Think of G as an object we'd like to be able to "explore" — we get dropped down on a node i, we look around, and we see which edges lead away from i. We'd like to be able to scan for these edges in an amount of time proportional to the number of edges that are actually incident to i. This will turn out to be crucial in many of the algorithms we consider. Using the adjacency matrix, however, our only option for determining the edges incident to i would be to read all of row i, keeping track of which entries were equal to 1. This takes time $\Theta(n)$, even when very few edges actually have ends equal to i. So in the case of the acquaintance-ship graph G of the U.S. that we've just been considering, for example, we'd like a representation of G that would let us scan the friends of a person i in roughly 1000 steps (proportional to the number of i's friends) rather than six billion steps (proportional to the full population).

Let's move on to a representation that has these features: its size is only proportional to the number of nodes and edges in G, and it lets us scan the "neighborhood" of a node quickly.

Adjacency lists. An adjacency list structure consists of an n-element array $\mathbf{V}$, where $\mathbf{V}[i]$ represents node i. $\mathbf{V}[i]$ simply points to a doubly linked list L_i that contains an entry for each edge $e = (i, j)$ incident to i; this entry for e records the other end of the edge as well. Thus, if we want to enumerate all the edges incident to node i, we first locate entry $\mathbf{V}[i]$, in constant time, and then walk through the doubly linked list L_i of edges that it points to. If

there are d incident edges, this takes time $O(d)$, independent of n.

Notice that each edge $e = (i, j)$ appears twice in the adjacency list representation; once as $e = (i, j)$ in the list L_i, and once as $e = (j, i)$ in the list L_j. We will require that these two entries have pointers from one to the other; this way, for example, if we have our hands on the copy of e in L_i and we'd like to delete it, we can quickly delete the copy in L_j as well by following the pointer to it.

It's important to note that there are some respects in which an adjacency matrix is better than an adjacency list. Using an adjacency list, we can't be told a pair of nodes i and j, and decide in constant time whether G contains the edge (i, j). To do this with the adjacency list, we'd need to walk through the list L_i of all edges incident to i, seeing if any had j as their other end; and this would take time proportional to the length of L_i.

It is easy to create hybrid versions of these two representations, which combine advantages of each. Suppose, for example, that we concurrently maintained a copy of the adjacency matrix of G as well as the adjacency list of G. The entries in the two structures would be "cross-linked" — the edge $e = (i, j)$ in the list L_i would have an extra pointer to the entry $A[i, j]$, and the entry $A[i, j]$ in the adjacency matrix would now have an extra pointer to the entry for $e = (i, j)$ in the list L_i. In this way, we get the ability to quickly scan the edges incident to i, as with adjacency lists, as well as the ability to determine in constant time whether i and j are joined by an edge. At the same time, this hybrid structure is not a pure improvement over the simple adjacency list, for it has inherited the enormous size ($\Theta(n^2)$) of the adjacency matrix, even when G has relatively few edges.

2.2 Paths, Cycles, and Trees

Since graphs so often model transportation or communication networks, a fundamental operation in graphs is that of traversing a sequence of nodes that are connected by edges. (Think of traveling from Ithaca to San Francisco on a sequence of airline flights, or tracing the route of a packet on the Internet through a sequence of intermediate routers.)

With this notion in mind, we define a *path* in a graph $G = (V, E)$ to be a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G. P is often called a path *from* v_1 *to* v_k, or a v_1-v_k path. We call such a sequence of nodes a *cycle* if $v_1 = v_k$ — in other words, the sequence "cycles back" to where it began. A path is called *simple* if all its vertices are distinct; it is called a *simple cycle* if $v_1, v_2, \ldots, v_{k_1}$ are all distinct, and $v_1 = v_k$.

We say that a graph is *connected* if for every pair of nodes u and v, there is a path from u to v. In thinking about what it means, structurally, for a graph to be connected, it turns out to be very helpful to think about the "simplest" possible connected graphs — those containing the minimal number of edges necessary for connectivity. In particular, consider a

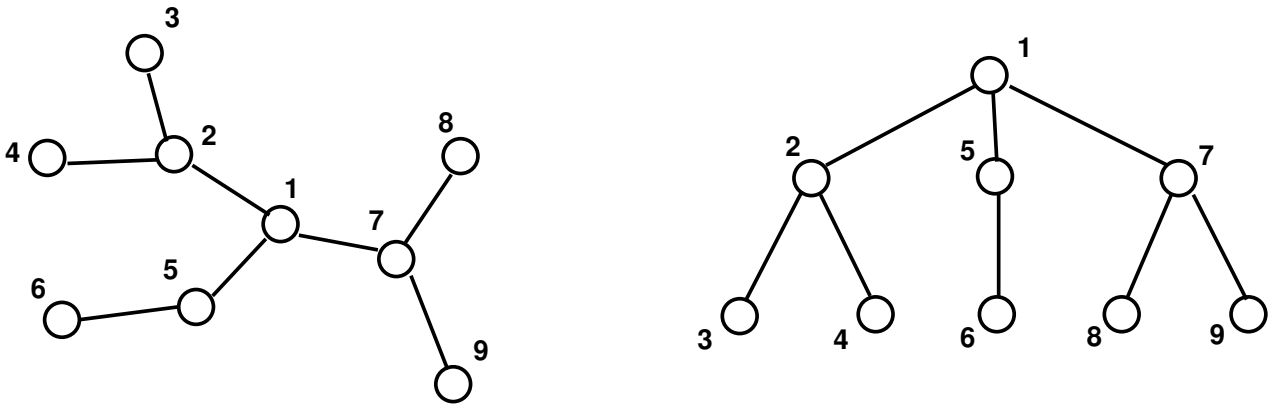


Figure 2.1: Two drawings of the same tree.

connected graph G with a simple cycle C. If $e = (u, v)$ is an edge on C, then G will remain connected even after we delete e — indeed, any path that previously used e can now be re-routed "the long way" around C, and so there is still a path joining each pair of nodes. So whenever we find a cycle in a graph G, we can delete an edge while keeping the graph connected; continuing in this way, we would end up with a graph that is still connected but has no more cycles. We will call such graphs *trees*: A tree is a connected graph with no cycles.

The two graphs pictured in Figure 2.1 are trees. In a strong sense, trees are indeed the "simplest" kind of connected graph: deleting any edge from a tree will disconnect it.

For thinking about the structure of a tree T, it is useful to *root* it at a particular node r. Physically, this is the operation of grabbing T at the node r, and letting the rest of it hang downward under the force of gravity, like a mobile. More precisely, we "orient" each edge of T away from r; for each other node v, we declare the *parent* of v to be the node u that directly precedes v on its path from r; we declare w to be a *child* of v if v is the parent of w. More generally, we say that w is a *descendent* of v (or v is an *ancestor* of w) if v lies on the path from the root to w; and we say that a node x is a *leaf* if it has no descendents. Thus, for example, the two pictures in Figure 2.1 correspond to the same tree T — the same pairs of nodes are joined by edges — but the drawing on the right represents the result of rooting T at node 1.

Rooted trees are fundamental objects in computer science, because they encode the notion of a *hierarchy*. For example, we can imagine the rooted tree in Figure 2.1 as corresponding to the organizational structure of a tiny 9-person company; employees 3 and 4 report to employee 2; employees 2, 5, and 6 report to employee 1; and so on. Many Web sites are organized according to a tree-like structure, to facilitate navigation. A typical computer science department's Web site will have an entry page as the root; the *People* page is a child of this entry page (as is the *Courses* page); pages entitled *Faculty* and *Students* are children

of the *People* page; individual professors' home pages are children of the *Faculty* page; and so on.

For our purposes here, rooting a tree T can make certain questions about T conceptually easy to answer. For example, given a tree T on n nodes, how many edges does it have? Each node other than the root has a single edge leading "upward" to its parent; and conversely, each edge leads upward from precisely one non-root node. Thus we have very easily proved the following fact.

(2.1) *Every n-node tree has exactly $n - 1$ edges.*

In fact, the following stronger statement is true, although we do not prove it here.

(2.2) *Let G be a graph on n nodes. Any two of the following statements implies the third.*

(i) G is connected.

(ii) G does not contain a cycle.

(iii) G has $n - 1$ edges.

We now turn to the role of trees in the fundamental algorithmic idea of *graph traversal*.

2.3 Graph Connectivity and Graph Traversal

Having built up some fundamental notions regarding graphs, we turn to a very basic algorithmic question: node-to-node connectivity. Suppose we are given a graph $G = (V, E)$, and two particular nodes s and t. We'd like to find an efficient algorithm that answers the question: is there a path from s to t in G? We will call this the problem of determining *s-t connectivity*.

For very small graphs, this question can often be answered easily by visual inspection. But for large graphs, it can take some work to search for a path — the challenge in solving mazes, for example, boils down to this question. How efficient an algorithm can we design for this task?

The basic idea in searching for a path is to "explore" the graph G starting from s, maintaining a set R consisting of all nodes that s can reach. Initially, we set $R = \{s\}$. If at any point in time, there is an edge (u, v) where $u \in R$ and $v \notin R$, then we claim it is safe to add v to R. Indeed, if there is a path P from s to u, then there is a path from s to v obtained by first following P and then following the edge (u, v). Suppose we continue this of growing the set R until there are no more edges leading out of R; in other words, we run the following algorithm.

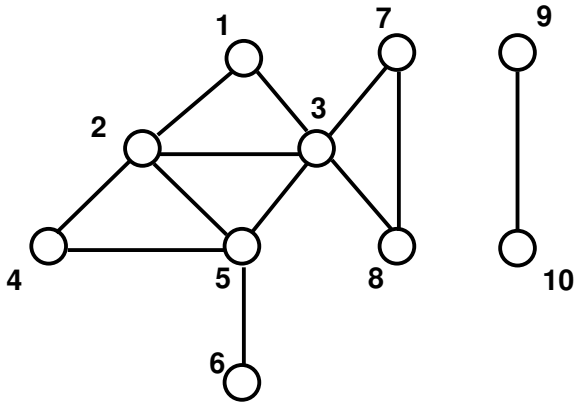


Figure 2.2:

```
R will consist of nodes to which s has a path.
Initially R = {s}.
While there is an edge (u,v) where u ∈ R and v ∉ R
  Add v to R.
Endwhile
```

Here is the key property of this algorithm.

(2.3) *The set R produced at the end of the algorithm consists of precisely the nodes to which s has a path.*

Proof. We have already argued that for any node $v \in R$, there is a path from s to v. In reality, our argument above hides a proof by induction on the number of iterations of the `While` loop: assuming that the set R produced after k steps of the loop contain only nodes reachable from R, then the node added in the $(k+1)^{\text{st}}$ step must also be reachable from s.

Now, consider a node $w \notin R$, and suppose by way of contradiction that there is an s-w path in G. Since $s \in R$ but $w \notin R$, there must be a first node v on P that does not belong to R; and this node v is not equal to s. Thus, there is a node u immediately preceding v on P, so (u, v) is an edge. Moreover, since v is the first node on P that does not belong to R, we must have $u \in R$. It follows that (u, v) is an edge where $u \in R$ and $v \notin R$; this contradicts the stopping rule for the algorithm. ■

We call this set R the *connected component* of G containing s. In view of (2.3) , our algorithm for determining s-t connectivity can simply produce the connected component R of G containing s, and determine whether $t \in R$. (Clearly, if we don't want the whole component, just the answer to the s-t connectivity question, then we can stop the growing of R as soon as t is discovered.) Observe that it is easy to recover the actual path from s to t, along the lines of the inductive argument in (2.3): we simply record, for each node v, the

edge (u, v) that was considered in the iteration in which v was added to S. Then, by tracing these edges backward from t, we proceed through a sequence of nodes that were added in earlier and earlier iterations, eventually reaching s; this defines an s-t path.

The algorithm that grows R is under-specified — how do we decide which edge to consider next? We will show two ways of specifying the answer to this question; they both lead to efficient connectivity algorithms, but with qualitatively different properties.

In both cases, "efficient" will mean the following. If G has n nodes and m edges, then in the worst case we may have to spend time proportional to $m + n$ just to look at the input. We will produce algorithms running in *linear time*; i.e. requiring time $O(m + n)$, which is the best possible.

Breadth-First Search

Perhaps the simplest way to grow the component R is in *layers.* We start with the node s, and add all nodes that are joined by an edge to s — this is the first layer. We then add all the nodes that are joined by an edge to any node in the first layer — this is the second layer. We continue in this way until we have a set of nodes with no edges leaving it; this is the component R. We call this algorithm *Breadth-First Search (BFS)*, since it explores G by considering all nearby nodes first, rather than exploring deeply in any particular direction.

Indeed, BFS is simply a particular implementation of our previous algorithm, defined by a particular way of choosing edges to explore. We summarize it as follows.

```
BFS(s):
  Mark s as "Visited".
  Initialize R = {s}.
  Define layer L_0 = {s}.
  While L_i is not empty
    For each node u ∈ L_i
      Consider each edge (u, v) incident to v
      If v is not marked "Visited" then
        Mark v "Visited"
        Add v to the set R and to layer L_{i+1}
      Endif
    Endfor
  Endwhile
```

If we store each layer L_i as a queue, then inserting nodes into layers and subsequently accessing them takes constant time per node. Furthermore, if we represent G using an adjacency list, then we spend constant time per edge over the course of the whole algorithm, since we consider each edge e at most once from each end. Thus, the overall time spent by the algorithm is $O(m + n)$.

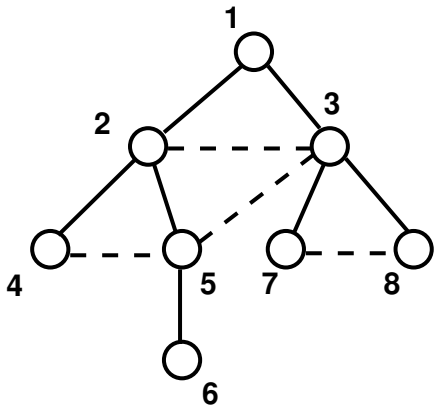


Figure 2.3: A breadth-first search tree T for the graph in Figure 2.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

The BFS algorithm has several useful properties. First, suppose we define the *distance* between two nodes u and v to be the minimum number of edges in a u-v path. (We can designate some symbol like ∞ to denote the distance between nodes that are not connected by a path.) The term "distance" here comes from imagining G as representing a communication or transportation network; if we wanted to get from u to v, we may well want a route with as few "hops" as possible. Now it is easy to see that breadth-first search, in addition to determining connectivity, is also computing distances; the nodes in layer L_i are precisely the nodes at distance i from s.

A further property of breadth-first search is that it produces, in a very natural way, a tree T on the set R, rooted at s. Specifically, consider a point in the BFS algorithm when the edges incident to a node $u \in L_i$ are being examined; we come to an edge (u, v) where v is not yet visited, and we add v to layer L_{i+1}. At this moment, we add the edge (u, v) to the tree T — u becomes the parent of v, representing the fact that u is "responsible" for bringing v into the component R. We call the tree T that is produced in this way a *breadth-first search tree* of R.

Figure 2.3 depicts a BFS tree rooted at node 1 for the graph from Figure 2.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T. In fact, the arrangement of these non-tree edges is very constrained relative to the tree T; as we now prove, they can only connect nodes in the same or adjacent layers.

(2.4) *Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G that is not an edge of T. Then i and j differ by at most* 1.

Proof. Suppose by way of contradiction that i and j differed by more than 1; in particular, suppose $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to

x were being examined. At this point, the only nodes marked "Visited" belonged to layers L_{i+1} and below; hence, y could not have been marked "Visited" at this point, and so it should have added to layer L_{i+1} during the examination of the edges incident to x. ■

Depth-First Search

Another natural method to find the nodes reachable from s is the approach you might take if the graph G were truly a maze of interconnected rooms, and you were walking around in it. You'd start from s and try the first edge leading out of it, to a node v. You'd then follow the first edge leading out of v, and continue in this way until you reached a "dead end" — a node for which you had already visited all its neighbors. You'd then back-track till you got to a node with an unvisited neighbor, and resume from there. We call this algorithm *Depth-First Search (DFS)*, since it explores G by going as deeply as possible, and only retreating when necessary.

DFS is also a particular implementation of the generic component-growing algorithm that we introduced initially. It is most easily described in recursive form: we can invoke "DFS" from any starting point, but maintain global knowledge of which nodes have already been visited.

```
DFS(u):
  Mark u as "Visited" and add u to R.
  For each edge (u, v) incident to u
    If v is not marked "Visited" then
      Add v to R.
      Recursively invoke DFS(v).
    Endif
  Endfor
```

To apply this to s-t connectivity, we simply declare all nodes initially to be not visited, and invoke $DFS(s)$.

With G represented using an adjacency list, we spend constant time per edge since we consider it at most once from each end; we also spend constant additional time per node since DFS(u) is invoked at most once for each node u. Thus the total running time is $O(m + n)$.

The DFS algorithm yields a natural rooted tree T in much the same way that BFS did: we make s the root, and make u the parent of v when u is responsible for the discovery of v. That is, whenenver v is marked "Visited" during the invocation of $DFS(u)$, we add the the edge (u, v) to T. The resulting tree is called a *depth-first search tree* of the component R.

Figure 2.4 depicts a DFS tree rooted at node 1 for the graph from Figure 2.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T. DFS trees look quite different from BFS trees; rather than having root-to-leaf paths that are as short as possible, they tend to be quite narrow and deep. However, as in the case of BFS,

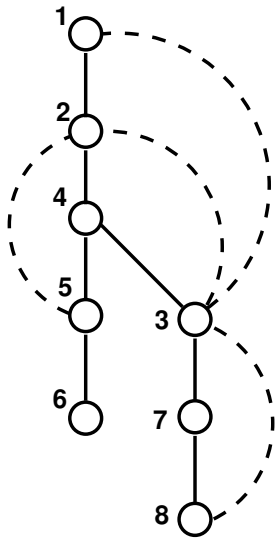


Figure 2.4: A depth-first search tree T for the graph in Figure 2.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

we can say something quite strong about the way in which non-tree edges of G must be arranged relative to the edges of a DFS tree T: as in the figure, non-tree edges can only connect ancestors of T to descendents.

To establish this, we first observe the following property of the DFS algorithm and the tree that it produces.

(2.5) *For a given recursive call $DFS(u)$, all nodes that are marked "Visited" between the invocation and end of this recursive call are descendents of u in T.*

Using (2.5), we prove

(2.6) *Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.*

Proof. Suppose that (x, y) is an edge of G that is not an edge of T, and suppose without loss of generality that x is reached first by the DFS algorithm. When the edge (x, y) is examined during the execution of $DFS(x)$, it is not added to T because y is marked "Visited." Since y was not marked "Visited" when $DFS(x)$ was first invoked, it is a node that was discovered between the invocation and end of the recursive call $DFS(x)$. It follows from (2.5) that y is a descendent of x. ■

Finding all Connected Components

Suppose we don't want to determine a path between a specific pair of nodes, but in fact want to produce *all* the connected components of G. One application would be to determine

whether G is a connected graph; this is the case if and only if it has a single connected component, rather than several.

We can easily use BFS or DFS to do this in time $O(m+n)$. Suppose the nodes are labeled $1, 2, \ldots, n$, and there is an array B that stores the "Visited/Unvisited" status of each node. We first grow the component R_1 containing 1, in time proportional to the number of nodes and edges in R_1. We then walk through the entries of the array B in sequence. Either we reach the end and discover that all nodes have been visited, or we come to the first node i that is not yet visited. The node i must belong to a different connected component, so we proceed to grow the component R_i containing it in time proportional to the number of nodes and edges in R_i. We then return to scanning the array B for unvisited nodes, *starting from the node i*, and continue in this way.

We thus eventually construct all the connected components of G. The time spent on the component-growing procedures is proportional to the sum of the sizes of all components, which is just the number of nodes and edges of G. The additional time spent identifying a new component to grow — by finding a node not yet visited — corresponds to just a single scan of the array B over the course of the algorithm, since we always pick up scanning where we left off. Thus, after $O(m + n)$ time, we have produced all the connected components of G.

2.4 Two Applications of Graph Traversal

We now discuss two applications that make direct use of the special structures of BFS and DFS trees. We first describe how to tell if a graph is bipartite; we then give an algorithm for identifying nodes whose deletion disconnects a graph.

Testing Whether a Graph is Bipartite

Recall the definition of a bipartite graph: it is one where the node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y. To make the discussion a little smoother, we can imagine that the nodes in the set X are colored *red*, and the nodes in the set Y are colored *blue*; with this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

In the previous chapter, we saw examples of bipartite graphs. Here start here by asking: what is an example of a non-bipartite graph, one where no such partition of V is possible?

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node. More generally, consider a cycle C of odd length, with nodes number $1, 2, 3, \ldots, 2k, 2k + 1$. If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on — coloring odd-numbered nodes

red and even-numbered nodes blue. But then we must color node $2k + 1$ red, and it has an edge to node 1, which is also red. This demonstrates that there's no way to partition C into red and blue nodes as required. More generally, if a graph G simply *contains* an odd cycle, then we can apply the same argument; thus we have established that

(2.7) *If a graph G is bipartite, then it cannot contain an odd cycle.*

It is easy to recognize that a graph is bipartite when appropriate sets X and Y (i.e. red and blue nodes) have actually been identified for us; and in many setting where bipartite graph arise, this is natural. But suppose we encounter a graph G with no annotation provided for us, and we'd like to determine for ourselves whether it is bipartite — i.e. whether there exists a partition into red and blue nodes as required. How difficult is this?

In fact, there is a very simple procedure to test for bipartiteness. First, we assume the graph G is connected, since otherwise we can first compute its connected components and analyze each of them separately. Now, we pick any node $s \in V$ and color it red — there is no loss in doing this, since s must receive some color. It follows that all the neighbors of s must be colored blue, so we do this. It then follows that all the neighbors of *these* nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored. At this point, either we have a valid red/blue coloring of G, in which every edge has ends of opposite colors, or there is some edge with ends of the same color. In this latter case, it seems clear that there's nothing we could have done: G simply is not bipartite. We now want to argue this point precisely, and also work out an efficient way to perform the coloring.

In fact, our description of the coloring procedure is almost identical to our description of BFS. Indeed, what we are doing is to color s red, then all of layer L_1 blue, then all of layer L_2 red, and so on. So in fact, by implementing the coloring on top of BFS, we can easily perform it in $O(m + n)$ time. The fact that we are correctly determining whether G is bipartite is now a consequence of the following claim.

(2.8) *Let G be a connected graph, and let $L_0, L_1, L_2, \ldots$ be the layers produced by a BFS starting at node s. Then exactly one of the following two things must hold.*

(i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

(ii) There is an edge of G joining two nodes of the same layer. In this case, G must contain on odd-length cycle, and so it cannot be bipartite.

Proof. In case (i), why does the specified coloring result in every edge having ends of opposite colors? By (2.4), every edge of G joins nodes either in the same layer, or in adjacent layers. Edges that join nodes of adjacenct layers have ends of opposite colors; and our assumption

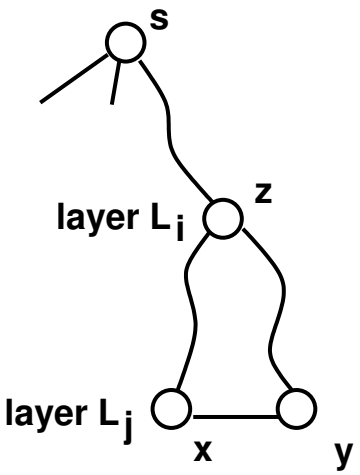


Figure 2.5: Discovering that a graph is not bipartite.

in (i) is that there are *no* edges joining nodes in the same layer. Thus, the specified coloring establishes that G is bipartite.

In case (ii), why must G contain an odd cycle? We are told that G contains an edge joining two nodes of the same layer; suppose this is the edge $x = (x, y)$, with $x, y \in L_j$. Consider the BFS tree T produced by our algorithm, and let z be the node that is in as low a layer as possible, subject to the condition that z is an ancestor of both x and y in T — for obvious reasons, we can call z the *least common ancestor* of x and y. Suppose $z \in L_i$, where $i < j$. We now have the situation pictured in Figure 2.5. We consider the cycle C defined by following the z-x path in T, then the edge e, and then the y-z path in T. The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number. ■

Finding Cut-Points in a Graph

The previous application illustrated how the properties of a BFS tree can be useful in reasoning about the structure of a graph. We now describe a problem for which the properties of a DFS tree — particularly the fact that non-tree edges only join ancestors to descendents — become very useful.

Given a connected graph $G = (V, E)$, we say that $u \in V$ is a *cut-point* if deleting u disconnects G — in other words, if $G - \{u\}$ is not connected. We can think of the cut-points as the "weak points" of G; the destruction of a single cut-point separates the graph into multiple pieces. For example, look at the connected graph G obtained by considering just the nodes 1-8 in Figure 2.2. This graph has two cut-points: the nodes 3 and 5.

How can we find cut-points efficiently? The DFS tree of G, as depicted in Figure 2.4, holds the key to this. Consider, for example, the sub-tree rooted at node 3 — node 3, acting

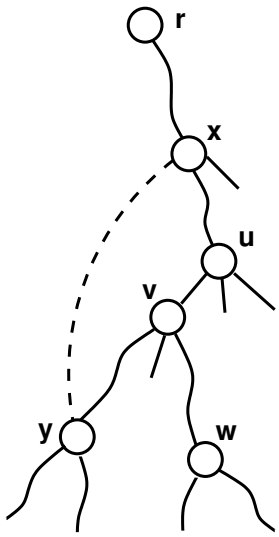


Figure 2.6: Traveling from the root to a node w while avoiding the node u.

as the "entry point" to this sub-tree, separates the nodes below it from the rest of the graph. In particular, since non-tree edges only connect ancestors and descendents, it is enough to observe that there is no edge that "jumps over" node 3, connecting one of its descendents to one of its ancestors, and so there is no way for the nodes below 3 to reach the rest of the graph except through 3. Node 4, on the other hand, lacks this property — both its descendents can reach the rest of the graph through edges that jump over 4.

We now make this kind of reasoning more concrete. We first say that a node u is *earlier* than a node v, relative to the execution of the DFS algorithm, if node u is marked as "Visited" before node v is. We write $u \preceq v$ to denote this. We define $earliest(u)$ to be the earliest node x such that some node in the sub-tree rooted at u is joined by a non-tree edge to x. So in the example of Figure 2.4, we have $earliest(4) = 1$, since node 3 has a non-tree edge to 1; on the other hand, $earliest(7) = 3$ since node 8 itself has a non-tree edge to 3. This latter fact — that $earliest(7) = 3$ — suggests why node 3 is a cut-point: the sub-tree rooted at 7 cannot jump over 3 to get to the rest of the graph.

By looking at these earliest reachable nodes, we can identify in general whether any non-tree edges jump over a given node u to one of its ancestors, and hence whether or not u is a cut-point. Here is the fact that makes this precise.

(2.9) *Let T be a DFS tree of G, with root r.*

(i) A node $u \neq r$ is a cut-point if and only if there is a child v of u for which $u \preceq earliest(v)$.

(ii) The root r is cut-point if and only if it has more than one child.

Proof. Statement (ii) is easier. If r has only one child in T, then even after deleting r there

are still paths in T connecting all other nodes. Conversely, if r has more than one child, then by (2.6), there are no non-tree edges connecting the sub-trees rooted at these children. Hence, deleting r will disconnect the nodes in these sub-trees from one another.

We now prove statement (i). For the first direction, suppose there is a child v of u for which $u \preceq \mathit{earliest}(v)$. Let X denote the set of nodes in sub-tree rooted at v; we claim that there is no edge from a node in X to any node in $G-X$ other than u. Indeed, by (2.6), such an edge would have to go to an ancestor of u; but all such ancestors are earlier than u in the order of the DFS. So it is not possible for such an edge to exist, since $u \preceq \mathit{earliest}(v)$.

Conversely, suppose that all children v of u have the property that $u \not\preceq \mathit{earliest}(v)$. Then we claim that for every node $w \neq u$, there is a path from r to w that does not use u; it will follow that $G-\{u\}$ is still connected. Clearly, r can reach any node that is not in the sub-tree rooted at u, just using the path in T. Now, consider a node w that is in the sub-tree rooted at u; it is also in the sub-tree rooted at v, for some particular child v of u. To get from r to w, we can proceed as follows. Let $x = \mathit{earliest}(v)$, and let y be a node in the sub-tree rooted at v for which (x, y) is an edge. Using edges in T, we can walk from r to x; we can then follow the edge (x, y); we can then use edges in T again to walk from y to v, and on to w. In this way, we have constructed an r-w path that avoids u. The construction is depicted schematically in Figure 2.6. ■

Given (2.9), we can determine the cut-points of G in linear time provided that we can compute the values of $\mathit{earliest}(u)$ for each node u in linear time. The simplest way to do this is straight from the definition of $\mathit{earliest}(\cdot)$, processing a DFS tree from the leaves upward.

```
To compute earliest(u) for all u:
First compute a DFS tree T of G rooted at r.
Now process the nodes in T from the leaves upward,
  so that a node u is only processed after all its children:
  To process node u:
    If u is a leaf, then earliest(u) is just the earliest node
      to which u is joined by a non-tree edge.
      We define earliest(u) = u if u has no incident non-tree edges.
    Else (u is not a leaf)
      Consider the set
        S = {u} ∪
            {w : (u, w) is a non-tree edge} ∪
            {earliest(v) : v is a child of u}
        Define earliest(u) to be the earliest node in S
    Endif
```

The computation of the DFS tree takes $O(m + n)$ time. After this, we spend constant time per edge to compute all the values $\mathit{earliest}(u)$, since we examine each edge at most once

from each end. Finally, we can implement the test in (2.9) in $O(m+n)$ time as well, so as to determine all the cut-points. Thus, the overall time is $O(m+n)$.

It is possible to combine the computation of the values $earliest(u)$ with the recursive procedure that actually performs the DFS; this eliminates the need for an explicitly "two-phase" algorithm that first builds the tree and then computes these values.

2.5 Extensions to Directed Graphs

Thus far, we have been looking at problems on undirected graphs; we now consider the extent to which these ideas carry over the case of directed graphs. Recall that in a directed graph, the edge (u, v) has a direction: it goes from u to v. In this way, the relationship between u and v is asymmetric, and this has qualitative effects on the structure of the resulting graph. For example, consider how different one's browsing experience would be on the World Wide Web if it were possible to follow a hyperlink in either direction.

At the same time, many of our basic definitions, representations, and algorithms have immediate analogues in the directed case. The notion of a *path* still makes sense in a directed graph, but the direction of edges is incorporated into the definition: a path is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that for each $i = 1, 2, \ldots, k-1$, there is an edge (v_i, v_{i+1}). Thus, it can easily happen that there is a path from u to v, but no path from v to u. The notion of a cycle carries over to the directed case analogously: all edges on the cycle must be oriented in the same direction.

The two basic representations — adjacency matrices and adjacency lists — also carry over to the directed case. The adjacency matrix is no longer symmetric, since we will have $A[i, j] \neq A[j, i]$ whenever one of (i, j) or (j, i) is an edge but the other isn't. In the adjacency list structure, it is often useful to have the entry $\mathbf{V}[i]$ point to two lists: the edges for which i is the tail, and (separately) the edges for which i the head. By analogy with the undirected case, we can have pointers cross-linking the appearance of the edge (i, j) in the tail list of i with the appearance of (i, j) in the head list of j.

Finally, the two basic traversal algorithms — breadth-first search and depth-first search — also carry over to the directed case. Each algorithm contains an inner loop in which, for a given node u, we look at all edges incident to u and determine which neighboring nodes have been visited. In a directed graph, we modify these algorithms so that they perform this scan over the edges for which u is the *tail*; otherwise, the algorithms remain exactly the same. The result of these algorithms is the set of nodes R to which u has a path, which may be quite different from the set of nodes R' that have a path to u. If we were interested in computing this latter set R', we could simply run DFS or BFS with the directions of all edges reversed.

We now consider an algorithmic problem that is specific to directed graphs, and has no

obvious analogue in the undirected case.

2.6 Directed Acyclic Graphs and Topological Ordering

If an undirected graph has no cycles, then it has an extremely simple structure — each of its connected components is a tree. But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure. For example, such graphs can have a large number of edges: if we start with the node set $\{1, 2, \ldots, n\}$ and include an edge (i, j) whenever $i < j$, then the resulting directed graph has $\binom{n}{2}$ edges but no cycles. If a directed graph has no cycles, we call it — naturally enough — a *directed acyclic graph*, or a *DAG* for short. (The term "DAG" is typically pronounced as a word, not spelled out as an acronym.)

DAGs are a very common structure in computer science, because they encode *precedence relations*, or *dependencies*, in the following way. Suppose we have a list of tasks labeled $\{1, 2, \ldots, n\}$ that need to be performed, and there are *dependencies* among them stipulating, for certain pairs i and j, that i must be performed before j. For example, the tasks may be courses, with pre-requisite requirements stating that certain courses must be taken before others. Or the tasks may correspond to a pipeline of computing jobs, with assertions that the output of job i is used in determining the input to job j, and hence job i must be done before job j.

We can represent such an inter-dependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j. If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG. Indeed, if it contained a cycle C, there would be no way to do any of the tasks in C: since each task in C cannot begin until some other one completes, no task in C could ever be done, since none could be done first.

Let's continue a little further with this picture of DAGs as precedence relations. Given a set of tasks with dependencies, it would be natural to seek a valid *order* in which the tasks could be performed, so that all dependencies are respected. Specifically, for a directed graph G, we say that a *topological ordering* of G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have $i < j$. In other words, all edges point "forwards" in the ordering. A topological ordering on tasks provides an order in which they can be safely performed; when we come to the task v_j, all the tasks that are required to precede it have already been done.

We can also view a topological ordering of G as providing an immediate "proof" that G has no cycles, via the following.

(2.10) *If G has a topological ordering, then G is a DAG.*

Proof. Suppose by way of contradiction that G has a topological ordering $v_1, v_2, \ldots, v_n$, and also has a cycle C. Let v_i be the lowest-indexed node on C, and let v_j be the node on C just

before v_i — thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ was a topological ordering. ■

The main question we consider here is the converse of (2.10): does every DAG have a topological ordering, and if so, how do we find one efficiently? A method to do this for every DAG would be very useful: it would show that for any precedence relation on a set of tasks without cycles, there is an efficiently computable order in which to perform the tasks.

In fact, the converse of (2.10) does hold, and we establish this via a linear-time algorithm to compute a topological ordering. The key to this lies in finding a way to get started: which node do we put at the beginning of the topological ordering? Such a node v_1 would need to have no in-coming edges, since any such edge would violate the defining property of the topological ordering, that all edges point forward. Thus, we need to prove the following fact.

(2.11) *In every DAG G, there is a node v with no in-coming edges.*

Proof. Let G be a directed graph in which every node has at least one in-coming edge. We show how to find a cycle in G; this will prove the claim. We pick any node v, and begin following edges backward from v: since v has at least one in-coming edge (u, v), we can walk backward to u; then, since u has at least one in-coming edge (x, u), we can walk backward to x; and so on. We can continue this process indefinitely, since every node we encounter has an in-coming edge. But after $n + 1$ steps, we will have visited some node w twice. If we let C denote the sequence of nodes encountered between successive visits to w, then clearly C forms cycle. ■

In fact, the existence of such a node v is all we need to produce a topological ordering of G by induction. We place v first in the topological ordering; this is safe, since all edges out of v will point forward. Now, $G-\{v\}$ is a DAG — deleting v cannot create any cycles — and so by induction it has a topological ordering which we can append after v. In fact, this argument is a complete proof of the desired converse of (2.10).

(2.12) *If G is a DAG, then G has a topological ordering.*

The inductive proof contains the following algorithm to compute a topological ordering of G.

```
To compute a topological ordering of G:
Find a node v with no in-coming edges and order it first.
Delete v from G.
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Identifying the node v and deleting it from G can be done in $O(n)$ time. Since the algorithm runs for n iterations, the total running time is $O(n^2)$.

This is not a bad running time; and if G is very dense, containing $\Theta(n^2)$ edges, then it is linear in the size of the input. But we may well want something better when the number of edges m is much less than n^2; in such a case, a running time of $O(m+n)$ could be a significant improvement over $\Theta(n^2)$.

In fact, we can achieve a running time of $O(m+n)$ using the same high-level algorithm — iteratively deleting nodes with no in-coming edges. We simply have to be more efficient in finding these nodes. We will maintain a queue S consisting of nodes without in-coming edges, which we initialize at the outset. In each iteration, we extract the node v that is at the front of the queue S, and we delete v from G. As we delete each of v's out-going edges (v, w), we check whether this was the last edge entering w; if so, we add w to the end of the queue S.

Observe that this is indeed just an implementation of the high-level inductive algorithm above: by the time each node v comes to the front of the queue S, there are no nodes either in S or still in the graph that have an edge to v, and so it is safe to place v next in the topological ordering. We summarize the algorithm as follows.

```
Improved algorithm to compute a topological ordering of G:
Initialize a queue S consisting of all nodes without in-coming edges.
While S is not empty
  Let v be the node at the front of S.
  Delete v from G:
    For each edge (v,w) in list of v's out-going edges
      Delete (v,w) from the list of w's in-coming edges
      If the list of w's in-coming edges is now empty then
        Add w to the end of S
      Endif
   Endfor
   Place v in the next position of the topological ordering
Endwhile
If all nodes have been deleted from G then
  The algorithm has produced a topological ordering of G
Else
  In the graph on the nodes that remain, every node has
    an in-coming edge, and so there is a cycle by  (2.11) .
Endif
```

What is the running time of this algorithm? Initializing S takes $O(n)$ time. We spend constant time per node when it comes to the front of the queue S. We spend constant time per edge (v, w), at the time when its tail v is being deleted from G. Thus, the overall running time is $O(m+n)$.

2.7 Exercises

1. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists. (They study butterflies.) Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught — thanks to the fact that many species look very similar to one another.

   One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups — those that belong to A, and those that belong to B — but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

   For each pair of specimens i and j, they study them carefully side-by-side; and if they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from opposite species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

   So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B; so more concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have opposite labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

   Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

2. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain the spanning tree T. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same spanning tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

3. When we discussed the problem of determining the cut-points in a graph, we mentioned

that one can compute the values $earliest(u)$ for all nodes u as part of the DFS computation — rather than computing the DFS tree first, and these values subsequently.

Give an algorithm that does this: show how to augment the recursive procedure $DFS(v)$ so that it still runs in $O(m + n)$, and it terminates with globally stored values for $earliest(u)$.

4. A number of recent stories in the press about the structure of the Internet and the Web have focused on some version of the following question: How far apart are typical nodes in these networks? If you read these stories carefully, you find that many of them are confused about the difference between the *diameter* of a network and the *average distance* in a network — they often jump back and forth between these concepts as though they're the same thing.

   As in the text, we say that the *distance* between two nodes u and v in a graph $G = (V, E)$ is the minimum number of edges in a path joining them; we'll denote this by $dist(u, v)$. We say that the *diameter* of G is the maximum distance between any pair of nodes; and we'll denote this quantity by $diam(G)$.

   Let's define a related quantity, which we'll call the *average pairwise distance* in G (denoted $apd(G)$). We define $apd(G)$ to be the average, over all $\binom{n}{2}$ sets of two distinct nodes u and v, of the distance between u and v. That is,

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u, v) \right] / \binom{n}{2}.$$

   Here's a simple example to convince yourself that there are graphs G for which $diam(G) \neq apd(G)$. Let G be a graph with three nodes u, v, w; and with the two edges $\{u, v\}$ and $\{v, w\}$. Then

$$diam(G) = dist(u, w) = 2,$$

   while

$$apd(G) = [dist(u, v) + dist(u, w) + dist(v, w)]/3 = 4/3.$$

   Of course, these two numbers aren't all *that* far apart in the case of this 3-node graph, and so it's natural to ask whether there's always a close relation between them. Here's a claim that tries to make this precise.

   > *Claim: There exists a positive natural number c so that for all graphs G, it is the case that*
   >
   > $$\frac{diam(G)}{apd(G)} \leq c.$$

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

5. Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around (really, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

   They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $n/2$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

   Here's a concrete way to formulate the question as a claim about graphs:

   > *Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $n/2$, then G is connected.*

   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

Chapter 3

Greedy Algorithms

In *Wall Street*, that iconic movie of the 80's, Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed ... is good. Greed is right. Greed works." In this chapter, we'll be taking a much more understated perspective as we investigate the pros and cons of short-sighted greed in the design of algorithms. Indeed, our aim is to approach a number of different computational problems with a recurring set of questions: Is greed good? Does greed work?

It is hard, if not impossible, to define precisely what is meant by a *greedy algorithm*. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to maximize some underlying criterion. One can often design many different "greedy algorithms" for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

When a greedy algorithm succeeds in solving a non-trivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. The same is true of problems in which a greedy algorithm can produce a solution that is guaranteed to be *close* to optimal, even if it does not achieve the precise optimum. These are the kinds of issues we'll be dealing with in this section: It's easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

The first two sections of this chapter will develop two basic methods for proving that a greedy algorithm produces an optimal solution to a problem. One can view the first approach as establishing that *"the greedy algorithm stays ahead"*. By this we mean that if one measures the greedy algorithm's progress in a step-by-step inductive fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution. The second approach is known as an *exchange argument*, and it is more general — one considers any possible solution to the problem, and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will

follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Following our introduction of these two styles of analysis, we focus on two of the most well-known applications of greedy algorithms: *shortest paths in a graph*, and the *minimum spanning tree problem*. They each provide nice examples of our analysis techniques. Finally, we consider a more complex application, the *minimum-cost arborescence problem*, which further extends of our notion of what a greedy algorithm can accomplish.

3.1 The Greedy Algorithm Stays Ahead

Interval Scheduling

Let's recall the *interval scheduling problem*, which was the first of the five representative problems we considered in the introduction to the course. We have a set of requests $\{1, 2, \ldots, n\}$; the i^{th} request corresponds to an interval of time starting at s_i and finishing at f_i. We'll say that a subset of the requests is *compatible* if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. (Compatible sets of maximum size will called *optimal*.)

In defining the problem, we assume that all requests are known to the scheduling algorithm when it is choosing the compatible subset. It would also be natural, of course, to think about the version of the problem in which the scheduler needs to make decisions about accepting or rejecting certain requests before knowing about the full set of requests: Customers (requestors) may well be impatient, they may give up and leave if the scheduler waits too long to gather information about all other requests. Towards the end of the course we will briefly discuss such *on-line* algorithms, which must make decisions as time proceeds, without knowledge of future input. For now we will be concerned with the *off-line* version of the problem in which all information is available to the algorithm at the start.

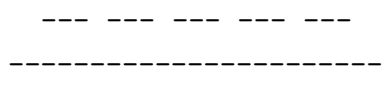
Both off-line and on-line problems arise in many applications; off-line scheduling problems come up in allocating lecture rooms for lectures, or exams for the semester, or a transportation timetable assuming that all routing data is known in advance.

Greedy Algorithms for Interval Scheduling. Using the interval scheduling problem, we can make our discussion of greedy algorithms above much more concrete. The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1. Once a request i_1 is accepted we reject all requests that are not compatible with i_1. We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2. We continue in this fashion until we run out of requests. The challenge in designing a good greedy algorithm is in deciding which simple rule to use for the selection — and there are many natural rules for this problem that do not give good solutions.

Let's try to think of some of the most natural rules, and see how they work.

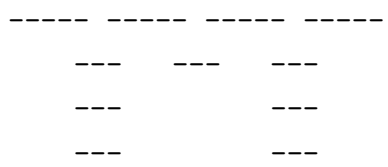
- Maybe the most obvious rule would be to always select the available request that starts earliest. That is, we always pick the request with minimal start time s_i. This way our resource starts being used as quickly as possible.

  This method does not yield an optimal solution. If the earliest request i is for a very long interval, then by accepting request i we have to possibly reject a lot of requests for shorter time intervals; since our goal is to satisfy as many requests as possible, we will end up with a sub-optimal solution. In a really bad case, say when the finish time f_i is the maximum among all requests, the accepted request i keeps our resource occupied for the whole time. In this case our greedy method would accept a single request, while the optimal solution could accept many. We represent this in the simple schematic picture below.



- The arguments above would suggest that we should accept first the request that requires the smallest interval of time; namely, the request for which $f_i - s_i$ is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but still we can get a sub-optimal schedule. For example, consider the picture below: If we have requests for the time intervals $0 - 10$, $9 - 11$, and $10 - 20$, then our greedy method would accept the short interval $9 - 11$ and hence would have to reject both of the other requests. Meanwhile, the optimum schedule rejects request $9 - 11$ and accepts the other two, which are compatible.

- In the previous greedy rule our problem was that the second request competes with both the first and the third; i.e., accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are not compatible, and accept the request which has the fewest number of non-compatible requests. This greedy choice would lead to the optimum solution in the previous example. In fact, it is quite a bit harder to design a bad example for this rule; but one can, and we've drawn an example in the picture below. The unique optimal solution in this example is to accept the four requests in the top row. The greedy method suggested here accepts the middle request in the second row, and thereby ensures a solution of size no greater than three.



The greedy rule that does lead to the optimal solution is based on a fourth idea: we should accept first the request finishes first, i.e., the request i for which f_i is as small as possible. This is also quite a natural idea: we ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

We state the algorithm a bit more formally. We will use R to denote the set of requests that we have neither accepted nor rejected yet, and use A to denote the set of accepted requests.

```
Initially let R be the set of all requests, and A be empty.
While R is not yet empty
   Choose a request i ∈ R that has smallest finishing time
        f_i = min_{j∈R} f_j
   Add request i to A.
   Delete all requests from R that are not compatible with request i.
EndWhile
Return the set A as the set of accepted requests
```

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous non-optimal versions of the greedy method also seemed promising at first.

As a start, we can immediately declare that the intervals in the set A returned by the algorithm are all compatible.

(3.1) *A is a compatible set of requests.*

What we need to show is that this solution is optimal. So, for purposes of comparison, let $\mathcal{O}$ be an optimal set of intervals. Ideally one might want to show that $A = \mathcal{O}$, but this is too much to ask: there may be many optimal solutions, and at best A is equal to a single one of them. So instead we will simply show that $|A| = |\mathcal{O}|$, i.e., that A is also an optimal solution.

The idea underlying the proof, as we suggested initially, will be to find a sense in which our greedy algorithm "stays ahead" of this solution $\mathcal{O}$. We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution $\mathcal{O}$, and show that the greedy algorithm is doing better in a step-by-step fashion.

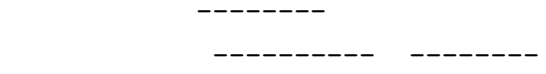
We introduce some notation to help with this proof. Let $i_1, \ldots, i_k$ be the set of requests in A in the order they were added to A. Note that $|A| = k$. Similarly, let the set of requests in $\mathcal{O}$ be denoted by $j_1, \ldots, j_m$. Our goal is to prove that $k = m$. Assume that the requests in $\mathcal{O}$ are also ordered in the natural left to right order of the corresponding intervals, i.e., the order of the start and finish points. Note that the requests in $\mathcal{O}$ are compatible, and this implies that the start points have the same order as the finish points.

Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f_{i_1} \leq f_{j_1}$. First we prove that this is also true for later requests: in the algorithm's schedule, the r^{th} accepted request finishes no later than the r^{th} request in the optimal schedule.

(3.2) *For all indices $r \leq k$ we have $f_{i_r} \leq f_{j_r}$.*

Proof. We will prove this statement by induction. For $r = 1$ the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.

Now let $r > 1$. We will assume as our induction hypothesis that the statement is true for $r - 1$, and we will try to prove it for r. In the figure below we use the lower line to indicate the requests j_{r-1} and j_r in the optimal schedule, and the upper line to indicate the request i_{r-1} from the algorithm's schedule.



By the induction hypothesis we have $f_{i_{r-1}} \leq f_{j_{r-1}}$. Since the optimal schedule consists of compatible intervals, we also know that $f_{j_{r-1}} \leq s_{j_r}$. Combining these two facts, we see that the request j_r is in our set R when request i_r is selected. The greedy algorithm selects the request i_r with smallest finish time, and request j_r is one of the options in R when it makes the selection, so we must have that $f_{i_r} \leq f_{j_r}$. ■

Thus, we have formalized the sense in which the greedy algorithm is remaining ahead of $\mathcal{O}$ — for each r, the r^{th} interval it selects finishes at least as soon as the r^{th} interval in $\mathcal{O}$. We now see why this implies the optimality of the greedy algorithm's set A.

(3.3) *The greedy algorithm returns an optimal set A.*

Proof. We will prove the statement by contradiction. If A is not optimal, than an optimal set $\mathcal{O}$ must have more requests, i.e., we must have $m > k$. Applying (3.2) with $r = k$, we get that $f_{i_k} \leq f_{j_k}$. Since $m > k$, there is a request j_{k+1} in $\mathcal{O}$. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty — a contradiction. ■

Implementation. We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the n requests in order of finishing time, and labeling them in this order; that is, we will assume that $f_i \leq f_j$ when $i < j$. This takes time $O(n \log n)$. In addition $O(n)$ time, we construct an array $S[1 \ldots n]$ with the property that $S[i]$ contains the value s_i.

We now select requests by processing the intervals in order of increasing f_i. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s_j \geq f_1$; we then select this one as well. More generally, if the most recent interval we've selected, we continue iterating through subsequent intervals until we reach the first j for which $s_j \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus, this part of the algorithm takes time $O(n)$ as well.

Variations. The interval scheduling problem we considered here is a quite a simple scheduling problem. There are many further complications that could arise in practical settings. The following are two issues that we will see later.

(i) In our problem we have only a single resource. But one could imagine having many similar lecture rooms, and each request asks to use one of them (any free room is fine) at a specified time.

(ii) Our goal was to maximize the number of satisfied requests. But we could picture the situation in which each request has a different value to us. For example, each request i could also have a weight w_i (the amount gained by satisfying request i), and the goal would be to maximize our income: the sum of the weights of all satisfied requests. This leads to the *weighted interval scheduling problem*, the second of the representative problems we described at the beginning of the course.

Clearly, there are many other variants and combinations that can arise. Interestingly, these problems have quite a range of difficulty. The purpose of this section has been to show that the simple interval scheduling problem discussed above can be solved optimally by a greedy algorithm. Our algorithm can in fact be extended to solve variation (i) in which we seek to schedule many identical resources. We will see a method for solving variation (ii) when we discuss dynamic programming.

Selecting Breakpoints

We now discuss another natural setting in which one can see a greedy algorithm "staying ahead" of all other algorithms, and ending up with an optimal solution.

Suppose that three of your friends, inspired by repeated viewings of that cult phenomenon *The Blair Witch Project*, have decided to hike the Appalachian Trail this summer. They

want to hike as much as possible per day, but — for obvious reasons — not after dark. On a map they've identified a large set of good *stopping points* for camping, and they're considering the following system for deciding when to stop for the day. Each time they come to a potential stopping point, they determine whether they can make it to the next one before nightfall. If they can make it, then they keep hiking; otherwise, they stop.

Despite many significant drawbacks, they claim this system does have one good feature. "Given that we're only hiking in the daylight," they claim, "it minimizes the number of camping stops we have to make."

Is this true? Might it not help to stop early on some day, so as to get better synchronized with camping opportunities on future days? The proposed system is a greedy algorithm, and we wish to determine whether it minimizes the number of stops needed.

To think about the fundamental issue at work here, we make a large number of simplifying assumptions. We'll model the Appalachian Trail as a long line segment of length L, and assume that your friends can hike d miles per day (independent of terrain, weather conditions, and so forth). We'll assume that the potential stopping points are located at distances $x_1, x_2, \ldots, x_n$ from the start of the Trail. We'll also assume (very generously) that your friends are always correct when they estimate whether they can make it to the next stopping point before nightfall.

We'll say that a set of stopping points is *valid* if the distance between each adjacent pair is at most d, the first is at distance at most d from the start of the Trail, and the last is at distance at most d from the end of the Trail. Thus, a set of stopping points is valid if one could camp only at these places, and still make it across the whole Trail. We'll assume, naturally, that the full set of n stopping points is valid; otherwise, there would be no way make it the whole way. Thus, the question is whether your friends' greedy algorithm — hiking as long as possible each day — is *optimal*, in the sense that it finds a valid set whose size is as small as possible.

Indeed, the algorithm is optimal; and we will prove this by identifying the natural sense in which the stopping points it chooses "stay ahead" of any other legal set of stopping points. Note an interesting contrast with the interval scheduling problem — there we needed to prove that a greedy algorithm maximized a quantity of interest, whereas here we seek to minimize a certain quantity.

Let $R = \{x_{p_1}, \ldots, x_{p_k}\}$ denote the set of stopping points chosen by the greedy algorithm, and suppose by way of contradiction that there is a valid set of stopping points $S = \{x_{q_1}, \ldots, x_{q_m}\}$ with $m < k$. We claim the following.

(3.4) *For each $j = 1, 2, \ldots, m$, we have $x_{p_j} \geq x_{q_j}$.*

Proof. We prove this by induction on j. The case $j = 1$ follows directly from the definition of the greedy algorithm — your friends travel as long as possible on the first day before stopping.

Now let $j > 1$ and assume that the claim is true for all $i < j$. Then

$$x_{q_j} - x_{p_{j-1}} \leq x_{q_j} - x_{q_{j-1}} \leq d,$$

where the first inequality follows from our assumption that the claim is true for $j-1$, and the second inequality follows from the fact that S is a valid set. This means that your friends have the option of hiking all the way from $x_{p_{j-1}}$ to x_{q_j} in one day; and hence the location x_{p_j} at which they finally stop can only be farther along than x_{q_j}. ■

(3.4) implies in particular that $x_{q_m} \leq x_{p_m}$. Now, if $m < k$, then we must have $x_{p_m} < L-d$, for otherwise your friends would never have stopped at the location $x_{p_{m+1}}$. Combining these two inequalities, we have concluded that $x_{q_m} < L - d$; but this contradicts the assumption that S is a valid set of stopping points.

Consequently, we cannot have $m < k$, and so we have proved

(3.5) *The greedy algorithm produces a valid set of stopping points of minimum possible size.*

3.2 Exchange Arguments

Scheduling to Minimize Lateness

A greedy algorithm similar to what we saw for interval scheduling works for a closely related problem as well. Proving its optimality for this problem, however, will require a more sophisticated kind of analysis.

Consider again a situation in which we have a single resource, and a set of n requests to use the resource for an interval of time. Assume that the resource is available starting at time s. In contrast to the previous problem, however, each request is now more flexible: Instead of a start time and finish time, the request i has a deadline d_i, and requires a contiguous time interval of length t_i, but it is willing to be scheduled at any time before the deadline. Each accepted request must be assigned an interval of time of length t_i, and different requests must be assigned non-overlapping intervals.

Selecting a maximum-size subset of requests that can be satisfied turns out to be computationally very difficult. Here we consider the following other natural objective function. Suppose that we plan to satisfy each request, but we are allowed to let certain requests run late. Thus, beginning at our overall start time s, we will assign each request i an interval of time of length t_i; let us denote this interval by $[s_i, f_i]$, with $f_i = s_i + t_i$. Unlike the previous problem, then, the algorithm must actually determine a start time (and hence a finish time) for each interval.

We say that a request i is *late* if it misses the deadline, i.e., if $f_i > d_i$. The *lateness* of a such a request i is defined to be $l_i = f_i - d_i$. We will say that $l_i = 0$ if request i is not

late. The goal in our new optimization problem will be to schedule all requests, using non-overlapping intervals, so as to minimize the *maximum lateness*, $L = \max_i l_i$. This problem arises naturally when scheduling jobs that need to use a single machine, and so we will refer to our requests as *jobs*.

The natural greedy algorithm for this problem is analogous to the greedy algorithm for the previous interval scheduling problem. We will consider the jobs in increasing order of their deadlines d_i, and schedule them in this order. This greedy rule for constructing schedules is often called *Earliest Deadline First*.

By renaming the jobs if necessary, we can assume that the jobs are labeled in the order of their deadlines, i.e., we have that

$$d_1 \leq \ldots \leq d_n.$$

We will simply schedule all jobs in this order. Again, let s be the start time for all jobs. Job 1 will start at time $s = s_1$ and end at time $f_1 = s_1 + t_1$; job 2 will start at time $s_2 = f_1$ and end at time $f_2 = s_2 + t_2$; and so forth. We will use f to denote the finishing time of the last scheduled job. We write this algorithm below.

```
Order the requests in order of their deadlines
Assume for simplicity of notation that d_1 ≤ ... ≤ d_n
Initially, f = s
Consider the requests i = 1, ..., n in this order
   Assign request i to the time interval from s_i = f to f_i = f + t_i
   Let f = f + t_i
End
Return the set of scheduled intervals [s_i, f_i] for i = 1, ..., n
```

As before, the greedy algorithm is quite natural here — we always work on the job with the closest deadline — but it is not clear that the resulting solution should be optimal. For example, if we were looking for things to worry about, we could observe that the decision rule it uses to order the jobs throws away half the data — the lengths of the jobs — and focuses only on their deadlines.

To reason about the optimality of the algorithm, we first observe that the schedule it produces has no "gaps" — times when the machine is not working, yet there are requests left. The time that passes during a gap will be called *idle time* — there is work to be done, yet for some reason the machine is sitting idle. Not only does the schedule A produced by our algorithm have no idle time; it is also very easy to show that there is an optimal schedule with this property. We do not write down a proof for this.

(3.6) *There is an optimal schedule with no idle time.*

Now, how can we prove that our schedule A is optimal, i.e., that its maximum lateness L is as small as possible? We will consider an optimal schedule $\mathcal{O}$. Our plan here is to gradually modify $\mathcal{O}$, preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule A found by the greedy algorithm. We refer to this type of analysis as an *exchange argument*.

We first try characterizing schedules in the following way. We say that a schedule A' has an *inversion* if a request i with deadline d_i is scheduled before another request j with earlier deadline $d_j < d_i$. Notice that, by definition, the schedule A produced by our algorithm has no inversions. If there are requests with identical deadlines, then there can be many different schedules with no inversions. However, we can show that all these schedules have the same maximum lateness.

(3.7) *All schedules with no inversions and no idle time have the same maximum lateness.*

Proof. If two different schedules have neither inversions nor idle time, then they differ only in the order in which jobs with identical deadlines are scheduled. Consider such a deadline d. In both schedules, the jobs with deadline d are scheduled consecutively (after all jobs with earlier deadlines, and before all jobs with later deadlines). Among the jobs with deadline d, the last one has the the greatest lateness, and this lateness does not depend on the order of the jobs. ■

The main step in showing the optimality of our algorithm is to establish that there is an optimal schedule that has no inversions and no idle time. To do this, we will start with any optimal schedule having no idle time; we will then convert it into a schedule with no inversions without increasing its maximum lateness. Thus, the resulting scheduling after this conversion will be optimal as well.

(3.8) *There is an optimal schedule that has no inversions and no idle time.*

Proof. By (3.6), there is an optimal schedule $\mathcal{O}$ with no idle time. The proof will consist of a sequence of statements. The first of these is simple to establish:

(a) *If $\mathcal{O}$ has an inversion, then there is a pair of requests i and j such that j is scheduled immediately after i and has $d_j < d_i$.*

Suppose $\mathcal{O}$ has at least one inversion, and let i and j be a pair of such adjacent inverted requests.

We will decrease the number of inversions in $\mathcal{O}$ by swapping the requests i and j in the schedule $\mathcal{O}$. The pair (i, j) formed an inversion in $\mathcal{O}$, this inversion is eliminated by the swap, and no new inversions are created. Thus we have

(b) *After swapping i and j we get a schedule with one fewer inversion.*

The hardest part of this proof is to argue that the inverted schedule is also optimal.

(c) *The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$.*

Proof of (c). We invent some notation to describe the schedule $\mathcal{O}$: assume that each request r is scheduled for the time interval $[s'_r, f'_r]$, and has lateness l'_r. Let $L' = \max_r l'_r$ denote the maximum lateness of this schedule. Let $\mathcal{O}''$ denote the swapped schedule; we will use s''_r, f''_r, l''_r, and L'' to denote the corresponding quantities in the swapped schedule. Now recall our two adjacent, inverted jobs i and j. All jobs other than jobs i and j finish at the same time in the two schedules. Using the notation just introduced, we have

$$s'_i \leq f'_i = s'_i + t_i = s'_j \leq f'_j = s'_j + t_j.$$

Job j will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job j.

The thing to worry about is clearly job i — its lateness has been increased, and what if this actually raises the maximum lateness of the whole schedule? Note that after the swap, job i will be finished at time f'_j, when job j was finished in the schedule $\mathcal{O}$. If job i is late, its lateness is $l''_i = f''_i - d_i = f'_j - d_i$. Our assumption $d_i > d_j$ implies that the lateness l''_i of request i in the new schedule is at most l'_j, the lateness of request j in the schedule $\mathcal{O}$. This shows that the swap does not increase the maximum lateness of the schedule. ■

We finish the proof of (3.8) by observing that the schedule $\mathcal{O}$ can have at most $\binom{n}{2}$ inversions (if all pairs are inverted), and hence after at most $\binom{n}{2}$ swaps we get an optimal schedule with no inversions. ■

The optimality of our greedy algorithm now follows immediately.

(3.9) *The schedule A produced by the greedy algorithm has optimal maximum lateness L.*

Proof. (3.8) proves that an optimal schedule with no inversions exists. Now by (3.7) all schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal. ■

Variations. There are many possible generalizations of this scheduling problem. For example, we assumed that all jobs are available to start at the common start time s. A natural, but harder, version of this problem would contain requests i that in addition to the deadline d_i and the requested time t_i would also have an earliest possible starting time r_i. This earliest possible starting time is usually referred to as the *release time*. Problems with release times

arises naturally in scheduling problems where requests can take the form "*Can I reserve the room for a 2 hour lecture, sometime between 1pm and 5pm?*" Our proof that the greedy algorithm finds an optimal solution relied crucially on the fact that all jobs are available at the common start time s. (Do you see where?) Unfortunately, as we will see later in the course, this more general version of the problem is much more difficult to solve optimally.

An Optimal Caching Strategy

We now consider a problem that involves processing a sequence of requests of a different form, and we develop an algorithm whose analysis requires a more subtle use of the exchange argument. The problem is that of *cache maintenance*.

To motivate caching, consider the following image. You're working on a long research paper, and your draconian library will only allow you to have eight books checked out at once. You know that you'll probably need more than this over the course of working on the paper, but any point in time, you'd like to have ready access to the eight books that are most relevant at that time. How should you decide which books to check out, and when should you return some in exchange for others, to minimize the number of times you have to exchange a book at the library?

This is precisely the problem that arises when dealing with a *memory hierarchy*: There is a small amount of data that can be accessed very quickly, and a large amount of data that requires more time to access; and you must decide which pieces of data to have close at hand.

Memory hierarchies have been a ubiquitous feature of computers since very early in their history. To begin with, data in the main memory of a processor can be accessed much more quickly than the data on its hard disk; but the disk has much more storage capacity. Thus, it is important to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible. The same phenomenon, qualitatively, occurs with on-chip caches in modern processors — these can be accessed in a few cycles, and so data can be retrieved from cache much more quickly than it can be retrieved from main memory. This is another level of hierarchy: small caches have faster access time than main memory, which in turn is smaller and faster to access than disk. And one can see extensions of this hierarchy in many other settings. When one uses a Web browser, the disk often acts as a cache for frequently visited Web pages — since going to disk is still much faster than downloading something over the Internet.

Caching is a general term for the process of storing a small amount of data in a fast memory, so as to reduce the amount of time spent interacting with a slow memory. In the examples above, the on-chip cache reduces the need to fetch data from main memory, the main memory acts as a cache for the disk, and the disk acts as a cache for the Internet. (And indeed, your desk acts as a cache for the campus library, and the assorted facts you're

able to remember without looking them up constitute a cache for the books on your desk.)

In order for caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a *cache maintenance* algorithm is responsible for determining what to keep in the cache, and what to evict from the cache when new data needs to be brought in.

Of course, as the caching problem arises in different settings, it involves various different considerations based on the underlying technology. For our purposes here, though, we take an abstract view of the problem that underlies most of these settings. We consider a set U of n pieces of data stored in *main memory*. We also have a faster memory, the *cache*, that can hold $k < n$ pieces of data at any one time. We will assume that the cache initially holds some k items. A sequence of data items $D = d_1, d_2, \ldots, d_m$ drawn from U is presented to us — this is the sequence of memory references we must process — and in processing them we must decide at all time which k items to keep in the cache. When item d_i is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and — if the cache is full — to *evict* some other piece of data that is currently in the cache, so as to make room for d_i. This is called a *cache miss*, and we want to have as few of these as possible.

Thus on a particular sequence of memory references, a cache maintenance algorithm determines an *eviction schedule* — specifying which items should be evicted from the cache at which points in the sequence — and this determines the contents of the cache and the number of misses over time. For example, suppose we have three items $\{a, b, c\}$, the cache size is $k = 2$, and we are presented with the sequence

$$a, b, c, b, c, a, b.$$

Suppose that the cache initially contains the items a and b. Then on the third item in the sequence, we could evict a so as to bring in c; and on the sixth item we could evict c so as to bring in a; we thereby incur two cache misses over the whole sequence. After thinking about it, one concludes that any eviction schedule for this sequence must include at least two cache misses.

Under real operating conditions, cache maintenance algorithms must process memory references $d_1, d_2, \ldots$ without knowledge of what's coming in the future; but for purposes of evaluating the quality of these algorithms, systems researchers very early on sought to understand the nature of the optimal solution to the caching problem. Given a full sequence S of memory references, what is the eviction schedule that incurs as few cache misses as possible?

In the 1960's, Les Belady showed that the following simple rule will always incur the minimum number of misses:

When d_i needs to be brought into the cache,

```
evict the item that is needed the farthest into the future.
```

We will call this the *Farthest-in-Future* algorithm. When it is time to evict something, we look at the next time that each item in the cache will be referenced, and choose the one for which this is as late as possible.

This is a very natural algorithm. At the same time, the fact that it is optimal on all sequences is somewhat more subtle than it first appears. Why evict the item that is needed farthest in the future, as opposed – for example – to the one that will be used least frequently in the future? Moreover, consider a sequence like

$$a, b, c, d, a, d, e, a, d, b, c$$

with $k = 3$ and items $\{a, b, c\}$ initially in the cache. The `Farthest-in-Future` rule will produce a schedule S that evicts c on the fourth step and b on the seventh step; a different eviction schedule S' evicts b on the fourth step and c on the seventh step, incurring the same number of misses. So in fact it's easy to find cases where schedules produced by rules other than `Farthest-in-Future` are just as good; and given this flexibility, why might a deviation from `Farthest-in-Future` early on not yield an actual savings farther along in the sequence? For example, on the seventh step in the above example, the schedule S' is actually evicting an item (c) that is needed *farther* into the future than the item evicted at this point by `Farthest-in-Future`, since `Farthest-in-Future` gave up c earlier on.

These are at least the kinds of things one should worry about before concluding that `Farthest-in-Future` really is optimal. In thinking about the example above, we quickly appreciate that it doesn't really matter whether b or c is evicted at the fourth step, since the other one should be evicted at the seventh step; so given a schedule where b is evicted first, we can swap the choices of b and c without changing the cost. This reasoning — swapping one decision for another — forms the first outline of an *exchange argument* that proves the optimality of `Farthest-in-Future`.

Before delving into this analysis, we clear up one important issue. All the cache maintenance algorithms that we've been considering so far produce schedules that only bring an item d into the cache in a step i if there is a request to d in step i, and d is not already in the cache. Let us call such a schedule *reduced* — it does the minimal amount of work necessary in a given step. But in general, one could imagine an algorithm that produced schedules that are not reduced, by bringing in items in steps when they are not requested. We now show that for every non-reduced schedule, there is an equally good reduced schedule.

Let S be a schedule that may not be reduced. We define a new schedule $\overline{S}$ — the *reduction* of S — as follows. In any step i where S brings in an item d that has not been requested, our constructing of $\overline{S}$ "pretends" to do this, but actually leaves d in main memory. It only actually brings d into the cache in the next step j after this when d is requested. In this way,

the cache miss incurred by $\overline{S}$ in step j can be charged to the earlier cache miss incurred by S in step i. Hence we have the following fact.

(3.10) *$\overline{S}$ is a reduced schedule that incurs at most as many misses as the schedule S.*

We now proceed with the exchange argument showing that `Farthest-in-Future` is optimal. Consider an arbitrary sequence D of memory references; let S_{FF} denote the schedule produced by `Farthest-in-Future`, and let S^* denote a schedule that incurs the minimum possible number of misses. We will now gradually "transform" the schedule S^* into the schedule S_{FF}, one eviction decision at a time, without increasing the number of misses.

Here is the basic fact we use to perform one step in the transformation.

(3.11) *Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through the first j items in the sequence, for a number j. Then there is a reduced schedule S' that makes the same eviction decisions as S_{FF} through the first $j+1$ items, and incurs no more misses than S does.*

Proof. Consider the $(j+1)^{\text{st}}$ request, to item $d = d_{j+1}$. Since S and S_{FF} have agreed up to this point, they have the same cache contents. So if d is in the cache for both, then no eviction decision is necessary (both schedules are reduced), and so S in fact agrees with S_{FF} through step $j+1$, and we can set $S' = S$. Similarly, if d needs to be brought into the cache, but S and S_{FF} both evict the same item to make room for d, then we can again set $S' = S$.

So the interesting case arises when d needs to be brought into the cache — and to do this S evicts item f while S_{FF} evicts item $e \neq f$. Here, S and S_{FF} do not already agree through step $j+1$ — S has e in cache while S_{FF} has f in cache. Hence we must actually do something non-trivial to construct S'.

As a first step, we should have S' evict e rather than f. Now we need to further ensure that S' incurs no more misses than S. An easy way to do this would be to have S' agree with S for the remainder of the sequence; but this is no longer possible, since S and S' have slightly different caches from this point onward. So instead, we'll have S' try to get its cache back to the same state as S as quickly as possible, while not incurring unnecessary misses — once the caches are the same, we can finish the construction of S' by just having it behave like S.

Specifically, from request $j+2$ onward, S' behaves exactly like S until one of the following things happens for the first time.

(i) There is a request to an item $g \neq e, f$ that is not in the cache of S, and S evicts e to make room for it. Since S' and S only differ on e and f, it must be that g is not in the cache of S' either; so we can have S' evict f, and now the caches of S and S' are the same. We can then have S' behave exactly like S for the rest of the sequence.

(ii) There is a request to f, and S evicts an item e'. If $e' = e$, then we're all set: S' can simply access f from the cache, and after this step the caches of S and S' will be the same. If $e' \neq e$, then we have S' evict e' as well, and bring in e from main memory; this too results in S and S' having the same caches. However, we must be careful here, since S' is no longer a reduced schedule — it brought in e when it wasn't immediately needed. So to finish this part of the construction, we further transform S' to its reduction $\overline{S'}$ using (3.10); this doesn't increase the number of misses of S', and it still agrees with S_{FF} through step $j+1$.

Hence in both these cases, we have a new reduced schedule S' that agrees with S_{FF} through the first $j+1$ items, and incurs no more misses than S does. And crucially — here is where we use the defining property of the `Farthest-in-Future` algorithm — one of these two cases will arise *before* there is a reference to e. This is because in step $j+1$, `Farthest-in-Future` evicted the item (e) that would be needed farthest in the future; so before there could be a request to e, there would have to be a request to f, and then case (ii) above would apply. ■

Using this result, it is easy to complete the proof of optimality. We begin with an optimal schedule S^*, and use (3.11) to construct a schedule S_1 that agrees with S_{FF} through the first step. We continue applying (3.11) inductively for $j = 1, 2, 3, \ldots, m$, producing schedules S_j that agree with S_{FF} through the first j steps. Each schedule incurs no more misses than the previous one; and by definition $S_m = S_{FF}$, since it agrees with it through the whole sequence. Thus we have

(3.12) *S_{FF} incurs no more misses than any other schedule S^*, and hence is optimal.*

Caching under Real Operating Conditions. As we discussed above, Belady's optimal algorithm provides a benchmark for caching performance; but in applications, one generally must make eviction decisions on the fly without knowledge of future requests. Experimentally, the best caching algorithms under this requirement seem to be variants of the *Least-Recently-Used* (*LRU*) principle, which proposes evicting the item from the cache that was referenced *longest ago*.

If one thinks about it, this is just Belady's algorithm with the direction of time reversed — longest in the past rather than farthest in the future. It is effective because applications generally exhibit *locality of reference* — a running program will generally keep accessing the things it has just been accessing. (It is easy to invent pathological exceptions to this principle, but these are relatively rare in practice.) Thus, one wants to keep the more recently referenced items in the cache.

Although we won't go into it here, it's worth mentioning that long after the adoption of *LRU* in practice, Sleator and Tarjan showed that one could actually provide some theoretical analysis of the performance of *LRU*, bounding the number of misses it incurs relative to `Farthest-in-Future`.

3.3 Shortest Paths in a Graph

Some of the basic algorithms for graphs are based on greedy design principles. Here we apply a greedy algorithm to the problem of finding shortest paths, and in the next section we look at the construction of minimum-cost spanning trees.

As we've seen, graphs are often used to model networks in which one travels from one point to another — traversing a sequence of highways through interchanges, or traversing a sequence of communication links through intermediate routers. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph. We may ask this as a point-to-point question: given nodes u and v, what is the shortest u-v path? Or we may ask for more information: given a *start node* s, what is the shortest path from s to each other node?

The concrete set-up of the shortest paths problem is as follows. We are given a directed graph $G = (V, E)$, with a designated *start node* s. We assume that s has a path to each other node in G. Each edge e has a length $\ell_e > 0$, indicating the time (or distance, or cost) it takes to traverse e. For a path P, the *length of* P — denoted $\ell(P)$ — is the sum of the lengths of all edges in P. Our goal is to determine the shortest path from s to each other node in the graph. We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph by simply replacing each undirected edge (u, v) of length ℓ by two directed edges (u, v) and (v, u), each of length ℓ.

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest paths problem. We begin by describing an algorithm that just determines the *length* of the shortest path from s to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance $d(u)$ from s; this is the "explored" part of the graph. Initially $S = \{s\}$, and $d(s) = 0$. Now, for each node $v \in V - S$, we determine the shortest path the can be constructed by traveling along a path through the explored part S to some $u \in S$, followed by the single edge (u, v). That is, we consider the quantity $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$. We choose the node $v \in V - S$ for which this quantity is minimized, add v to S, and define $d(v)$ to be the value $d'(v)$.

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes.
    For each u ∈ S, we store a distance d(u).
Initially S = {s} and d(s) = 0.
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible.
    Add v to S and define d(v) = d'(v).
EndWhile
```

It is simple to produce the s-u paths corresponding to these distances. As each node v is added to the set S, we simply record the edge (u, v) on which it achieved the value $\min_{e=(u,v):u\in S} d(u) + \ell_e$. The path P_v is implicitly represented by these edges: if (u, v) is the edge we have stored for v, then P_v is just (recursively) the path P_u followed by the single edge (u, v). In other words, to construct P_v, we simply start at v, follow the edge we have stored for v in the reverse direction to u; then follow the edge we have stored for u in the reverse direction to its predecessor; and so on until we reach s. Note that s must be reached, since our backwards walk from v visits nodes that were added to S earlier and earlier.

Now we must prove the correctness of Dijkstra's algorithm, by showing that these paths P_u really are shortest paths. Dijkstra's algorithm is greedy in the sense that we always form the shortest new s-v path we can make from a path in S followed by a single edge. And we prove its correctness using our first style of analysis, showing a concrete sense in which it "stays ahead" of any other solution.

In previous analysis based on the "greedy algorithm stays ahead" principle, the underlying problem always had a natural one-dimensional character — so it was clear what "staying ahead" meant. But what do we mean here? It turns out that the set S has a special property that is not obvious from the statement of the algorithm — nodes are added to it in increasing order of their distance from s. Thus, the first k nodes discovered by Dijkstra's algorithm are at least as close to s as the first k nodes discovered by any algorithm.

(3.13) *Consider the point in the algorithm at which $|S| = k$. Then S consists of the k closest nodes to s, and for each $u \in S$, the path P_u is a shortest s-u path.*

Note that this fact immediately establishes the correctness of Dijkstra's algorithm, by applying it when $|S| = n$, in which case S includes all nodes.

Proof of (3.13). We prove this by induction on k. The case $k = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds for some value of $k \geq 1$ and set S; we now grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our s-v path P_v.

By induction hypothesis, P_u is the shortest s-u path for each $u \in S$. Now, consider any other s-v path P; we wish to show that it is at least as long as P_v. See Figure 3.1. Let z be the first node on P that is not in S, let y be the node on P just before z, and let P' be the sub-path of P from s to y. Then $\ell(P') \geq \ell(P_y)$, and so $\ell(P') + \ell(y, z) \geq d'(z)$. But the full path length $\ell(P)$ is at least as large as $\ell(P') + \ell(y, z)$, and by our choice of v we have $d'(z) \geq d'(v)$. Thus, $\ell(P) \geq d'(v) = \ell(P_v)$, establishing that no other s-v path is shorter than P_v.

Moreover, the argument in the previous paragraph also establishes that for any node $z \notin S$, the length of the shortest s-z path is at least $d'(z)$, which is at least $d'(v)$. Thus v is as close to s as any other node in $V - S$. ■

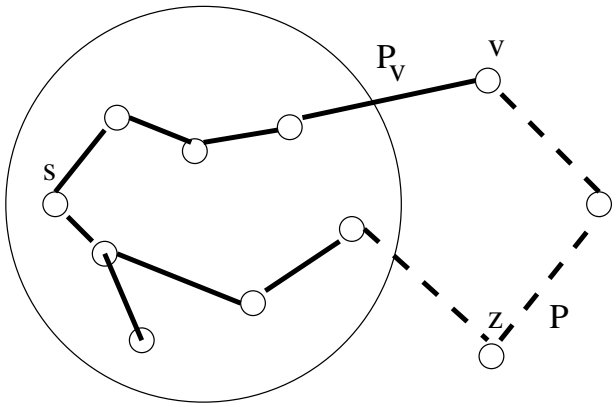


Figure 3.1: The shortest path P_v and an alternate path P in the graph.

Here are two things to notice about Dijkstra's algorithm and its analysis. First, the algorithm does not always find shortest paths if some of the edges can have negative lengths. (Do you see where the proof breaks?) Many shortest-path applications involve negative edge lengths, and a more complex algorithm — due to Bellman and Ford — is required for this case. We will see this algorithm when we consider the topic of dynamic programming.

The second observation is that Dijkstra's algorithm is, in a sense, even simpler than we've described here. Recall how breadth-first search discovered the nodes of G one "layer" at a time, from a start node s. Dijkstra's algorithm is really a "continuous" version of breadth-first search, motivated by the following picture. Suppose the edges of G formed a system of pipes filled with water, joined together at the nodes; each edge e has length ℓ_e and a fixed cross-sectional area. Now suppose an extra droplet of water falls at node s, and starts a wave from s. As the wave expands out of node s at a constant speed, the expanding sphere of wavefront reaches nodes in increasing order of their distance from s. It is easy to believe (and also true) that the path taken by the wavefront to get to any node v is a shortest path. Indeed, it is easy to see that this is exactly the path to v found by Dijkstra's algorithm, and that the nodes are discovered by the expanding water in the same order that they are discovered by Dijkstra's algorithm.

Implementation. To conclude our discussion of Dijkstra's algorithm, we consider its running time. There are $n-1$ iterations of the `While` loop for a graph with n nodes, as each iteration adds a new node v to S. Selecting the correct node v efficiently is a more subtle issue. One's first impression is that each iteration would have to consider each node $v \notin S$, and go through all the edges between S and v to determine the minimum $\min_{e=(u,v):u\in S} d(u)+\ell_e$, so that we can select the node v for which this minimum is smallest. For a graph with m edges, computing all these minima can take $O(m)$ time, so this would lead to an implementation that runs in $O(mn)$ time.

We can do considerably better if we use the right data structures. First, we will explicitly maintain the values of the minima $d'(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$ for each node $v \in V - S$, rather than recomputing them in each iteration. We can further improve the efficiency by keeping the nodes $V - S$ in a *priority queue* with $d'(v)$ as their keys. A priority queues is a data structure that should be familiar from earlier courses. It is designed to maintain a set of n elements, each with a key; it can efficiently insert elements, delete elements, change an element's key, and extract the element with minimum key. We will need the third and fourth of the above operations: `ChangeKey` and `ExtractMin`.

How do we implement Dijkstra's algorithm using a priority queue? To select the node v that should be added to the set S, we need the `ExtractMin` operation. To see how to update the keys, consider an iteration in which node v is added to S, and let $w \notin S$ be a node that remains in the priority queue. What do we have to do to update the value of $d'(w)$? If (v, w) is not an edge, then we don't have to do anything: the set of edges considered in the minimum $\min_{e=(u,w):u\in S} d(u) + \ell_e$ is exactly the same before and after adding v to S. If $e' = (v, w) \in E$, on the other hand, then the new value for the key is $\min(d'(w), d(v) + \ell_{e'})$. If $d'(w) > d(v) + \ell_{e'}$ then we need to use the `ChangeKey` operation to decrease the key of node w appropriately. This `ChangeKey` operation can occur at most once per edge, when the tail of the edge e' is added to S. In summary, we have the following result.

(3.14) *Using a priority queue, Dijkstra's algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin`*, and m* `ChangeKey` *operations.*

Using a simple heap-based priority queue as discussed in previous courses, each priority queue operation can be made to run in $O(\log n)$ time. Thus the overall time for the implementation is $O(m \log n)$.

3.4 The Minimum Spanning Tree Problem

We now apply an exchange argument in the context of a second fundamental problem on graphs — the minimum spanning tree problem. Suppose we have a set of locations $V = \{v_1, v_2, \ldots, v_n\}$, and we want to build a communication network on top of them. The network should be connected — there should be a path between every pair of nodes — but subject to this requirement, we wish to build it as cheaply as possible.

For certain pairs (v_i, v_j), we may build a direct link between v_i and v_j for a certain cost $c(v_i, v_j) > 0$. Thus, we can represent the set of possible links that may be built using a graph $G = (V, E)$, with a positive *cost* c_e associated with each edge $e = (v_i, v_j)$. The problem is to find a subset of the edges $T \subseteq E$ so that the graph (V, T) is connected, and the total cost

$\sum_{e \in T} c_e$ is as small as possible. (We will assume that the full graph G is connected; otherwise, no solution is possible.)

Here is a basic thing to notice.

(3.15) *Let T be a minimum-cost solution to the network design problem defined above. Then (V, T) is a tree.*

Proof. By definition, (V, T) must be connected; we show that it also will contain no cycles. Indeed, suppose it contained a cycle C, and let e be any edge on C. We claim that $(V, T-\{e\})$ is still connected; for any path that previously used the edge e can now go "the long way" around the remainder of the cycle C instead. It follows that $(V, T - \{e\})$ is also a valid solution to the problem, and it is cheaper — a contradiction. ■

We will call a subset $T \subseteq E$ a *spanning tree* of G if (V, T) is a tree. In view of (3.15), our network design problem is generally called the *minimum spanning tree problem.*

Unless G is a very simple graph, it will have exponentially many different spanning trees, whose structures may look very different from one another. It is not at all clear how to find the cheapest among these efficiently. In this section, we will discuss two simple greedy algorithms for the problem. Both algorithms are very natural, and the hard part in each case is to actually prove that they produce a minimum spanning tree for every graph.

In describing both algorithms, we will make the simplifying assumption that all edge costs are distinct from one another (i.e. no two are equal). This assumption makes make things cleaner to think about, while preserving the all main issues in the problem.

Prim's Algorithm

The first approach we discuss, Prim's algorithm, is a natural adaptation of Dijkstra's algorithm to the present setting. Rather than seeking short paths from a single source, we now want to link up all the nodes of an undirected graph; this new goal is in a sense more "symmetric." At all times, we maintain a *growing set* $S \subseteq V$, together with a spanning tree T on S. (S will start out as a single arbitrary node.) We grow S one node at a time; but rather than looking at the distances of nodes from a particular source in S, we use an even simpler greedy rule: we just choose the node v that costs the least to add in this step. Given a node v, what's the least we have to pay to attach v to S? We need to join it to some node in S by an edge, so the cost is $\min_{e=(u,v):u \in S} c_e$. This "attachment cost" is the crucial quantity in Prim's algorithm.

```
Prim's Algorithm (G, c)
Initially S = {s} and T = ∅
While S ≠ V
```

```
    Select a node v ∉ S which has an edge into S and
        for which the attachment cost min_{e=(u,v):u∈S} c_e is as small as possible
    Add v to S
    Add the edge e where the minimum is obtained to T
  EndWhile
  Return the spanning tree T
```

It is fairly easy to see that the set of edges T returned by the algorithm is indeed a spanning tree of G.

(3.16) *If the graph G is connected, then the set of edges T returned by the algorithm is a spanning tree of G.*

Proof. First we make sure that the algorithm is well defined: could we reach a point where $S \neq V$, but there is no $v \notin S$ with an edge into S? In this case, the algorithm could never finish the `While` loop. But this cannot happen, for in a connected graph G every subset of nodes $S \subseteq V$ has an edge leaving the set; i.e., an edge (v, w) with $v \notin S$ and $w \in S$.

To show that the set of edges T forms a spanning tree, we prove the following fact by induction on the number of steps of the algorithm: in each iteration, the graph (S, T) is a tree. (In other words, T "spans" S at all times.) This is clearly true when $S = \{s\}$. And if it is true for some set S, then it remains true after we add a node v to S, and an edge (v, w) ($w \in S$) to T — the resulting graph is still connected, and no cycle has been created. ■

As has been the case with previous greedy algorithms, making up the algorithm was not difficult. What is not at all obvious is that Prim's algorithm is computing a spanning tree of minimum cost. With Dijkstra's algorithm, we ultimately came up with an intuitive reason why it was doing the right thing — it was essentially visiting nodes by a uniformly expanding "wavefront." But Prim's algorithm seems harder to believe in — why shouldn't it be the case that adding a more expensive edge initially would make it possible to add many cheap edges later on, so that suppressing our greed in the short term might pay off in the long run? And here is another thing to worry about: We know that we're trying to minimize a quantity (the spanning tree cost) that is a sum of many terms; yet nowhere does Prim's algorithm ever add two numbers together.

We now prove that the algorithm does find a minimum spanning tree. This result holds even without the assumption that all edge costs are distinct; but adopting this assumption will make the proof somewhat cleaner, and it will yield an additional consequence: the minimum spanning tree of G is unique.

The key to our analysis is the following fact. Its proof contains the fundamental exchange argument we need for reasoning about the algorithm — given any spanning tree, we will transform it into one that is no more expensive, and "closer" to the one found by Prim's algorithm.

(3.17) *Assume that all edge costs are distinct. For any subset $\emptyset \neq S \subsetneq V$, let edge e be the minimum-cost edge with one end in S and the other in $V - S$. Then every minimum-cost spanning tree contains the edge e.*

Proof. Let T be a minimum-cost spanning tree that does not contain e. We need to show that T does not have the minimum possible cost. Write $e = (v, w)$. T is a spanning tree, so there must is a path P in T from v to to w. Starting at v, suppose we follow the nodes of P in sequence; there is a first node w' on P that is in $V - S$. See Figure 3.2. Let $v' \in S$ be the node just before w' on P, and let $e' = (v', w')$ be the edge joining them. Thus, e' is an edge of T with one end in S and the other in $V - S$.

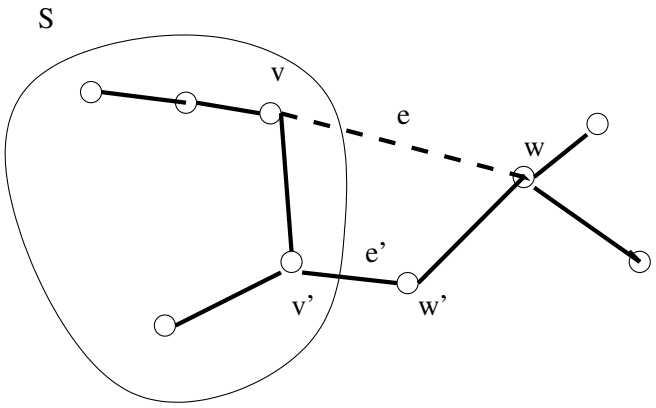


Figure 3.2: A spanning tree T that does not contain edge e.

Let's consider the set of edges $T' = T - \{e'\} \cup \{e\}$. First we claim that T' is a spanning tree. Clearly it has $n-1$ edges. Also, (V, T') is connected, since (V, T) is connected, and any path in (V, T) that used the edge $e' = (v', w')$ can now be "re-routed" in (V, T') to follow the portion of P from v' to v, then the edge e, and then the portion of P from w to w'.

We noted above that the edge e' has one end in S and the other in $V - S$. But e is the cheapest edge with this property, and so $c_e < c_{e'}$. (The inequality is strict since no two edges have the same cost.) Thus the total cost of T' is less than that of T, which proves that T is not a minimum-cost spanning tree. ■

Now, to show that Prim's algorithm produces a minimum spanning tree, we observe that every time we add an edge e to the tree, e is the minimum-cost edge leaving the set S, and so we can apply (3.17).

(3.18) *If all edge costs are distinct, then the spanning tree T returned by Prim's algorithm is the unique minimum-cost spanning tree in G.*

Proof. Assume by way of contradiction that $T^* \neq T$ is a minimum spanning tree. Consider the first iteration in which Prim's algorithm adds an edge $e \notin T^*$. Let S be the set of nodes

already connected at the start of this iteration; and let $e = (v, w)$ with $v \notin S$ and $w \in S$. We claim that e is the minimum-cost edge with one end in S and the other end in $V - S$. For if there were a cheaper edge $e' = (v', w')$ with $v' \notin S$ and $w' \in S$, then it would have been considered in this iteration; and the algorithm would have selected such a node v' and edge e', rather than v and e. Thus, (3.17) implies that e is in every minimum spanning tree, which contradicts our assumption that T^* was minimum.

Hence, no tree other than T can be a minimum spanning tree, which proves the claim. ■

As we noted above, one can prove by very similar means that Prim's algorithm finds a minimum spanning tree even when the edge costs are not all distinct; but in this case, it is not necessarily the unique minimum spanning tree.

Implementation. The proof of correctness of Prim's algorithm was quite different from the proof of Dijkstra's; but their implementations are almost identical. By analogy with Dijkstra's algorithm, we need to be able to decide which node v to add next to S, by maintaining the attachment costs $a(v) = \min_{e=(u,v):u \in S} c_e$ for each node $v \in V - S$. As before, we keep the nodes in a priority queue with $a(v)$ as the keys; we select a node with an `ExtractMin` operation, and update the attachment costs using `ChangeKey` operations. There are $n - 1$ iterations in which we perform `ExtractMin`, and we perform `ChangeKey` at most once for each edge. Thus we have

(3.19) *Using a priority queue, Prim's algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin`*, and m* `ChangeKey` *operations.*

Using a heap-based priority queue, we can implement both `ExtractMin` and `ChangeKey` in $O(\log n)$ time, and so get an overall running time of $O(m \log n)$. Alternatively, we can use an array-based priority queue, taking $O(n)$ time on each `ExtractMin`, but only $O(1)$ for `ChangeKey` , and so get an overall running time of $O(n^2)$, which is better for dense graphs.

Kruskal's Algorithm

We derived the optimality of Prim's algorithm from the very general statement of (3.17) . Looking more closely at the proof of its optimality, we can ask what was really important in the analysis. Could we construct a minimum spanning tree by greedily considering edges in any order at all? Clearly this would be too much to expect. What (3.17) proves is that we can safely add an edge to our solution as long as it is the minimum-cost edge leaving some set S.

There is another basic greedy algorithm for the minimum spanning tree problem that follows this principle, and in a sense it is even simpler than Prim's algorithm. This is

Kruskal's algorithm, and it behaves as follows. It initially sorts all the edges in order of increasing weight. It then considers them in a single pass; when it comes to the edge e, it includes it in the tree if and only if it does not form a cycle when added to the set of edges already chosen.

```
Kruskal's Algorithm (G, c)
Sort the edges in order of increasing cost.
Initially T = ∅
For each edge e = (v, w) in the sorted order
   If there is currently no path from v to w in (V, T) then
      (Adding e won't create a cycle)
      Add e to T.
   EndIf
EndFor
Return the set of edges T
```

Note the basic difference between Prim and Kruskal. In Prim's algorithm we grow a single connected component S, and as the algorithm proceeds we add more and more nodes to this one component. In Kruskal's algorithm there are many components growing at the same time, and edges are added to connect up these separate components.

(3.20) *If all edge costs are distinct, then the set of edges T returned by Kruskal's algorithm is the unique minimum-cost spanning tree in G.*

Proof. First we argue that (V, T) is a tree. By the definition of the algorithm, it has no cycles. Suppose it is not connected; then there are two nodes v and w with no path between them. Let S be the set of all nodes to which v has a path in (V, T). Since G is connected, and $\phi \neq S \subsetneq V$, we know there is at least one edge with exactly one end in S; let $\hat{e}$ be such an edge. Note that $\hat{e} \notin T$. But then when Kruskal's algorithm considered $\hat{e}$, it could have added it without forming a cycle, a contradiction. Thus (V, T) is a tree.

The proof that T is a minimum spanning tree is almost the same as our proof of the optimality of Prim's algorithm: each edge added by Kruskal's algorithm is the cheapest edge leaving some set S, and so it is guaranteed to belong to every minimum spanning tree, by (3.17). More concretely, suppose that $T^* \neq T$ is a minimum spanning tree, and consider the first edge $e = (v, w) \notin T^*$ that Kruskal's algorithm adds to T. At the point just before e is added, let S be the set of all nodes to which v has a path in (V, T). e is the cheapest edge with exactly one end in S, and so by (3.17), every minimum spanning tree contains e. This contradicts our assumption that T^* was minimum. ■

One can prove similarly that Kruskal's algorithm also produces a minimum spanning tree when the edge costs are not all distinct.

What do we need in order to implement Kruskal's algorithm? The first step, sorting the edges, takes time $O(m \log m)$. For the remainder of the algorithm, we need to maintain the connected components of (V, T) as they change under the insertion of new edges. It is possible to design a data structure for maintaining the connected components that allows us to test whether or not to add each edge e in $O(\log m)$ time; consequently, we can implement Kruskal's algorithm in $O(m \log m)$ time. We will not discuss the details of this implementation here.

A Clustering Perspective

We motivated the construction of minimum spanning trees through the problem of finding a low-cost network connecting a set of sites. But minimum spanning trees arise in a range of different settings, several of which appear on the surface to be quite different from one another. An appealing example is the role that minimum spanning trees play in the area of *clustering*.

Clustering arises whenever one has a collection of objects — say a set of photographs, or documents, or micro-organisms — that one is trying to classify, or organize into coherent groups. Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is. One common approach is to define a *distance function* on the objects, with the interpretation that objects at larger distance from one another are less similar to each other. For points in the physical world, distance may actually be related to their physical distance; but in many applications, distance takes on a much more abstract meaning. For example, we could define the distance between two species to be the number of years since they diverged in the course of evolution; we could define the distance between two images in a video stream as the number of corresponding pixels at which their intensity values differ by at least some threshold.

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are "close," and objects in different groups are "far apart." Starting from this vague set of goals, the field of clustering branches into a vast number of technically different approaches, each seeking to formalize this general notion of what a good set of groups might look like.

Clusterings of Maximum Spacing. Minimum spanning trees play a role in one of the most basic formalizations, which we describe here. Suppose we are given a set U of n objects, labeled $p_1, p_2, \ldots, p_n$. For each pair, p_i and p_j, we have a numerical distance $d(p_i, p_j)$. We require only that $d(p_i, p_i) = 0$; that $d(p_i, p_j) > 0$ for distinct p_i and p_j; and that distances are symmetric: $d(p_i, p_j) = d(p_j, p_i)$.

Suppose we are seeking to divide the objects in U into k groups, for a given parameter k. We say that a *k-clustering* of U is a partition of U into k non-empty sets $C_1, C_2, \ldots, C_k$. We

define the *spacing* of a k-clustering to be the minimum distance between any pair of points lying in different clusters. Given that we want points in different clusters to be far apart from one another, a natural goal is to seek the k-clustering with maximum possible spacing.

The question now becomes the following. There are exponentially many different k-clusterings of a set U; how can we efficiently find the one that has maximum spacing?

To do this, we consider growing a graph on the vertex set U. The connected components will be the clusters, and we will try to bring nearby points together into the same cluster as rapidly as possible. (This way, they don't end up as points in different clusters that are very close together.) Thus, we start by drawing an edge between the closest pair of points. We then draw an edge between the next closest pair of points. We continue adding edges between pairs of points, in order of increasing distance $d(p_i, p_j)$. In this way, we are growing a graph H on U edge-by-edge, with connected components corresponding to clusters. Notice that we are only interested in the connected components of the graph H, not the full set of edges; so if we are about to add the edge (p_i, p_j) and find that p_i and p_j already belong to the same cluster, we will refrain from adding the edge — it's not necessary, since it won't change the set of components. In this way, our graph-growing process will never create a cycle; so H will actually be a union of trees. Each time we add an edge that spans two distinct components, it is as though we have merged the two corresponding clusters. In the clustering literature, the iterative merging of clusters in this way is often termed *single-link clustering*, a special case of *hierarchical agglomerative clustering*. ("Agglomerative" here means that we combine clusters; "single-link" means that we do so as soon as a single link joins them together.)

What is the connection to minimum spanning trees? It's very simple: although our graph-growing procedure was motivated by this cluster-merging idea, *our procedure is precisely Kruskal's minimum spanning tree algorithm.* We are doing exactly what Kruskal's algorithm would do if given a graph G on U in which there was an edge of cost $d(p_i, p_j)$ between each pair of nodes (p_i, p_j). The only different is that we seek a k-clustering, so we stop the procedure once we obtain k connected components.

In other words, we are running Kruskal's algorithm but stopping it just before it adds its last $k-1$ edges. This is equivalent to taking the full minimum spanning tree T (as Kruskal's algorithm would have produced it), deleting the $k-1$ most expensive edges (the ones that we never actually added), and defining the k-clustering to be the resulting connected components $C_1, C_2, \ldots, C_k$. Thus, iteratively merging clusters is equivalent to computing a minimum spanning tree and deleting the most expensive edges.

So two superficially different approaches yield the same set of clusters $C_1, C_2, \ldots, C_k$. Have we achieved our goal of producing clusters that are as spaced apart as possible? The following claim shows that we have.

(3.21) *The components $C_1, C_2, \ldots, C_k$ formed by deleting the $k-1$ most expensive edges*

of the minimum spanning tree T constitute a k-clustering of maximum spacing.

Proof. Let $\mathcal{C}$ denote the clustering $C_1, C_2, \ldots, C_k$. The spacing of $\mathcal{C}$ is precisely the length d^* of the $(k-1)^{\text{st}}$ most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's algorithm would have added next, at the moment we stopped it.

Now, consider some other k-clustering $\mathcal{C}'$, which partitions U into non-empty sets $C'_1, C'_2, \ldots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.

Since the two clusterings $\mathcal{C}$ and $\mathcal{C}'$ are not the same, it must be that one of our clusters C_r is not a subset of any of the k sets C'_s in $\mathcal{C}'$. Hence there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$; say $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.

Since p_i and p_j belong to the same component C_r, it must be that Kruskal's algorithm added all the edges of a p_i-p_j path P before we stopped it. In particular, this means that each edge on P has length at most d^*. Now, we know that $p_i \in C'_s$ but $p_j \notin C'_s$; so let p' be the first node on P that does not belong to C'_s, and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \leq d^*$, since the edge (p, p') was added by Kruskal's algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \leq d^*$. This completes the proof. ■

3.5 Minimum-Cost Arborescences: A Multi-Phase Greedy Algorithm

As we've seen more and more examples of greedy algorithms, we've come to appreciate that there can be considerable diversity in the way they operate. Many greedy algorithms make some sort of an initial "ordering" decision on the input, and then process everything in a one-pass fashion. Others make more incremental decisions — still local and opportunistic, but without a global "plan" in advance. In this lecture, we consider a problem that stresses our intuitive view of greedy algorithms still further. The problem is to compute a minimum-cost *arborescence* of a directed graph. This is essentially an analogue of the the minimum spanning tree problem for directed, rather than undirected, graphs; we will see that the move to directed graphs introduces significant new complications. At the same time, the style of the algorithm has a strongly "greedy" flavor, since it still constructs a solution according to a local, myopic rule.

We begin with the basic definitions. Let $G = (V, E)$ be a directed graph in which we've distinguished one node $r \in V$ as a *root*. An *arborescence* (with respect to r) is essentially a directed tree rooted at r — specifically, it is a subgraph $T = (V, F)$ such that T is a spanning tree of G if we ignore the direction of edges; and there is a path in T from r to each other node $v \in V$ if we take the direction of edges into account. Figure 3.3 gives an example of two different arborescences in the same directed graph.

There is a useful equivalent way to characterize arborescences, and this is as follows.

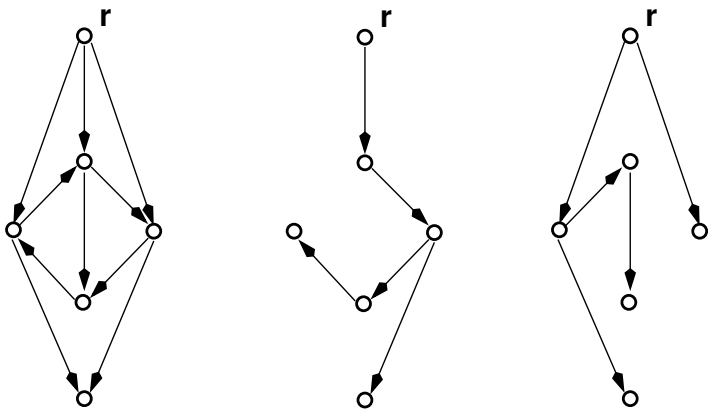


Figure 3.3: Two arborescences in the same underlying graph.

(3.22) *A subgraph $T = (V, F)$ of G is an arborescence with respect to root r if and only if T has no cycles, and for each node $v \neq r$, there is exactly one edge in F that enters v.*

Proof. If T is an arborescence with root r, then indeed each other node v has exactly one entering edge: this is simply the last edge on the unique r-v path.

Conversely, suppose T has no cycles, and each node $v \neq r$ has exactly one entering edge. Then in order to establish that T is an arborescence, we need only show that there is a directed path from r to each other node v. Here is how to construct such a path. We start at v, and repeatedly follow edges in the backward direction. Since T has no cycles, we can never return to a node we've previously visited, and thus this process must terminate. But r is the only node without incoming edges, and so the process must in fact terminate by reaching r; the sequence of nodes thus visited yields a path (in the reverse direction) from r to v. ■

Just as every connected graph has a spanning tree, it is easy to show that a directed graph has an arborescence rooted at r provided that r can reach every node. Indeed, in this case, the edges traversed in a breadth-first search beginning at r will form an arborescence.

(3.23) *A directed graph G has an arborescence rooted at r if and only if there is a directed path from r to each other node.*

Now, here is the central problem for this lecture: we are given a directed graph $G = (V, E)$, with a distinguished root node r and with a non-negative cost $c_e \geq 0$ on each edge, and we wish to compute an arborescence rooted at r of minimum total cost. (We will refer to this as the *optimal* arborescence.) We will assume throughout that G at least has an arborescence rooted at r; by (3.23), this can be easily checked at the outset.

Given the relationship between arborescences and trees, the minimum-cost arborescence problem certainly has a strong initial resemblance to the minimum spanning tree problem for

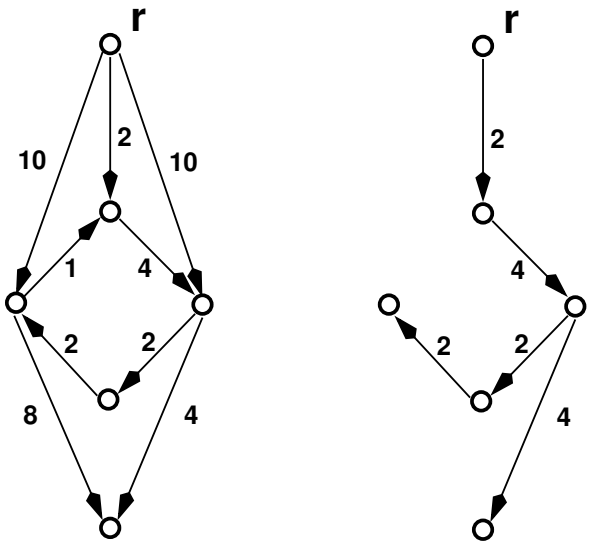


Figure 3.4: A directed graph with edge costs, and the optimal arborescence rooted at r.

undirected graphs. Thus, it's natural to start by asking whether the ideas we developed for that problem can be carried over directly to this setting. For example, must the minimum-cost arborescence contain the cheapest edge in the whole graph? Can we safely delete the most expensive edge on a cycle, confident that it cannot be in the optimal arborescence?

Clearly the cheapest edge e in G will not belong to the optimal arborescence if e enters the root; for the arborescence we're seeking is not supposed to have any edges entering the root. But even if the cheapest edge in G belongs to *some* arborescence rooted at r, it need not belong to the optimal one, as the example of Figure 3.4 shows. Indeed, including the edge of cost 1 in Figure 3.4 would prevent us from including the edge of cost 2 out of the root r (since there can only be one entering edge per node); and this in turn would force us to incur an unacceptable cost of 10 when we included one of the other edges out of r. This kind of argument never clouded our thinking in the minimum spanning tree problem, where it was always safe to plunge ahead and include the cheapest edge; it suggests that finding the optimal arborescence may be a significantly more complicated task. (It's worth noticing that the optimal arborescence in Figure 3.4 also includes the most expensive edge on a cycle; with a different construction, one can even cause the optimal arborescence to include the most expensive edge in the whole graph.)

Despite this, it is possible to design a greedy type of algorithm for this problem; it's just that our myopic rule for choosing edges has to be a little more sophisticated. First, let's consider a little more carefully what goes wrong with the general strategy of including the cheapest edges. Here's a particular version of this strategy: for each node $v \neq r$, select the cheapest edge entering v (breaking ties arbitrarily), and let F^* be this set of $n-1$ edges. Now consider the subgraph (V, F^*). Since we know that the optimal arborescence needs to

have exactly one edge entering each node $v \neq r$, and (V, F^*) represents the cheapest possible way of making these choices, we have the following fact.

(3.24) *If (V, F^*) is an arborescence, then it is a minimum-cost arborescence.*

So the difficulty is that (V, F^*) may not be an arborescence. In this case, (3.22) implies that (V, F^*) must contain a cycle C, which does not include the root. We now must decide how to proceed in this situation.

To make matters somewhat clearer, we begin with the following observation. Every arborescence contains exactly one edge entering each node $v \neq r$; so if we pick some node v and subtract a uniform quantity from the cost of every edge entering v, then the total cost of every arborescence changes by exactly the same amount. This means, essentially, that the actual cost of the cheapest edge entering v is not important; what matters is the cost of all other edges entering v *relative* to this. Thus, let y_v denote the minimum cost of any edge entering v. For each edge $e = (u, v)$, with cost $c_e \geq 0$, we define its *modified cost* c'_e to be $c_e - y_v$. Note that since $c_e \geq y_v$, all the modified costs are still non-negative. More crucially, our discussion motivates the following fact.

(3.25) *T is an optimal arborescence in G subject to costs $\{c_e\}$ if and only if it is an optimal arborescence subject to costs $\{c'_e\}$.*

Proof. Consider an arbitrary arborescence T. The difference between its cost with costs $\{c_e\}$ and $\{c'_e\}$ is exactly $\sum_{v \neq r} y_v$; i.e.

$$\sum_{e \in T} c_e - \sum_{e \in T} c'_e = \sum_{v \neq r} y_v.$$

This is because an arborescence has exactly one edge entering each node v in the sum. Since the difference between the two costs is independent of the choice of the arborescence T, we see that T has minimum cost subject to $\{c_e\}$ if and only if it has minimum cost subject to $\{c'_e\}$. ■

We now consider the problem in terms of the costs $\{c'_e\}$. All the edges in our set F^* have cost 0 under these modified costs; and so if (V, F^*) contains a cycle C, we know that all edges in C have cost 0. This suggests that we can afford to use as many edges from C as we want (consistent with producing an arborescence), since including edges from C doesn't raise the cost.

Thus, our algorithm continues as follows. We contract C into a single *super-node*, obtaining a smaller graph $G' = (V', E')$. Here, V' contains the nodes of $V - C$, plus a single node c^* representing C. We transform each edge $e \in E$ to an edge $e' \in E'$ by replacing each end of e that belongs to C with the new node c^*. This can result in G' having parallel edges

(i.e. edges with the same ends), which is fine; however, we delete self-loops from E' — edges that have both ends equal to c^*. We recursively find an optimal arborescence in this smaller graph G', subject to the costs $\{c'_e\}$. The arborescence returned by this recursive call can be converted into an arborescence of G by including all but one edge on the cycle C.

In summary, here is the full algorithm.

```
For each node v ≠ r
   Let y_v be the minimum cost of an edge entering node v.
   Modify the costs of all edges e entering v to c'_e = c_e − y_v.
Choose one 0-cost edge entering each v ≠ r, obtaining a set F*.
If F* forms an arborescence, then return it.
Else there is a directed cycle C ⊆ F*
   Contract C to a single super-node, yielding a graph G' = (V', E').
   Recursively find an optimal arborescence (V', F') in G'
              with costs {c'_e}.
   Extend (V', F') to an arborescence (V, F) in G
              by adding all but one edge of C.
```

It is easy to implement this algorithm so that it runs in polynomial time. But does it lead to an optimal arborescence? Before concluding that it does, we need to worry about the following point: not every arborescence in G corresponds to an arborescence in the contracted graph G'; could we perhaps "miss" the true optimal arborescence in G by focusing on G'? What is true is the following: the arborescences of G' are in one-to-one correspondence with arborescences of G *that have exactly one edge entering the cycle* C; and these corresponding arborescences have the same cost with respect to $\{c'_e\}$, since C consists of 0-cost edges. (We say that an edge $e = (u, v)$ *enters* C if $v \in C$; it does not matter in this definition whether or not u also belongs to C.) So to prove that our algorithm finds an optimal arborescence in G, we must prove that G has an optimal arborescence with exactly one edge entering C. We do this now.

(3.26) *Let C be a cycle in G consisting of edges of cost 0, such that $r \notin C$. Then there is an optimal arborescence rooted at r that has exactly one edge entering C.*

Proof. Consider any arborescence T in G. Since r has a path in T to every node, there is at least one edge of T that enters C. If T enters C exactly once, then we are done. Otherwise, suppose that T enters C more than once; we show how to modify it to obtain an arborescence of no greater cost that enters C exactly once.

Let $e = (a, b)$ be an edge entering C that lies on as short a path from r as possible; this means in particular that no edges on the path from r to e can enter C. We delete all edges of T that enter C, except for the edge e. We add in all edges of C except for the one edge that enters b, the head of edge e. Let T' denote the resulting subgraph of G.

We claim that T' is also an arborescence. This will establish the result, since the cost of T' is clearly no greater than that of T: the only edges of T' that do not also belong to T have cost 0. So why is T' an arborescence? First, observe that T' has exactly one edge entering each node $v \neq r$, and no edge entering r. So T' has exactly $n-1$ edges, and hence if we can show there is an r-v path in T' for each v, then T' must be connected in an undirected sense, and hence a tree. Thus it would satisfy our initial definition of an arborescence.

So consider any node $v \neq r$; we must show there is an r-v path in T'. If $v \in C$, we can use the fact that the path in T from r to e has been preserved in the construction of T'; thus, we can reach v by first reaching e, and then following the edges of the cycle C. Now suppose that $v \notin C$, and let P denote the r-v path in T. If P did not touch C, then it still exists in T'. Otherwise, let w be the last node in $P \cap C$, and let P' be the sub-path of P from w to v. Observe that all the edges in P' still exist in T'. We have already argued that w is reachable from r in T', since it belongs to C; concatenating this path to w with the sub-path P' gives us a path to v as well. ■

We can now put all the pieces together to argue that our algorithm is correct.

(3.27) *The algorithm finds an optimal arborescence rooted at r in G.*

Proof. The proof is by induction on the number of nodes in G. If the edges of F form an arborescence, then the algorithm returns an optimal arborescence by (3.24). Otherwise, we consider the problem with the modified costs $\{c'_e\}$, which is equivalent by (3.25). After contracting a 0-cost cycle C to obtain a smaller graph G', the algorithm produces an optimal arborescence in G' by the inductive hypothesis. Finally, by (3.26), there is an optimal arborescence in G that corresponds to the optimal arborescence computed for G'. ■

3.6 Exercises

1. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive to the New York station one-by-one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive — otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and want your opinion on whether the situation can be improved. Here is how they are thinking: maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allowing the next few trucks to be better packed.

Prove that the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling problem — it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

2. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges — what's being bought — are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them — e.g. they may want to know if the four events

   `buy Yahoo, buy eBay, buy Yahoo, buy Oracle`

   occur in this sequence S, in order but not necessarily consecutively.

   They begin with a finite collection of possible *events* (e.g. the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g. `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So for example, the sequence of four events above is a subsequence of the sequence

   `buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle`

   Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time $O(m + n)$ whether S' is a subsequence of S.

3. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within 4 miles of one of the base stations.

   Give an efficient algorithm that achieves this goal, using as few base stations as possible.

4. Consider the following variation on the *Interval Scheduling Problem* from lecture. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

   Given a list of n such jobs, your goal is to accept *as many jobs as possible* (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n, the number of jobs. You may assume for simplicity that no two jobs have the same start or end times.

   Example: Consider the following four jobs, specified by *(start-time, end-time)* pairs.

   (6 pm, 6 am), (9 pm, 4 am), (3 am, 2 pm), (1 pm, 7 pm).

   The unique solution would be to pick the two jobs (9 pm, 4 am) and (1 pm, 7 pm), which can be scheduled without overlapping.

5. Consider the following scheduling problem. You have a n jobs, labeled $1, \ldots, n$, which must be run one at a time, on a single processor. Job j takes time t_j to be processed. We will assume that no two jobs have the same processing time; that is, there are no two distinct jobs i and j for which $t_i = t_j$.

   You must decide on a schedule: the order in which to run the jobs. Having fixed an order, each job j has a *completion time* under this order: this is the total amount of time that elapses (from the beginning of the schedule) before it is done being processed.

   For example, suppose you have a set of three jobs $\{1, 2, 3\}$ with

   $$t_1 = 3, \quad t_2 = 1, \quad t_3 = 5,$$

   and you run them in this order. Then the completion time of job 1 will be 3, the completion of job 2 will be $3 + 1 = 4$, and the completion time of job 3 will be $3 + 1 + 5 = 9$.

   On the other hand, if you run the jobs in the reverse of the order in which they're listed (i.e. 3, 2, 1), then the completion time of job 3 will be 5, the completion of job 2 will be $5 + 1 = 6$, and the completion time of job 1 will be $5 + 1 + 3 = 9$.

   (a) Give an algorithm that takes the n processing times $t_1, \ldots, t_n$, and orders the jobs so that the *sum* of the completion times of all jobs is as small as possible. (Such an order will be called *optimal*.)

The running time of your algorithm should be polynomial in n. You should give a complete proof of correctness of your algorithm, and also briefly analyze the running time. As above, you can assume that no two jobs have the same processing time.

(b) Prove that if no two jobs have the same processing time, then the optimal order is *unique*. In other words, for any order other than the one produced by your algorithm in (a), the sum of the completion times of all jobs is not as small as possible.

You may find it helpful to refer to parts of your analysis from (a).

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swin 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swins the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he/she's out and starts biking, a third contestant begins swimming ... and so on.)

   Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running. Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts.

   What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it re-compiles its index. Fortunately, the company has at its disposal a single large super-computer together with an essentially unlimited supply of high-end PC's.

   They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can performed completely independently of each other. Each job consists of two stages: first it needs to be *pre-processed* on the super-computer, and then it needs to be *finished* on one of the PC's. Let's say that job J_i needs p_i seconds of time on the super-computer followed by f_i seconds of time on a PC.

Since there are at least n PC's available on the premises, the finishing of the jobs can be performed fully in parallel — all the jobs can be processed at the same time. However, the super-computer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the super-computer. As soon as the first job in order is done on the super-computer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the super-computer; when the second job is done on the super-computer, it can proceed to a PC regardless of whether or not the first job is done or not (since the PC's work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the super-computer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PC's. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

8. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

   Now, because you're just one user, the link does not want you taking up too much bandwidth — so it imposes the following constraint, using a fixed parameter r:

   > ($*$) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.

   Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value.

   We say that a schedule is *valid* if it satisfies the constraint ($*$) imposed by the link.

   The problem is: Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with

$$(b_1, t_1) = (2000, 1), \quad (b_2, t_2) = (6000, 2), \quad (b_3, t_3) = (2000, 1),$$

and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order $1, 2, 3$, is valid, since the constraint $(*)$ is satisfied:

$t = 1$: the whole first stream has been sent, and $2000 < 5000 \cdot 1$
$t = 2$: half the second stream has also been sent,
and $2000 + 3000 < 5000 \cdot 2$
Similar calculations hold for $t = 3$ and $t = 4$.

(a) Consider the following claim:

Claim: There exists a valid schedule if and only if each stream i satisfies $b_i < rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

9. **(a)** Suppose you're a consultant for a communications company in northern New Jersey, and they come to you with the following problem. Consider a fiber-optic cable that passes through a set of n *terminals*, $t_1, \ldots, t_n$, in sequence. (I.e. it begins at terminal t_1, then passes through terminals $t_2, t_3, \ldots, t_{n-1}$, and ends at t_n.) Certain pairs of terminals wish to establish a *connection* on which they can exchange data; for t_i to t_j to establish a connection, they need to reserve access to the portion of the cable that runs between t_i and t_j.

   Now, the magic of fiber-optic technology is that you can accomodate all connections simultaneously as follows. You assign a *wavelength* to each connection in such a way that two connections requiring overlapping portions of the cable need to be assigned different wavelengths. (So you can assign the same wavelength more than once, provided it is to connections using non-overlapping portions of the cable.) Of course, you could safely assign a different wavelength to every single connection, but this would be wasteful: the goal is to use as few distinct wavelengths as possible.

   Define the *load* of the set of connection to be the maximum number of connections that require access to any single point on the cable. The load gives a natural lower

bound on the number of distinct wavelengths you need: if the load is L, then there is some point on the cable through which L connections will be sending data, and each of these needs a different wavelength.

For an arbitrary set of connections (each specified by a pair of terminals) having a load of L, is it always possible to accomodate all connections using only L wavelengths? If so, give an algorithm to assign each connection one of L possible wavelengths in a conflict-free fashion; if not, give an example of a set of connections requiring a number of wavelengths greater than its load.

(b) Instead of routing on a linear cable, let's look at the problem of routing on a *ring*. So we consider a circular fiber-optic cable, passing through terminals $t_1, \ldots, t_n$ in clockwise order. For t_i and t_j to establish a connection, they must reserve the portion of the cable extending clockwise from t_i to t_j.

The rest of the set-up is the same as in part (a), and we can ask the same question: For an arbitrary set of connections (each specified by a pair of terminals) having a load of L, is it always possible to accomodate all connections using only L wavelengths? If so, give an algorithm to assign each connection one of L possible wavelengths in a conflict-free fashion; if not, give an example of a set of connections requiring a number of wavelengths greater than its load.

10. Timing circuits are a crucial component of VLSI chips; here's a simple model of such a timing circuit. Consider a complete binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

    The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

    Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem: we want the leaves to be completely synchronized, and all receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain of the edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

    Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew, and the total edge length is as small as possible.

Example. Consider the tree in accompanying figure, with letters naming the nodes and numbers indicating the edge lengths.

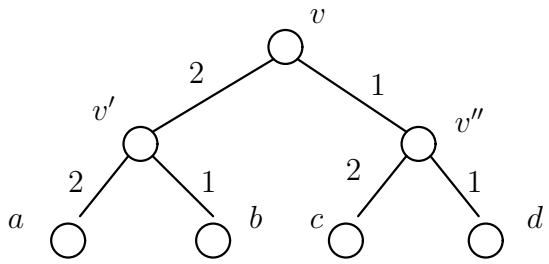


Figure 3.5: An instance of the zero-skew problem.

The unique optimal solution for this instance would be to take the three length-1 edges, and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

11. Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, v_2, \ldots, v_n\}$, then the degree of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.

12. Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They've researched all the travel options, and have drawn up a directed graph whose nodes represent intermediate destinations, and edges represent the roads between them.

    In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter, and may cause large travel delays. They've found an excellent travel Web site that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backwards in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

13. Suppose you are given an undirected graph G, with edge weights that you may assume are all distinct. G has n vertices and m edges. A particular edge e of G is specified. Give a algorithm with running time $O(m + n)$ to decide whether e is contained in a minimum-weight spanning tree of G.

14. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum cost spanning tree T in G. Now assume that a new edge is added, connecting two nodes $v, w \in V$ with cost c.

    a Give an efficient algorithm to test if T remains the minimum cost spanning tree with the new edge added. Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumption you make about what data structure is used to represent the tree T and the graph G.

    b Suppose T is no longer the minimum cost spanning tree. Give a linear time algorithm to update the tree T to the new minimum cost spanning tree.

15. One of the basic motivations behind the minimum spanning tree problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here, we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

    Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge weights that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest weight.

    A spanning tree T of G is a *minimum bottleneck spanning tree* if there is no spanning tree T' of G with a lighter bottleneck edge.

    (a) Is every minimum bottleneck tree of G a minimum spanning tree of G? Prove or give a counter-example.

    (b) Is every minimum spanning tree of G a minimum bottleneck tree of G? Prove or give a counter-example.

    (c)$(*)$ Give an algorithm with running time $O(m+n)$ that, on input G, computes a minimum bottleneck spanning tree of G. (*Hint: You may use the fact that the median of a set of k numbers can be computed in time $O(k)$.*)

16. In trying to understand the combinatorial structure of spanning trees, we can consider the space of *all* possible spanning trees of a given graph, and study the properties of this space. This is a strategy that has been applied to many similar problems as well.

    Here is one way to do this. Let G be a connected graph, and T and T' two different spanning trees of G. We say that T and T' are *neighbors* if T contains exactly one edge that is not in T', and T' contains exactly one edge that is not in T.

    Now, from any graph G, we can build a (large) graph $\mathcal{H}$ as follows. The nodes of $\mathcal{H}$ are the spanning trees of G, and there is an edge between two nodes of $\mathcal{H}$ if the corresponding spanning trees are neighbors.

    Is it true that for any connected graph G, the resulting graph $\mathcal{H}$ is connected? Give a proof that $\mathcal{H}$ is always connected, or provide an example (with explanation) of a connected graph G for which $\mathcal{H}$ is not connected.

17. Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a connected graph $G = (V, E)$ with n nodes. Each edge e is a fiber-optic cable that is owned by one of two companies — creatively named X and Y — and leased to CluNet.

    Their plan is to choose a spanning tree T of G, and upgrade the links corresponding to the edges of T. Their business relations people have already concluded an agreement with companies X and Y stipulating a number k so that in the tree T that is chosen, k of the edges will be owned by X and $n - k - 1$ of the edges will be owned by Y.

    CluNet management now faces the following problem: It is not at all clear to them whether there even *exists* a spanning tree T meeting these conditions, and how to find one if it exists. So this is the problem they put to you: give a polynomial-time algorithm that takes G, with each edge labeled X or Y, and either (i) returns a spanning tree with exactly k edges labeled X, or (ii) reports correctly that no such tree exists.

18. Suppose you are given a connected graph $G = (V, E)$, with a weight w_e on each edge e. On the first problem set, we saw that when all edge weights are distinct, G has a unique minimum-weight spanning tree. However, G may have many minimum-weight spanning trees when the edge weights are not all distinct. Here, we formulate the question: can Kruskal's algorithm be made to find all the minimum-weight spanning trees of G?

    Recall that Kruskal's algorithm sorted the edges in order of increasing weight, then greedily processed edges one-by-one, adding an edge e as long as it did not form a cycle. When some edges have the same weight, the phrase "in order of increasing weight" has to be specified a little more carefully: we'll say that an ordering of the edges is *valid* if the corresponding sequence of edge weights is non-decreasing. We'll say that a *valid*

execution of Kruskal's algorithm is one that begins with a valid ordering of the edges of G.

For any graph G, and any minimum spanning tree T of G, is there a valid execution of Kruskal's algorithm on G that produces T as output? Give a proof or a counter-example.

19. Every September, somewhere in a far-away mountainous part of the world, the county highway crews get together and decide which roads to keep clear through the coming winter. There are n towns in this county, and the road system can be viewed as a (connected) graph $G = (V, E)$ on this set of towns, each edge representing a road joining two of them. In the winter, people are high enough up in the mountains that they stop worrying about the *length* of roads and start worrying about their *altitude* — this is really what determines how difficult the trip will be.

    So each road — each edge e in the graph — is annotated with a number a_e that gives the altitude of the highest point on the road. We'll assume that no two edges have exactly the same altitude value a_e. The *height* of a path P in the graph is then the maximum of a_e over all edges e on P. Finally, a path between towns i and j is declared to be *winter-optimal* if it achieves the minimum possible height over all paths from i to j.

    The highway crews are going to select a set $E' \subseteq E$ of the roads to keep clear through the winter; the rest will be left unmaintained and kept off limits to travelers. They all agree that whichever subset of roads E' they decide to keep clear, it should clearly have the property that (V, E') is a connected subgraph; and more strongly, for every pair of towns i and j, the height of the winter-optimal path in (V, E') should be no greater than it is in the full graph $G = (V, E)$. We'll say that (V, E') is a *minimum-altitude connected subgraph* if it has this property.

    Given that they're going to maintain this key property, however, they otherwise want to keep as few roads clear as possible. One year, they hit upon the following conjecture:

    > *The minimum spanning tree of G, with respect to the edge weights a_e, is a minimum-altitude connected subgraph.*

    (In an earlier problem, we claimed that there is a unique minimum spanning tree when the edge weights are distinct. Thus, thanks to the assumption that all a_e are distinct, it is okay for us to speak of *the* minimum spanning tree.)

    Initially, this conjecture is a somewhat counter-intuitive claim, since the minimum spanning tree is trying to minimize the *sum* of the values a_e, while the goal of minimizing altitude seems to be asking for a fairly different thing. But lacking an argument to the contrary, they begin considering an even bolder second conjecture:

A subgraph (V, E') is a minimum-altitude connected subgraph if and only if it contains the edges of the minimum spanning tree.

Note that this second conjecture would immediately imply the first one, since the minimum spanning tree contains its own edges.

So here's the question:

(a) Is the first conjecture true, for all choices of G and altitudes a_e? Give a proof or a counter-example with explanation.

(b) Is the second conjecture true, for all choices of G and altitudes a_e? Give a proof or a counter-example with explanation.

20. One of the first things you learn in calculus is how to minimize a differentiable function like $y = ax^2 + bx + c$, where $a > 0$. The minimum spanning tree problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved — rather than a continuum of possibilities — and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

    One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a *time-varying edge cost* given by a function $f_e : \mathbf{R} \to \mathbf{R}$. Thus, at time t, it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we'll denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ is minimized.

    Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) : e \in E\}$, and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

21. Suppose we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, together with a distance function d on the set P; as usual, d is simply a function on pairs of points in P with the properties that $d(p_i, p_j) = d(p_j, p_i) > 0$ if $i \neq j$, and that $d(p_i, p_i) = 0$ for each i.

    We define a *stratified metric* on P to be any distance function τ that can be constructed as follows. We build a rooted tree T with n leaves, and we associate with each node

v of T (both leaves and internal nodes) a *height* h_v. These heights must satisfy the properties that $h(v) = 0$ for each leaf v, and if u is the parent of v in T, then $h(u) \geq h(v)$. We place each point in P at a distinct leaf in T. Now, for any pair of points p_i and p_j, their distance $\tau(p_i, p_j)$ is defined as follows. We determine the least common ancestor v in T of the leaves containing p_i and p_j, and define $\tau(p_i, p_j) = h_v$.

We say that a stratified metric τ is *consistent* with our distance function d if for all pairs i, j, we have $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Give a polynomial-time algorithm that takes the distance function d and produces a stratified metric τ with the following properties:

(i) τ is consistent with d, and

(ii) if τ' is any other stratified metric consistent with d, then $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for each pair of points p_i and p_j.

22. Let's go back to the original motivation for the minimum spanning tree problem: we are given a connected, undirected graph $G = (V, E)$ with positive edge lengths $\{\ell_e\}$, and we want to find a spanning subgraph of it. Now, suppose we are willing to settle for a subgraph $H = (V, F)$ that is "denser" than a tree, and we are interested in guaranteeing that for each pair of vertices $u, v \in V$, the length of the shortest u-v path in H is not much longer than the length of the shortest u-v path in G. By the *length* of a path P here, we mean the sum of ℓ_e over all edges e in P.

Here's a variant of Kruskal's algorithm designed to produce such a subgraph.

- First, we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)
- We then construct a subgraph $H = (V, F)$ by considering each edge in order.
- When we come to edge $e = (u, v)$, we add e to the subgraph H if there is currently no u-v path in H. (This is what Kruskal's algorithm would do as well.) On the other hand, if there is a u-v path in H, we let d_{uv} denote the total length of the shortest such path; again, length is with respect to the values $\{\ell_e\}$. We add e to H if $3\ell_e < d_{uv}$.

In other words, we add an edge even when u and v are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let $H = (V, F)$ be the subgraph of G returned by the algorithm.

(a) Prove that for every pair of nodes $u, v \in V$, the length of the shortest u-v path in H is at most 3 times the length of the shortest u-v path in G.

(b)(∗) Despite its ability to approximately preserve shortest-path distances, the subgraph H produced by the algorithm cannot be too dense. Let $f(n)$ denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all n-node input graphs with edge lengths. Prove that

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0.$$

23. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

    We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a sub-tree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

    Show that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(n^{O(k)})$.

24. Recall the problem of computing a minimum-cost arborescence in a directed graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge. Here we will consider the case in which G is a directed acyclic graph; that is, it contains no directed cycles.

    As in general directed graphs, there can in general be many distinct minimum-cost solutions. Suppose we are given a directed acyclic graph $G = (V, E)$, and an arborescence $A \subseteq E$ with the guarantee that for every $e \in A$, e belongs to *some* minimum-cost arborescence in G. Can we conclude that A itself must be a minimum-cost arborescence in G? Give a proof, or a counter-example with explanation.

25. Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the min-cost arborescence algorithm.

    (a) The algorithm discussed in Section 3.5 works as follows: we modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strongly connected components of this subgraph.

    (b) In the course of the algorithm, we defined y_v to be the min cost of an edge entering v, and we modified the costs of all edges e entering node v to be $c'_e = c_e - y_v$. Suppose we instead use the following modified cost: $c''_e = \max(0, c_e - 2y_v)$. This new change is likely to turn more edges 0 cost. Suppose, now we find an arborescence T of 0 cost. Prove that this T has cost at most twice the cost of the minimum cost arborescence in the original graph.

(c)(∗) Assume you do not find an arborescence of 0 cost. Contract all 0-cost strongly connected components, and recursively apply the same procedure on the resulting graph till an arborescence is found. Prove that this T has cost at most twice the cost of the minimum cost arborescence in the original graph.

26. (∗) Suppose you are given a directed graph $G = (V, E)$ in which each edge has a cost of either 0 or 1. Also, suppose that G has a node r such that there is a path from r to each other node in G. You are also given an integer k. Give a polynomial-time algorithm that either constructs an arborescence rooted at r of cost *exactly* k, or reports (correctly) that no such arborescence exists.

Chapter 4

Divide and Conquer

Divide-and-conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these sub-problems into an overall solution. In many cases, it can be a simple and powerful method.

We will not devote too much time to divide-and-conquer as a technique in its own right, for two main reasons. First, it is an idea that should already be familiar from earlier courses. many of the most fundamental methods for sorting and searching follow the divide-and-conquer paradigm; these include binary search, QUICKSORT, and MERGESORT. Second, it is a technique that pervades the design of many algorithms, and we will see it implicitly in a number of subsequent chapters.

For our coverage of divide-and-conquer here, we focus on what is arguably its most widespread impact — taking problems for which a naive approach requires $O(n^2)$ running time, and producing an algorithm with the much better running time of $O(n \log n)$. Improvements of this type are very often a consequence of the following general approach:

> (†) Break the input into two pieces of equal size; solve the two sub-problems on these pieces separately by recursion; and then spend linear time to combine the two results into an overall solution.

The MERGESORT algorithm for sorting a list of n numbers does precisely this: it recursively sorts the front half and back halves separately, and then "merges" these two sorted halves in an additional $O(n)$ steps. We will prove below that any algorithm with this general structure has an $O(n \log n)$ running time; and this fact actually goes a long way towards explaining the ubiquity of the function $O(n \log n)$ in computer science.

4.1 A Useful Recurrence Relation

Consider an algorithm that follows the technique outlined in (†), and let $T(n)$ denote its maximum running time on input instances of size n. The algorithm divides the input into

pieces of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, spends $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ to solve these two sub-problems recursively, and then spends $O(n)$ to combine the solutions. Thus, the running time satisfies the following *recurrence relation*:

(4.1) *$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$ for $n \geq 2$, and $T(1) = c_1$ for a constant c_1.*

You'll often see this written as $T(n) \leq 2T(n/2) + O(n)$, under the (often unstated) assumption that n is power of 2.

This recurrence relation can be used to derive a tight upper bound on $T(n)$. In the analysis we use $\log n$ to mean the base 2 logarithm $\log_2 n$; but recall that the base of the logarithm does not matter inside $O(\cdot)$ notation, since different bases result in a change of only a constant factor.

(4.2) *Any function $T(\cdot)$ satisfying (4.1) is bounded by $O(n \log n)$, when $n > 1$.*

Proof. Let us make the constant inside the "$O(\cdot)$" notation of (4.1) explicit: we write it as $c'n$ where c' is an absolute constants. Assume that $n > 1$ and let $k = \lceil \log_2 n \rceil$. Note that applying (4.1) directly, we have $T(2) \leq 2c_1 + c'$; let c denote this constant quantity.

We claim that for $n > 1$ we have $T(n) \leq ckn$, and prove this by induction on n. (Note that we start the induction proof at $n = 2$ since $\log 1 = 0$. The claim is clearly true when $n = 2$, since $T(2) \leq c$ by definition.

Now, for a general value of n, write $n_1 = \lceil n/2 \rceil$ and $n_2 = \lfloor n/2 \rfloor$. A key observation, which is not difficult to show, is that $\lceil \log_2 n_i \rceil \leq k - 1$ for $i = 1, 2$.

We use this fact together with the induction hypothesis for $T(n_1)$ and $T(n_2)$:

$$\begin{aligned} T(n) &\leq T(n_1) + T(n_2) + c'n \\ &\leq T(n_1) + T(n_2) + cn \\ &\leq c(k-1)n_1 + c(k-1)n_2 + cn \\ &= c(k-1)(n_1 + n_2) + cn \\ &= ckn. \end{aligned}$$

■

We apply this result in the remainder of the chapter. We consider two problems which initially seem to require quadratic time, and show how to use the approach described above to get a running time of $O(n \log n)$. In both cases, it takes some work to make the general divide-and-conquer strategy actually succeed.

4.2 Counting Inversions

Variants of the MERGESORT technique can be used to solve some problems that are not directly related to sorting elements.

A number of sites on the Web try to match your preferences (for books, movies, restaurants) with those of other people out on the Internet. You rank a set of n movies, and then the site consults its database to look for other people who had "similar" rankings. But what's a good way to measure, numerically, how similar two people's rankings are? Clearly an identical ranking is very similar, and a completely reversed ranking is very different; we want something that interpolates through the middle region.

Let's consider comparing your ranking and a stranger's ranking of the same set of n movies. A natural method would to label the movies from 1 to n according to your ranking, then order these labels according to the stranger's ranking, and see how many pairs are "out of order." More concretely, we will consider the following problem. We are given a sequence of n numbers $a_1, \ldots, a_n$; we will assume that all the numbers are distinct. We want to define a measure that tells us how far this list is from being in ascending order; the value of the measure should be 0 if $a_1 < a_2 < \ldots < a_n$, and should increase as the numbers become more scrambled.

A natural way to quantify this notion is by counting the number of *inversions*. We say that two indices $i < j$ form an inversion if $a_i > a_j$, i.e., if the two elements a_i and a_j are "out of order." We will seek to determine the number of inversions in the sequence $a_1, \ldots, a_n$. Note that if the sequence is in ascending order, then there are no inversions; if the sequence is in descending order (i.e. as bad as possible), then every pair forms an inversion, and so there are $n(n-1)/2$ of them.

What is the simplest algorithm to count inversions? Clearly, we could look at every pair of numbers (a_i, a_j) and determine whether they constitute an inversion; this would take $O(n^2)$ time.

We now show how to count the number of inversions much more quickly, in $O(n \log n)$ time. Note that since there can be a quadratic number of inversions, such an algorithm must be able to compute the total number without ever *looking* at each inversion individually. The basic idea is to follow the strategy (†) defined above. We set $m = \lceil n/2 \rceil$ and divide the list into the two pieces $a_1, \ldots, a_m$ and $a_{m+1}, \ldots, a_n$. We first count the number of inversions in each of these two halves separately. Then, we count the number of inversions (a_i, a_j), where the two numbers belong to different halves; the trick is that we must do this part in $O(n)$ time, if we want to apply (4.2) . Note that these first-half/second-half inversions have a particularly nice form: they are precisely the pairs (a_i, a_j) where a_i is in the first half, a_j is in the second half, and $a_i > a_j$.

To help with counting the number of inversions between the two halves, we will make the algorithm recursively sort the numbers in the two halves as well. Having the recursive

step do a bit more work (sorting as well as counting inversions) will make the "combining" portion of the algorithm easier.

So the crucial routine in this process is `Merge-and-Count`. Suppose we have recursively sorted the first and second halves of the list, and counted the inversions in each. We now have two sorted lists A and B, containing the first and second halves respectively. We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$, $b \in B$, and $a > b$. By our discussion above, this is precisely what we will need for the "combining" step that computes the number of first-half/second-half inversions.

The `Merge-and-Count` routine walks through the sorted lists A and B, removing elements from the front and appending them to the sorted list C. In one step, it compares the elements a_i and b_j being pointed to in each list, removes the smaller one from its list, and appends it to the end of list C. Now, because A and B are sorted, it is very easy to keep track of the number of inversions we encounter. Every time the element a_i is appended to C, no new inversions are encountered — since a_i is smaller than everything left in list B, and it comes before all of them. On the other hand, if b_j is appended to list C, then it is smaller than all the remaining items in A, and it comes after all of them — so we increase our count of the number of inversions by the number of elements remaining in A. This is the crucial idea: in constant time, we have accounted for a potentially large number of inversions. See Figure 4.1 for an illustration of this process.

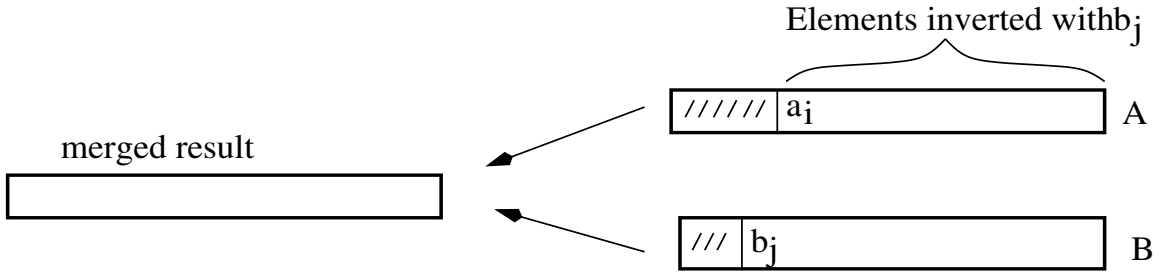


Figure 4.1: Merging the first half of the list A with the second half B

We use this `Merge-and-Count` routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list L.

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements.
         B contains the remaining ⌊n/2⌋ elements.
      (r_A, A)=Sort-and-Count(A)
```

```
    (r_B, B)=Sort-and-Count(B)
    (r, L)=Merge-and-Count(A, B)
Endif
Return r = r_A + r_B + r, and the sorted list L
```

Since our `Merge-and-Count` procedure takes $O(n)$ time, the running time $T(n)$ of the full `Sort-and-Count` procedure satisfies the recurrence (4.1). By (4.2), we have

(4.3) *The* `Sort-and-Count` *algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.*

4.3 Finding the Closest Pair of Points

We now describe another problem that can be solved by an algorithm in the style we've been discussing; but finding the right way to "merge" the solutions to the two sub-problems it generates requires quite a bit of ingenuity. The problem itself is very simple to state: given n points in the plane, find the pair that is closest together.

The problem was considered by M.I. Shamos and D. Hoey in the early 1970's, as part of their project to work out efficient algorithms for basic computational primitives in geometry. These algorithms formed the foundations of the then-fledgling field of *computational geometry*, and they have found their way into areas such as graphics, computer vision, geographic information systems, and molecular modeling. And although the closest-pair problem is one of the most natural algorithmic problems in geometry, it is surprisingly hard to find an efficient algorithm for it. It is immediately clear that there is an $O(n^2)$ solution — compute the distance between each pair of points and take the minimum — and so Shamos and Hoey asked whether an algorithm asymptotically faster than quadratic could be found. It took quite a long time before they resolved this question, and the $O(n \log n)$ algorithm we give below is essentially the one they discovered.

We begin with a bit of notation. Let us denote the set of points by $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates (x_i, y_i); and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find the pair of points p_i, p_j which minimizes $d(p_i, p_j)$.

We will assume that no two points in P have the same x-coordinate or the same y-coordinate. This makes the discussion cleaner; and it's easy to eliminate this assumption either by initially applying a rotation to the points that makes it true, or by slightly extending the algorithm we develop here.

It's instructive to consider the one-dimensional version of this problem for a minute, since it is much simpler and the contrasts are revealing. How would we find the closest pair of points on a line? We'd first sort them, in $O(n \log n)$ time, and then we'd walk through the

sorted list, computing the distance from each point to the one that comes after it. It is easy to see that one of these distances must be the minimum one.

In two dimensions, we could try sorting the points by their y-coordinate (or x-coordinate), and hoping that the two closest points were near one another in the order of this sorted list. But it is easy to construct examples in which they are very far apart, preventing us from adapting our one-dimensional approach.

Instead, our plan will be to apply the style of divide-and-conquer used in MERGESORT: we find the closest pair among the points in the "left half" of P and the closest pair among the points in the "right half" of P; and then we need to use this information to get the final solution in linear time. This last part is the catch: the distances that have not been considered by either of our recursive calls are precisely those that occur between a point in the left half and a point in the right half; there are $\Omega(n^2)$ such distances, yet we need to find the smallest one in $O(n)$ time after the recursive calls return. If we can do this, our solution will be complete: it will be the smallest of the values computed in the recursive calls and this minimum "left-to-right" distance.

Setting up the Recursion. Let's get a few easy things out of the way first. It will be very useful if every recursive call, on a set $P' \subseteq P$, begins with two lists: a list P'_x in which all the points in P' have been sorted by increasing x-coordinate, and a list P'_y in which all the points in P' have been sorted by increasing y-coordinate. We can ensure that this remains true throughout the algorithm as follows.

First, before any of the recursion begins, we sort all the points in P by x-coordinate and again by y-coordinate, producing lists P_x and P_y. Attached to each entry in each list is a record of the position of that point in both lists.

The first level of recursion will work as follows, with all further levels working in a completely analogous way. We define Q to be the set of points in the first $\lceil n/2 \rceil$ positions of the list P_x (the "left half") and R to be the set of points in the final $\lfloor n/2 \rfloor$ positions of the list P_x (the "right half"). See Figure 4.2. By a single pass through each of P_x and P_y, in $O(n)$ time, we can create the following four lists: Q_x, consisting of the points in Q sorted by increasing x-coordinate; Q_y, consisting of the points in Q sorted by increasing y-coordinate; and analogous lists R_x and R_y. For each entry of each of these lists, as before, we record the position of the point in both lists it belongs to.

We now recursively determine the closest pair of points in Q (with access to the lists Q_x and Q_y). Suppose that q_0^* and q_1^* are (correctly) returned as a closest pair of points in Q. Similarly, we determine the closest pair of points in R, obtaining r_0^* and r_1^*.

Combining the Solutions. The general machinery of divide-and-conquer has gotten us this far, without our really having delved into the structure of the closest-pair problem.

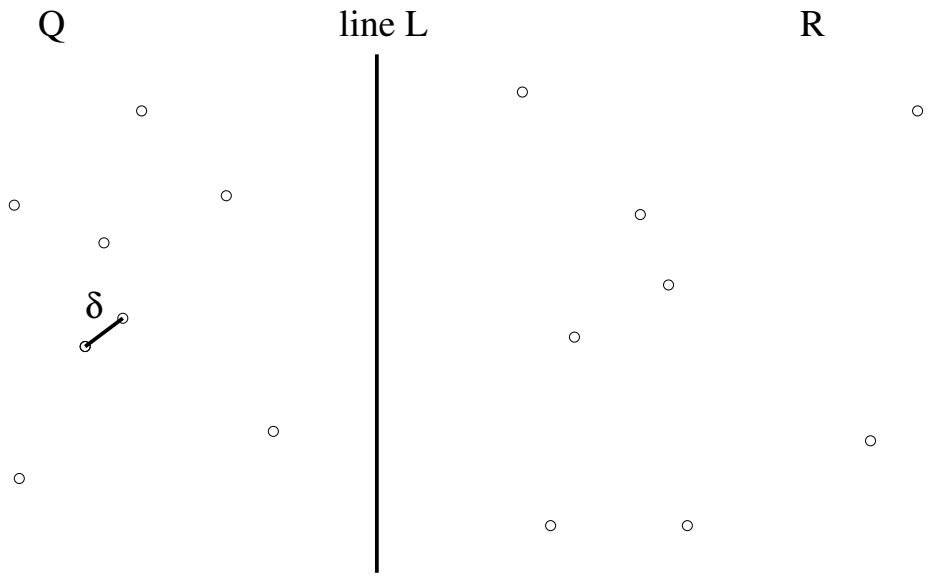


Figure 4.2: The first level of recursion

But it still leaves us with the problem that we saw looming originally: how do we use the solutions to the two sub-problems as part of a linear-time "combining" operation?

Let δ be the minimum of $d(q_0^*,q_1^*)$ and $d(r_0^*,r_1^*)$. The real question is: are there points $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$? If not, then we have already found the closest pair in one of our recursive calls. But if there are, then the closest such q and r form the closest pair in P.

Let x^* denote the x-coordinate of the rightmost point in Q, and let L denote the vertical line described by the equation $x = x^*$. This line L "separates" Q from R. Here is a simple fact:

(4.4) *If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L.*

Proof. Suppose such q and r exist; we write $q = (q_x, q_y)$ and $r = (r_x, r_y)$. By the definition of x^*, we know that $q_x \leq x^* \leq r_x$. Then we have

$$x^* - q_x \leq r_x - q_x \leq d(q,r) < \delta$$

and

$$r_x - x^* \leq r_x - q_x \leq d(q,r) < \delta,$$

so each of q and r has an x-coordinate within δ of x^*, and hence lies within distance δ of the line L. ■

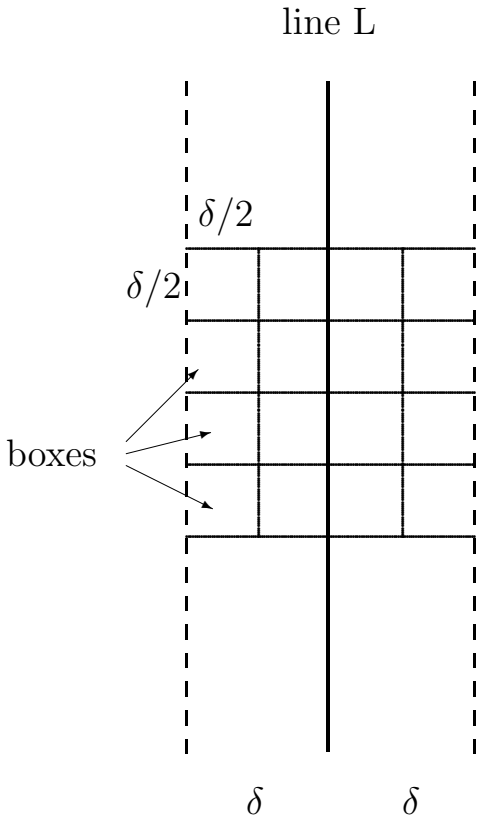
So if we want to find a close q and r, we can restrict our search to the narrow band consisting only of points in P within δ of L. Let $S \subseteq P$ denote this set, and let S_y denote the list consisting of the points in S sorted by increasing y-coordinate. By a single pass through the list P_y, we can construct S_y in $O(n)$ time.

We can restate (4.4) as follows, in terms of the set S.

(4.5) *There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ for which $d(s, s') < \delta$.*

It's worth noticing at this point that S might in fact be the whole set P, in which case (4.4) and (4.5) really seem to buy us nothing. But this is actually far from true, as the following amazing fact shows.

(4.6) *If $s, s' \in S$ have the property that $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.*



Proof. Consider the subset Z of the plane consisting of all points within distance δ of L. We partition Z into *boxes*: squares with horizontal and vertical sides of length $\delta/2$. One *row* of Z will consist of four boxes whose horizontal sides have the same y-coordinates.

Suppose two points of S lay in the same box. Since all points in this box lie on the same side of L, these two points either both belong to Q or both belong to R. But any two points in the same box are within distance $\delta \cdot \sqrt{2}/2 < \delta$, which contradicts our definition of δ as the minimum distance between any pair of points in Q or in R. Thus, each box contains at most one point of S.

Now suppose that $s, s' \in S$ have the property that $d(s, s') < \delta$, and that they are at least 16 positions apart in S_y. Assume without loss of generality that s has the smaller y-coordinate. Then since there can be at most one point per box, there are at least three rows of Z lying between s and s'. But any two points in Z separated by at least three rows must be a distance of at least $3\delta/2$ apart — a contradiction. ■

We note that the value of 15 can be reduced; but for our purposes at the moment, the important thing is that it is an absolute constant.

In view of (4.6), we can conclude the algorithm as follows. We make one pass through S_y, and for each $s \in S_y$, we compute its distance to each of the next 15 points in S_y. (4.6) implies that in doing so, we will have computed the distance of each pair of points in S (if any) that are at distance less than δ from one another. So having done this, we can compare the smallest such distance to δ, and we can report one of two things: (i) the closest pair of points in S, if their distance is less than δ; or (ii) the (correct) conclusion that no pairs of points in S are within δ of one another. In case (i), this pair is the closest pair in P; in case (ii), the closest pair found by our recursive calls is the closest pair in P.

Note the resemblance between this procedure and the algorithm we rejected at the very beginning, which tried to make one pass through P in order of y-coordinate. The reason such an approach works now is due to the extra knowledge (the value of δ) we've gained from the recursive calls, and the special structure of the set S.

This concludes the description of the "combining" part of the algorithm, since by (4.5) we have now determined whether the minimum distance between a point in Q and a point in R is less than δ, and if so, we have found the closest such pair.

A complete description of the algorithm and its proof of correctness are implicitly contained in the discussion so far, but for the sake of concreteness, we now summarize both.

Summary. A high-level description of the algorithm is the following, using the notation we have developed above.

```
Closest-Pair(P)
  Construct P_x and P_y   (O(n log n) time)
  (p_0^*,p_1^*) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
```

```
  find closest pair by measuring all pairwise distances

Construct Q_x,Q_y,R_x,R_y (O(n) time)
(q_0^*,q_1^*) = Closest-Pair-Rec(Q_x, Q_y)
(r_0^*,r_1^*) = Closest-Pair-Rec(R_x, R_y)

δ = min(d(q_0^*,q_1^*), d(r_0^*,r_1^*))
x^* = maximum x-coordinate of a point in set Q
L = {(x,y) : x = x^*}
S = points in P within distance δ of L.

Construct S_y (O(n) time)
For each point s ∈ S_y, compute distance from s
   to each of next 15 points in S_y.
   Let s,s' be pair achieving minimum of these distances
   (O(n) time)

If d(s,s') < δ then
   Return (s,s')
Else if d(q_0^*,q_1^*) < d(r_0^*,r_1^*) then
   Return (q_0^*,q_1^*)
Else
   Return (r_0^*,r_1^*)
```

(4.7) *The algorithm correctly outputs a closest pair of points in P.*

Proof. As we've noted, all the components of the proof have already been worked out above; so here we just summarize how they fit together.

We prove the correctness by induction on the size of P, the case of $|P| \leq 3$ being clear. For a given P, the closest pair in the recursive calls is computed correctly by induction. By (4.6) and (4.5), the remainder of the algorithm correctly determines whether any pair of points in S is at distance less than δ, and if so returns the closest such pair. Now the closest pair in P either has both elements in one of Q or R, or it has one element in each. In the former case, the closest pair is correctly found by the recursive call; in the latter case, this pair is at distance less than δ, and it is correctly found by the remainder of the algorithm. ■

(4.8) *The running time of the algorithm is $O(n \log n)$.*

Proof. The initial sorting of P by x- and y-coordinate takes time $O(n \log n)$. The running time of the remainder of the algorithm satisfies the recurrence (4.1), and hence is $O(n \log n)$ by (4.2). ■

4.4 Exercises

1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values — so there are $2n$ values total — and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

   However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   Give an algorithm that finds the median value using at most $O(\log n)$ queries.

2. Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume be all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

   We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

3. $(*)$ *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction — when Woody is standing in front of Buzz you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.

   The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n non-vertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost — intuitively, some portion of it can be seen if you look down from "$y = \infty$."

   Give an algorithm that takes n lines as input, and in $O(n \log n)$ time returns all of the ones that are visible. The accompanying figure gives an example.

4. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n "smart-cards" that they've confiscated, suspecting them of being used in fraud. Each smart-card is

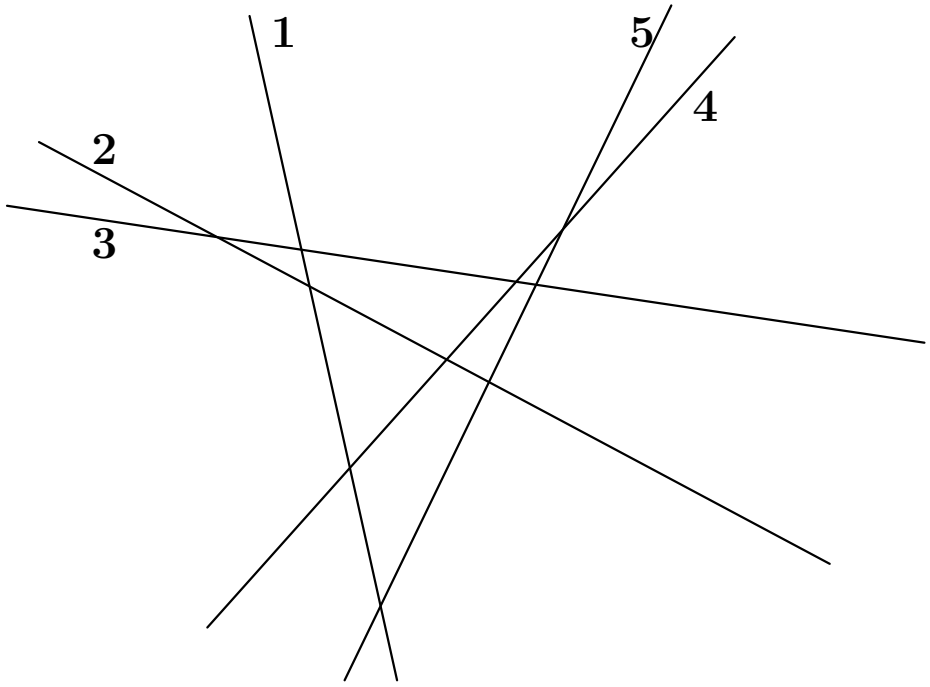


Figure 4.3: An instance with five lines (labeled "1"–"5" in the figure). All the lines except for "2" are visible.

a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many smart-cards corresponding to it, and we'll say that two smart-cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a smart-card directly, but the bank has a high-tech "equivalence tester" that takes two smart-cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Chapter 5

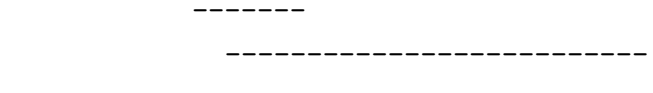
Dynamic Programming

5.1 Weighted Interval Scheduling: The Basic Set-up

We have seen that a particular greedy algorithm produces an optimal solution to the interval scheduling problem, where the goal is to accept as large a set of non-overlapping intervals as possible. We also discussed a more general version of the problem, *weighted interval scheduling*, in which each interval has a certain *value* to us, and we want to accept a set of maximum value.

In this section, we'll consider the following natural version of the weighted interval scheduling problem: the *value* of each interval is proportional to its length, so our goal is to accept a set of intervals of maximum total length. This arises naturally in the case in which our resource is available for rent, for a fixed rate per hour; then the amount of revenue we accrue from a given interval is proportional to its length, and we want to maximize our revenue. We'll see at the end that the solution we develop for this problem carries over pretty much directly to the case in which each interval has an arbitrary value.

Using the same types of examples we applied to the original interval scheduling problem, we can show that essentially every natural greedy algorithm one might think of can fail to find the optimal solution. Even our previously successful algorithm, in which we sort by increasing finish times, does not work for the current problem; see the picture below:



Since we last looked at interval scheduling problems, we've also seen the divide-and-conquer technique. But it seems difficult to apply this to the current problem as well. For example, we could divide the intervals into two sets, solve the problem optimally on each set, and then combine these solutions; but the solutions will presumably overlap, and it is not clear how to deal with this. Alternately we could divide "time" into two halves, and try solving the problem optimally in each half. But it may well be the case that there is no

dividing point on the time line that would not cut through several intervals; and then it is not clear how even to set up the sub-problems.

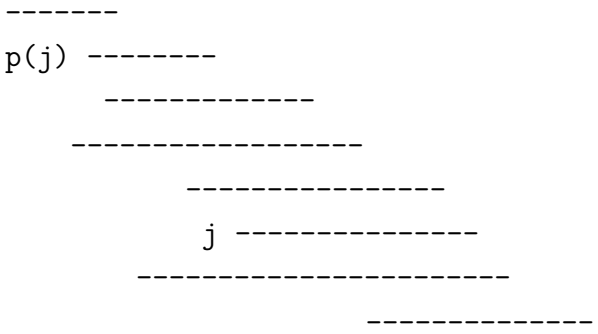
There is, however, a very efficient algorithm for this problem. It is based on the idea of breaking things down into sub-problems, but it manages the set of sub-problems in a more careful way than we saw with divide-and-conquer. There are two styles in which to develop the algorithm, conceptually different but arriving at essentially the same final result. We'll describe both of them, in an attempt to get at the subtle nature of the underlying idea: *dynamic programming*.

Style #1: Branching with Memoization

We keep the notation from our previous encounter with interval scheduling problems: We have n requests labeled $1, \ldots, n$, with each request i specifying a start time s_i and a finish time f_i. The *length* ℓ_i of request i is the difference between its finish time and its start time: $\ell_i = f_i - s_i$. Two requests are *compatible* if they do not overlap. The goal of our current problem is to select a subset $S \subseteq \{1, \ldots, n\}$ of mutually compatible requests, so as to maximize the sum of the lengths of the requests in S,

$$\sum_{i \in S} \ell_i = \sum_{i \in S} f_i - s_i.$$

Let's suppose that the requests are sorted in order of non-decreasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$. We'll say a request i comes *before* a request j if $i < j$. Here's a bit of notation that will be very useful for us: for a request j, let $p(j)$ denote the largest-numbered request before j that is compatible with j. We define $p(j) = 0$ if no request before j is compatible with j. Note that the set of *all* compatible requests before j is simply $\{1, 2, \ldots, p(j)\}$. An example is illustrated below.



Now, let's consider an optimal solution $\mathcal{O}$, ignoring for now that we have no idea what it is. Here's something completely obvious that we can say about $\mathcal{O}$: either request n (the last one) belongs to $\mathcal{O}$, or it doesn't. Suppose we explore both sides of this dichotomy a little further. If $n \in \mathcal{O}$, then clearly no interval strictly between $p(n)$ and n can belong to

$\mathcal{O}$. Moreover, in this case, $\mathcal{O}$ must include an *optimal* solution to the problem consisting of requests $\{1, \ldots, p(n)\}$ — for if it didn't, we could replace its choice of requests from $\{1, \ldots, p(n)\}$ with a better one, with no danger of overlapping request n. On the other hand, if $n \notin \mathcal{O}$, then $\mathcal{O}$ is simply equal to the optimal solution to the problem consisting of requests $\{1, \ldots, n-1\}$. This is by completely analogous reasoning: we're assuming that $\mathcal{O}$ does not include request n; so if it does not choose the optimal set of requests from $\{1, \ldots, n-1\}$, we could replace it with a better one.

For any value of i between 1 and n, let $OPT(i)$ denote the value of an optimal solution to the problem consisting of requests $\{1, \ldots, i\}$. The optimal value we're seeking is precisely $OPT(n)$. So for our optimal solution $\mathcal{O}$, we've observed that either $n \in \mathcal{O}$, in which case $OPT(n) = \ell_n + OPT(p(n))$, or $n \notin \mathcal{O}$, in which case $OPT(n) = OPT(n-1)$. Since these are precisely the two possible choices ($n \in \mathcal{O}$ or $n \notin \mathcal{O}$), we can further say:

(5.1) $OPT(n) = \max(\ell_n + OPT(p(n)), OPT(n-1))$.

And how do we decide whether n belongs to an optimal solution? This too is easy: it belongs to an optimal solution if and only if the first of the options above is at least as good as the second; in other words,

(5.2) *Request n belongs to an optimal solution if and only if*

$$\ell_n + OPT(p(n)) \geq OPT(n-1).$$

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller sub-problems.

Indeed, this recurrence equation (5.1) directly gives us a recursive algorithm to compute $OPT(n)$, assuming that we have already sorted the requests by finishing time and computed the values of $p(j)$ for each j.

```
Compute-Opt(n)
  If n = 0 then
    Return 0
  Else
    v = Compute-Opt(p(n))
    v' = Compute-Opt(n − 1)
    If ℓ_n + v ≥ v' then
      Return ℓ_n + v
    Else
      Return v'
    Endif
  Endif
```

The correctness of the algorithm follows directly from (5.1).

If we really implemented the algorithm as just written, it would take exponential time to run in the worst case; this is not surprising, since each call generates two new calls, while potentially only shrinking the problem size from n to $n - O(1)$. Thus we have not achieved a polynomial-time solution.

A fundamental observation, which forms the second crucial component of a dynamic programming solution, is that our recursive algorithm is really only solving $n + 1$ different sub-problems: `Compute-Opt(0)`, `Compute-Opt(1)`, ... `Compute-Opt(`n`)`. The fact it runs in exponential time as written, is simply due to the spectacular redundancy in the number of times it issues each of these calls.

How could we eliminate all this redundancy? We could store the value of `Compute-Opt(`i`)` in a globally accessible place the first time we compute it, and then simply use this pre-computed value in place of all future recursive calls. This technique of saving values that have already been computed is often called *memoization*.

We implement the above strategy in the more "intelligent" procedure `M-Compute-Opt`. This procedure will make use of an array $M[0 \dots n]$; $M[i]$ will start with the value "empty," but will hold the value of `Compute-Opt(`i`)` as soon as it is first determined. To determine $OPT(n)$, we invoke `M-Compute-Opt(`n`)`.

```
M-Compute-Opt(n)
  If M[n] = "empty" then
    v = Compute-Opt(n)
    M[n] = v.
    Return v.
  Else
    Return M[n]
  Endif
```

```
Compute-Opt(n)
  If n = 0 then
    Return 0
  Else
    v = M-Compute-Opt(p(n))
    v' = M-Compute-Opt(n - 1)
    If ℓ_n + v ≥ v' then
      Return ℓ_n + v
    Else
      Return v'
    Endif
  Endif
```

Clearly, this looks very similar to our previous implementation of the algorithm; however, memoization has brought the running time way down.

(5.3) *The running time of* `M-Compute-Opt(`n`)` *is* $O(n)$.

Proof. The time spent in a single call to `Compute-Opt` or `M-Compute-Opt` is $O(1)$, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued to either of these procedures. Since the implementation itself gives no explicit upper bound, we invoke the strategy of looking for a good measure of "progress."

The most useful progress measure here is the number of entries in M that are not "empty." Initially this number is 0; but each call to `Compute-Opt` increases the number by 1. Since M has only $n + 1$ entries, there can be at most $n + 1$ calls to `Compute-Opt`.

Now, each invocation of `M-Compute-Opt` is either the initial one, or it comes from `Compute-Opt`. But each call to `Compute-Opt` generates only two calls to `M-Compute-Opt`; and there are at most $n + 1$ calls to `Compute-Opt`. Thus the number of calls to `M-Compute-Opt` is at most $1 + 2(n + 1) = 2n + 3$.

It follows that the entire algorithm has running time $O(n)$. ■

Computing a solution, in addition to its value. So far we have simply computed the *value* of an optimal solution; presumably we want a full optimal set of requests as well. It is easy to extend `M-Compute-Opt` to do this: we change `Compute-Opt(`i`)` to keep track of the optimal solution in addition to its value. We would maintain an additional array S so that $S[i]$ contains an optimal set of intervals among $\{1, 2, \ldots, i\}$. Enhancing the code to maintain the solutions in the array S costs us an $O(n)$ blow-up in the running time: While a position in the M array can be updated in $O(1)$ time, writing down a set in the S array takes $O(n)$ time. We can avoid this $O(n)$ blow-up by not explicitly maintaining the S, but rather by recovering the optimal solution from values saved in the array M after the optimum value has been computed.

Using our observation from (5.2) , n belongs to some optimal solution for the set of requests $\{1, \ldots, i\}$ if and only if in the comparison "$\ell_n + v \geq v'$" the left-hand-side is at least as large as the right-hand-side. Using this observation for all values $i \leq n$ we get the following procedure, which "traces back" through the array M to find the set of intervals in an optimal solution.

```
Find-Solution(i)
  If i = 0 then
    Output nothing.
  Else
    v = M[p(i)]
```

```
    v' = M[i − 1]
    If ℓ_i + v ≥ v' then
      Output i and the result of Find-Solution(p(i)).
    Else
      Output the result of Find-Solution(i − 1)
    Endif
  Endif
```

(5.4) *Given the array M of the optimal values of the sub-problems as computed by* `M-Compute-Opt`*, the code* `Find-Solution` *returns the optimal set S in $O(n)$ time.*

Proof. The time spent in a single call to `Find-Solution` is $O(1)$, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued. A call to `Find-Solution`(i) issues at most one recursive call to a problem with smaller i value, so the number of calls ever issued by `Find-Solution`(n) is at most n. ■

Style #2: Building up Solutions to Sub-Problems

If we think for a little while, and unwind what the memoized version of `Compute-Opt` is doing, we see that it's really just building up entries in an array. One could argue that it's simpler just to do that with a `For` loop, as follows:

```
Iterative-Compute-Opt(n)
  Array M[0 ... n]
  M[0] = 0
  For i = 1, 2, ..., n
    If ℓ_i + M[p(i)] ≥ M[i − 1] then
      M[i] = ℓ_i + M[p(i)]
    Else
      M[i] = M[i − 1]
    Endif
  Endfor
  Return M[n]
```

Using (5.1) one can immediately prove by induction that the value $M[n]$ is an optimal solution for the set of requests $\{1, \ldots, n\}$. The running time can be bounded as follows. There are n iterations of the `For` loop, each iteration takes $O(1)$ time, and thus the overall time of the algorithm as implemented above is $O(n)$. Once the array M is computed we can run `Find-Solution`(n) to find the optimal set S.

Different people have different intuitions about dynamic programming algorithms; in particular, some people find it easier to invent such algorithms in the first style, others in

the second. Here we've followed the route of memoization, and only devised an iterative "building-up" algorithm once our intuition for the problem was firmly in place.

However, it is possible to develop "building-up" algorithms from scratch as well. Essentially, for such an algorithm, one needs a collection of sub-problems derived from the original problem that satisfy the following basic properties:

(i) There are only a polynomial number of sub-problems.

(ii) The solution to the original problem can be easily computed from the solutions to the sub-problems. (For example, the original problem may actually *be* one of the sub-problems.)

(iii) There is an easy-to-compute recurrence, as in (5.1) and (5.2), allowing one to determine the solution to a sub-problem from the solutions to some number of "smaller" sub-problems.

Naturally, these are informal guidelines; in particular, the notion of "smaller" in part (iii) will depend on the type of recurrence one has.

In future uses of dynamic programming, we will specify our algorithm more compactly by using the iterative approach, but will often use ideas from the recursive approach to design the collection of sub-problems we use.

Weighted Interval Scheduling

The general weighted interval scheduling problem consists of n requests, with each request i specified by an interval and a non-negative *value*, or *weight*, w_i. The goal is to accept a compatible set of requests of maximum total value.

We started by saying that we would consider the special case of the problem in which w_i is defined to be the length ℓ_i of interval i, for $i = 1, \ldots, n$. But if we go back over our solution to this problem, we see that we never actually used the fact that ℓ_i was the length of i, as opposed to a completely arbitrary non-negative value. Indeed, it is easy to check that facts (5.1) and (5.2) remain true if in place of the lengths $\{\ell_i\}$ we assume general non-negative values $\{w_i\}$, and thus our algorithms in fact solve the general weighted interval scheduling problem as well.

5.2 Segmented Least Squares: Multi-way Choices

We now discuss a different type of problem, which illustrates a slightly more complicated style of dynamic programming. In the previous section, we developed a recurrence based on a fundamentally *binary* choice: either the interval n belonged to an optimal solution or it didn't. In the problem we consider here, the recurrence will involve what might be called

"multi-way choices" — at each step, we have a polynomial number of possibilities to consider for the structure of the optimal solution. As we'll see, the dynamic programming approach adapts to this more general situation very naturally.

As a separate issue, the problem developed in this section is also a nice illustration of how a clean algorithmic definition can formalize a notion that initially seems too fuzzy and intuitive to work with mathematically.

Often when looking at scientific or statistical data, plotted on a two-dimensional set of axes, one tries to pass a "line of best fit" through the data as in Figure 5.1.

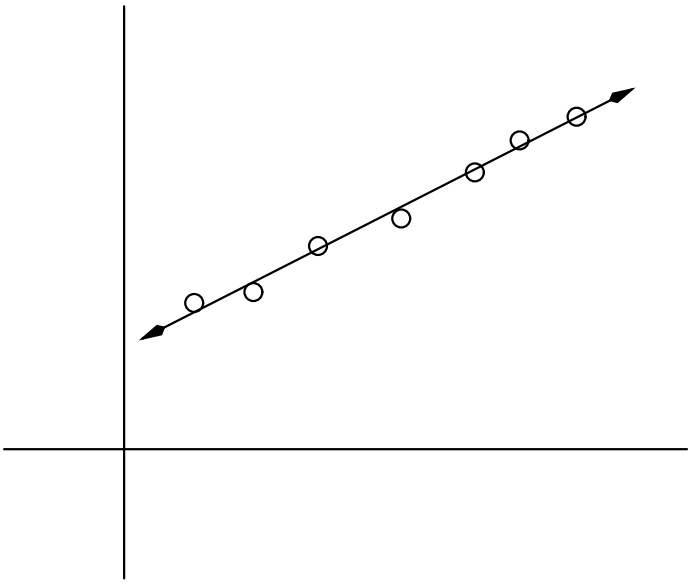


Figure 5.1: A "line of best fit."

This is a foundational problem in statistics and numerical analysis, formulated as follows. Suppose our data consists of a set P of n points in the plane, denoted $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$; and suppose $x_1 < x_2 < \cdots < x_n$. Given a line L defined by the equation $y = ax + b$, we say that the *error* of L with respect to P is the sum of its squared "distances" to the points in P:

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

A natural goal is then to find the line with minimum error; this turns out to have a nice closed-form solution that can be easily derived using calculus. Skipping the derivation here, we simply state the result: The line of minimum error is $y = ax + b$, where

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \qquad \text{and} \qquad b = \frac{\sum_i y_i - a \sum_i x_i}{n}.$$

Now, here's a kind of issue that these formulas weren't designed to cover. Often we have data that looks something like the picture in Figure 5.2. In this case, we'd like to make a statement like: "The points lie roughly on a sequence of two lines." How could we formalize this concept?

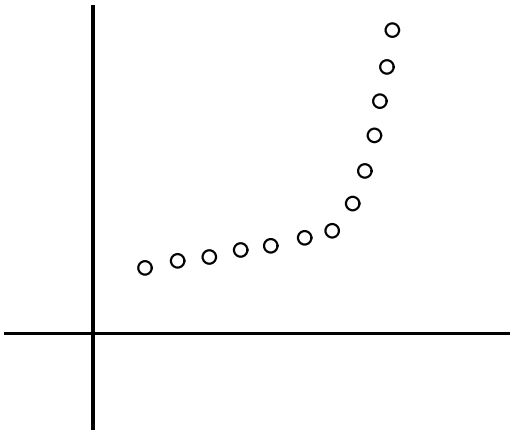


Figure 5.2: Two lines would be better.

Essentially, any single line through the points in the figure would have a terrible error; but if we use two lines, we could achieve quite a small error. So we could try formulating a new problem as follows: rather than seek a single line of best fit, we are allowed to pass an arbitrary *set* of lines through the points, and we seek a set of lines that minimizes the error. But this fails as a good problem formulation, because it has a trivial solution: if we're allowed to fit the points with an arbitrarily large set of lines, we could fit the points perfectly by having a different line pass through each pair of consecutive points in P.

At the other extreme, we could try "hard-coding" the number two into the problem: we could seek the best fit using at most two lines. But this too misses a crucial feature of our intuition: we didn't start out with a pre-conceived idea that the points lay approximately on two lines; we concluded that from looking at the picture. For example, most people would say that the points in Figure 5.3 lie approximately on three lines.

Thus, intuitively, we need a problem formulation that requires us to fit the points well, using as few lines as possible. We now formulate a problem — the *segmented least squares* problem – that captures these issues quite cleanly.

As before, we are given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, with $x_1 < x_2 < \cdots < x_n$. We will use p_i to denote the point (x_i, y_i). We must first partition P into some number of *segments*. Each *segment* is a subset of P that represents a contiguous set of x-coordinates; that is, it is a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i \leq j$. Then, for each segment S in our partition of P, we compute the line minimizing the error with respect to the points in S, according to the formulas above.

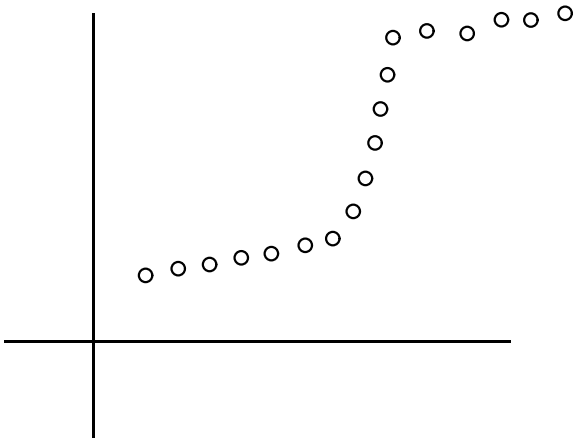


Figure 5.3: Three lines would be better.

The *penalty* of a partition is defined to be a sum of the following terms:

(i) The number of segments into which we partition P, times a fixed, given multiplier $C > 0$.

(ii) For each sequence, the error value of the optimal line through that segment.

Our goal in the *segmented least squares* problem is to find a partition of minimum penalty. This minimization captures the trade-offs we discussed above. We are allowed to consider partitions into any number of segments; as we increase the number of segments, we reduce the penalty terms in part (ii) of the definition, but we increase the term in part (i). (The multiplier C is provided with the input, and by tuning C, we can penalize the use of additional lines to a greater or lesser extent.)

There are exponentially many possible partitions of P, and initially it is not clear that we should be able to find the optimal one efficiently. We now show how to use dynamic programming to find a partition of minimum penalty in time polynomial in n. (We will make the reasonable assumption, as always, that each arithmetic operation we perform takes a constant amount of time.)

Designing the algorithm. To begin with, we should recall the ingredients we need in a dynamic programming algorithm from page 123. We want a polynomial number of "sub-problems," the solutions of which should yield a solution to the original problem; and we should be able to build up solutions to these sub-problems using a recurrence. As with the weighted interval scheduling problem, it helps to think about some simple properties of the optimal solution. Note, however, that there is not really a direct analogy to weighted interval scheduling: there we were looking for a *subset* of n objects, whereas here we are seeking to *partition* n objects.

For segmented least squares, the following observation is very useful: The last point p_n belongs to a single segment in the optimal partition, and that segment begins at some earlier point p_j. This is the type of observation that can suggest the right set of sub-problems: if we knew the identity of the *last* segment $p_j, \ldots, p_n$ (see Figure 5.4), then we could remove those points from consideration and recursively solve the problem on the remaining points $p_1, \ldots, p_{j-1}$.

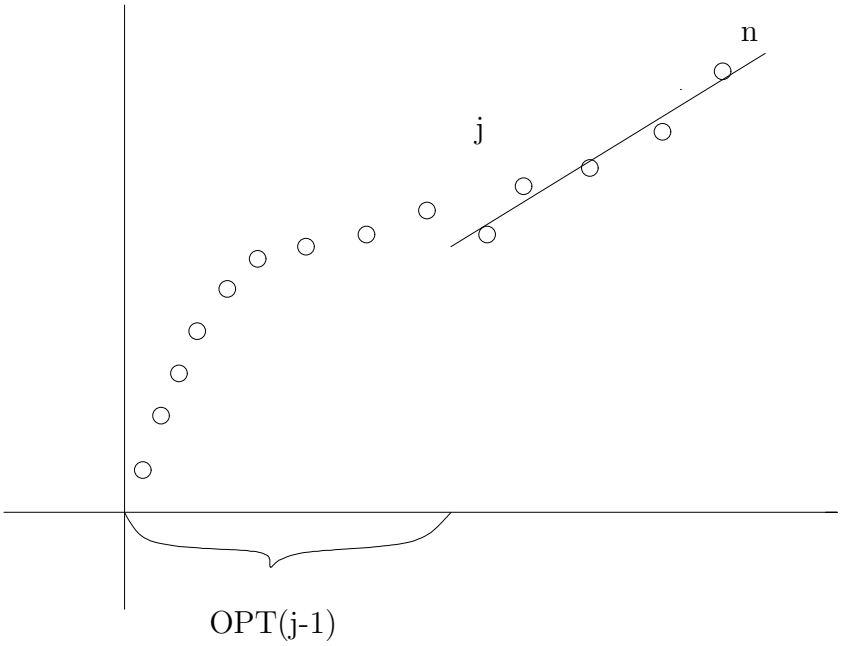


Figure 5.4: The last segment.

Suppose we let $OPT(i)$ denote the optimum solution for the points $p_1, \ldots, p_i$, and we let $e_{j,n}$ denote the minimum error of any line with respect to $p_j, p_{j+1}, \ldots, p_n$. (We will write $OPT(0) = 0$ as a boundary case.) Then our observation above says that we should find the best way to produce a final segment — paying the error plus an additive C for this segment — together with an optimal solution for the remaining points. In other words, we have justified the following recurrence.

(5.5) *If the last segment of the optimal partition is $p_i, \ldots, p_n$, then the value of the optimal solution is $OPT(n) = e_{i,n} + C + OPT(i-1)$. Therefore we have*

$$OPT(n) = \min_{1 \le i \le n} e_{i,n} + C + OPT(i-1),$$

and the segment $p_i, \ldots, p_n$ is used in an optimum solution if and only if the minimum is obtained using index i.

The hard part in designing the algorithm is now behind us. From here, we simply build up the solutions $OPT(i)$ in order of increasing i.

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For i = 1, ..., n
    For j = 1, ..., i
       Compute the least squares error e_{j,i} for the segment p_j, ..., p_i
    Endfor
    Use the recurrence (5.5) to compute M[i].
  Endfor
  Return M[n].
```

The correctness of the algorithm follows immediately from (5.5).

As in our algorithm for weighted interval scheduling, we can trace back through the array M to compute an optimum partition.

```
Find-Segments(i)
  If i = 0 then
    Output nothing
  Else
    Find a j that minimizes e_{j,i} + C + M[j − 1]
    Output the segment {p_j, ..., p_i} and the result of
           Find-Segments(j − 1)
  Endif
```

Finally, we consider the running time of `Segmented-Least-Squares`. The algorithm has n iterations, for values $i = 1, \ldots, n$. For each value of i we have to consider i options for the value of j, compute the least squares error $e_{j,i}$ for all i of them, and choose the minimum. We use the formula given above to compute the errors $e_{j,i}$, spending $O(n)$ on each; thus, the overall time to compute the array entry $M[i]$ is $O(n^2)$. As there are $n + 1$ array positions, the total time is $O(n^3)$.[1]

5.3 Subset Sums and Knapsacks: Adding a Variable

We're seeing more and more that issues in scheduling provide a rich source of practically motivated algorithmic problems. So far we've considered problems in which requests are

[1] One can actually design essentially the same dynamic programming algorithm to run in total time $O(n^2)$. The bottleneck in the current version is in computing $e_{j,i}$ for all pairs (j, i); by cleverly saving intermediate results, these can all be computed in total time $O(n^2)$. This immediately reduces the time for each array entry $M[i]$ to $O(n)$, and hence the overall time to $O(n^2)$. We won't go into the details of this faster version, but it is an interesting exercise to consider how one might compute all values $e_{j,i}$ in $O(n^2)$ time.

specified by a given interval of time on a resource, as well as problems in which requests have a duration and a deadline, but do not mandate a particular interval during which they need to be done.

In this section we consider a version of the second type of problem, with durations and deadlines, which is difficult to solve directly using the techniques we've seen so far. We will use dynamic programming to solve the problem, but with a twist — the "obvious" set of sub-problems will turn out not be enough, and so we end up creating a richer collection of sub-problems. As we will see below, this is done by adding a new variable to the recurrence underlying the dynamic program.

In our problem formulation, we have a single machine that can process jobs, and we have a set of requests $\{1, 2, \ldots, n\}$. We are only able to use this resource for the period between time 0 and time W, for some number W. Each request corresponds to a job that requires time w_i to process. If our goal is to process jobs so as to keep the machine as busy as possible up to the "cut-off" W, which jobs should we choose?

More formally, we are given n items $\{1, \ldots, n\}$, and each has a given nonnegative weight w_i (for $i = 1, \ldots, n$). We are also given a bound W. We would like to select a subset S of the items so that $\sum_{i \in S} w_i \leq W$ and, subject to this restriction, $\sum_{i \in S} w_i$ is as large as possible.

This problem is a natural special case of a more general problem called the *knapsack problem*, where each request i has both a *value* v_i and a *weight* w_i. The goal in this more general problem is to select a subset of maximum total value, subject to the restriction that its total weight not exceed W. Knapsack problems often show up as sub-problems in other, more complex problems. The name "knapsack" refers to the problem of filling a knapsack of capacity W as full as possible (or packing in as much value as possible), using a subset of the items $\{1, \ldots, n\}$. We will use *weight* or *time* when referring to the quantities w_i and W.

Since this sort of resembles other scheduling problems we've seen before, it's natural to ask whether a greedy algorithm can find the optimal solution. It appears that the answer is no — in any case, no efficient greedy rule is known that always constructs an optimal solution. One natural greedy approach to try would be to sort the items by decreasing weight — or at least to do this for all items of weight at most W — and then start selecting items in this order as long as the total weight remains below W. But if W is a multiple of 2, and we have three items with weights $\{W/2 + 1, W/2, W/2\}$, then we see that this greedy algorithm will not produce the optimal solution. Alternately, we could sort by *increasing* weight and then do the same thing; but this fails on inputs like $\{1, W/2, W/2\}$.

The goal of this lecture is to show how to use dynamic programming to solve this problem. Recall the main principles of dynamic programming: we have to come up with a polynomial number of sub-problems, so that each sub-problem can be solved easily from "smaller" sub-problems, and the solution to the original problem can be obtained easily once we know the solutions to all the sub-problems. As usual, the hard part in designing a dynamic

programming algorithm lies in figuring out a good set of sub-problems.

A False Start. One general strategy, which worked for us in the case of weighted interval scheduling, is to consider sub-problems involving only the first i requests. We start by trying this strategy here. We use the notation $OPT(i)$, analogously to the notation used before, to denote the best possible solution using a subset of the requests $\{1, \ldots, i\}$. The key to our method for the weighted interval scheduling problem was to concentrate on an optimal solution $\mathcal{O}$ to our problem and consider two cases, depending whether or not the last request n is accepted or rejected by this optimum solution. Just as last time, we have the first part, which follows immediately from the definition of $OPT(i)$.

- If $n \notin \mathcal{O}$ then $OPT(n) = OPT(n-1)$.

Next we have to consider the case in which $n \in \mathcal{O}$. What we'd like here is a simple recursion, which tells us the best possible value we can get for solutions that contain the last request n. For weighted interval scheduling this was easy, as we could simply delete each request that conflicted with request n. In the current problem, this is not so simple. Accepting request n does not immediately imply that we have to reject any other request. Instead, it means that for the subset of requests $S \subseteq \{1, \ldots, n-1\}$ that we will accept, we have less available weight left: a weight of w_n is used on the accepted request n, and we only have $W - w_n$ weight left for the set S of remaining requests that we accept. See Figure 5.5.

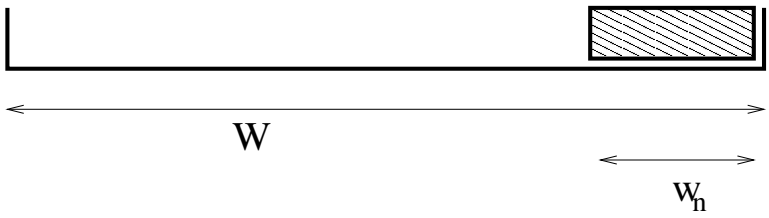


Figure 5.5: A knapsack of size W with the item w_n included.

A Better Solution. This suggests that we need more sub-problems: To find out the value for $OPT(n)$ we not only need the value of $OPT(n-1)$, but we also need to know the best solution we can get using a subset of the first $n-1$ items, and total allowed weight $W - w_n$. We are therefore going to use many more sub-problems: one for each initial set $\{1, \ldots, i\}$ of the items, and each possible value for the remaining available weight w. Assume that W is an integer, and all requests $i = 1, \ldots, n$ have integer weights w_i. We will have a sub-problem for each $i = 0, 1, \ldots, n$ and each integer $0 \leq w \leq W$. We will use $OPT(i, w)$ to denote the

value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w, i.e.,

$$OPT(i, w) = \max_S \sum_{j \in S} w_j,$$

where the maximum is over subsets $S \subseteq \{1, \ldots, i\}$ that satisfy $\sum_{j \in S} w_j \leq w$. Using this new set of sub-problems, we will be able to express the value $OPT(i, w)$ as a simple expression in terms of values from smaller problems. Moreover, $OPT(n, W)$ is the quantity we're looking for in the end. As before, let $\mathcal{O}$ denote an optimum solution for the original problem.

- If $n \notin \mathcal{O}$ then $OPT(n, W) = OPT(n-1, W)$
- If $n \in \mathcal{O}$ then $OPT(n, W) = w_n + OPT(n-1, W - w_n)$.

When the n^{th} item is too big, i.e., $W < w_n$, then we must have $OPT(n, W) = OPT(n-1, W)$. Otherwise, we get the optimum solution allowing all n requests by taking the better of these two options. This gives us the recursion:

(5.6) *If $W < w_n$ then $OPT(n, W) = OPT(n-1, W)$. Otherwise*

$$OPT(n, W) = \max(OPT(n-1, W), w_n + OPT(n-1, W - w_n)).$$

As before, we want to design an algorithm that builds up a table of all $OPT(i, w)$ values while computing each of them at most once.

```
Subset-Sum(n, W)
  Array M[0...n, 0...W]
  For w = 0, ..., W
     M[0, w] = 0
  For i = 1, 2, ..., n
     For w = 0, ..., W
        Use the recurrence (5.6) to compute M[i, w]
      Endfor
  Endfor
  Return M[n, W]
```

There is an appealing pictorial way in which one can think about the computation of the algorithm. To compute the value $M[i, w]$ we used two other values $M[i-1, w]$ and $M[i, w - w_i]$, as depicted by Figure 5.6.

Using (5.6) one can immediately prove by induction that the returned value $M[n, W]$ is the optimum solution value for the requests $1, \ldots, n$ and available weight W.

Next we will worry about the running time of this algorithm. As before in the case of the weighted interval scheduling, we are building up a table of solutions M, and we compute each of the values $M[i, w]$ in $O(1)$ time using the previous values. Thus the running time is proportional to the number of entries in the table.

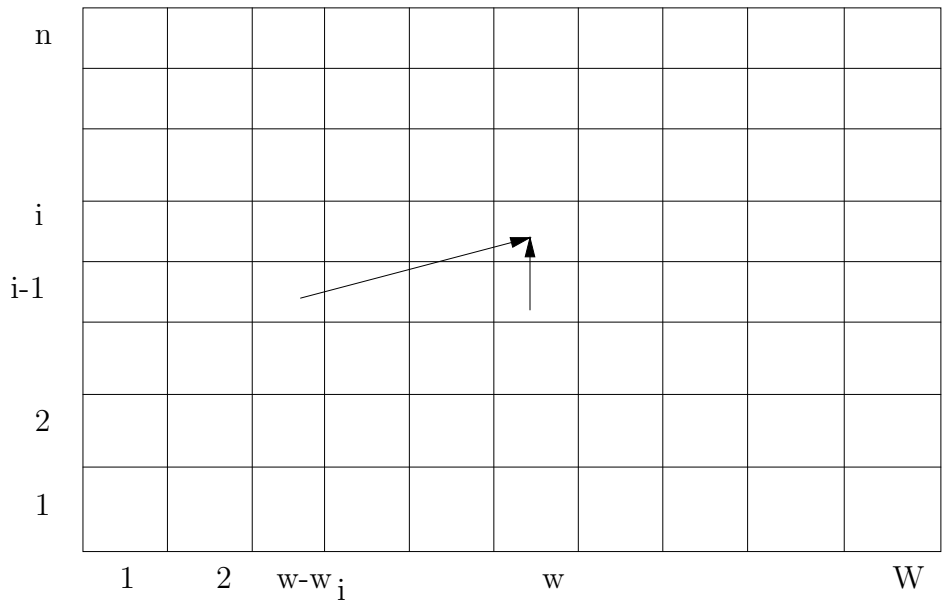


Figure 5.6: How the table of $OPT(i, w)$ values is computed.

(5.7) *The* `Subset-Sum`(n, W) *algorithm correctly computes the optimal value of the problem, and runs in* $O(nW)$ *time.*

Note that this method is not as efficient as our dynamic program for the weighted interval scheduling problem. Indeed, its running time is not a polynomial function of n; rather, it is a polynomial function of n and W, the largest integer involved in defining the problem. We call such algorithms *pseudo polynomial*. Pseudo polynomial algorithms can be reasonably efficient when the numbers $\{w_i\}$ involved in the input are reasonably small; however, they become less practical as these numbers grow large.

To recover an optimal set S of items, we can trace back through the array M by a procedure similar to those we developed in the previous sections.

(5.8) *Given a table* M *of the optimal values of the sub-problems, the optimal set* S *can be found in* $O(n)$ *time.*

The Knapsack Problem

The knapsack problem is a bit more complex then the scheduling problem we discussed above. Consider a situation in which each item i has a nonnegative weight w_i as before, and also a distinct *value* v_i. Our goal is now to find a subset S of maximum value $\sum_{i \in S} v_i$ subject to the restriction that the total weight of the set should not exceed W: $\sum_{i \in S} w_i \leq W$.

It is not hard to extend our dynamic programming algorithm to this more general problem. We use the analogous set of sub-problems, $OPT(i, w)$ to denote the value of the optimal

solution using a subset of the items $\{1, \ldots, i\}$ and maximum available weight w. We consider an optimal solution $\mathcal{O}$, and identify two cases depending on whether or not $n \in \mathcal{O}$.

- If $n \notin \mathcal{O}$ then $OPT(n, W) = OPT(n-1, W)$.
- If $n \in \mathcal{O}$ then $OPT(n, W) = v_n + OPT(n-1, W - w_n)$.

This implies the following analogue of (5.6).

(5.9) *If $W < w_n$ then $OPT(n, W) = OPT(n-1, W)$. Otherwise*

$$OPT(n, W) = \max(OPT(n-1, W), v_n + OPT(n-1, W - w_n)).$$

Using this recursion we can write down an analogous dynamic programming algorithm.

(5.10) `Knapsack`*(n, W) takes $O(nW)$ time, and correctly computes the optimal values of the sub-problems.*

As was done before we can trace back through the table M containing the optimal values of the sub-problems, to find an optimal solution in $O(n)$ time.

5.4 RNA Secondary Structure: Dynamic Programming Over Intervals

In the Knapsack problem, we were able to formulate a dynamic programming algorithm by adding a new variable. A different but very common way by which one ends up adding a variable to a dynamic program is through the following scenario. We start by thinking about the set of sub-problems on $\{1, 2, \ldots, j\}$, for all choices of j, and find ourselves unable to come up with a natural recurrence. We then look at the larger set of sub-problems on $\{i, i+1, \ldots, j\}$ for all choices of i *and* j (where $i \leq j$), and find a natural recurrence relation on these sub-problems. In this way, we have added the second variable i; the effect is to consider a sub-problem for every contiguous *interval* in $\{1, 2, \ldots, n\}$.

There are a few canonical problems that fit this profile; those of you who have studied parsing algorithms for context-free grammars have probably seen at least one dynamic programming algorithm in this style. Here we focus on the problem of RNA secondary structure prediction, a fundamental issue in computational biology.

As one learns in introductory biology classes, Watson and Crick posited that double-stranded DNA is "zipped" together by complementary base-pairing. Each strand of DNA can be viewed as a string of *bases*, where each base is drawn from the set $\{A, C, G, T\}$. The bases A and T pair with each other, and the bases C and G pair with each; it is these A-T and C-G pairings that hold the two strands together.

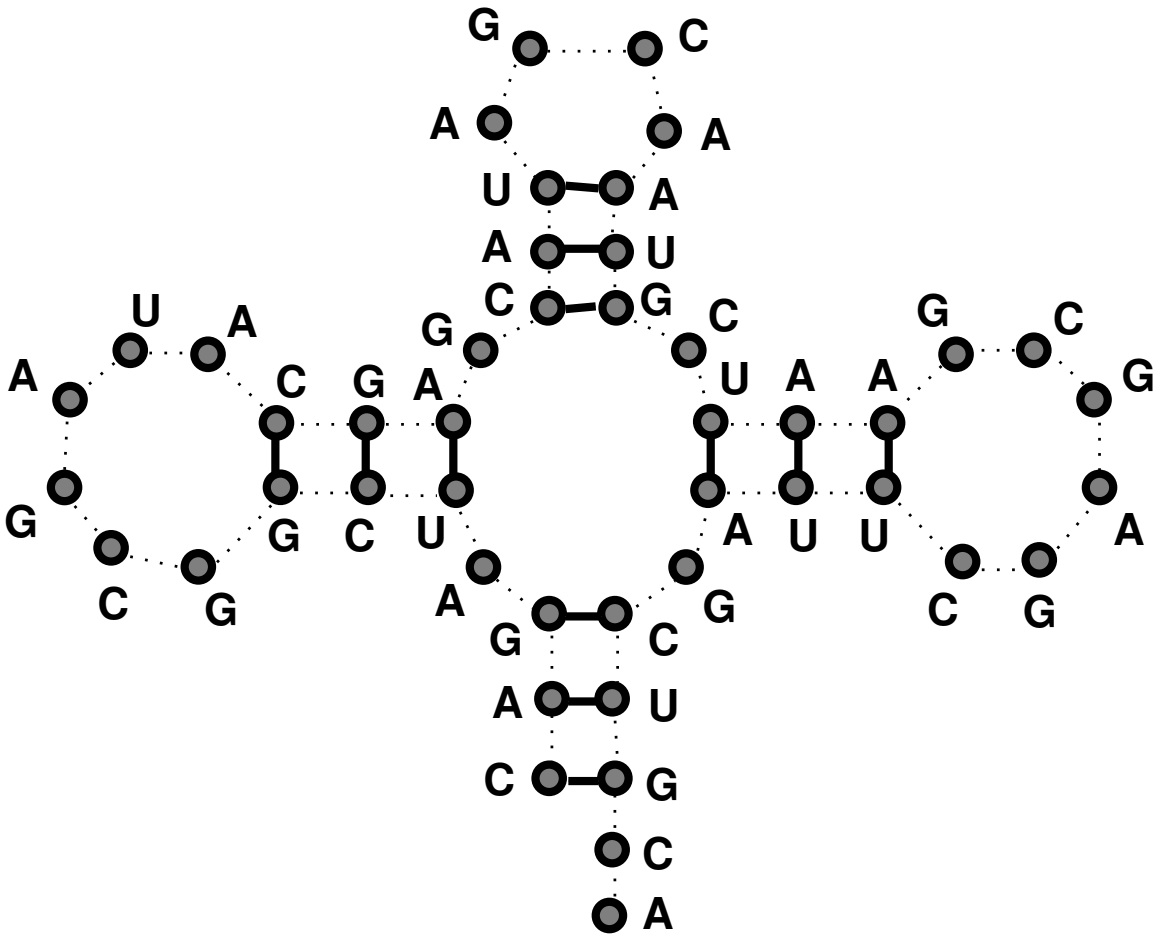


Figure 5.7: An RNA secondary structure. Dotted connections indicate adjacent elements of the sequence; solid connections indicate pairs that are matched.

Now, single-stranded RNA molecules are key components in many of the processes that go on inside a cell, and they follow more or less the same structural principles. However, unlike double-stranded DNA, there's no "second strand" for the RNA to stick to; so it tends to loop back and form base pairs with itself, resulting in interesting shapes like the one depicted in Figure 5.7. The set of pairs (and resulting shape) formed by the RNA molecule through this process is called the *secondary structure*, and understanding the secondary structure is essential for understanding the behavior of the molecule.

For our purposes, a single-stranded RNA molecule can be viewed as a sequence of n symbols (bases) drawn from the alphabet $\{A, C, G, U\}$.[2] Let $B = b_1 b_2 \cdots b_n$ be a single-stranded RNA molecule, where each $b_i \in \{A, C, G, U\}$. To a first approximation, one can model its secondary structure as follows. As usual, we require that A pairs with U, and

[2]Note that the symbol T from the alphabet of DNA has been replaced by a U, but this is not important for us here.

C pairs with G; we also require that each base can pair with at most one other base — in other words, the set of base pairs forms a *matching*. It also turns out that secondary structures are (again, to a first approximation) "knot-free," which we will formalize as a kind of "non-crossing" condition below.

Thus, concretely, we say that a *secondary structure on* B is a set of pairs $S = \{(b_i, b_j)\}$ that satisfies the following conditions:

(i) *(No sharp turns.)* The ends of each pair in S are separated by at least four intervening bases; that is, if $(b_i, b_j) \in S$, then $i < j - 4$.

(ii) The elements of any pair in S consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).

(iii) S is a matching: no base appears in more than one pair.

(iv) *(The non-crossing condition.)* If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

Note that the RNA secondary structure in Figure 5.7 satisfies properties (i) through (iv). From a structural point of view, condition (i) arises simply because the RNA molecule cannot bend too sharply; and conditions (ii) and (iii) are the fundamental Watson-Crick rules of base-pairing. Condition (iv) is the striking one, since it's not obvious why it should hold in nature. But while there are sporadic exceptions to it in real molecules (via so-called "pseudo-knotting"), it does turn out to be a very good approximation to the spatial constraints on real RNA secondary structures.

Now, out of all the secondary structures that are possible for a single RNA molecule, which are the ones that are likely to arise under physiological conditions? The usual hypothesis is that a single-stranded RNA molecule will form the secondary structure with the optimum total free energy. The correct model for the free energy of a secondary structure is a subject of much debate; but a first approximation here is to assume that the free energy of a secondary structure is proportional simply to the *number* of base pairs that it contains.

Thus, having said all this, we can state the basic RNA secondary structure prediction problem very simply. We want an efficient algorithm that takes a single stranded RNA molecule $B = b_1 b_2 \cdots b_n$ and determines a secondary structure S with the maximum possible number of base pairs.

A first attempt at dynamic programming. The natural first attempt to apply dynamic programming would presumably be based on the following sub-problems: we say that $OPT(j)$ is the maximum number of base pairs in a secondary structure on $b_1 b_2 \cdots b_j$. By the no-sharp-turns condition above, we know that $OPT(j) = 0$ for $j \leq 5$; and we know that $OPT(n)$ is the solution we're looking for.

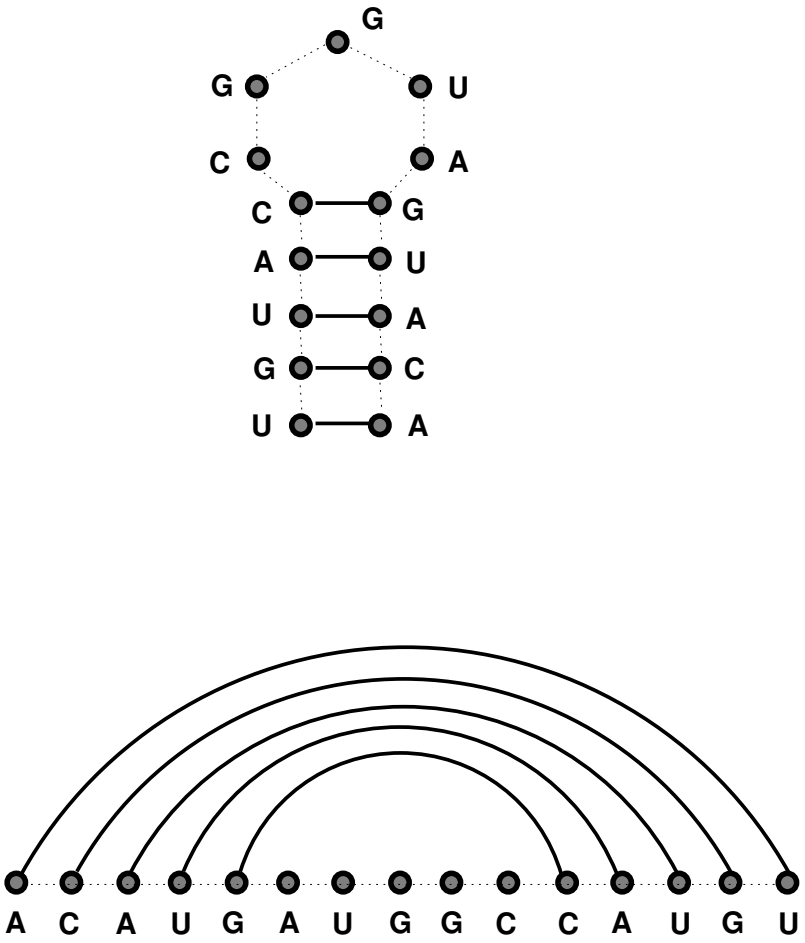


Figure 5.8: Two views of an RNA secondary structure. In the second view, the string has been "stretched" lengthwise, and edges connecting matched pairs appear as non-crossing "bubbles" over the string.

The trouble comes when we try writing down a recurrence that expresses $OPT(j)$ in terms of the solutions to smaller sub-problems. We can get partway there: in the optimal secondary structure on $b_1 b_2 \cdots b_j$, it's the case that either

- b_j is not involved in a pair; or
- b_j pairs with b_t for some $t < j - 4$.

In the first case, we just need to consult our solution for $OPT(j-1)$. The second case is depicted in Figure 5.9(a); because of the non-crossing condition, we now know that no pair can have one end between 1 and $t-1$ and the other end between $t+1$ and $j-1$. We've therefore effectively isolated two new sub-problems: one on the bases $b_1 b_2 \cdots b_{t-1}$, and the other on the bases $b_{t+1} \cdots b_{j-1}$. The first is solved by $OPT(t-1)$, but the second is not on our list of sub-problems — *because it does not begin with* b_1.

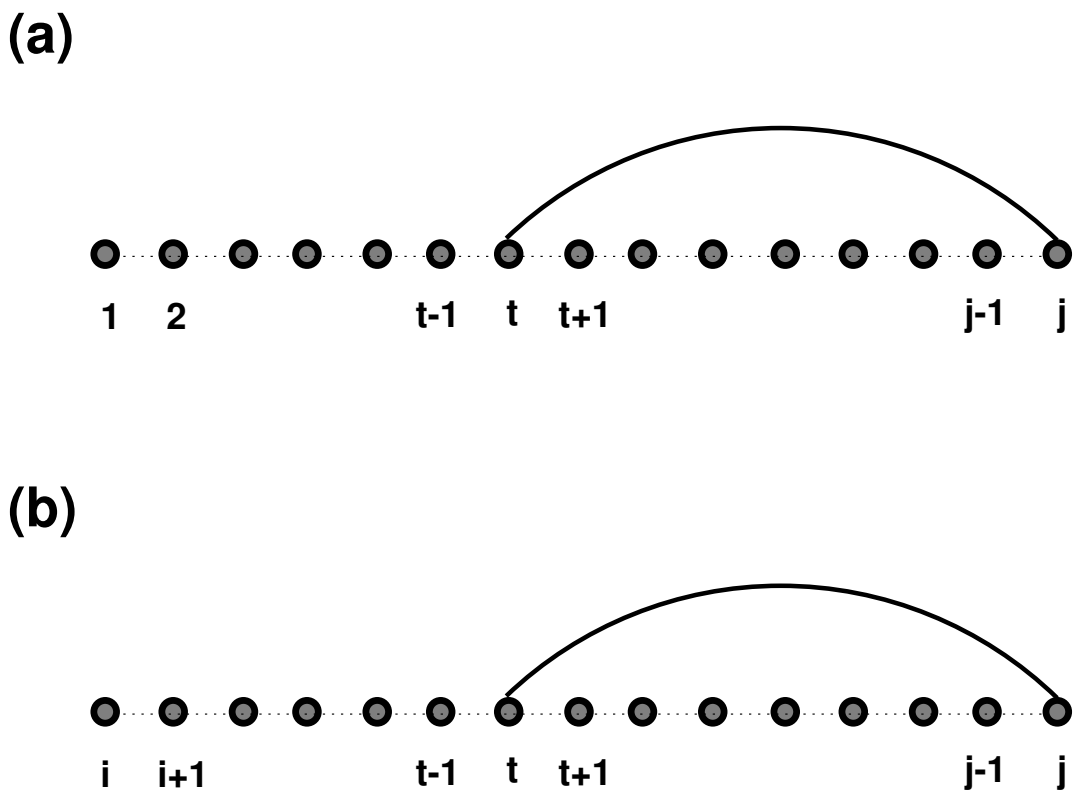


Figure 5.9: Schematic view of the dynamic programming recurrence using (a) one variable, and (b) two variables.

This is the insight that makes us realize we need to add a variable. We need to be able to work with sub-problems that do not begin with b_1; in other words, we need to consider sub-problems on $b_i b_{i+1} \cdots b_j$ for all choices of $i \leq j$.

Dynamic programming over intervals. Once we make this decision, our previous reasoning leads straight to a successful recurrence. Let $OPT(i, j)$ denote the maximum number of base pairs in a secondary structure on $b_i b_{i+1} \cdots b_j$. The no-sharp-turns condition lets us initialize $OPT(i, j) = 0$ whenever $i \geq j + 4$.

Now, in the optimal secondary structure on $b_i b_{i+1} \cdots b_j$, we have the same alternatives as before:

- b_j is not involved in a pair; or
- b_j pairs with b_t for some $t < j - 4$.

In the first case, we have $OPT(i, j) = OPT(i, j - 1)$. In the second case, depicted in Figure 5.9(b); we recur on the two sub-problems $OPT(i, t - 1)$ and $OPT(t + 1, j - 1)$; as

argued above, the non-crossing condition has isolated these two sub-problems from each other.

We have therefore justified the following recurrence.

(5.11) $OPT(i, j) = \max(OPT(i, j-1), \max(1 + OPT(i, t-1) + OPT(t+1, j-1)))$, *where the* max *is taken over* t *such that* b_t *and* b_j *are an allowable base pair (under the Watson-Crick condition (ii)).*

Now we just have to make sure we understand the proper order in which to build up the solutions to the sub-problems. The form of (5.11) reveals that we're always invoking the solution to sub-problems on *shorter* intervals: those for which $j - i$ is smaller. Thus, things will work without any trouble if we build up the solutions in order of increasing interval length.

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For i = 1, 2, ... n
  For j = i + 5, i + 6, ..., n
    Compute OPT(i, j) using the recurrence in  (5.11)
  Endfor
Endfor
Return OPT(1, n)
```

As always, we can recover the secondary structure itself (not just its value) by recording how the minima in (5.11) are achieved, and tracing back through the computation.

It is easy to bound the running time: there are $O(n^2)$ sub-problems to solve, and evaluating the recurrence in (5.11) takes time $O(n)$ for each. Thus, the running time is $O(n^3)$.

5.5 Sequence Alignment

Dictionaries on the Web seem to get more and more useful: often it now seems easier to pull up a book-marked on-line dictionary than to get a physical dictionary down from the bookshelf. And many on-line dictionaries offer functions that you can't get from a printed one: if you're looking for a definition and type in a word it doesn't contain — say, "ocurrance" — it will come back and ask, "Perhaps you mean 'occurrence?' ". How does it do this? Did it truly know what you had in mind?

Let's defer the second question to a different course, and think a little about the first one. To decide what you probably meant, it would be natural to search the dictionary for the word most "similar" to the one you typed in. To do this, we have to answer the question: how should we define similarity between two words or strings?

Intuitively we'd like to say that "ocurrance" and "occurrence" are similar because we can make the two words identical if we add a 'c' to the first word, and change the 'a' to an 'e'.

Since neither of these changes seems so large, we conclude that the words are quite similar. To put it another way, we can *nearly* line up the two words letter by letter:

```
o-currance
occurrence
```

The "-" symbol indicates a *gap* — we had to add a gap to the second word to get it to line up with the first. Moreover, our lining up is not perfect in that an "e" is lined up with an "a".

We want a model in which similarity is determined roughly by the number of gaps and mismatches we incur when we line up the two words. Of course, there are many possible ways to line up the two words; for example, we could have written

```
o-curr-ance
occurre-nce
```

which involves three gaps and no mismatches. Which is better: one gap and one mismatch, or three gaps and no mismatches?

This discussion has been made easier because we know roughly what the correspondence "ought" to look like. When the two strings don't look like English words — for example, "abbbaabbbbaab" and "ababaaabbbbbab" — it may take a little work to decide whether they can be lined up nicely or not:

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Dictionary interfaces and spell-checkers are not the most computationally intensive application for this type of problem. In fact, determining similarities among strings is one of the central computational problems facing molecular biologists today.

Strings arise very naturally in biology: an organism's *genome* — its full set of genetic material — is divided up into giant linear DNA molecules known as chromosomes, each of which serves conceptually as a one-dimensional chemical storage device. Indeed, it does not obscure reality very much to think of it as an enormous linear *tape*, containing a string over the alphabet $\{A, C, G, T\}$.[3] The string of symbols encodes the instructions for building protein molecules; using a chemical mechanism for reading portions of the chromosome, a cell can construct proteins that in turn control its metabolism.

Why is similarity important in this picture? We know that the sequence of symbols in an organism's genome directly determines everything about the organism. So suppose we have two strains of bacteria, X and Y, which are closely related evolutionarily. Suppose further that we've determined that a certain substring in the DNA of X codes for a certain kind of

[3]Adenine, cytosine, guanine, and thymine, the four basic units of DNA.

toxin. Then if we discover a very "similar" substring in the DNA of Y, we might be able to hypothesize, before performing any experiments at all, that this portion of the DNA in Y codes for a similar kind of toxin. This use of computation to guide decisions about biological experiments is one of the hallmarks of the field of *computational biology*.

All this leaves us with the same question we asked initially, typing badly spelled words into our on-line dictionary. How should we define the notion of *similarity* between two strings?

In the early 1970's, the two molecular biologists Needleman and Wunsch proposed a definition of similarity which, basically unchanged, has become the standard definition in use today. Its position as a standard was reinforced by its simplicity and intuitive appeal, as well as through its independent discovery by several other researchers around the same time. Moreover, this definition of similarity came with an efficient dynamic programming algorithm to compute it. In this way, the paradigm of dynamic programming was independently discovered by biologists some twenty years after Bellman first articulated it.

The definition is motivated by the considerations we discussed above, and in particular the notion of "lining up" two strings. Suppose we are given two strings X and Y: X consists of the sequence of symbols $x_1x_2 \cdots x_m$ and Y consists of the sequence of symbols $y_1y_2 \cdots y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings X and Y, and consider a matching of these sets; recall that a *matching* is a set of ordered pairs with the property that each item occurs in at most one pair. We say that matching M of these two sets is an *alignment* if there are no "crossing" pairs: if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. Intuitively an alignment gives a way of "lining up" the two strings, by telling us which pairs of positions will be lined up with one another. Thus, for example,

```
stop-
-tops
```

corresponds to the alignment $\{(2, 1), (3, 2), (4, 3)\}$.

Our definition of similarity will be based on finding the *optimal* alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y.

- First, there is a parameter $\delta > 0$ that defines a "gap penalty." For each position that is not matched in M — it is a "gap" — we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a "mismatch cost" of α_{pq} for lining up p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $\alpha_{x_iy_j}$ for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p — there is no mismatch cost to line up a letter with another copy of itself — though this will not be necessary in anything that follows.

- The *cost* of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

The process of minimizing this cost is often referred to as *sequence alignment* in the biology literature. The quantities δ and $\{\alpha_{pq}\}$ are external parameters that must be plugged into software for sequence alignment; indeed, a lot of work goes into choosing the settings for these parameters. From our point of view, in designing an algorithm for sequence alignment, we will take them as given. To go back to our first example, notice how these parameters determine which alignment of "ocurrance" and "occurrence" we should prefer: the first is strictly better if and only if $\delta + \alpha_{ae} < 3\delta$.

Computing an Optimal Alignment. We now have a concrete numerical definition for the similarity between strings X and Y: it is the minimum cost of an alignment between X and Y. Let's denote this cost by $\sigma(X, Y)$. The lower this cost is, the more similar we declare the strings to be. We now turn to the problem of computing $\sigma(X, Y)$, and the optimal alignment that yields it, for a given pair of strings X and Y.

One of the approaches we could try for this problem is dynamic programming, and we are motivated by the following basic dichotomy:

- In the optimal alignment M, either $(m, n) \in M$ or $(m, n) \notin M$. (That is, either the last symbols in the two strings are matched to each other, or they aren't.)

By itself, this fact would be too weak to provide us with a dynamic programming solution. Suppose, however, that we compound it with the following basic fact.

(5.12) *Let M be any alignment of X and Y. If $(m, n) \notin M$, then either the m^{th} position of X or the n^{th} position of Y is not matched in M.*

Proof. Suppose by way of contradiction that $(m, n) \notin M$, but there are numbers $i < m$ and $j < n$ so that $(m, j) \in M$ and $(i, n) \in M$. But this contradicts our definition of *alignment*: we have $(i, n), (m, j) \in M$ with $i < m$ but $n > i$ so the pairs (i, n) and (m, j) cross. ■

Given (5.12), we can turn our original dichotomy into the following, slightly less trivial, set of alternatives.

(5.13) *In an optimal alignment M, at least one of the following is true:*

(i) $(m, n) \in M$*; or*

(ii) the m^{th} position of X is not matched; or

(iii) the n^{th} position of Y is not matched.

Now, let $OPT(i, j)$ denote the minimum cost of an alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$. If case (i) of (5.13) holds, we pay $\alpha_{x_m y_n}$ and then align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_{n-1}$; we get $OPT(m, n) = \alpha_{x_m y_n} + OPT(m-1, n-1)$. If case (ii) holds, we pay a gap cost of δ since the m^{th} position of X is not matched, and then we align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_n$. In this way, we get $OPT(m, n) = \delta + OPT(m-1, n)$. Similarly, if case (iii) holds, we get $OPT(m, n) = \delta + OPT(m, n-1)$.

Thus we get the following fact.

(5.14) *The minimum alignment costs satisfy the following recurrence:*

$$OPT(m, n) = \min[\alpha_{x_m y_n} + OPT(m-1, n-1), \delta + OPT(m-1, n), \delta + OPT(m, n-1)].$$

Moreover, (m, n) is in an optimal alignment M if and only if the minimum is achieved by the first of these values.

We have maneuvered ourselves into a position where the dynamic programming algorithm has become clear: we build up the values of $OPT(i, j)$ using the recurrence in (5.14). There are only $O(mn)$ sub-problems, and $OPT(m, n)$ is the value we are seeking.

We now specify the algorithm to compute the value of the optimal alignment. For purposes of initialization, we note that that $OPT(i, 0) = OPT(0, i) = i\delta$ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

```
Alignment(X,Y)
  Array A[0...n, 0...m]
  For i = 0, ..., m
     A[i, 0] = iδ
  Endfor
  For j = 0, ..., n
     A[0, j] = jδ
  Endfor
  For j = 1, ..., n
     For i = 1, ..., m
        Use the recurrence (5.14) to compute A[i, j]
     Endfor
  Endfor
  Return A[m, n]
```

As in previous dynamic programming algorithms, we can "trace back" through the array A, using the second part of fact (5.14), to construct the alignment itself. The correctness of the algorithm follow directly from (5.14). Their running time is $O(mn)$, since the array A has $O(mn)$ entries, and at worst we spend constant time on each.

There is an appealing pictorial way in which people think about this sequence alignment algorithm. Suppose we build a two-dimensional $m \times n$ grid graph G_{XY}, with the rows labeled

by prefixes of the string X, the columns labeled by prefixes of Y, and directed edges as in Figure 5.10.

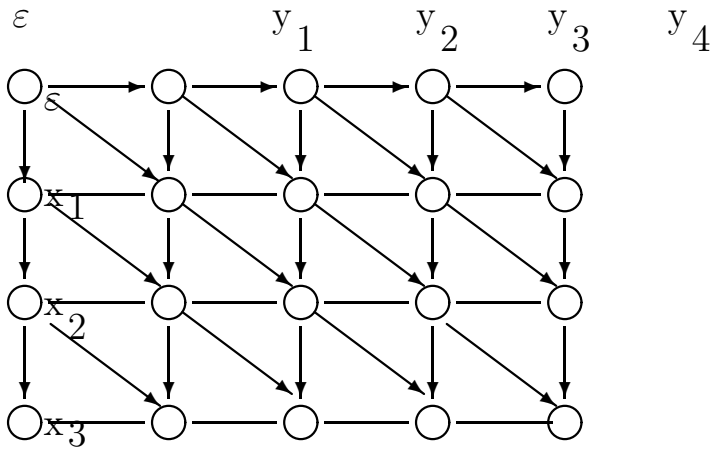


Figure 5.10: A graph-based picture of sequence alignment.

We number the rows from 0 to m and the columns from 0 to n; we denote the node in the i^{th} row and the j^{th} column by the label (i, j). We put *costs* on the edges of G_{XY}: the cost of each horizontal and vertical edge is δ, and the cost of the diagonal edge from $(i-1, j-1)$ to (i, j) is $\alpha_{x_i y_j}$.

The purpose of this picture now emerges: the recurrence in (5.14) for $OPT(i, j)$ is precisely the recurrence one gets for the minimum-cost path in G_{XY} from $(0, 0)$ to (i, j). Thus we can show

(5.15) *Let $f(i, j)$ denote the minimum cost of a path from $(0, 0)$ to (i, j) in G_{XY}. Then all for i, j, we have $f(i, j) = OPT(i, j)$.*

Proof. We can easily prove this by induction on $i + j$. When $i + j = 0$, we have $i = j = 0$, and indeed $f(i, j) = OPT(i, j) = 0$.

Now consider arbitrary values of i and j, and suppose the statement is true for all pairs (i', j') with $i'+j' < i+j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have

$$\begin{aligned} f(i,j) &= \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] \\ &= \min[\alpha_{x_i y_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)] \\ &= OPT(i, j), \end{aligned}$$

where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using (5.14). ■

Thus, the value of the optimal alignment is the length of the shortest path in G_{XY} from $(0, 0)$ to (m, n). (We'll call any path in G_{XY} from $(0, 0)$ to (m, n) a *corner-to-corner path*.) Moreover, the diagonal edges used in a shortest path correspond precisely to the pairs used in a minimum-cost alignment. These connections to the shortest path problem in the graph G_{XY} do not directly yield an improvement in the running time for the sequence alignment problem; however, they do help one's intuition for the problem, and have been useful in suggesting algorithms for more complex variations on sequence alignment.

5.6 Sequence Alignment in Linear Space

We have discussed the running time requirements of the sequence alignment algorithm, but — in keeping with our main focus in dynamic programming — we have not explicitly tabulated the space requirements. This is not difficult to do: we need only maintain the array A holding the values of $OPT(\cdot, \cdot)$, and hence the space required is $O(mn)$.

The real question is: should we be happy with $O(mn)$ as a space bound? If our application is to compare English words, or even English sentences, it is quite reasonable. In biological applications of sequence alignment, however, one often compares very long strings against one another; and in these cases, the $\Theta(mn)$ space requirement can potentially be a more severe problem than the $\Theta(mn)$ time requirement. Suppose, for example, that we are comparing two strings of length $100,000$ each. Depending on the underlying processor, the prospect of performing roughly ten billion primitive operations might be less cause for worry than the prospect of working with a single ten-gigabyte array.

Fortunately, this is not the end of the story. In this section we describe a very clever enhancement of the sequence alignment algorithm that makes it work in $O(mn)$ time using only $O(m + n)$ space. For ease of description, we'll describe various steps in terms of paths in the graph G_{XY}, with the natural equivalence back to the sequence alignment problem. Thus, when we seek the pairs in an optimal alignment, we can equivalently ask for the edges in a shortest corner-to-corner path in G_{XY}.

A Space-Efficient Algorithm for the Optimal Value. We first show that if we only care about the *value* of the optimal alignment, and not the alignment itself, it is easy to get away with linear space. The crucial observation is that to fill in an entry of the array A, the recurrence in (5.14) only needs information from the current column of A and the previous column of A. Thus we will we "collapse" the array A to an $m \times 2$ array B: as the algorithm iterates through values of j, $B[i, 0]$ will hold the "previous" column's value $A[i, j-1]$, and $B[i, 1]$ will hold the "current" column's value $A[i, j]$.

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m, 0...1]
```

```
For i = 0, ..., m
    B[i, 0] = iδ
Endfor
For j = 1, ..., n
    B[0, 1] = jδ
    For i = 1, ..., m
        B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                      δ + B[i − 1, 1], δ + B[i, 0]].
    Endfor
    For i = 1, ..., m
        B[i, 0] = B[i, 1]
    Endfor
Endfor
```

It is easy to verify that when this algorithm completes, the array entry $B[i, 1]$ holds the value of $OPT(i, n) = f(i, n)$, for $i = 0, 1, \ldots, m$. Moreover, it uses $O(mn)$ time and $O(m+n)$ space. The problem is: where is the alignment itself? We haven't left enough information around to be able to run a procedure like `Find-Alignment`; and as we think about it, we see that it would be very difficult to try "predicting" what the alignment is going to be as we run our space-efficient procedure. In particular, as we compute the values in the j^{th} column of the (now implicit) array A, we could try hypothesizing that a certain entry has a very small value, and hence that the alignment that passes through this entry is a promising candidate to be the optimal one. But this promising alignment might run into big problems later on, and a different alignment that currently looks much less attractive will turn out to be the optimal one.

There is in fact a solution to this problem — we will be able to recover the alignment itself using $O(m + n)$ space — but it requires a genuinely new idea. The insight is based on employing the *divide-and-conquer* technique that we've seen earlier in the course. We begin with a simple alternative way to implement the basic dynamic programming solution.

A Backward Formulation of the Dynamic Program. Recall that we use $f(i, j)$ to denote the length of the shortest path from $(0, 0)$ to (i, j) in the graph G_{XY}. Let's define $g(i, j)$ to be the length of the shortest path from (i, j) to (m, n) in G_{XY}. The function g provides an equally natural dynamic programming approach to sequence alignment, except that we build it up in reverse: we start with $g(m, n) = 0$, and the answer we want is $g(0, 0)$. By strict analogy with (5.14), we have the following recurrence for g.

(5.16) *For $i < m$ and $j < n$ we have $g(i, j) = \min[\alpha_{x_{i+1}y_{j+1}} + g(i + 1, j + 1), \delta + g(i, j + 1), \delta + g(i + 1, j)]$.*

This is just the recurrence one obtains by taking the graph G_{XY}, "rotating" it so that the node (m, n) is in the upper left corner, and using the previous approach. Using this picture,

we can also work out the full dynamic programming algorithm to build up the values of g, *backwards* starting from (m, n).

Combining the Forward and Backward Formulations. So now we have symmetric algorithms which build up the values of the functions f and g. The idea will be to use these two algorithms in concert to find the optimal alignment. First, here are two basic facts summarizing some relationships between the functions f and g.

(5.17) *The length of the shortest corner-to-corner path in G_{XY} that passes through (i, j) is $f(i, j) + g(i, j)$.*

Proof. Let ℓ_{ij} denote the length of the shortest corner-to-corner path in G_{XY} that passes through (i, j). Clearly any such path must get from $(0, 0)$ to (i, j), and then from (i, j) to (m, n). Thus its length is at least $f(i, j)+g(i, j)$, and so we have $\ell_{ij} \geq f(i, j)+g(i, j)$. On the other hand, consider the corner-to-corner path that consists of a minimum-length path from $(0, 0)$ to (i, j), followed by a minimum-length path from (i, j) to (m, n). This path has length $f(i, j) + g(i, j)$, and so we have $\ell_{ij} \leq f(i, j) + g(i, j)$. It follows that $\ell_{ij} = f(i, j) + g(i, j)$. ■

(5.18) *Let k be any number in $\{0, \ldots, n\}$, and let q be an index that minimizes the quantity $f(q, k) + g(q, k)$. Then there is a corner-to-corner path of minimum length that passes through the node (q, k).*

Proof. Let ℓ^* denote the length of the shortest corner-to-corner path in G_{XY}. Now, fix a value of $k \in \{0, \ldots, n\}$. The shortest corner-to-corner path must use *some* node in the k^{th} column of G_{XY} — let's suppose it is node (p, k) — and thus by (5.17)

$$\ell^* = f(p, k) + g(p, k) \geq \min_q f(q, k) + g(q, k).$$

Now consider the index q that achieves the minimum in the right-hand-side of this expression; we have

$$\ell^* \geq f(q, k) + g(q, k).$$

By (5.17) again, the shortest corner-to-corner path using the node (q, k) has length $f(q, k)+g(q, k)$, and since ℓ^* is the minimum length of *any* corner-to-corner path, we have

$$\ell^* \leq f(q, k) + g(q, k).$$

It follows that $\ell^* = f(q, k) + g(q, k)$. Thus the the shortest corner-to-corner path using the node (q, k) has length ℓ^*, and this proves (5.18). ■

Using (5.18) and our space-efficient algorithms to compute the *value* of the optimal alignment, we will proceed as follows. We divide G_{XY} along its center column and compute

the value of $f(i, n/2)$ and $g(i, n/2)$ for each value of i, using our two space-efficient algorithms. We can then determine the minimum value of $f(i, n/2) + g(i, n/2)$, and conclude via (5.18) that there is a shortest corner-to-corner path passing through the node $(i, n/2)$. Given this, we can search for the shortest path recursively in the portion of G_{XY} between $(0, 0)$ and $(i, n/2)$, and in the portion between $(i, n/2)$ and (m, n). The crucial point is that we apply these recursive calls sequentially, and re-use the working space from one call to the next; thus, since we only work on one recursive call at a time, the total space usage is $O(m + n)$. The key question we have to resolve is whether the running time of this algorithm remains $O(mn)$.

In running the algorithm, we maintain a globally accessible list P which will hold nodes on the shortest corner-to-corner path as they are discovered. Initially, P is empty. P need only have $m + n$ entries, since no corner-to-corner path can use more than this many edges. We also use the following notation: $X[i : j]$, for $1 \leq i \leq j \leq m$, denotes the substring of X consisting of $x_i x_{i+1} \cdots x_j$; and we define $Y[i : j]$ analogously. We will assume for simplicity that n is a power of 2; this assumption makes the discussion much cleaner, although it can be easily avoided.

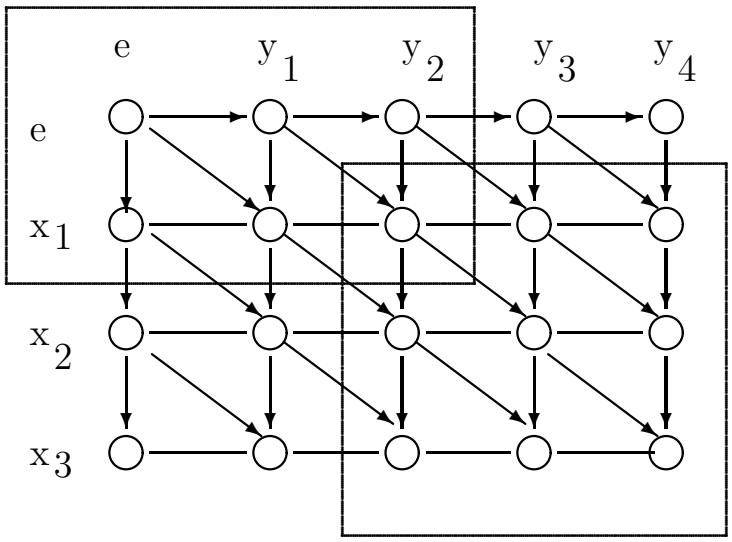
```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X.
  Let n be the number of symbols in Y.
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y).
  Call Space-Efficient-Alignment(X,Y[1 : n/2]),
     obtaining array B.
  Call Backwards-Space-Efficient-Alignment(X,Y[n/2 + 1 : n]),
     obtaining array B'.
  Let q be the index minimizing B[q, 1] + B'[q, 1].
  Add (q, n/2) to global list P.
  Divide-and-Conquer-Alignment(X[1 : q],Y[1, n/2])
  Divide-and-Conquer-Alignment(X[q + 1 : n],Y[n/2 + 1, n])
  Return P.
```

As an example of the first level of recursion, consider the figure below. If the "minimizing index" q turns out to be 1, we get the two sub-problems pictured.

The arguments above already establish that the algorithm returns the correct answer, and that it uses $O(m + n)$ space. Thus, we need only verify the following fact.

(5.19) *The running time of* `Divide-and-Conquer-Alignment` *on strings of length m and n is $O(mn)$.*

Proof. Let $T(m, n)$ denote the maximum running time of the algorithm on strings of length m and n. The algorithm performs $O(mn)$ work to build up the arrays B and B'; it then



runs recursively on strings of size q and $n/2$, and on string of size $m-q$ and $n/2$. Thus, for some constant c, we have

$$\begin{aligned} T(m,n) &\leq cmn + T(q,n/2) + T(m-q,n/2) \\ T(m,2) &\leq cm \\ T(2,n) &\leq cn. \end{aligned}$$

Now we claim that this recurrence implies $T(m,n) \leq 2cmn$; in other words, our space-efficient strategy has at worst doubled the running time. We prove this by induction, with the case of $m \leq 2$ and $n \leq 2$ following immediately from the inequalities above. For general m and n, we have

$$\begin{aligned} T(m,n) &\leq cmn + T(q,n/2) + T(m-q,n/2) \\ &\leq cmn + 2cqn/2 + 2c(m-q)n/2 \\ &= cmn + cqn + cmn - cqn \\ &= 2cnm. \end{aligned}$$

■

5.7 Shortest Paths in a Graph

For the final two sections, we focus on the problem of finding shortest paths in a graph, together with some closely related issues.

Let $G = (V, E)$ be a directed graph. Assume that each edge $(i, j) \in E$ has an associated *weight* c_{ij}. The weights can be used to model a number of different things; we will picture

here the interpretation in which the weight c_{ij} represents a *cost* for going directly from node i to node j in the graph.

Earlier, we discussed *Dijkstra's algorithm* for finding shortest paths in graphs with positive edge costs. Here we consider the more complex problem in which we seek shortest paths when costs may be negative. Among the motivations for studying this problem, here are two that particularly stand out. First, negative costs turn out to be crucial for modeling a number of phenomena with shortest paths. For example, the nodes may represent agents in a financial setting, and c_{ij} represents the cost of a transaction in which we buy from agent i and then immediately sell to agent j. In this case, a path would represent a succession of transactions, and edges with negative costs would represent transactions that result in profits. Second, the algorithm that we develop for dealing with edges of negative cost turns out, in certain crucial ways, to be more flexible and *decentralized* than Dijkstra's algorithm. As a consequence, it has important applications for the design of distributed routing algorithms that determine the most efficient path in a communication network.

In this section and the next we will consider the following two related problems.

- Given a graph G with weights, as described above, decide if G has a negative cycle, i.e., a directed cycle C such that
$$\sum_{ij \in C} c_{ij} < 0.$$
- If the graph has no negative cycles, find a path P from an origin node s to a destination node t with minimum total cost:
$$\sum_{ij \in P} c_{ij}$$
should be as small as possible for any s-t path. This is called both the *minimum-cost path problem* and the *shortest path problem*.

In terms of our financial motivation above, a negative cycle corresponds to a profitable sequence of transactions that takes us back to our starting point: we buy from i_1, sell to i_2, buy from i_2, sell to i_3, and so forth, finally arriving back at i_1 with a net profit. Thus, negative cycles in such a network can be viewed as good *arbitrage opportunities*.

It makes sense to consider the minimum-cost s-t path problem under the assumption that there are no negative cycles. If there is a negative cycle C, a path P_s from s to the cycle, and another path P_t from the cycle to t, then we can build an s-t path of arbitrarily negative cost: we first use P_s to get to the negative cycle C, then we go around C as many times as we want, and then use P_t to get from C to the destination t.

Let's begin by recalling Dijkstra's algorithm for the shortest path problem when there are no negative costs. The method computes a shortest path from the origin s to every other node v in the graph, essentially using a greedy algorithm. The basic idea is to maintain a

set S with the property that the shortest path from s to each node in S is known. We start with $S = \{s\}$ — since we know the shortest path from s to s has cost 0 when there are no negative edges — and we add elements greedily to this set S. As our first greedy step, we consider the minimum cost edge leaving node s, i.e., $\min_{i \in V} c_{si}$. Let v be a node on which this minimum is obtained. The main observation underlying Dijkstra's algorithm is that the shortest path from s to v is the single-edge path $\{s, v\}$. Thus we can immediately add the node v to the set S. The path $\{s, v\}$ is clearly the shortest to v if there are no negative edge costs: any other path from s to v would have to start on an edge out of s that is at least as expensive as edge sv.

The above observation is no longer true if we can have negative edge costs. A path that starts on an expensive edge, but then uses many edges with negative cost, can be cheaper than a path that starts on a cheap edge. This suggests that the Dijkstra-style greedy approach will not work here.

Another natural idea is to first modify the cost c_{ij} by adding some large constant M to each, i.e., we let $c'_{ij} = c_{ij} + M$ for each edge $(i, j) \in E$. If the constant M is large enough, then all modified costs are non-negative, and we can use Dijstra's algorithm to find the minimum-cost path subject to costs c'. However, this approach also fails. The problem here is that changing the costs from c to c' changes the minimum cost path. For example, if a path P consisting of 3 edges is somewhat cheaper than another path P' that has 2 edges, than after the change in costs, P' will be cheaper, since we only add $2M$ to the cost of P' while adding $3M$ to the cost of P.

We will try to use dynamic programming to solve the problem of finding a shortest path from s to t when there are negative edge costs but no negative cycles. We could try an idea that has worked for us so far: sub-problem i could be to find a shortest path using only the first i nodes. This idea does not immediately work, but it can be made to work with some effort. Here, however, we will discuss a simpler and more efficient solution, the *Bellman-Ford algorithm.* The development of dynamic programming as a general algorithmic technique is often credited to the work of Bellman in the 1950's; and the Bellman-Ford shortest path algorithm was one of the first applications.

The dynamic programming solution we develop will be based on the following crucial observation.

(5.20) *If G has no negative cycles, then there is a shortest path from s to t that is simple, and hence has at most $n - 1$ edges.*

Proof. Since every cycle has non-negative cost, the shortest path $\mathcal{P}$ from s to t with the fewest number of edges does not repeat any vertex v. For if $\mathcal{P}$ did repeat a vertex v, we could remove the portion of $\mathcal{P}$ between consecutive visits to v, resulting in a path of no greater cost and fewer edges. ■

Let's use $OPT(i, v)$ to denote the minimum cost of a v-t path using at most i edges. By (5.20), our original problem is to compute $OPT(n-1, s)$. We could instead design an algorithm whose sub-problems correspond to the minimum cost of an s-v path using at most i edges. This would form a more natural parallel with Dijkstra's algorithm, but it would not be as natural in the context of the routing protocols we discuss later.

We now need a simple way to express $OPT(i, v)$ using smaller sub-problems. We will see that the most natural approach involves the consideration of many different cases; this is another example of the principle of "multi-way choices" that we saw in the algorithm for the *segmented least squares problem*.

Let's fix an optimal path $\mathcal{P}$ representing $OPT(i, v)$ as depicted on Figure 5.11.

- If the path $\mathcal{P}$ uses at most $i-1$ edges, then we have $OPT(i, v) = OPT(i-1, v)$
- If the path $\mathcal{P}$ uses i edges, and the first edge is (v, w), then $OPT(i, v) = c_{vw} + OPT(i-1, w)$.

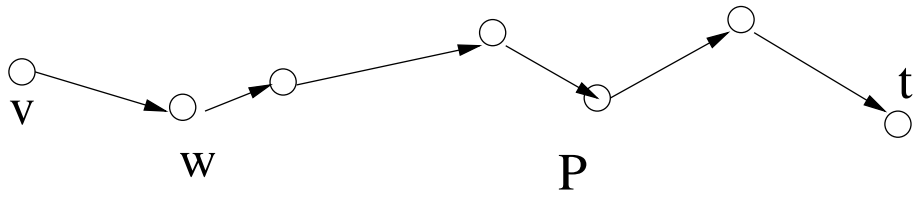


Figure 5.11: The minimum cost path P from v to t using at most i edges.

This leads to the following recursive formula.

(5.21) *If $i > 0$ then*

$$OPT(i, v) = \min(OPT(i-1, v), \min_{w \in V}(OPT(i-1, w) + c_{wv})).$$

Using this recurrence, we get the following dynamic programming algorithm to compute the value $OPT(n-1, t)$.

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0...n-1, V]
  For v ∈ V in any order
    M[0, v] = ∞
  Endfor
  M[0, t] = 0
  For i = 1, ..., n-1
    For v ∈ V in any order
```

```
    M = M[i − 1, v]
    M′ = min_{w∈V}(c_vw + M[i − 1, w])
    M[i, v] = min(M, M′)
  Endfor
Endfor
Return M[n − 1, s]
```

The correctness of the method follows directly from the statement (5.21). We can bound the running time as follows. The table M has n^2 entries; and each entry can take $O(n)$ time to compute, as there are at most n nodes w we have to consider.

(5.22) *The* `Shortest-Path` *method correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in $O(n^3)$ time.*

Given the table M containing the optimal values of the sub-problems, the shortest path using at most i edges can be obtained in $O(in)$ time, by tracing back through smaller sub-problems.

Improved Versions

A big problem with the above version of the Bellman-Ford algorithm (and in fact, with many dynamic programming algorithms) is that it uses too much memory. For a graph with n nodes, we used a table M of size n^2. Our first change to the algorithm will be aimed at decreasing the memory requirement. We will no longer record $M[i, v]$ for each value i; instead we will use and update a single value $M[v]$ for each node v, the length of the shortest path from v to t that we have found so far. One can change the above algorithm to proceed in rounds; in each round we let $M' = \min_{w \in V}(c_{vw} + M[w])$, and update $M[v]$ to be $\min(M[v], M')$. Just as before, the code will have a "`For` $i = 1, \ldots, n-1$" loop, but the only point of the loop is to count the number of iterations. The following lemma is not hard to show.

(5.23) *Throughout the algorithm $M[v]$ is the length of some path from v to t, and after i rounds of updates the value $M[v]$ is no larger than the length of the shortest path from v to t using at most i edges.*

Now we can use (5.20) to show that after $n - 1$ iterations we are done.

Further, we can also improve the running time. A graph with n nodes can have close to n^2 directed edges. When we work with a graph for which the number of edges m is significantly less than n^2, it is often useful to write the running time in terms of both m and n; this way, we can quantify our speed-up when we work with a graph that doesn't have very many edges.

If we are a little more careful in the analysis of the method above, we can improve the running time bound to $O(mn)$ without significantly changing the algorithm itself.

(5.24) *The* `Shortest-Path` *method can be implemented in $O(mn)$ time, using $O(n)$ memory.*

Proof. The improvement is obtained through two changes to the algorithm. The improvement in memory usage has been discussed above. To obtain the improvement in the running time, we consider the line that computes

$$M' = \min_{w \in V}(c_{vw} + M[i-1, w])$$

while building up the array entry $M[i, v]$. We assumed it could take up to $O(n)$ time to compute this minimum, since there are n possible nodes w. But of course, we need only compute this minimum over all nodes w for which v has an edge to w; let us use n_v to denote this number. Then it takes time $O(n_v)$ to compute the array entry $M[i, v]$. We have to compute an entry for every node v and every index $0 \leq i \leq n-1$, so this gives a running time bound of $O\left(n \sum_{v \in V} n_v\right)$.

This bound can be written $O(mn)$, as shown by the following fact. ■

(5.25) $\sum_{v \in V} n_v = m$

Proof. Each edge enters exactly one node, which implies the statement. ■

Note that the path whose length is $M[v]$ after i iterations, can have substantially more edges than i. For example, if the graph is a single path, and we perform updates in the order the edges appear on the path, then we get the final shortest path values in just one iteration. To take advantage of the fact that the $M[v]$ values may reach the length of the shortest path in fewer than $n-1$ iterations, we need to be able to terminate without reaching iteration $n-1$. This can be done using the following observation: if we reach an iteration i in which *no* $M[v]$ value changes, then the algorithm can terminate — since there will be no further changes in any subsequent iteration. Note that it is not enough for a *particular* $M[v]$ value to remain the same; in order to safely terminate, we need for all these values to remain the same for a single iteration.

Shortest Paths and Distance Vector Protocols

One important application of the shortest path problem is for routers in a communication network to determine the most efficient path to a destination. We represent the network using a graph in which the nodes correspond to routers, and there is an edge between v and w if the two routers are connected by a direct communication link. We define a cost c_{vw} representing the delay on the link (v, w); the shortest path problem with these costs is to determine the path with minimum delay from a source node s to a destination t. Delays are

naturally non-negative, so one could use Dijkstra's algorithm to compute the shortest path. However, Dijkstra's shortest path computation requires global knowledge of the network: we need to maintain a set S of nodes for which shortest paths have been determined, and make a global decision about which node to add next to S. While routers could be made to run a protocol in the background that gathers enough global information to implement such an algorithm, it is cleaner and more flexible to use algorithms that require only local knowledge of neighboring nodes.

If we think about it, the Bellman-Ford algorithm discussed above has just such a "local" property. Suppose we let each node v maintain its value $M[v]$; then to update this value, v needs only obtain the value $M[w]$ from each neighbor w, and then compute

$$\min_{w \in V}(c_{vw} + M[i-1, w])$$

based on the information obtained.

We now discuss two improvements to the Bellman-Ford algorithm that make it better suited for routers, and at the same time also make it a faster algorithm in practice. First, our current implementation of the Bellman-Ford algorithm can be thought of as a "pull"-based algorithm. In each iteration i, each node v has to contact each neighbor w, and "pull" the new value $M[w]$ from it. If a node w has not changed its value, then there is no need for v to get the value again — however, v has has no way of knowing this fact, and so it must execute the "pull" anyway.

This wastefulness suggeests a symmetric "push"-based implementation, where values are only transmitted when they change. Specifically, each node w whose distance value $M[w]$ changes in an iteration informs all its neighbors of the new value in the next iteration; this allows them to update their values accordingly. If $M[w]$ has not changed, then the neighbors of w already have the current value, and there is no need to "push" it to them again. Here is a concrete description of the push-based implementation:

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[V]
  For v ∈ V in any order
    M[v] = ∞
  Endfor
  M[t] = 0
  For i = 1, ..., n − 1 or While some value changes
    For w ∈ V in any order
      If M[w] has been updated in the previous iteration then
         For all edges (v, w) in any order
              M[v] = min(M[v], c_vw + M[w])
         Endfor
```

```
  Endfor
Endfor
Return M[s]
```

In this algorithm nodes are sent updates of their neighbors' distance values in rounds, and each node may send out updates in each iteration. This complete synchrony cannot be enforced for nodes that are independently operating routers. However, even if nodes update their distance values asynchronously, it is not hard to see the followng: the distances will eventually converge to the correct values assuming only that the costs c_{vw} remain constant and each node whose value changes eventually sends out the required updates.

The algorithm we developed uses a single destination t, and all nodes $v \in V$ compute their shortest path to t. More generally, we are presumably interested in finding distances and shortest paths between all pairs of nodes in a graph. To obtain such distances, we effectively use n separate computations, one for each destination. Such an algorithm is referred to as a *distance vector protocol*, since each node maintains a vector of distances to every other node in the network.

Problems with the distance vector protocol. One of the major problems with the distributed implementation of Bellman-Ford on routers — the protocol we have been discussing above — is that it does not deal well with cases in which edges are deleted, or edge costs increase significantly. If an edge (v, w) is deleted (say the link goes down), it is natural for node v to react as follows: it should check whether its shortest path to some node t used the edge (v, w), and, if so, it should increase the distance using other neighbors. Notice that this increase in distance from v can now trigger increases at v's neighbors, if they were relying on a path through v, and these changes can cascade through the network. Consider an example in which the original (undirected) graph has two edges (s, v) and (v, t) of cost 1 each as shown on Figure 5.12.

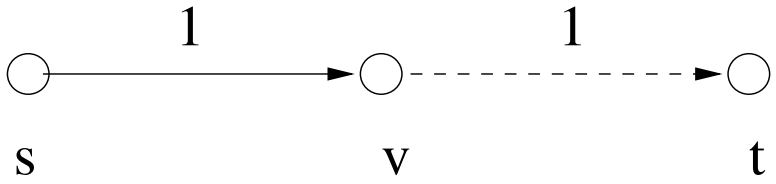


Figure 5.12: The problem of counting to infinity.

Now, suppose the edge (v, t) in Figure 5.12 is deleted. How does node v react? Unfortunately, it does not have a global map of the network; it only knows the shortest-path distances of each of its neighbors to t. Thus, it does not know that the deletion of (v, t) has eliminated all paths from s to t. Instead, it sees that $M[s] = 2$, and so it updates $M[v] = c_{vs} + M[s] = 3$

— assuming that it will use its cost-1 edge to s, followed by the supposed cost-2 path from s to t. Seeing this change, node s will update $M[s] = c_{sv} + M[v] = 4$ — based on its cost-1 edge to v, followed by the supposed cost-3 path from v to t. Nodes s and v will continue updating their distance to t until one of them finds an alternate route; in the case, as here, that the network is truly disconnected, these updates will continue indefinitely — a behavior known as the problem of *counting to infinity*.

To avoid this problem, the designers of routing algorithms have tended to move from distance vector protocols to more expressive *path vector protocols*, in which each node stores not just the distance and first hop of their path to a destination, but a representation of the entire path. Given knowledge of the paths, nodes can avoid updating their paths to use edges they know to be deleted; at the same time, they require significantly more storage to keep track of the full paths. The path-vector approach is used in the *border gateway protocol (BGP)* in the Internet core.

5.8 Negative Cycles in a Graph

In this section, we consider graphs that have negative cycles. There are two natural questions we will consider.

- How do we decide if a graph contains a negative cycle?
- How do we actually construct a negative cycle in a graph that contains one?

The algorithm developed for finding negative cycles will also lead to an improved practical implementation of the Bellman-Ford algorithm from the previous section.

It turns out that the ideas we've seen so far will allow us to find negative cycles that have a path reaching a sink t. Before we develop the details of this, let's compare the problem of finding a negative cycle that can reach a given t with the seemingly more natural problem of finding a negative cycle *anywhere* in the graph, regardless of its position related to a sink. It turns out that if we develop a solution to the first problem, we'll be able to obtain a solution to the second problem as well, in the following way. Suppose we start with a graph G, add a new node t to it, and connect each other node v in the graph to node t via an edge of cost 0 as shown on Figure 5.13. Let us call the new "augmented graph" G'.

(5.26) *The augmented graph G' has a negative cycle reachable C such that there is a path from C to the sink t, if and only if the original graph has a negative cycle.*

Proof. Assume G has a negative cycle. Then this cycle C clearly has an edge to t in G', since all nodes have an edge to t.

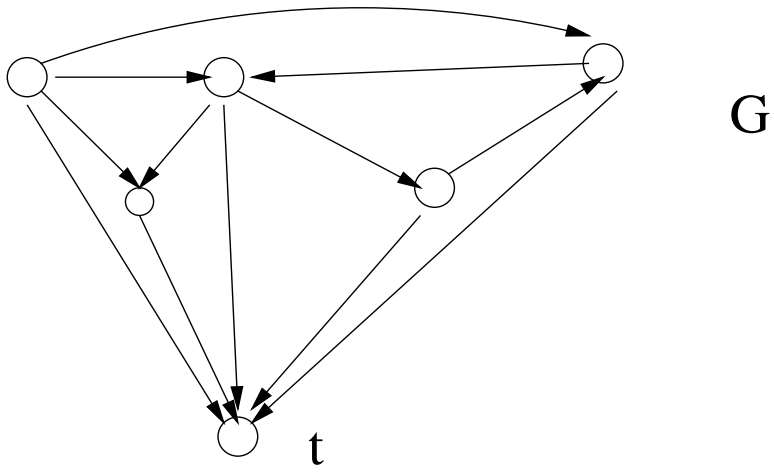


Figure 5.13: The augmented graph.

Now suppose G' has a negative cycle with a path to t. Since no edge leaves t in G', this cycle cannot contain t. Since G' is the same as G aside from the node t, it follows that this cycle is also a negative cycle of G. ■

So it is really enough to solve the problem of deciding whether G has a negative cycle that has a path to a given sink node t, and we do this now. To solve this problem, we first extend our definitions of $OPT(i, v)$ for values $i \geq n$. With the presence of a negative cycle in the graph, (5.20) no longer applies, and indeed the shortest path may get shorter and shorter as we go around a negative cycle. In fact, for any node v on a negative cycle that has a path to t, we have the following.

(5.27) *If node v can reach node t and is contained in a negative cycle, then*

$$\lim_{i \to \infty} OPT(i, v) = -\infty.$$

If the graph has no negative cycles, then (5.20) implies following statement.

(5.28) *If there are no negative negative cycles in G, then $OPT(i, v) = OPT(n-1, v)$ for all nodes v and all $i \geq n$.*

But for how large an i do we have to compute the values $OPT(i, v)$ before concluding that the graph has no negative cycles? For example, a node v may satisfy the equation $OPT(n, v) = OPT(n-1, v)$, and yet still lie on a negative cycle. (Do you see why?) However, it turns out that we will be in good shape if this equation holds for all nodes.

(5.29) *There is no negative cycle reachable from s if and only if $OPT(n, v) = OPT(n-1, v)$ for all nodes v.*

Proof. (5.28) has already proved the forward direction of this statement. Now, suppose $OPT(n, v) = OPT(n-1, v)$ for all nodes v. The values of $OPT(n+1, v)$ can be computed from $OPT(n, v)$; but all these values are the same as the corresponding $OPT(n-1, v)$. It follows that we will have $OPT(n+1, v) = OPT(n-1, v)$. Extending this reasoning to future iterations, we see that none of the values will ever change again, i.e., $OPT(i, v) = OPT(n-1, v)$ for all nodes v and all $i \geq n$. Thus, there cannot be a negative cycle C that has a path to t; such a cycle C would contain a node w, and by (5.27) the values $OPT(i, w)$ would have to become arbitrarily negative as i increased. ■

(5.29) gives an $O(mn)$ method to decide if G has a negative cycle reachable from s. We compute values of $OPT(i, v)$ for nodes of G and for values of i up to n. By (5.29), there is no negative cycle if and only if there is some value of i at which $OPT(i, v) = OPT(i-1, v)$ for all nodes v.

So far we have determined whether or not the graph has a negative cycle with a path from the cycle to t, but we have not actually found the cycle. To find a negative cycle, we consider a node v such that $OPT(n, v) \neq OPT(n-1, v)$: for this node, a path $\mathcal{P}$ from v to t of cost $OPT(n, v)$ must use *exactly* n edges. We find this minimum-cost path $\mathcal{P}$ from v to t by tracing back through the sub-problems. As in our proof of (5.20), a simple path can only have $n-1$ edges, so $\mathcal{P}$ must contain a cycle C. We claim that this cycle C has negative cost.

(5.30) *If G has n nodes, and $OPT(n, v) \neq OPT(n-1, v)$, then a path $\mathcal{P}$ from v to t of cost $OPT(n, v)$ contains a cycle C, and C has negative cost.*

Proof. First observe that the path $\mathcal{P}$ must have n edges, as $OPT(n, v) \neq OPT(n-1, v)$, and so every path using $n-1$ edges has cost greater than that of the path $\mathcal{P}$. In a graph with n nodes, a path consisting of n edges must repeat a node somewhere; let w be a node that occurs on $\mathcal{P}$ more than once. Let C be the cycle on $\mathcal{P}$ between two consecutive occurrence of node w. If C were not a negative cycle, then deleting C from $\mathcal{P}$ would give us an v-t path with fewer than n edges, and no greater cost. This contradicts our assumption that $OPT(n, v) \neq OPT(n-1, v)$, and hence C must be a negative cycle. ■

(5.31) *The algorithm above finds a negative cycle in G, if such a cycle exists, and runs in $O(mn)$ time.*

An Improved Version

Earlier we saw that if a graph G has no negative cycles, the algorithm can often be stopped early: if for some value of $i \leq n$, we have $OPT(i, v) = OPT(i-1, v)$ *for all nodes* v, then the values will not change on any further iteration either. Is there an analogous early

termination rule for graphs that have negative cycles, so that we can sometimes avoid running all n iterations?

We can use Statement (5.20) to detect the presence of negative cycles, but to do this we need to "count to $n-1$". This problem is analogous to the problem of counting to infinity discussed in relation to the distance vector protocol. In fact, the two problems have similar underlying causes: the repeated change in the distance values is caused by using cycles repeatedly in a single path. One could adapt the path vector solution to solve this problem, by maintaining paths explicitly and terminating whenever a path has a negative cycle; but maintaining path vectors can take as much as $O(n)$ extra time and memory.

In the case of Internet routing one resorts to maintaining paths so as to keep the protocol distributed. However, if we're thinking about finding negative cycles using a traditional algorithm, then allowing simple global computations will make it possible to solve the problem without having to maintain complete paths. Here we will discuss a solution to this problem where each node v maintains the first node $f[v]$ after v on the shortest path to the destination t. To maintain $f[v]$ we must update this value whenever the distance $M[v]$ is updated. In other words, we add the line

```
If M[v] > M[w] + c_vw
   then F[v] = w
```

before updating the $M[v]$ value. Note that once the final distance values are computed, we can now find the shortest path by simply following the selected edges: v to $f[v] = v_1$, to $f[v_1] = v_2$, and so forth.

Let P denote the directed "pointer" graph whose nodes are V, and whose edges are $\{(v, f[v])\}$. The main observation is the following:

(5.32) *If the pointer graph P contains a cycle C, then this cycle must have negative cost.*

Proof. Notice that if $f[v] = w$ at any time, then we must have $M[v] \geq c_{vw} + M[w]$. Indeed, the left- and right-hand sides are equal when $f[v]$ is set to w; and since $M[w]$ may decrease, this equation may turn into an inequality.

Let $v_1, v_2, \ldots, v_k$ be the nodes along the cycle C in the pointer graph, and assume that (v_k, v_1) is the last edge to have been added. Now, consider the values right before this last update. At this time we have $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ for all $i = 1, \ldots, k-1$, and we also have $M[v_k] > c_{v_k v_1} + M[v_1]$ since we are about to update $M[v_k]$ and change $f[v_k]$ to v_1. Adding all these inequalities, the $M[v_i]$ values cancel, and we get $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$: a negative cycle, as claimed. ■

To take advantage of this observation, we would like to determine whether a cycle is created in the pointer graph P every time we add a new edge (v, w) with $f[v] = w$. (Consider Figure 5.14 for an example.) The most natural way to do this is to follow the current path

from w to the terminal t in time proportional to the length of this path. If we encounter v along this path, then a cycle has been formed, and hence by (5.32) the graph has a negative cycle. However, if we do this, then we could spend as much as $O(n)$ time following the path to t and still not find a cycle. Next we discuss a method that works does not require an $O(n)$ blow-up in the running time.

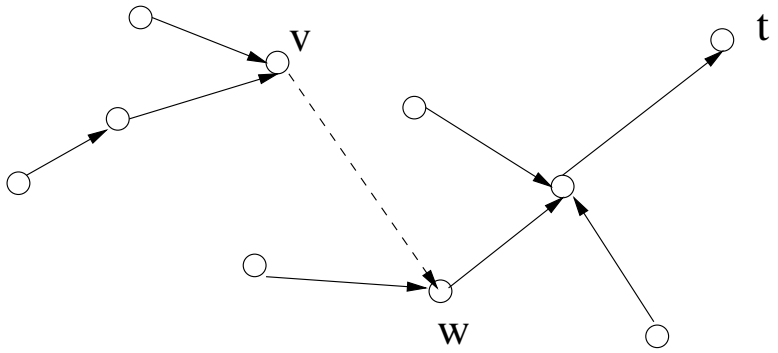


Figure 5.14: The pointer graph P with new edge (v, w) being added.

We know that before the new edge (v, w) was added, the pointer graph was a directed tree. Another way to test whether the addition of (v, w) creates a cycle is to consider all nodes in the subtree directed towards v. If w is in this subtree then (v, w) forms a cycle; otherwise it does not. To be able to find all nodes in the subtree directed towards v, we need to have each node v maintain a list of all other nodes whose selected edges point to v. Given these pointers we can find the subtree in time proportional to the time size of the subtree pointing to v, at most $O(n)$ as before. However, here we will be able to make additional use of the work done. Notice that the current distance value $M[x]$ for all nodes x in the subtree was derived from node v's old value. We have just updated v's distance, and hence we know that the distance values of all these node will be updated again. We'll mark each ofthese nodes x as "inactive", delete the edge $(x, f[x])$ from the pointer graph, and not use x for future updates until its distance value changes. This can save a lot of future work in updates, but what is the effect on the worst case running time? We can spend as much as $O(n)$ extra time marking nodes inactive after every update in distances. However, a node can be marked inactive only if it was active before, so the time spent on marking nodes inactive is at most as much at the time the algorithm spends updating distances. The time spent by the algorithm on operations other than marking nodes inactive is $O(mn)$ by Statement (5.31) , and hence we see that the new implementation of the algorithm still runs in $O(mn)$ time, using $O(n)$ space. In fact, this new version is in practice the fastest implementation of the algorithm even for graphs that do not have negative cycles, or even negative-cost edges.

5.9 Exercises

1. Suppose you're running a lightweight consulting business — just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

   Each month, you can either run your business from an office in New York (NY), or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

   However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed *moving cost* of M to switch base offices.

   Given a sequence of n months, a *plan* is a sequence of n locations — each one equal to either NY or SF — such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

   The problem is: Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

   Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

   Then the plan of minimum cost would be the sequence of locations

   $$[NY, NY, SF, SF],$$

   with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

   (a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

   ```
   For i = 1 to n
     If N_i < S_i then
       Output "NY in Month i"
   ```

```
  Else
    Output "SF in Month i"
End
```

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an example of an instance in which every optimal plan must move (i.e. change locations) at least three times.

Provide an explanation, of at most three sentences, saying why your example has this property.

(c) Give an algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

2. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

   Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

   Consider, for example, the 5-node path drawn in the figure below. The *weights* are the numbers drawn next to the nodes.

   The goal in this question is to solve the following algorithmic problem:

   > (*) *Find an independent set in a path G whose total weight is as large as possible.*

   (a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm:
  Start with S equal to the empty set.
  While some node remains in G
    Pick a node v_i of maximum weight.
    Add v_i to S.
    Delete v_i and its neighbors from G.
  end while
  Return S
```

(b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let S_1 be the set of all v_i where i is an odd number.
Let S_2 be the set of all v_i where i is an even number.
/* Note that S_1 and S_2 are both independent sets.  */
Determine which of S_1 or S_2 has greater total weight,
  and return this one.
```

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is a *line-graph* if it has the following properties:

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following algorithmic problem:

> *Given a line-graph G, find the length of the longest path that begins at v_1 and ends at v_n.*

Thus, a correct answer for the line-graph in the figure would be 3: the longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of a line-graph on which it does not return the correct answer.

```
Set w = v_1.
Set L = 0
While there is an edge out of the node w
  Choose the edge (w, v_j)
     for which j is as small as possible.
  Set w = v_j
  Increase L by 1.
end while
Return L as the length of the longest path.
```

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes a line graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

4. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g. setting up a Web site for a class of fifth-graders at the local elementary school) and those that are *high-stress* (e.g. protecting America's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers.) The basic question, each week, is whether to take on a low-stress job or a high-stress job.

   If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

   So given a sequence of n weeks, a *plan* is specified by a choice of "low-stress", "high-stress", or "none" for each of the n weeks — with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

   The problem is: Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

   Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

   (a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

```
For iterations i = 1 to n
  If h_{i+1} > ℓ_i + ℓ_{i+1} then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i + 1"
    Continue with iteration i + 2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i + 1
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, and returns the *value* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

5. Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily-traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$ (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location x_i, you receive a revenue of $r_i > 0$.

   You want to place billboards at a subset of the sites in $\{x_1, \ldots, x_n\}$ so as to maximize your total revenue, subject to the following restrictions.

   (i) (*The environmental constraint.*) You cannot build two billboards within less than 5 miles of one another on the highway.

   (ii) (*The boundary constraint.*) You cannot build a billboard within less than 5 miles of the western or eastern ends of the highway.

   A subset of sites satisfying these two restrictions will be called *valid.*

   Example. Suppose $M = 20$, $n = 4$,

$$\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\},$$

and

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}.$$

Then the optimal solution would be to place billboards at x_1 and x_3, for a total revenue of 10.

Give an algorithm that takes an instance of this problem as input, and returns the maximum total revenue that can be obtained from any valid subset of sites.

The running time of the algorithm should be polynomial in n. Include a brief analysis of the running time of your algorithm, and a proof that it is correct.

6. You're trying to run a large computing job, in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

   Here's the problem you're faced with. Your job can only run on one of the machines in any given minute. Over each of the next n minutes you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

   So given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute — with the property that choices A and B cannot appear in consecutive minutes. E.g. if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i + 1$ must be *move*, and then your choice for minute $i + 2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

   The problem is: Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

   Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
In minute 1, choose machine achieving the larger of a_1, b_1.
Set i = 2
While i ≤ n
   What was the choice in minute i − 1?
   If A:
      If b_{i+1} > a_i + a_{i+1} then
         Choose move in minute i and B in minute i + 1
         Proceed to iteration i + 2
      Else
         Choose A in minute i
         Proceed to iteration i + 1
      Endif
   If B:  behave as above with roles of A and B reversed.
EndWhile
```

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

7. Suppose you're consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. A typical instance of the problem is: they're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose

during this time period, they wanted to buy 1000 shares on some day, and sell all these shares on some (later) day. They want to know: when should they have bought and when should they have sold in order to have made as much money as possible? (If there was no way to make money during the n days, you should report this instead.)

Example: Suppose $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then you should return *"buy on 2, sell on 3"*; i.e. buying on day 2 and selling on day 3 means they would have made \$4 per share, the maximum possible for that period.

Clearly, there's a simple algorithm that takes time $O(n^2)$: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

Show how to find the correct numbers i and j in time $O(n)$.

8. Eventually your friends from the previous problem move up to more elaborate simulations, and they're hoping you can still help them out. As before, they're looking at n consecutive days of a given stock, at some point in the past. The days are numbered $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day.

   For certain (possibly large) values of k, they want to study what they call *k-shot strategies*. A k-shot strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$, where $0 \leq m \leq k$ and

   $$1 \leq b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leq n.$$

   We view these as a set of up to k non-overlapping intervals, during each of which the investors buy 1000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given k-shot strategy is simply the profit obtained from the m buy-sell transactions, namely

   $$1000 \sum_{i=1}^{m} p(s_i) - p(b_i).$$

   The investors want to assess the value of k-shot strategies by running simulations on their n-day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the k-shot strategy with the maximum possible return. Since k may be relatively large in these simulations, your running time should be polynomial in both n and k; it should not contain k in the exponent.

9. Suppose you're consulting for a company that manufactures PC equipment, and ships it to distributors all over the country. For each of the next n weeks they have a projected *supply* s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of 4 consecutive weeks at a time.

A *schedule*, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of 4 contiguous weeks at a time. The *cost* of the schedule is the total amount paid to A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, \ldots, s_n$, and returns a *schedule* of minimum cost.

Example: Suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a blocks of 4 consecutive weeks, and then company A for the final three weeks.

10. Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: it will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

    Now, you have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. So as to figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ (rough estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *non-decreasing*: if $h < h'$ then $f_i(h) \leq f_i(h')$.)

    So the problem is: given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your

algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

11. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with one another (e.g. by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from one another and lose the ability to communicate directly.

    In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (I.e. we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

    Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on to an edge set E_b. For $i = 0, 1, 2, \ldots, b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse", it would look precisely like the sequence of graphs $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

    Now, consider two particular nodes $s, t \in V$. For an s-t path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \ldots, P_b$ so that for each i, P_i is an s-t path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes* — points at which the identity of the path switches. Formally, we define $\mathit{changes}(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$

    Fix a constant $K > 0$. We define the *cost* of the sequence of paths $P_0, P_1, \ldots, P_b$ to be

$$\mathit{cost}(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot \mathit{changes}(P_0, P_1, \ldots, P_b).$$

    (a) Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

    (b) Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

12. Recall the scheduling problem from the text in which we sought to minimize the maximum *lateness*. There are n jobs, each with a deadline d_i and a required processing time t_i, and all jobs are available to be scheduled starting at time s. For a job i to be done it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned non-overlapping intervals. As usual, an assignment of times in this way will be called a *schedule*.

    In this problem, we consider the same set-up, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We'll say that a subset J of the jobs is *schedulable* if there is a schedule for the jobs in J so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size, and give a schedule for this subset that allows each job to finish by its deadline.

    (a) Prove that there is an optimal solution J (i.e. a schedulable set of maximum size) in which the jobs in J are scheduled in increasing order of their deadlines.

    (b) Assume that all deadlines d_i and required times t_i are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs n, and the maximum deadline $D = \max_i d_i$.

13. Consider the sequence alignment problem over a four-letter alphabet $\{z_1, z_2, z_3, z_4\}$, with a cost δ for each insertion or deletion, and a cost α_{ij} for a substitution of z_i by z_j (for each pair $i \neq j$). Assume that δ and each α_{ij} is a positive integer.

    Suppose you are given two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the *unique* minimum-cost alignment between A and B.

14. Consider the following inventory problem. You are running a store that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are *stored* until the beginning of the next month. You can store at most S trucks, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

    - There are two parts to the cost. First, storage: it costs C for every truck on hand that is not needed that month. Scecond, ordering fees: it costs K for every order placed.

- In each month you need enough trucks to satisfy the demand d_i, but the amount left over after satisfying the demand for the month should not exceed the inventory limit S.

Give an algorithm that solves this problem in time that is polynomial in n and S.

15. You are consulting for an independently operated gas-station, and it faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order they need to pay a fix price P for delivery in addition to the cost of the gas ordered. However, it cost c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost. They are planning to close for winder break, and want their tank to be empty, as they are afraid that any gas left in the tank would freeze over during break. Luckily, years of experience gives them accurate projections for how much gas they will need each day until winter break. Assume that there are n days left till break, and they need g_i gallons of gas for each of day $i = 1, ..., n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide which days they should place orders, and how much to order to minimize their total cost.

    The following two observations might help.

    - If $g_1 > 0$ then the first order has to arrive the morning of day 1.
    - If the next order is due to arrive on day i, then the amount ordered should be $\sum_{j=1}^{i-1} g_j$.

16. Through some friends of friends, you end up on a consulting visit to the cutting-edge biotech firm Clones 'R' Us. At first you're not sure how your algorithmic background will be of any help to them, but you soon find yourself called upon to help two identical-looking software engineers tackle a perplexing problem.

    The problem they are currently working on is based on the *concatenation* of sequences of genetic material. If X and Y are each strings over a fixed alphabet Σ, then XY denotes the string obtained by *concatenating* them — writing X followed by Y. CRU has identified a "target sequence" A of genetic material, consisting of m symbols, and they want to produce a sequence that is as similar to A as possible. For this purpose, they have a set of (shorter) sequences $B_1, B_2, \ldots, B_k$, consisting of $n_1, n_2, \ldots, n_k$ symbols respectively. They can cheaply produce any sequence consisting of copies of the strings in $\{B_i\}$ concatenated together (with repetitions allowed).

    Thus, we say that a *concatenation over* $\{B_i\}$ is any sequence of the form $B_{i_1} B_{i_2} \cdots B_{i_\ell}$, where each $i_j \in \{1, 2, \ldots, k\}$. So B_1, $B_1 B_1 B_1$, and $B_3 B_2 B_1$ are all concatenations

over $\{B_i\}$. The problem is to find a concatenation over $\{B_i\}$ for which the sequence alignment cost is as small as possible. (For the purpose of computing the sequence alignment cost, you may assume that you are given a cost δ for each insertion or deletion, and a substitution cost α_{ij} for each pair $i, j \in \Sigma$.)

Give a polynomial-time algorithm for this problem.

17. Suppose we want to replicate a file over a collection of n servers, labeled $S_1, S_2, \ldots, S_n$. To place a copy of the file at server S_i results in a *placement cost* of c_i, for an integer $c_i > 0$.

    Now, if a user requests the file from server S_i, and no copy of the file is present at S_i, then the servers $S_{i+1}, S_{i+2}, S_{i+3} \ldots$ are searched in order until a copy of the file is finally found, say at server S_j, where $j > i$. This results in an *access cost* of $j - i$. (Note that the lower-indexed servers $S_{i-1}, S_{i-2}, \ldots$ are not consulted in this search.) The access cost is 0 if S_i holds a copy of the file. We will require that a copy of the file be placed at server S_n, so that all such searches will terminate, at the latest, at S_n.

    We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a *configuration* is a choice, for each server S_i with $i = 1, 2, \ldots, n-1$, of whether to place a copy of the file at S_i or not. (Recall that a copy is always placed at S_n.) The *total cost* of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all n servers.

    Give a polynomial-time algorithm to find a configuration of minimum total cost.

18. $(*)$ Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

    We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a sub-tree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

    Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

19. Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a *rising trend* as follows.

    They have the closing price for a given stock recorded for n days in succession; let these prices be denoted $P[1], P[2], \ldots, P[n]$. A *rising trend* in these prices is a subsequence of the prices $P[i_1], P[i_2], \ldots, P[i_k]$, for days $i_1 < i_2 < \ldots < i_k$, so that

- $i_1 = 1$, and
- $P[i_j] < P[i_{j+1}]$ for each $j = 1, 2, \ldots, k-1$.

Thus a rising trend is a subsequence of the days — beginning on the first day and not necessarily contiguous — so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose $n = 7$, and the sequence of prices is

$$10, 1, 2, 11, 3, 4, 12.$$

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

(a) Show that the following algorithm does not correctly return the *length* of the longest rising trend, by giving an instance on which it fails to return the correct answer.

```
Define i = 1.
       L = 1.
For j = 2 to n
  If P[j] > P[i] then
    Set i = j.
    Add 1 to L.
  Endif
Endfor
```

In your example, give the actual length of the longest rising trend, and say what the above algorithm returns.

(b) Give an algorithm that takes a sequence of prices $P[1], P[2], \ldots, P[n]$ and returns the *length* of the longest rising trend.

The running time of your algorithm should be polynomial in the length of the input. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

20. Consider the Bellman-Ford minimum-cost path algorithm from the text, assuming that the graph has no negative cost cycles. This algorithm is both fairly slow and also memory-intensive. In many applications of dynamic programming, the large memory requirements can become a bigger problem than the running time. The goal of

this problem is to decrease the memory requirement. The pseudo-code SHORTEST-PATH(G, s, t) in the text maintains an array $M[0...n-1; V]$ of size n^2, where $n = |V|$ is the number of nodes on the graph.

Notice that the values of $M[i, v]$ are computed only using $M[i-1, w]$ for some nodes $w \in V$. This suggests he following idea: can we decrease the memory needs of the algorithm to $O(n)$ by maintaining only two columns of the M matrix at any time? Thus we will "collapse" the array M to an $2 \times n$ array B: as the algorithm iterates through values of i, $B[0, v]$ will hold the "previous" column's value $M[i-1, v]$, and $B[1, v]$ will hold the "current" column's value $M[i, v]$.

```
Space-Efficient-Shortest-Path(G, s, t)
  n = number of nodes in G
  Array B[0...1, V]
  For v ∈ V in any order
    B[0, v] = ∞
  Endfor
  B[0, s] = 0
  For i = 1, ..., n − 1
    For v ∈ V in any order
      M = B[0, v]
      M' = min_{w∈V:(w,v)∈E} (B[0, w] + c_wv)
      B[1, v] = min(M, M')
    Endfor
    For v ∈ V in any order
      B[0, v] = B[1, v]
    Endfor
  Endfor
  Return B[1, t]
```

It is easy to verify that when this algorithm completes, the array entry $B[1, v]$ holds the value of $OPT(n-1, v)$, the minimum-cost of a path from s to v using at most $n-1$ edges, for all $v \in V$. Moreover, it uses $O(n^3)$ time and only $O(n)$ space. You do not need to prove these facts.

The problem is: where is the shortest path? The usual way to find the path involves tracing back through the $M[i, v]$ values, using the whole matrix M, and we no longer have that. The goal of this problem is to show that if the graph has no negative cycles, then there is enough information saved in the last column of the matrix M, to recover the shortest path in $O(n^2)$ time.

Assume G has no negative or even zero length cycles. Give an algorithm FIND-PATH(t, G, B) that uses only the array B (and the graph G) to find the the minimum-

cost path from s to t in $O(n^2)$ time.

21. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm trades shares in n different companies. For each pair $i \neq j$ they maintain a trade ratio r_{ij} meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional, i.e., $r_{ij} = \frac{2}{3}$ means that you can trade 3 shares of i to get a 2 shares of j.

    A *trading cycle* for a sequence of shares $i_1, i_2, \ldots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than what you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

    Give a polynomial time algorithm that finds such an opportunity cycle, if one exists. Hint: a useful construction not covered in lecture is the augmented graph used in the statement (4.4.7).

22. As we all know, there are many sunny days in Ithaca, NY; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on rainy day. The ranking officer decides to postpone the picnic, and must notify everyone by phone. Here is the mechanism she uses to do this.

    Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus, the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has as a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u. See Figure 1, in which A is the ranking officer, B and D are the direct subordinates of A, and C is the direct subordinate of B.

    To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates one at a time. The process continues this way, until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 1, A would not be allowed to call C.

    Now, we can picture this process as being divided into *rounds*: In one *round*, each

person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates. For example, in Figure 1, it will take only two rounds if A starts by calling B, but it will take three rounds if A starts by calling D.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

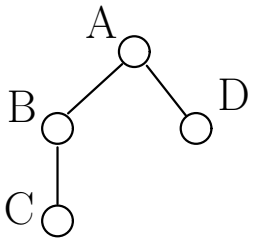


Figure 5.15: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

23. In a word processor, the goal of "pretty-printing" is to take text with a ragged right margin — like this:

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

— and turn it into text whose right margin is as "even" as possible — like this:

```
Call me Ishmael.  Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even." So

suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font, and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word but the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line — it's the number of spaces left at the right margin.

Give an efficient to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines (including the last line) is minimized.

24. You're consulting for a group of people, who would prefer not be mentioned here by name, whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0's and 1's over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

    This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0's and 1's, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a prefix of $x = 101$.

    We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters 1,2,5,7,8,9 form 101101 — a repetition of x — and the remaining characters 3,4,6 form 000 — a repetition of y.

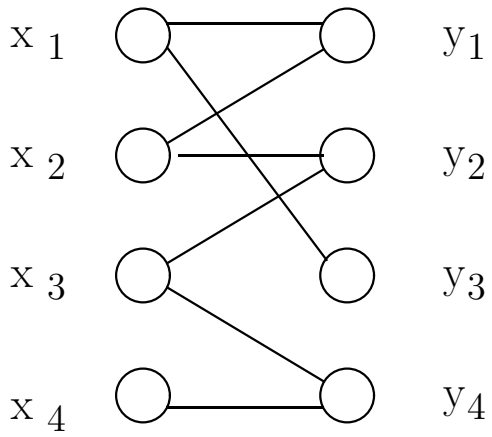
    In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: we want to make sure it "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y, and decides if s is an interleaving of x and y.

Chapter 6

Network Flow

In this chapter, we focus on a rich set of algorithmic problems that grow, in a sense, out of one of the original problems we formulated at the beginning of the course: *bipartite matching*.

Recall the set-up of the bipartite matching problem. A *bipartite graph* $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y. We often draw bipartite graphs as in the figure below, with the nodes in X in a column on the left, the nodes in Y in a column on the right, and each edge crossing from the left column to the right column.



Now, we've already seen the notion of a *matching* at several points in the course: we've used the term to describe collections of pairs over a set, with the property that no element of the set appears in more than one pair. (Think of men (X) matched to women (Y) in the stable matching problem, or characters in a sequence alignment problem.) In the case of a graph, the edges constitute pairs of nodes, and we consequently say that a *matching* in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M. M is a *perfect matching* if every node appears in *exactly* one edge of M.

Matchings in bipartite graphs can model situations in which objects are being *assigned* to other objects. Many such situations were mentioned earlier in the course when we introduced graphs, and bipartite graphs. One natural example arises when the nodes in X represent jobs, the nodes in Y represent machines, and an edge (x_i, y_j) indicates that machine y_j is capable of processing job x_i. A perfect matching is then a way of assigning each job to a machine that can process it, with the property that each machine is assigned exactly one job. Bipartite graphs can represent many other relations that arise between two distinct sets of objects, such as the relation between customers and stores; or houses and nearby fire stations; and so forth.

One of the oldest problems in combinatorial algorithms is that of determining the size of the largest matching in a bipartite graph G. (As a special case, note that G has a perfect matching if and only if $|X| = |Y|$ and it has a matching of size $|X|$.) This problem turns out to be solvable by an algorithm that runs in polynomial time, but the development of this algorithm needs ideas fundamentally different from the techniques that we've seen so far.

Rather than developing the algorithm directly, we begin by formulating a general class of problems — *network flow* problems — that includes bipartite matching as a special case. We then develop a polynomial-time algorithm for a general problem in this class — the *maximum flow* problem — and show how this provides an efficient algorithm for maximum bipartite matching as well.

6.1 The Maximum Flow Problem

One often uses graphs to model *transportation networks* — networks whose edges carry some sort of traffic, and whose nodes act as "switches" passing traffic between different edges. Consider, for example, a highway system in which the edges are highways and the nodes are interchanges; or a computer network, in which the edges are links that can carry packets, and the nodes are switches; or a fluid network, in which edges are pipes that carry water, and the nodes are junctures where pipes are plugged together. Network models of this type have several ingredients: *capacities* on the edges, indicating how much they can carry; *source* nodes in the graph, which generate traffic; *sink* (or destination) nodes in the graph, which can "absorb" traffic as it arrives; and finally, the traffic itself, which is transmitted across the edges.

We'll be considering graphs of this form, and refer to the traffic as *flow* — an abstract entity that is generated at source nodes, transmitted across edges, and absorbed at sink nodes. Formally, we'll say that a *flow network* is a directed graph $G = (V, E)$ with the following features.

- Associated with each edge e is a *capacity*, which is a non-negative number that we denote c_e.

- There is a single *source* node $s \in V$.
- There is a single *sink* node $t \in V$.

Nodes other than s and t will be called *internal* nodes.

We will make two assumptions about the flow networks we deal with — first, that no edge enters the source s and no edge leaves the sink t; and second, that that all capacity values are integers. These assumptions make things cleaner to think about, and while they eliminate a few pathologies, they preserve essentially all the issues we want to think about.

Figure 6.1 gives a picture of a flow network with 4 nodes and 5 edges, and capacity values given next to each edge.

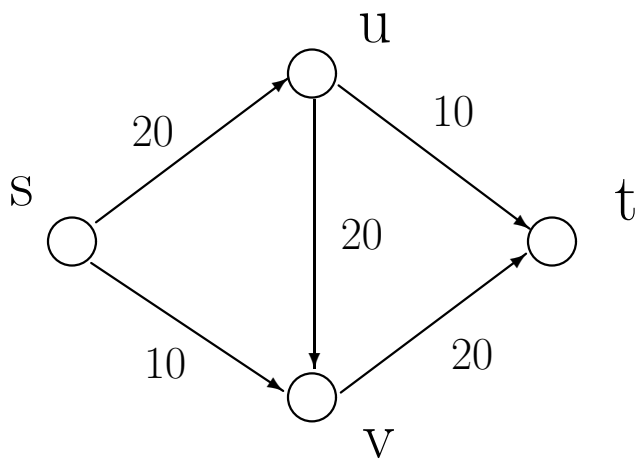


Figure 6.1: A flow network.

Next we define what it means for our network to carry traffic, or flow. We say that an *s-t flow* is a function f that maps each edge e to a non-negative real number, $f : E \to \mathbf{R}^+$; the value $f(e)$ intuitively represents the amount of flow carried by edge e. A flow f must satisfy the following two properties.

(i) *(Capacity conditions.)* For each $e \in E$, $0 \leq f(e) \leq c_e$.

(ii) *(Conservation conditions.)* For each node v other than s and t, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

Here $\sum_{e \text{ into } v} f(e)$ sums the flow value $f(e)$ over all edges entering node v, while $\sum_{e \text{ out of } v} f(e)$ is the sum of flow values over all edges leaving node v.

Thus, the flow on an edge cannot exceed the capacity of the edge. For every node other than the source and the sink, the amount of flow entering must equal the amount of flow

leaving. The source has no entering edges (by our assumption), but it is allowed to have flow going out; in other words, it can generate flow. Symmetrically, the sink is allowed to have flow coming in, even though it has no edges leaving it. The *value* of a flow f, denoted $\nu(f)$, is defined to be the amount of flow generated at the source:

$$\nu(f) = \sum_{e \text{ out of } s} f(e).$$

To make the notation a little more compact, we define $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$ and $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$. We can extend this to sets of vertices; if $S \subseteq V$, we define $f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e)$ and $f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$. In this terminology, the conservation condition for nodes $v \neq s, t$ becomes $f^{\text{in}}(v) = f^{\text{out}}(v)$; and we can write $\nu(f) = f^{\text{out}}(s)$.

Given a flow network, a natural goal is to arrange the traffic so as to make as efficient use of the available capacity as possible. Thus, the basic algorithmic problem we will consider is the following: given a flow network, find a flow of maximum possible value.

As we think about designing algorithms for this problem, it's useful to consider how the structure of the flow network places upper bounds on the maximum value of an s-t flow. Here is a basic "obstacle" to the existence of large flows. Suppose we divide the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. Then, intuitively, any quantum of flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such "cut" of the graph puts a bound on the maximum possible flow value. The maximum flow algorithm that we develop here will be intertwined with a proof that the maximum flow value equals the minimum capacity of any such division, called the minimum cut. As a bonus, our algorithm will also compute the minimum cut. We will see that the problem of finding cuts of minimum capacity in a flow network turns out to be at least as valuable, from the point of view of applications, as that of finding a maximum flow.

6.2 Computing Maximum Flows

Suppose we wanted to find a maximum flow in a network; how should we go about doing this? It takes some testing out to decide that an approach such as dynamic programming doesn't seem to work — at least, there is no algorithm known for the maximum flow problem that could really be viewed as naturally belonging to the dynamic programming paradigm. In the absence of other ideas, we could go back and think about simple greedy approaches, to see where they break down.

Suppose we start with zero flow: $f(e) = 0$ for all e. Clearly this respects the capacity and conservation conditions; the problem is that its value is 0. We now try to increase the

value of f by "pushing" flow along a path from s to t, up to the limits imposed by the edge capacities. Thus, in the figure above, we might choose the path consisting of the edges $\{(s,u),(u,v),(v,t)\}$ and increase the flow on each of these edges to 20, and $f(e) = 0$ for the other two. In this way, we still respect the capacity conditions — since we only set the flow as high as the edge capacities would allow — and the conservation conditions — since when we increase flow on an edge entering an internal node, we also increase it on an edge leaving the node. Now the value of our flow is 20, and we can ask: is this the maximum possible for the graph in the figure? If we think about it, we see that the answer is "no," since it is possible to construct a flow of value 30. The problem is that we're now stuck — there is no s-t path on which we can directly push flow without exceeding some capacity — and yet we do not have a maximum flow. What we need is a more general way of pushing flow from s to t, so that in a situation such as this, we have a way to increase the value of the current flow.

Essentially, we'd like to perform the following operation. We push 10 units of flow along (s,v); this now results in too much flow coming into v. So we "undo" 10 units of flow on (u,v); this restores the conservation condition at v, but results in too little flow leaving u. So, finally, we push 10 units of flow along (u,t), restoring the conservation condition at u. We now have a valid flow, and its value is 30. See Figure 6.2 where the dark edges are carrying flow before the operation, and the dashed edges form the new kind of augmentation.

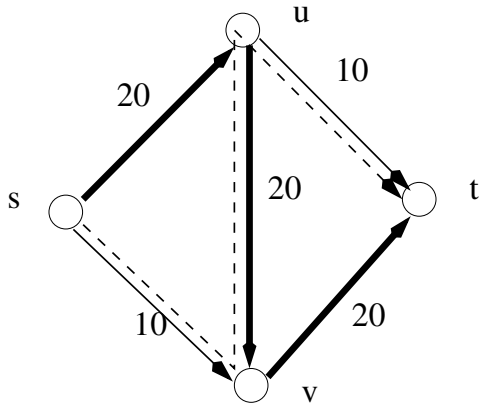


Figure 6.2: Augmenting flow using the edge (u,v) backwards.

This is a more general way of pushing flow: we can push *forward* on edges with leftover capacity, and we can push *backward* on edges that are already carrying flow, to divert it in a different direction. We now define the *residual graph*, which provides a systematic way to search for forward-backward operations such as this.

Given a flow network G, and a flow f on G, we define the *residual graph* G_f of G with respect to f as follows.

- The node set of G_f is the same as that of G.

- For each edge $e = (u, v)$ of G on which $f(e) < c_e$, there are $c_e - f(e)$ "leftover" units of capacity on which we could try pushing flow forwards. So we include the edge $e = (u, v)$ in G_f, with a capacity of $c_e - f(e)$. We will call edges included this way *forward edges*.

- For each edge $e = (u, v)$ of G on which $f(e) > 0$, there are $f(e)$ units of flow that we can "undo" if we want to, by pushing flow backward. So we include the edge $e' = (v, u)$ in G_f, with a capacity of $f(e)$. Note that e' has the same ends as e, but its direction is reversed; we will call edges included this way *backward edges*.

This completes the definition of the residual graph G_f. Note that each edge e in G can give rise to one or two edges in G_f: if $0 < f(e) < c_e$ it results in both a forward edge and a backward edge being included in G_f. Thus, G_f has at most twice as many edges as G. We will sometimes refer to the capacity of an edge in the residual graph as a *residual capacity*, to help distinguish it from the capacity of the corresponding edge in the original flow network G.

Now we want to make precise the way in which we "push" flow from s to t in G_f. Let P be a simple s-t path in G_f — i.e. P does not visit any node more than once. We define `bottleneck`(P) to be the minimum residual capacity of any edge on P. We now define the following operation `augment`(f, P), which yields a new flow f' in G.

```
augment(f, P)
  Let b = bottleneck(P).
  For each edge e ∈ P
    If e is a forward edge then
      increase f(e) in G by b.
    Else (e is a backward edge)
      decrease f(e) in G by b.
    Endif
  Endfor
  Return(f)
```

It was purely to be able to perform this operation that we defined the residual graph; to reflect the importance of `augment`, one often refers to any s-t path in the residual graph as an *augmenting path*.

The result of `augment`(f, P) is a new flow f' in G, obtained by increasing and decreasing the flow values on edges of P. Let us first verify that f' is indeed a flow.

(6.1) *f' is a flow in G.*

Proof. We must verify the capacity and conservation conditions.

Since f' differs from f only on edges of P, so we need check the capacity conditions only on these edges. Thus, let e be an edge of P. Informally, the capacity condition continues to

hold because if e is a forward edge, we specifically avoided increasing the flow on e above c_e; and if e is a backward edge, we specifically avoided decreasing the flow on e below 0. More concretely, note that $\texttt{bottleneck}(P)$ is no larger than the residual capacity of e. If e is a forward edge, then its residual capacity is $c_e - f(e)$; thus we have

$$0 \leq f(e) \leq f'(e) = f(e) + \texttt{bottleneck}(P) \leq f(e) + (c_e - f(e)) = c_e,$$

so the capacity condition holds. If e is a backward edge, then its residual capacity is $f(e)$, so we have

$$c_e \geq f(e) \geq f'(e) = f(e) - \texttt{bottleneck}(P) \geq f(e) - f(e) = 0,$$

and again the capacity condition holds.

We need to check the conservation condition at each internal node that lies on the path P. Let v be such a node; we can verify that the change in the amount of flow entering v is the same as the change in the amount of flow exiting v; since f satisfied the conservation condition at v, so must f'. Technically there are four cases to check, depending on whether the edge of P that enters v is a forward or backward edge, and whether the edge of P that exits v is a forward or backward edge. However, each of these cases is easily worked out, and we leave them to the reader. ■

This augmentation operation captures the type of forward and backward pushing of flow that we discussed earlier. Let's now consider the following algorithm to compute an s-t flow in G.

```
Max-Flow (G, s, t, c)
  Initially f(e) = 0 for all e in G.
  While there is an s-t path in the residual graph G_f
    Let P be a simple s-t path in G_f
    f' = augment(f, P)
    Update f to be f'
    Update the residual graph G_f to be G_{f'}
  Endwhile
  Return f
```

We'll call this the *Ford-Fulkerson algorithm* (or, briefly, the *F-F algorithm*), since it was developed by Ford and Fulkerson in 1956. The F-F algorithm is really quite simple. What is not at all clear is whether its central `While` loop terminates, and whether the flow returned is a maximum flow. The answers to both of these questions turn out to be a little subtle.

First, consider some properties that the algorithm maintains by induction on the number of iterations of the `While` loop, relying on our assumption that all capacities are integers.

(6.2) *At every intermediate stage of the F-F algorithm, the flow values $\{f(e)\}$ and the residual capacities in G_f are integers.*

Proof. The statement is clearly true before any iterations of the `While` loop. Now suppose it is true after j iterations. Then since all residual capacities in G_f are integers, the value `bottleneck`(P) for the augmenting path found in iteration $j+1$ will be an integer. Thus the flow f' will have integer values, and hence so will the capacities of the new residual graph. ■

We can use this property to prove that the F-F algorithm terminates. As at previous points in the course, we will look for a measure of *progress* that will imply termination.

First, we show that the flow value strictly increases when we apply an augmentation.

(6.3) *Let f be a flow in G, and let P be a simple s-t path in G_f. Then $\nu(f') = \nu(f) +$* `bottleneck`(P)*; and since* `bottleneck`$(P) > 0$*, we have $\nu(f') > \nu(f)$.*

Proof. The first edge e of P must be an edge out of s in the residual graph G_f; and since the path is simple, it does not visit s again. Since G has no edges entering s, the edge e must be a forward edge. We increase the flow on this edge by `bottleneck`(P), and we do not change the flow on any other edge incident to s. Therefore the value of f' exceeds the value of f by `bottleneck`(P). ■

We need one more observation to prove termination: we need to be able to bound the maximum possible flow value. Here's one upper bound: even if all the edges out of s could be completely saturated with flow, the value of the flow would be $\sum_{e \text{ out of } s} c_e$. Let C denote this sum. Thus we have $\nu(f) \leq C$ for all s-t flows f. Using statement (6.3), we can now prove termination.

(6.4) *Suppose, as above, that all capacities in the flow network G are integers. Then the F-F algorithm terminates in at most C iterations of the* `While` *loop.*

Proof. Note that C is the value of the cut where $A = \{s\}$ and $B = V - \{s\}$ has all other nodes. By the capacity condition, we know that no flow in G can have a value greater than C. (C may be a huge overestimate of the maximum value of a flow in G, but it's handy for us as a finite, simply stated bound.)

Now, by (6.3), the value of the flow maintained by the F-F algorithm increases in each iteration; so by (6.2), it increases by at least 1 in each iteration. Since it starts with the value 0, and cannot go higher than C, the `While` loop in the F-F algorithm can run for at most C iterations. ■

Next, we consider the running time of the F-F algorithm. Let n denote the number of nodes in G, m denote the number of edges in G. We will assume that $m \geq n - 1$; this assumption is true, for example, if there are paths in G from s to every other node.

(6.5) *Suppose, as above, that all capacities in the flow network G are integers. Then the F-F algorithm can be implemented to run in $O(mC)$ time.*

Proof. We know from (6.4) that the algorithm terminates in at most C iterations of the `While` loop. We therefore consider the amount of work involved in one iteration, when the current flow is f.

The residual graph G_f has at most $2m$ edges, since each edge of G gives rise to at most two edges in the residual graph. We will maintain it using two linked lists for each node v, one containing the edges entering v, and one containing the edges exiting v. To find an s-t path in G_f, we can use breadth-first search or depth-first search, which run in $O(m+n)$ time; by our assumption that $m \geq n-1$, $O(m+n)$ is the same as $O(m)$. The procedure `augment`(f, P) takes time $O(n)$, as the path P has at most $n-1$ edges. Given the new flow f', we can build the new residual graph in $O(m)$ time: For each edge e of G, we construct the correct forward and backwards edges in $G_{f'}$. ■

A somewhat more efficient version of the algorithm would maintain the linked lists of edges in the residual graph G_f as part of the `augment` procedure that changes the flow f via augmentation.

6.3 Cuts in a Flow Network

Our next goal is to show that the flow that is returned by the F-F algorithm has the maximum possible value of any flow in G. To make progress towards this goal, we return to an issue that we raised in Section 6.1 — the way in which the structure of the flow network places upper bounds on the maximum value of an s-t flow. We have already seen one upper bound: the value $\nu(f)$, of any s-t-flow f, is at most $C = \sum_{e \text{ out of } s} c_e$. Sometimes this bound is useful, but sometimes it is very weak. We now use the notion of a *cut* to develop a much more general means of placing upper bounds on the maximum flow value.

Consider dividing the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. As in our discussion at the end of Section 6.1, any such division places an upper bound on the maximum possible flow value, since all the flow must cross from A to B somewhere. Formally, we say that an *s-t cut* is a partition (A, B) of the vertex set V, so that $s \in A$ and $t \in B$. The *capacity* of a cut (A, B), which we will denote $c(A, B)$, is simply the sum of the capacities of all edges out of A: $c(A, B) = \sum_{e \text{ out of } A} c_e$.

Cuts turn out to provide very natural upper bounds on the values of flows, as expressed by our intuition above. We make this precise via a sequence of facts.

(6.6) *Let f be any s-t flow, and (A, B) any s-t cut. Then $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.*

This statement is actually much stronger than a simple upper bound: it says that by watching the amount of flow f sends across a cut, we can exactly *measure* the flow value: it is the

total amount that leaves A, minus the amount that "swirls back" into A. This makes sense intuitively, although the proof requires a little manipulation of sums.

Proof of (6.6). We know that $\nu(f) = f^{\text{out}}(s)$. By assumption we have the $f^{\text{in}}(s) = 0$, as the source s has no entering edges, so we can write $\nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Since every node v in A other than s is internal, we know that $f^{\text{out}}(v) - f^{\text{in}}(v) = 0$ for all such nodes. Thus,

$$\nu(f) = \sum_{v \in A} f^{\text{out}}(v) - f^{\text{in}}(v),$$

since the only term in this sum that is non-zero is the one in which v is set to s.

Let's try to rewrite the sum on the right as follows. If an edge e has both ends in A, then $f(e)$ appears once in the sum with a "+" and once with a "−", and hence these two terms cancel out. If e has only its tail in A, then $f(e)$ appears just once in the sum, with a "+". If e has only its head in A, then $f(e)$ also appears just once in the sum, with a "−". Finally, if e has neither end in A, then $f(e)$ doesn't appear in the sum at all. In view of this, we have

$$\sum_{v \in A} f^{\text{out}}(v) - f^{\text{in}}(v) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

Putting together these two equations, we have the statement of (6.6). ■

If $A = \{s\}$ then $f^{\text{out}}(A) = f^{\text{out}}(s)$, and $f^{\text{in}}(A) = 0$ as there are no edges entering the source by assumption. So the statement for this set $A = \{s\}$ is exactly the definition of the flow value $\nu(f)$.

Note that if (A, B) is a cut, then the edges into B are precisely the edges out of A. Similarly the edges out of B are precisely the edges into A. Thus we have $f^{\text{out}}(A) = f^{\text{in}}(B)$ and $f^{\text{in}}(A) = f^{\text{out}}(B)$, just by comparing the definitions for these two expressions. So we can rephrase (6.7) in the following way.

(6.7) *Let f be any s-t flow, and (A, B) any s-t cut. Then $\nu(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.*

If we set $A = V - \{t\}$ and $B = \{t\}$ in (6.7), we have $\nu(f) = f^{\text{in}}(B) - f^{\text{out}}(B) = f^{\text{in}}(t) - f^{\text{out}}(t)$. By our assumption the sink t has no leaving edges, so we have that $f^{\text{out}}(t) = 0$. This says that we could have originally defined the *value* of a flow equally well in terms of the sink t: it is the amount of flow arriving at the sink.

A very useful consequence of (6.6) is the following upper bound.

(6.8) *Let f be any s-t flow, and (A, B) any s-t cut. Then $\nu(f) \leq c(A, B)$.*

Proof.

$$\begin{aligned} \nu(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\ &\leq f^{\text{out}}(A) \end{aligned}$$

$$\begin{aligned} &= \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c_e \\ &= c(A, B). \end{aligned}$$

Here, the first line is simply (6.6); we pass from the first to the second since $f^{\text{in}}(A) \geq 0$, and we pass from the third to the fourth by applying the capacity conditions to each term of the sum. ■

In a sense, (6.8) looks weaker than (6.6), since it is only an inequality rather than an equality. However, it will be extremely useful for us, since its right-hand-side is independent of any particular flow f. What (6.8) says is that *the value of every flow is upper-bounded by the capacity of every cut.* In other words, if we exhibit any s-t cut in G of some value c^*, we know immediately by (6.8) that there cannot be an s-t flow in G of value greater than c^*. Conversely, if we exhibit any s-t flow in G of some value ν^*, we know immediately by (6.8) that there cannot be an s-t cut in G of value less than ν^*.

6.4 Max-Flow Equals Min-Cut

Let $\overline{f}$ denote the flow that is returned by the F-F algorithm. We want to show that $\overline{f}$ has the maximum possible value of any flow in G, and we do this by the method discussed at the end of the previous section: we exhibit an s-t cut (A^*, B^*) for which $\nu(\overline{f}) = c(A^*, B^*)$. This immediately establishes that $\overline{f}$ has the maximum value of any flow, and that (A^*, B^*) has the minimum capacity of any s-t cut.

The F-F algorithm terminates when the flow f has no s-t path in the residual graph G_f. This turns out to be the only property needed for proving its maximality.

(6.9) *If f is an s-t-flow such that there is no s-t path in the residual graph G_f, then there is an s-t cut (A^*, B^*) in G for which $\nu(f) = c(A^*, B^*)$. Consequently, f has the maximum value of any flow in G.*

Proof. The statement claims the existence of a cut satisfying a certain desirable property; thus, we must now identify such a cut. To this end, let A^* denote the set of all nodes v in G for which there is an s-v path in G_f. Let B^* denote the set of all other nodes: $B^* = V - A^*$.

First, we establish that (A^*, B^*) is indeed an s-t cut. It is clearly a partition of V. $s \in A^*$ since there is always a path from s to s. Moreover, $t \notin A^*$ by the assumption that there is no s-t path in the residual graph; hence $t \in B^*$ as desired.

Next, suppose that $e = (u, v)$ is an edge in G for which $u \in A^*$ and $v \in B^*$. We claim that $f(e) = c_e$. For if not, e would be a forward edge in the residual graph G_f. Since $u \in A^*$,

there is an s-u path in G_f; appending e to this path, we would obtain an s-v path in $G_{\overline{f}}$, contradicting our assumption that $v \in B^*$.

Now suppose that $e' = (u', v')$ is an edge in G for which $u' \in B^*$ and $v' \in A^*$. We claim that $f(e') = 0$. For if not, e' would give rise to a backward edge $e'' = (v', u')$ in the residual graph G_f. Since $v' \in A^*$, there is an s-v' path in G_f; appending e'' to this path, we would obtain an s-u' path in G_f, contradicting our assumption that $u' \in B^*$.

So all edges out of A^* are completely saturated with flow, while all edges into A^* are completely unused. We can now use (6.6) to reach the desired conclusion:

$$\begin{aligned} \nu(f) &= \overline{f}^{\text{out}}(A^*) - \overline{f}^{\text{in}}(A) \\ &= \sum_{e \text{ out of } A^*} \overline{f}(e) - \sum_{e \text{ into } A^*} \overline{f}(e) \\ &= \sum_{e \text{ out of } A^*} c_e - 0 \\ &= c(A^*, B^*). \end{aligned}$$

■

Note how, in retrospect, we can see why the two types of residual edges — forward and backward — are crucial in analyzing the two terms in the expression from (6.6), which we use to establish that the flow $\overline{f}$, obtained by the F-F algorithm, is a maximum flow.

As a bonus, we have obtained the following amazing fact through the analysis of the algorithm.

(6.10) *In every flow network, there is a flow f^* and a cut (A^*, B^*) so that $\nu(f^*) = c(A^*, B^*)$.*

The point is that f^* in (6.10) must be a maximum s-t-flow; for if there were a flow f' of greater value, the value of f' would exceed the capacity of (A^*, B^*), and this would contradict (6.8). Similarly, it follows that (A^*, B^*) in (6.10) is a *minimum cut* — no other cut can have smaller capacity — for if there were a cut (A', B') of smaller capacity, it would be less than the value of f^*, and this again would contradict (6.8). Due to these implications, (6.10) is often called the *Max-Flow Min-Cut Theorem*, and phrased as follows.

(6.11) *In every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t.*

We also observe that our algorithm can easily be extended to compute a minimum s-t-cut (A^*, B^*), as follows.

(6.12) *Given a flow $\overline{f}$ of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.*

Proof. We simply follow the construction in the proof of (6.9). We construct the residual graph $G_{\overline{f}}$, and perform breadth-first search or depth-first search to determine the set A^* of all nodes that s can reach. We then define $B^* = V - A^*$, and return the cut (A^*, B^*). ■

Note that there can be many minimum-capacity cuts in a graph G; the procedure in the proof (6.5) is simply finding a particular one of these cuts, starting from a maximum flow $\overline{f}$.

Integer-Valued Flows. Among the many corollaries emerging from our analysis of the F-F algorithm, here is another extremely important one. By (6.2), we maintain an integer-valued flow at all times, and by (6.9), we conclude with a maximum flow. Thus we have

(6.13) *If all capacities in the flow network are integers, then there is a maximum flow f for which every flow value $f(e)$ is an integer.*

Note that (6.13) does not claim that *every* maximum flow is integer-valued; only that *some* maximum flow has this property. Curiously, although (6.13) makes no reference to the F-F algorithm, our algorithmic approach here provides what is probably the easiest way to prove it.

Real numbers as capacities? Finally, before moving on, we can ask how crucial our assumption of integer capacities was. (Ignoring (6.13) and (6.5), which clearly needed it.) First, we notice that allowing capacities to be equal to rational numbers does not make the situation any more general, since we always determine the least common multiple of all capacities, and multiply them all by this value to obtain an equivalent problem with integer capacities.

But what if we have real numbers as capacities? Where in the the proof did we rely on the capacities being integers? In fact, we relied on it quite crucially: we used (6.2) to establish, in (6.4), that the value of the flow increased by at least 1 in every step. With real numbers as capacities, we should be concerned that the value of our flow keeps increasing, but in increments that become arbitrarily smaller and smaller; and hence we have no guarantee that the number of iterations of the loop is finite. And this turns out to be an extremely real worry, for the following reason: *With pathological choices for the augmenting path, the F-F algorithm with real-valued capacities can run forever.*

However, one can still prove that the Max-Flow Min-Cut Theorem (6.10) is true even if the capacities may be real numbers. Note that (6.9) assumed only that the flow f has no s-t-path in its residual graph G_f, in order to conclude that there is an s-t-cut of equal value. Clearly, for any flow f of maximum value, the residual graph has no s-t-path — otherwise there would be a way to increase the value of the flow. So one can prove (6.10) in the case

of real-valued capacities by simply establishing that for every flow network, there exists a maximum flow.

Capacities in any real application are integers or rational numbers. However, the problem of pathological choices for the augmenting paths can manifest itself even with integer capacities: it can make the F-F algorithm take a gigantic number of iterations. In the next section, we discuss how to select augmenting paths so as to avoid the potential bad behavior of the algorithm.

6.5 Choosing Good Augmenting Paths

In the previous section, we saw that any way of choosing an augmenting path increases the value of the flow, and this led to an $O(C)$ bound on the number of augmentations, where $C = \sum_{e \text{ out of } s} c_e$. When C is not very large, this can be a reasonable bound; however, in general it is very weak.

To get a sense for how bad this bound can be, consider the example graph in the beginning of this chapter; but this time assume the capacities are as follows: the edges (s, v), (s, u), (v, t) and (u, t) have capacity 100, and the edge (u, v) has capacity 1. It is easy to see that the maximum flow has value 200, and has $f(e) = 100$ for the edges $(s, v), (s, u), (v, t)$ and (u, t) and value 0 on the edge (u, v). This flow can be obtained by a sequence of 2 augmentations, using the paths of nodes s, u, t and path s, v, t. But consider how bad the F-F algorithm can be with pathological choices for the augmenting paths. Suppose we start with augmenting path P_1 of nodes s, u, v, t in this order. This path has $\texttt{bottleneck}(P_1) = 1$. After this augmentation we have $f(e) = 1$ on the edge $e = (u, v)$, so the reverse edge is in the residual graph. For the next augmenting path we choose the path P_2 of the nodes s, v, u, t in this order. In this second augmentation we get $\texttt{bottleneck}(P_2) = 1$ as well. After this second augmentation we have $f(e) = 0$ for the edge $e = (u, v)$, so the edge is again in the residual graph. Suppose we alternate between choosing P_1 and P_2 for augmentation. In this case each augmentation will have 1 as the bottleneck capacity, and it will take 200 augmentations to get the desired flow of value 200. This is exactly the bound we proved in (6.4).

The goal of this section is to show that with a better choice of paths we can improve this bound significantly. A large amount of work has been devoted to finding good ways of choosing augmenting paths in the maximum flow problem, so as terminate in as few iterations as possible. We focus here on one of the most natural approaches. Recall that augmentation increases the value of the maximum flow by the bottleneck capacity of the selected path; so if we choose paths with large bottleneck capacity, we will be making a lot of progress. A natural idea is to select the path that has the largest bottleneck capacity. Selecting such a path at each iteration can slow down the iterations by quite a bit. We will avoid this slowdown by not worrying about selecting the path that has *exactly* the largest

bottleneck capacity. Instead, we will maintain a so-called *scaling parameter* Δ, and we will look for paths that have bottleneck capacity at least Δ.

Let $G_f(\Delta)$ be the subset of the residual graph consisting only of edges with residual capacity at least Δ. We will work with values of Δ that are powers of 2. The algorithm is as follows.

```
Scaling Max-Flow(G, c)
  Initially f(e) = 0 for all e in G.
  Initially set Δ to be the largest power of 2
        that is no larger than the maximum capacity out of s:
        Δ ≤ max_{e out of s} c_e.
    While Δ ≥ 1
       While there is an s-t path in the graph G_f(Δ)
          Let P be a simple s-t path in G_f(Δ)
          f' = augment(f, P)
          Update f to be f'
       Endwhile
       Δ = Δ/2
    Endwhile
Return f
```

First observe that the new `Scaling Max-Flow` algorithm is really just an implementation of the original `Ford-Fulkerson` algorithm. The new loops, the value Δ, and the restricted residual graph $G_f(\Delta)$ is only used to guide the selection of residual path — with the goal of using edges with large residual capacity for as long as possible. Hence, all the properties that we proved about the original `Max-Flow` algorithm are also true for this new version: the flow remains integer-valued throughout the algorithm, and hence all residual capacities are integer-valued.

(6.14) *If the capacities are integer-valued, then throughout the* `Scaling Max-Flow` *algorithm the flow and the residual capacities remain integer-valued. This implies that when* $\Delta = 1$, $G_f(\Delta)$ *is the same as* G_f, *hence when the algorithm terminates the flow* f *is of maximum value.*

Next we consider the running time. We call an iteration of the outside `While` loop — with a fixed value of Δ — the Δ-*scaling phase*. It is easy to give an upper bound on the number of different Δ-scaling phases, in terms of the value $C = \sum_{e \text{ out of } s} c_e$ that we also used in the previous section. The initial value of Δ is at most C, it drops by factors of 2, and it never gets below 1. Thus,

(6.15) *The number of iterations of the outside* `While` *loop is at most* $\lceil \log_2 C \rceil$.

The harder part is to bound the number of augmentations done in each scaling phase. The idea here is that we are using paths that augment the flow by a lot, and so there should be relatively few augmentations. During the Δ-scaling phase we only use edges with residual capacity at least Δ. Using (6.3), we have

(6.16) *During the Δ-scaling phase, each augmentation increases the flow value by at least Δ.*

The key insight is that at the end of the Δ-scaling phase, the flow f cannot be not too far from the maximum possible value.

(6.17) *Let f be the flow at the end of the Δ-scaling phase. There is an s-t cut (A^*, B^*) in G for which $c(A^*, B^*) \leq \nu(f) + m\Delta$, where m is the number of edges in the graph G. Consequently, the maximum flow in the network has value at most $\nu(f) + m\Delta$.*

Proof. This proof is analogous to our proof of (6.9), which established that the flow returned by the original `Max-Flow` algorithm is of maximum value.

As in that proof, we must identify the cut promised in the first statement above. Let A^* denote the set of all nodes v in G for which there is an s-v path in $G_f(\Delta)$. Let B^* denote the set of all other nodes: $B^* = V - A^*$. We can see that (A^*, B^*) is indeed an s-t cut as otherwise the phase would not have ended.

Now consider an edge $e = (u, v)$ in G for which $u \in A^*$ and $v \in B^*$. We claim that $c_e < f(e) + \Delta$, for if not, e would be a forward edge in the graph $G_f(\Delta)$. Since $u \in A^*$, there is an s-u path in $G_f(\Delta)$; appending e to this path, we would obtain an s-v path in $G_f(\Delta)$, contradicting our assumption that $v \in B^*$. Similarly, we get that for any edge $e' = (u', v')$ in G for which $u' \in B^*$ and $v' \in A^*$, $f(e') < \Delta$. For if not, e' would give rise to a backward edge $e'' = (v', u')$ in the graph $G_f(\Delta)$. Since $v' \in A^*$, there is an s-v' path in $G_f(\Delta)$; appending e'' to this path, we would obtain an s-u' path in $G_f(\Delta)$, contradicting our assumption that $u' \in B^*$.

So all edges e out of A^* are almost saturated — they satisfy $c_e < f(e) + \Delta$ — and all edges into A^* are almost empty — they satisfy $f(e) < \Delta$. We can now use (6.6) to reach the desired conclusion:

$$\begin{aligned}
\nu(f) &= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\
&\geq \sum_{e \text{ out of } A^*} (c_e - \Delta) - \sum_{e \text{ into } A^*} \Delta \\
&= \sum_{e \text{ out of } A^*} c_e - \sum_{e \text{ out of } A^*} \Delta - \sum_{e \text{ into } A^*} \Delta \\
&\geq c(A^*, B^*) - m\Delta.
\end{aligned}$$

Here the first inequality follows from our bounds on the flow values of edges across the cut, and the second inequality follows from the simple fact that the graph only contains m edges total.

The maximum flow value is bounded by the capacity of any cut. We use the cut (A^*, B^*) to obtain the bound claimed in the second statement. ■

(6.18) *The number of augmentations in a scaling phase is at most $2m$.*

Proof. The statement is clearly true in the first scaling phase — we can use each of the edges out of s only for at most one augmentation in that phase. Now consider a later scaling phase Δ, and let f_0 be the flow at the end of the *previous* scaling phase. In that phase we used $\Delta' = 2\Delta$ as our parameter. By (6.17) the maximum flow is at most $\nu(f)+m\Delta' = \nu(f)+2m\Delta$. In the Δ-scaling phase, each augmentation increases the flow by at least Δ, and hence there can be at most $2m$ augmentations. ■

An augmentation takes $O(m)$ time, including the time required to set up the graph and find the appropriate path. We have at most $\lceil \log_2 C \rceil$ scaling phases, and at most $2m$ augmentations in each scaling phase. Thus we have the following result.

(6.19) *The* `Scaling Max-Flow` *algorithm in a graph with m edges and integer capacities finds a maximum flow in at most $2m\lceil \log_2 C \rceil$ augmentations. It can be implemented to run in at most $O(m^2 \log_2 C)$ time.*

When C is large, this time bound is much better than the $O(mC)$ bound that applied to an arbitrary implementation of the F-F algorithm. Consider that in our example at the beginning of this section, we had capacities of size 100; but we could just as well have used capacities of size 2^{100}; in this case, the generic F-F algorithm could take time proportional to 2^{100}, while the scaling algorithm will take time proportional to $\log_2(2^{100}) = 100$. One way to view this distinction is as follows: the generic F-F algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the number of *bits* needed to specify the capacities in the input to the problem. As a result, the scaling algorithm is running in time polynomial in the size of the input — i.e. the number of edges and the numerical representation of the capacities — and so it meets our traditional goal of achieving a polynomial-time algorithm. Bad implementations of the F-F algorithm, which require time $\Omega(mC)$, do not meet this standard of polynomiality.

Could we ask for something qualitatively better than what the scaling algorithm guarantees? Here is one thing we could hope for: our example graph had 4 nodes and 5 edges; so it would be nice to run in time polynomial in the numbers 4 and 5, completely independently of the values of the capacities (except for having to do arithmetic operations using these numbers). Such an algorithm, which is polynomial in $|V|$ and $|E|$ independent of the

numerical values assigned to the edges, is called a *strongly polynomial algorithm*. In fact, the first polynomial algorithms for the maximum flow problem, discovered independently by Dinitz and by Edmonds and Karp, were also strongly polynomial; and had running times of $O(mn^2)$ and $O(m^2n)$ respectively. Both algorithms were based on the F-F algorithm and used in each iteration the augmenting paths with fewest edges. There has since been a huge amount of work devoted to improving the running times of maximum flow algorithms; there are currently algorithms that achieve running times of $O(mn \log n)$, $O(n^3)$, and $O(\min(n^{2/3}m^{1/2})m \log n \log C)$, where the last bound assumes that all capacities are integral and at most C.

6.6 The Preflow-Push Maximum Flow Algorithm

From the very beginning, our discussion of the maximum flow problem has been centered around the idea of an augmenting path in the residual graph — but in fact, there are some very powerful techniques for maximum flow that are not explicitly based on augmenting paths. In this section we study one such technique, the `Preflow-Push` algorithm.

Algorithms based on augmenting paths maintain a flow f, and use the `augment` procedure to increase the value of the flow. By way of contrast, the `Preflow-Push` algorithm will, in essence, increase the flow on an edge-by-edge basis. Changing the flow on a single edge is will typically violate the conservation condition, and so the algorithm will have to maintain something less well-behaved than a flow — something that does not obey conservation — as it operates.

We say that an *s-t preflow* (*preflow*, for short) is a function f that maps each edge e to a non-negative real number, $f : E \to \mathbf{R}^+$. A preflow f must satisfy the capacity conditions:

(i) For each $e \in E$, $0 \leq f(e) \leq c_e$.

In place of the conservation conditions, we require only inequalities: each node other than s must have at least as much flow entering as leaving:

(ii) For each node v other than the source s, we have

$$\sum_{e \text{ into } v} f(e) \geq \sum_{e \text{ out of } v} f(e).$$

We will call the difference

$$e_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$$

the *excess* of preflow at node v. Notice, that a preflow where all nodes other than s and t have zero excess is a flow, and the value of the flow is exactly $e_f(t)$. We can still define the

concept of a residual graph G_f for a preflow f, just as we did for a flow. The algorithm will "push" flow along edges of the residual graph (using both forward and backward edges).

The `Preflow-Push` algorithm will maintain a preflow and work on converting the preflow into an flow. There is a nice physical intuition behind the algorithm. We will also assign each node v a label $h(v)$ that we will think of as the *height* of that node, and will push flow from nodes with higher label to those with lower labels, following the intuition that fluid flows downhill. To make this precise, a *labeling* is a function $h : V \to \mathbf{Z}_{\geq 0}$ from the nodes to the non-negative integers. We will also refer to the labels as *heights* of the nodes. We will say that a labeling h and a s-t-preflow f are *compatible*, if

(i) (*Source and sink conditions.*) $h(t) = 0$ and $h(s) = n$,

(ii) (*Steepness conditions.*) For all edges $(v, w) \in E_f$ in the residual graph $h(v) \leq h(w)+1$.

Intuitively, the height difference n between the source and the sink is meant to ensure that the flow starts high enough to flow from s towards the sink t, while the steepness condition will help by making the descent of the flow gradual enough to make it to the sink.

The key property of a compatible labeling preflow and labeling is that there can be no s-t path in the residual graph.

(6.20) *If s-t-preflow f is compatible with a labeling h, than there is no s-t-path in the residual graph G_f.*

Proof. We prove the statement by contradiction. Let P be a simple s-t-path in the residual graph G. Assume that the nodes along P are $s, v_1, \ldots, v_k = t$. By definition of a labeling compatible with preflow f we have that $h(s) = n$. The edge (s, v_1) is in the residual graph, hence $h(v_1) \geq h(s) - 1 = n - 1$. Using induction i and using the steepness condition for the edge (v_{i-1}, v_i), we get that for all nodes v_i in path P the height is at least $h(v_i) \geq n - i$. Notice that the last node of the path is $v_k = t$, hence we get that $h(t) \geq n - k$. However, $h(t) = 0$ by definition; and $k < n$ as the path P is simple. This contradiction proves the claim. ■

Recall from (6.9) that if there is no s-t path in the residual graph G_f of a *flow* f than the flow has maximum value. This implies the following corollary.

(6.21) *If s-t-flow f is compatible with a labeling h, than f is a flow of maximum value.*

Note that (6.20) applies to preflows, while (6.21) is more restrictive in that it applies only to flows. Thus `Preflow-Push` algorithm will maintain a preflow f and a labeling h compatible with f, and it will work on modifying f and h so as to move f toward being a flow. Once f actually becomes a flow, we can invoke (6.21) to conclude that it is maximum. In light of this, we can view the `Preflow-Push` algorithm as being in a way orthogonal to

the Ford-Fulkerson algorithm. The Ford-Fulkerson algorithm maintains a feasible flow while changing it gradually towards optimality. The `Preflow-Push` algorithm, on the other hand, maintains a condition that would imply the optimality of a preflow f, *if it were to be a feasible flow*, and the algorithm gradually transforms the preflow f into a flow.

To start the algorithm we will need to define an initial preflow f and labeling h that are compatible. We will use $h(v) = 0$ for all $v \neq s$, and $h(s) = n$, as our initial labeling. To make a preflow f compatible with this labeling, we need to make sure that no edges leaving s are in the residual graph (as these edges do not satisfy the steepness condition). To this end we define the initial preflow as $f(e) = c_e$ for all edges $e = (s, v)$ leaving the source, and define $f(e) = 0$ for all other edges.

(6.22) *The initial preflow f and labeling h are compatible.*

Next we will discuss the steps the algorithm makes towards turning the preflow f into a feasible flow, while keeping it compatible with some labeling h. Consider any node v that has excess — i.e., $e_f(v) > 0$. If there is any edge e in the residual graph G_f that leaves v and goes to a node w at a lower height (note that $h(w)$ is at most 1 less than $h(v)$ due to the steepness condition), then we can modify f by pushing some of the excess flow from v to w. Will will call this a *push* operation.

`push`(f, h, v, w)
  `Applicable if` $e_f(v) > 0$, $h(w) < h(v)$ `and` $(v, w) \in E_f$.
  `If` $e = (v, w)$ `is a forward edge then`
    `let` $\delta = \min(e_f(v), c_e - f(e))$ `and`
    `increase` $f(e)$ `by` δ.
  `If` $e = (v, w)$ `is a backwards edge then`
    `let` $\delta = \min(e_f(v), f(e))$ `and`
    `decrease` $f(e)$ `by` δ.
  `Return(`f, h`)`

If we cannot push the excess of v along any edge leaving v then we will need to raise v's height. We will call this a *relabel* operation.

`relabel`(f, h, v)
  `Applicable if` $e_f(v) > 0$, `and`
     `for all edges` $(v, w) \in E_f$ `we have` $h(w) \geq h(v)$.
  `Increase` $h(v)$ `by 1.`
  `Return(`f, h`)`

So in summary the `Preflow-Push` algorithm is as follows.

`Preflow-Push`(G, s, t, c)

```
Initially h(v) = 0 for all v ≠ s and h(s) = n and
f(e) = c_e for all e = (s, v) and f(e) = 0 for all other edges.
While there is a node v ≠ t with excess e_f(v) > 0
  Let v be a node with excess
  If there is w such that push(f, h, v, w) can be applied then
     push(f, h, v, w)
  Else
     relabel(f, h, v)
Endwhile
Return(f)
```

As usual this algorithm is somewhat under-specified. For an implementation of the algorithm we will have to specify which node with excess to choose, and how to efficiently select an edge on which to push. However, it is clear that each iteration of this algorithm can be implemented in polynomial time. (We'll discuss later how to implement it reasonably efficiently.). Further, it is not hard to see that the preflow f and the labeling h are compatible throughout the algorithm. If the algorithm terminates — something that is far from obvious based on its description — then there are no non-sinks with positive excess, and hence the preflow f is in fact a flow. It then follows from (6.21) that f would be a maximum flow at termination.

We summarize these observations as follows.

(6.23) *Throughout the* `Preflow-Push` *algorithm:*

(i) the labels are non-negative integers;

(ii) f is a preflow, and if the capacities are integral then the preflow f is integral; and

(iii) the preflow f and labeling h are compatible.

If the algorithm returns a preflow f, then f is a flow of maximum value.

Proof. By (6.22) the initial preflow f and labeling h are compatible. We will show using induction on the number of `push` and `relabel` operations that f and h satisfy the properties of the statement. The push operation modifies the preflow f, but the bounds on δ guarantee that the the f returned satisfies the capacity constraints, and the all excesses remain non-negative, so f is a preflow. To see that preflow f and the labeling h are compatible, note that `push`(f, h, v, w) can add one edge to the residual graph, the reverse edge (v, w), and this edge does satisfy the steepness condition. The `relabel` operation increases the label of v, and hence increases the steepness of all edges leaving v. However, it only applies when no edge leaving v in the residual graph is going downwards, and hence the preflow f and the labeling h are compatible after relabeling.

The algorithm terminates if no node other than s or t has excess. In this case, f is a flow by definition; and since the preflow f and the labeling h remain compatible throughout the algorithm, (6.21) implies that f is a flow of maximum value. ■

Next, we will consider the number of `push` and `relabel` operations. First, we will prove a limit on the `relabel` operations, and this will help prove a limit on the maximum number of `push` operations possible. We only consider a node v for either `push` or `relabel` when v has excess. The only source of flow in the network is the source s, hence intuitively the excess at v must have originated at s. The following consequence of this fact will be key to the analysis.

(6.24) *Let f be a preflow. If the node v has excess, then there is a path in G_f from v to the source s.*

Proof. Let A denote all the nodes w such that there is a path from w to s in the residual graph G_f, and let $B = V - A$. We need to show that all nodes with excess are in A.

Notice that $s \in A$. Further, no edges $e = (x, y)$ leaving A can have positive flow, as an edge with $f(e) > 0$ would give rise to a reverse edge (y, x) in the residual graph, and then y would have been in A. Now consider the sum of excesses in the set B, and recall that each node in B has non-negative excess, as $s \notin B$.

$$0 \leq \sum_{v \in B} e_f(v) = \sum_{v \in B} (f^{\text{in}}(v) - f^{\text{out}}(v))$$

Now let's rewrite the sum on the right as follows. If an edge e has both ends in B, then $f(e)$ appears once in the sum with a "+" and once with a "−", and hence these two terms cancel out. If e has only its head in B, then e leaves A and we know that all edges leaving A have $f(e) = 0$. If e has only its tail in B, then $f(e)$ appears just once in the sum, with a "−". So we get

$$0 \leq \sum_{v \in B} e_f(v) = -f^{\text{out}}(B).$$

Since flows are non-negative, we see that the sum of the excesses in B is zero; since individual excess in B is non-negative, they must therefore all be 0. ■

Now we are ready to prove that the labels do not change too much. The algorithm never changes the label of s (as the source never has positive excess). Each other node v starts with $h(v) = 0$, and its label increases by 1 every time it changes. So we simply need to give a limit on how high a label can get. Recall that n denotes the number of nodes in V.

(6.25) *Throughout the algorithm all nodes have $h(v) \leq 2n - 1$.*

Proof. The initial labels $h(t) = 0$ and $h(s) = n$ do not change during the algorithm. Consider some other node $v \neq s, t$. The algorithm changes v's label only when applying the relabel operation, so let f and h be the preflow and labeling returned by a `relabel`(f, h, v) operation. By Statement (6.24) there is a path P in the residual graph G_f from v to s. Let $|P|$ denote the number of edges in P, and note that $|P| \leq n - 1$. The steepness condition implies that heights of the nodes can decrease by at most 1 along each edge in P, and hence $h(v) - h(s) \leq |P|$, which proves the statement. ■

Labels are monotone increasing throughout the algorithm, so this statement immediately implies a limit on the number of relabeling operations.

(6.26) *Throughout the algorithm each node is relabeled at most $2n - 1$ times, and the total number of relabeling operation is less than $2n^2$.*

Next we will bound the number of `push` operations. We will distinguish two kinds of push operations. A `push`(f, h, v, w) is *saturating* if either $e = (v, w)$ is a forward edge in E_f and $\delta = c_e - f(e)$, or (v, w) is a backwards edge with $e = (w, v)$ and $\delta = f(e)$. I.e., the push is saturating if after the push the edge (v, w) is no longer in the residual graph. All other `push` operations will be referred to as *non-saturating*.

(6.27) *Throughout the algorithm the number of saturating* `push` *operations is at most $2nm$.*

Proof. Consider an edge (v, w) in the residual graph. After a saturating `push`(f, h, v, w) we have that $h(v) = h(w) + 1$, and the edge (v, w) is no longer in the residual graph G_f. Before we can `push` again along this edge, first we have to push from w to v to make the edge (v, w) appear in the residual graph. However, in order to push from w to v, we first need for w's label to increase by at least 2 (so that w is above v). The label of w can increase by 2 at most $n - 1$ times, so a saturating push from v to w can occur at most n times. Each edge $e \in E$ can give rise to two edges in the residual graph, so overall we can have at most $2nm$ saturating pushes. ■

The hardest part of the analysis is proving a bound on the number of non-saturating pushes, and this also will be the bottleneck for the theoretical bound on the running time.

(6.28) *Throughout the algorithm the number of non-saturating* `push` *operations is at most $2n^2m$.*

Proof. For this proof we will use a so-called *potential function method*. For a preflow f and a compatible labeling h, we define

$$\Phi(f, h) = \sum_{v : e_f(v) > 0} h(v)$$

to be the sum of the heights of all nodes with positive excess. (Φ is often called a *potential* since it resembles the "potential energy" of all nodes with positive excess.)

In the initial preflow and labeling all nodes with positive excess are at height 0, so $\Phi(f,h) = 0$. $\Phi(f,h)$ remains non-negative throughout the algorithm. A non-saturating $\texttt{push}(f,h,v,w)$ decreases $\Phi(f,h)$ by at least 1, as after the push the node v will have no excess, and w, the only node that gets new excess from the operation, is at a height one less than v. However, each saturating `push` and each `relabel` operation can increase $\Phi(f,h)$. A `relabel` operation increases $\Phi(f,h)$ by exactly 1. There are at most $2n^2$ `relabel` operations, so the total increase in $\Phi(f,h)$ due to `relabel` operations is $2n^2$. A saturating $\texttt{push}(f,h,v,w)$ operation does not change labels, but can increase $\Phi(f,h)$ since the node w may suddenly acquire positive excess after the push. This would increase $\Phi(f,h)$ by the height of w, which is at most $2n-1$. There are at most $2nm$ saturating `push` operations, so the total increase in $\Phi(f,h)$ due to `push` operations is at most $2mn(2n-1)$. So between the two causes $\Phi(f,h)$ can increase by at most $4mn^2$ during the algorithm.

But since Φ remains non-negative throughout, and it decreases by at least one on each non-saturating `push`, it follows that there can be at most $4mn^2$ non-saturating `push` operations. ■

There has been a lot of work devoted to choosing node selection rules for this algorithm to improve the worst case running time. Next we show that if one always selects the node with positive excess at the maximum height, then there will be at most $O(n^3)$ non-saturating `push` operations.

(6.29) *If at each step we choose the node with excess at maximum height than the number of non-saturating* `push` *operations through the algorithm is at most* $2n^3$.

Proof. The algorithm selects a node with excess at maximum height. Consider this height $H = \max_{v:e_f(v)>0} h(v)$ as the algorithm proceeds. This maximum height H can only increase due to relabeling (as flow is always pushed to nodes at less height), and so the total increase in H throughout the algorithm is at most $2n^2$ by (6.25) . H starts out 0 and remains non-negative, so the number of times H changes is at most $4n^2$.

Now consider the behavior of the algorithm over a phase of time in which H remains constant. We claim that each node can have at most one non-saturating `push` during this phase. Indeed, during this phase, flow is being pushed from nodes at height H to nodes at height $H-1$; and after a non-saturating `push` from v, it must receive flow from a node at height $H+1$ before we can push from it again.

Since there are at most n non-saturating `push` operations between each change to H, and H changes at most $4n^2$ times, the total number of non-saturating `push` operations is at most $4n^3$. ■

As a follow-up to (6.29), it is interesting to note that experimentally the computational bottleneck of the method is the number of relabeling operations, and better experimental running time is obtained by variants that work on increasing labels faster than one-by-one. This is a point that we pursue further in some of the exercises.

Finally, we need to briefly discuss how to implement this algorithm efficiently. Maintaining a few simple data structures will allow us to effectively implement the operations of the algorithm in constant time each, and overall implement the algorithm in time $O(mn)$ plus the number of non-saturating `push` operations. Hence the above generic algorithm will run in $O(mn^2)$ time, while the version that always selects the node at maximum height will run in $O(n^3)$ time.

We can maintain all nodes with excess on a simple list and so we will be able to select a node with excess in constant time. One has to be a bit more careful to be able to select a node with maximum height H in constant time. In order to do this we will maintain a linked list of all nodes with excess at every possible height. Note that whenever a node v gets relabeled, or continues to have positive excess after a `push`, it remains a node with maximum height H. Thus, we only have to select a new node after a `push` when the current node v no longer has positive excess. If node v was at height H, then the new node at maximum height will also be at height H or, if no node at height H has excess, then the maximum height will be $H - 1$ — since the previous `push` out of v pushed flow to a node at height $H - 1$.

Now assume we have selected a node v, and we need to select an edge (v, w) on which to apply `push`(f, h, v, w) (or `relabel`(f, h, v) if no such w exists). To be able to select an edge quickly we will use the adjacency list representation of the graph. More precisely, we will maintain, for each node v, all possible edges leaving v in the residual graph (both forwards and backwards edges) in a linked list, and with each edge we keep its capacity and flow value. Note that this way we have two copies of each edge in our data structure: a forwards and a backwards copy. These two copies will each have pointers to one another, so that updates done at one copy can be carried over to the other one in $O(1)$ time. We will select edges leaving a node v for `push` operations in the order they appear on node v's list. To facilitate this selection we will maintain a pointer `current`(v) for each node v to the last edge on the list that has been considered for a `push` operation. So, if node v no longer has excess after a non-saturating `push` out of node v, the pointer `current`(v) will stay at this edge, and we will use the same edge for the next `push` operation out of v. After a saturating `push` out of node v, we advance `current`(v) to the next edge on the list.

The key observation is that after advancing the pointer `current`(v) from an edge (v, w), we will not want to apply `push` to this edge again until relabel v.

(6.30) *After the* `current`(v) *pointer is advanced from an edge* (v, w) *we cannot apply* `push` *to this edge till* v *gets relabeled.*

Proof. At the moment `current`(v) is advanced from the edge (v, w) there is some reason `push` cannot be applied to this edge. Either $h(w) \geq h(v)$, or the edge is not in the residual graph. In the first case, we clearly need to relabel v before applying a `push` on this edge. In the later case, one needs to apply `push` to the reverse edge (w, v) to make (v, w) re-enter the residual graph. However, when we apply `push` to edge (w, v) then w is above v, and so v needs to be relabeled before one can push flow from v to w again. ■

Since edges do not have to be considered again for `push` before relabeling, we get the following.

(6.31) *When the* `current`(v) *pointer reaches the end of the edge list for* v*, the* `relabel` *operation can be applied to node* v*.*

After relabeling node v we reset `current`(v) to the first edge on the list, and start considering edges again in the order they appear on v's list.

(6.32) *The running time of the* `Preflow-Push` *algorithm, implemented using the above data structures is* $O(mn)$ *plus* $O(1)$ *for each non-saturating* `push` *operation. In particular, the generic* `Preflow-Push` *algorithm runs in* $O(n^2m)$ *time, while the version when we always select the node at maximum height runs in* $O(n^3)$ *time.*

Proof. The initial flow and relabeling is set up on $O(m)$ time. Both `push` and `relabel` operations can be implemented in $O(1)$ time, once the operation has been selected. Consider a node v. We know that v can be relabeled at most $2n$ times throughout the algorithm. We will consider the total time the algorithm spends on finding the right edge to `push` flow on out of node v between two times that node v gets relabeled. If node v has d_v out-going edges, then by (6.31) we spend $O(d_v)$ time on advancing the `current`(v) pointer between consecutive relabelings of v. Thus, the total time spent on advancing the `current` pointers throughout the algorithm is $O(\sum_{v \in V} nd_v) = O(mn)$, as claimed. ■

6.7 Applications: Disjoint Paths and Bipartite Matchings

Next we develop two simple applications of maximum flow and minimum cuts in graphs. Much before the work of Ford and Fulkerson, Menger in 1927 studied the closely related disjoint paths problem. A set of paths is *edge-disjoint* if their edge sets are disjoint, i.e., no two paths share an edge, though multiple paths may go through some of the same nodes. Given a graph $G = (V, E)$ with two distinguished nodes $s, t \in V$, the *edge-disjoint paths problem* is to find the maximum number of edge-disjoint s-t paths in G. This problem can be solved naturally using flows. Let the capacity of each edge $e = (v, w)$ be 1. We claim

that the maximum flow has value k if and only if there are k edge-disjoint paths in G from s to t.

(6.33) *There are k edge-disjoint paths in G from s to t if and only if the value of the maximum value of an s-t-flow in G is at least k.*

Proof. First suppose there are k edge-disjoint paths. We can make each of these paths carry one unit of flow: we set the flow as $f(v, w) = 1$ for all edges on the paths, and $f(v, w) = 0$ on all other edges, to define a feasible flow of value k.

Conversely, consider a flow in the network G of value k. By (6.43), we know that there is a feasible flow with integer flow values. Since all edges have a capacity bound of 1, and the flow is integer-valued, each edge that carries flow has exactly one unit of flow on it.

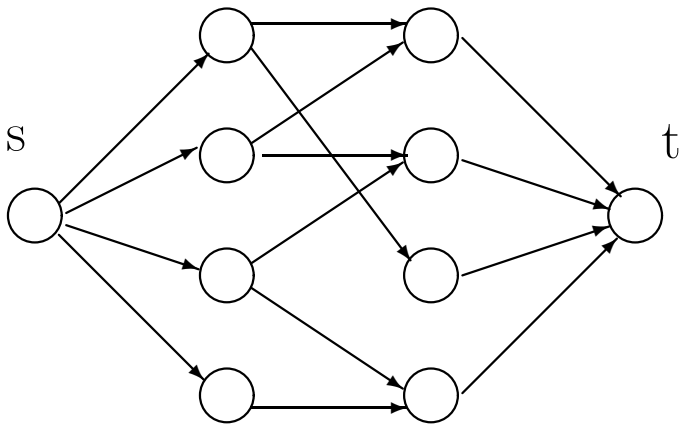
Consider an edge (s, u) that carries one unit of flow. It follows by conservation that there is some edge (u, v) carries one unit of flow. If we continue in this way, we construct a path P from s to t, so that each edge on this path carries one unit of flow. We can apply this construction to each edge of the form (s, u) carrying one unit of flow; in this way, we produce k paths from s to t, each consisting of edges that carry one unit of flow. We can make these k paths are edge-disjoint: if there are multiple edges carrying flow into a node v, than, by conservation, there are at least as many edge carrying flow out of v, so each path entering v can use a different edge to leave the node. ■

For this flow problem, we discover that $C = \sum_{e \text{ out of } s} c_e \leq |V| = n$, as there are at most $|V|$ edges out of s, each of which has capacity 1. Thus, by using the $O(mC)$ bound in (6.5) we get an integer maximum flow in $O(mn)$ time. To obtain the edge-disjoint paths, we decompose the flow into paths as was done in the proof of (6.33). It is not hard to find such a decomposition also in $O(mn)$ time. The number of edges in the paths is at most m, so we get an $O(mn)$ running time by adding each new edge to the current path in $O(n)$ time.

(6.34) *The F-F algorithm can be used to find a maximum set of edge-disjoint s-t paths in directed graph G in $O(mn)$ time.*

It's interesting that if we were to use the "better" bound of $O(m^2 \log_2 C)$ that we developed in the previous section, we'd get the inferior running time of $O(m^2 \log n)$ for this problem. There is nothing contradictory in this — the $O(m^2 \log_2 C)$ bound was designed to be good for instances in which C is very large relative to m and n; whereas in the bipartite matching problem, $C = n$.

The Max-Flow Min-Cut Theorem (6.11) can be used to give the following characterization of the maximum number of edge-disjoint paths. This characterization was originally discovered by Menger in 1927. We say that a set $F \subseteq E$ of edges *disconnects* t from s if after removing the edges F from the graph G no s-t paths remain in the graph.



(6.35) *In every directed graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects t from s.*

Proof. If the removal of a set $F \subseteq E$ of edges disconnects t from s, then all s-t-paths must use at least one edge from F, and hence the number of edge-disjoint s-t paths is at most $|F|$.

To prove the other direction we will use the Max-Flow Min-Cut Theorem (6.11). By (6.33) the maximum number of edge-disjoint paths is the value ν of the maximum s-t flow. Now (6.11) states that there is an s-t-cut (A, B) with capacity ν. Let F be the set of edges that go from A to B. Each edge has capacity 1, so $|F| = \nu$, and by the definition of an s-t cut removing these ν edges from G disconnects t from s. ■

The Bipartite Matching Problem

One of our original goals in developing the maximum flow problem was to be able to solve the bipartite matching problem, and we now show how to do this. Recall that a *bipartite graph* $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y. A *matching* M in G is a subset of the edges $M \subseteq E$ such that each node appears in at most one edge in M. The *maximum matching problem* is that of finding a matching in M of largest possible size.

Note that the graph defining a matching problem is undirected, while flow networks are directed. Yet, the idea of using the maximum flow algorithm to find a maximum matching will be quite simple. Beginning with the graph G in a bipartite matching problem, we construct a flow network G' as follows. First, we direct all edges in G from X to Y. We then add a node s, and an edge (s, x) from s to each node in X. We add a node t, and an edge (y, t) from each node in Y to t. Finally, we give each edge in G' a capacity of 1.

We can now show that integer-valued flows in G' encode matchings in G in a fairly transparent fashion. First, suppose there is a matching in G consisting of k edges $(x_{i_1}, y_{i_1}), \ldots, (x_{i_k}, y_{i_k})$.

Then consider the flow f that sends one unit along each path of the form s, x_{i_j}, y_{i_j}, t — that is, $f(e) = 1$ for each edge on one of these paths. One can verify easily that the capacity and conservation conditions are indeed met, and that f is an s-t flow of value k.

Conversely, suppose there is a flow f' in G' of value k. By the integrality theorem for maximum flows (6.13), we know there is an integer-valued flow f of value k — and since all capacities are 1, this means that $f(e)$ is equal to either 0 or 1 for each edge e. Now, consider the set M' of edges of the form (x, y) on which the flow value is 1.

Here are three simple facts about the set M'

- M' contains k edges.

To see this, consider the cut (A, B) in G' with $A = \{s\} \cup X$. The value of the flow is the total flow leaving A, minus the total flow entering A. The first of these terms is simply the cardinality of M', since these are the edges leaving A that carry flow, and each carries exactly one unit of flow. The second of these terms is 0, since there are no edges entering A. Thus, M' contains k edges.

- Each node in X is the tail of at most one edge in M'.

To see this, suppose $x \in X$ were the tail of at least two edges in M'. Since our flow is integer-valued, this means that at least two units of flow leave from x. By conservation of flow, at least two units of flow would have to come into x — but this is not possible, since only a single edge of capacity 1 enters x. Thus x is the tail of at most one edge in M'. By the same reasoning, we can show

- Each node in Y is the head of at most one edge in M'.

Combining these facts, we see that if we view M' as a set of edges in the original bipartite graph G, we get a matching of size k. In summary, we have proved the following fact.

(6.36) *The size of the maximum matching in G is equal to the value of the maximum flow in G'; and the edges in such a matching in G are the edges that carry flow from X to Y in G'.*

Note the crucial way in which the integrality theorem (6.13) figured in this construction — we needed to know there is a maximum flow in G' that takes only the values 0 and 1.

Now let's consider how quickly we can compute a maximum matching in G. Let $n = |X| = |Y|$, and let m be the number of edges of G. We'll tacitly assume that $m \geq n$ — since we may as well assume that there is at least one edge incident to each node in the original problem. The time to compute a maximum matching is dominated by the time to compute an integer-valued maximum flow in G', since converting this to a matching in G is simple. For this flow problem, we have that $C = \sum_{e \text{ out of } s} c_e = |X| = n$, as s has an edge of capacity 1 to each node of X. Thus, by using the $O(mC)$ bound in (6.5), we get the following.

(6.37) *The F-F algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time.*

It is worthwhile to consider what the augmenting paths mean in the network G'. Consider the matching M consisting of edges (x_1, y_1), (x_2, y_2) and (x_4, y_4) in the bipartite graph at the beginning of this chapter. Let f be the corresponding flow in G'. This matching is not maximum, so f is not a maximum s-t flow, and hence there is an augmenting path in the residual graph G'_f. There is only one edge in G_f leaving node s: the edge (s, x_3). Similarly the only edge entering t in G_f is edge (y_3, t). The augmenting path in the residual graph G_f goes through the nodes $s, x_3, y_2, x_2, y_1, x_1, y_3, t$ in this order. Note that the edges (x_2, y_2) and (x_1, y_1) are used backwards, and all other edges were used forwards. The effect of this augmentation is to take the edges used backwards out of the matching, and replace them with the edges going forwards. Because the augmenting path goes from s to t, there is one more forward edge than backward edge; thus, the size of the matching increases by one.

Before we conclude this section, we consider the structure of perfect matchings in bipartite graphs. Algorithmically, we've seen how to find perfect matchings: we use the algorithm above to find a maximum matching, and then check if this matching is perfect.

But let's ask a slightly less algorithmic question. Not all bipartite graphs have perfect matchings. What does a bipartite graph without a perfect matching look like? Is there an easy way to see that a bipartite graph does not have a perfect matching — or at least an easy way to convince someone the graph has no perfect matching, after we run the algorithm? More concretely, it would be nice if the algorithm, upon concluding that there is no perfect matching, could produce a short "certificate" of this fact. The certificate could allow someone to be quickly convinced that there is no perfect matching, without having to pore over a trace of the entire execution of the algorithm.

What might such a certificate look like? For example, if there are nodes $x_1, x_2 \in X$ that have only one incident edge each, and the other end of each edge is the same node y, then clearly the graph has no perfect matching: both x_1 and x_2 would need to get matched to the same node y. More generally, consider a subset of nodes $A \subseteq X$, and let $\Gamma(A) \subseteq Y$ denote the set of all nodes that are adjacent to nodes in A. If the graph has a perfect matching, then each node is A has to be matched to a different node in $\Gamma(A)$, so $\Gamma(A)$ has to be at least as large as A. This gives us the following fact.

(6.38) *If a bipartite graph $G = (V, E)$ with two sides X and Y has a perfect matching, then for all $A \subseteq X$ we must have $|\Gamma(A)| \geq |A|$.*

This statement suggests a type of certificate that a graph does not have a perfect matching: a set $A \subseteq X$ such that $|\Gamma(A)| < |A|$. But is the converse of (6.38) also true? Is is the case that whenever there is no perfect matching, there is a set A like this that proves

it? The answer turns out to be yes, provided we add the obvious condition that $|X| = |Y|$ (without which there could certainly not be a perfect matching). This statement is known in the literature as *Hall's theorem*, though versions of it were discovered independently by a number of different people — perhaps first by König — in the early part of this century. The proof of the statement also provides a way to find such a subset A in polynomial time.

(6.39) *Assume that the bipartite graph $G = (V, E)$ has two sides X and Y such that $|X| = |Y|$. Then the graph G either has a perfect matching, or there is a subset $A \subseteq X$ such that $|\Gamma(A)| < |A|$. A perfect matching or an appropriate subset A can be found in $O(mn)$ time.*

Proof. We will use the same graph G' as in the proof of the (6.36). Assume that $|X| = |Y| = n$. By (6.36) the graph G has a maximum matching if and only if the value of the maximum flow in G' is n.

We need to show that if the value of the maximum flow is less than n, then there is a subset A as claimed in the statement. By the Max-Flow Min-Cut theorem (6.10), if the maximum flow value is less than n, then there is a cut (A', B') with capacity less than n in G'. We claim that the set $A = X \cap A'$ has the claimed property. This will prove both parts of the statement, as we've seen in (6.5) that a minimum cut (A', B') can also be found by running the F-F algorithm.

First we claim that one can modify the minimum cut so as to ensure that $\Gamma(A) \subseteq A'$. To do this, consider a node $y \in \Gamma(A))$ that belongs to B'. We claim that by moving y from B' to A' we do not increase the capacity of the cut. For what happens when we move y from B' to A'? The edge (y, t) now crosses the cut, increasing the capacity by one. But previously there was *at least* one edge (x, y) with $x \in A$, since $y \in \Gamma(A)$; all these edges used to cross the cut, and don't anymore. Thus, overall, the capacity of the cut cannot increase. (Note that we don't have to be concerned about nodes $x \in X$ that are not in A. The two ends of the edge (x, y) will be on different sides of the cut, but this edge does not add to the capacity of the cut, as it goes from B' to A'.)

Next consider the capacity of this minimum cut (A', B') that has $\Gamma(A) \subseteq A'$. Since all the neighbors of A belong to A', we see that the only edges out of A' are either edges that leave the sink s or that enter the sink t. Thus the capacity of the cut is exactly

$$c(A', B') = |X \cap B'| + |Y \cap A'|.$$

Notice that $|X \cap B'| = n - |A|$, and $|Y \cap A'| \geq |\Gamma(A)|$. Now the assumption that $c(A', B') < n$ implies that

$$n - |A| + |\Gamma(A)| \leq |X \cap B'| + |Y \cap A'| = c(A', B') < n.$$

Comparing the first and the last terms, we get the claimed inequality $|A| < |\Gamma(A)|$. ∎

One way to understand the idea of Hall's theorem is as follows. We can decide if the graph G has a perfect matching by checking if the maximum flow in a related graph G' has value at least n. By the Max-Flow Min-Cut theorem, there will be an s-t cut of capacity less than n if the maximum flow value in G' has value less than n. The condition in Hall theorem provides a natural meaning for cuts in G' of capacity less than n, in terms of the original graph G.

6.8 Extensions to the Maximum Flow Problem

Much of the power of the maximum flow problem has essentially nothing to do with the fact that it models traffic in a network — rather, it lies in the fact that many problems with a non-trivial combinatorial search component can be solved in polynomial time because they can be reduced to the problem of finding a maximum flow or a minimum cut in a directed graph.

Bipartite matching is a natural first application in this vein; in the next section, we investigate a range of further applications. To begin with, we stay with the picture of flow as an abstract kind of "traffic," and look for more general conditions we might impose on this traffic. These more general conditions will turn out to be useful for some of our further applications.

In particular, we focus on two generalizations of maximum flow; we will see that they can all be reduced to the basic maximum flow problem.

Circulations with Demands

One simplifying aspect of our initial formulation of the maximum flow problem is that we had only a single source s and a single sink t. Now suppose that there can be a set S of sources generating flow, and a set T of sinks that can absorb flow. As before, there is an integer capacity on each edge.

With multiple sources and sinks it is a bit unclear how to decide which source or sink to favor in a maximization problem. So instead of maximizing the flow value we will consider a problem where sources have fixed *supply* values and sinks have fixed *demand* values, and our goal is to ship flow from nodes with available supply to those with given demands. Picture, for example, that the network represents a system of highways in which we want to ship products from factories (which have supply) to retail outlets (which have demand). In this type of problem, we will not be seeking to maximize a particular value; rather, we simply want to satisfy all the demand using the available supply.

Thus, we are given a flow network $G = (V, E)$ with capacities on the edges. Now, associated with each node $v \in V$ is a *demand* d_v. If $d_v > 0$, this indicates that the node v has a *demand* of d_v for flow — the node is a sink in T, and it wishes to receive d_v units more

flow than it sends out. If $d_v < 0$, this indicates that v has a *supply* of $-d_v$ — the node is a source in S and it wishes to send out $-d_v$ units more flow than it receives. If $d_v = 0$ than the node v is neither a source nor a sink. We will assume that all capacities and demands are integers.

In this setting, we say that a *circulation* with demands $\{d_v\}$ is a function f that assigns a non-negative real number to each edge, and satisfies the following two conditions.

(i) *(Capacity conditions.)* For each $e \in E$, $0 \leq f(e) \leq c_e$.

(ii) *(Demand conditions.)* For every v, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Now, instead of considering a maximization problem, we are concerned with a *feasibility problem* — we want to know whether there *exists* a circulation that meets conditions (i) and (ii).

Here is a simple condition that must hold in order for a feasible circulation to exist — the total supply must equal the total demand.

(6.40) *If there exists a feasible circulation with demands $\{d_v\}$, then $\sum_v d_v = 0$.*

Proof. Suppose there exists a feasible circulation f in this setting. Then $\sum_v d_v = \sum_v f^{\text{in}}(v) - f^{\text{out}}(v)$. Now, in this latter expression, the value $f(e)$ for each edge $e = (u, v)$ is counted exactly twice: once in $f^{\text{out}}(u)$, and once in $f^{\text{in}}(v)$. These two terms cancel out; and since this holds for all values $f(e)$, the overall sum is 0. ■

Thanks to (6.40), we know that

$$\sum_{v:d_v>0} d_v = \sum_{v:d_v<0} -d_v.$$

Let D denote this common value.

It turns out that we can reduce the problem of finding a feasible circulation with demands $\{d_v\}$ to the problem of finding a maximum s-t flow in a different network. The reduction looks very much like the one we used for bipartite matching: we attach a "super-source" s^* to each node in S, and a "super-sink" t^* to each node in T. More specifically, we create a graph G' from G by adding new nodes s^* and t^* to G. For each node $v \in T$ — i.e., each node v with $d_v > 0$ — we add an edge (v, t) with capacity d_v. For each node $v \in S$ — i.e., each node with $d_v < 0$ — we add an edge (s, v) with capacity $-d_v$. We carry the remaining structure of G over to G' unchanged. In this graph G', we will be seeking a maximum s^*-t^* flow.

Note that there cannot be an s^*-t^* flow in G' of value greater than D, since the cut (A, B) with $A = \{s^*\}$ only has capacity D. Now, if there is a feasible circulation f with demands $\{d_v\}$ in G, then by sending a flow value of $-d_v$ on each edge (s^*, v), and a flow value of d_v

on each edge (v, t^*), we obtain an s^*-t^* flow in G' of value D, and so this is a maximum flow. Conversely, suppose there is a (maximum) s^*-t^* flow in G' of value D. It must be that every edge out of s^*, and every edge into t^*, is completely saturated with flow. Thus, if we delete these edges, we obtain a circulation f in G with $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ for each node v. Further, if there is a flow of value D in G', then there is such a flow that takes integer values.

In summary, we have proved the following.

(6.41) *There is a feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s^*-t^* flow in G' has value D. If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

At the end of the previous section we used the Max-Flow Min-Cut theorem to derive the characterization (6.39) of bipartite graphs that do not have perfect matchings. We can give an analogous characterization for graphs that do not have a feasible circulation. The characterization uses the notion of a *cut*, adapted to the present setting. In the context of circulation problems with demands, a cut (A, B) is any partition of the nodes set V into two sets, with no restriction on which side of the partition the sources and sinks fall. We include the characterization here without a proof.

(6.42) *The graph G has a feasible circulation with demands $\{d_v\}$ if and only if for all cuts (A, B)*

$$\sum_{v \in B} d_v \leq c(A, B).$$

It is important to note that our network has only a single "kind" of flow. Although the flow is supplied from multiple sources, and absorbed at multiple sinks, we cannot place restrictions on which source will supply the flow to which sink; we have to let our maximum flow algorithm decide this. A harder problem, which we may address briefly at the end of the course, is *multicommodity flow* — here sink t_i must be supplied with flow that originated at source s_i, for each i.

Circulations with Demands and Lower Bounds

Finally, let us generalize the previous problem a little. In many applications, we not only want to satisfy demands at various nodes; we also want to force the flow to make use of certain edges. This can be enforced by placing *lower bounds* on edges, as well as the usual upper bounds imposed by edge capacities.

Consider a flow network $G = (V, E)$ with a *capacity* c_e and a *lower bound* ℓ_e on each edge e. We will assume $0 \leq \ell_e \leq c_e$ for each e. As before, each node v will also have a demand d_v, which can be either positive or negative. We will assume that all demands, capacities, and lower bounds are integers.

The given quantities have the same meaning as usual; a lower bound ℓ_e means that the flow value on e must be *at least* ℓ_e. Thus, a circulation in our flow network must satisfy the following two conditions.

(i) *(Capacity conditions.)* For each $e \in E$, $\ell_e \leq f(e) \leq c_e$.

(ii) *(Demand conditions.)* For every v, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

As before, we wish to decide whether there exists a *feasible circulation* — one that satisfies these conditions.

Our strategy will be to reduce this to the problem of finding a circulation with demands, but no lower bounds. (We've seen that this latter problem, in turn, can be reduced to a standard maximum flow problem.) The idea is as follows. We know that on each edge e, we need to send at least ℓ_e units of flow — so suppose that we define an initial circulation f_0 simply by $f_0(e) = \ell_e$. f' satisfies all the capacity conditions (both lower and upper bounds); but it presumably does not satisfy all the demand conditions. In particular,

$$f_0{}^{\text{in}}(v) - f_0{}^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e.$$

Let us denote this quantity by L_v. If $L_v = d_v$, then we have satisfied the demand condition at v; but if not, then we need to superimpose a circulation f_1 on top of f_0 which will clear the remaining "imbalance" at v: so we need $f_1{}^{\text{in}}(v) - f_1{}^{\text{out}}(v) = d_v - L_v$. And how much capacity do we have with which to do this? Having already sent ℓ_e units of flow on each edge e, we have $c_e - \ell_e$ more units to work with.

These considerations directly motivate the following construction. Let the graph G' have the same nodes and edges, with capacities and demands, but no lower bounds. The capacity of edge e will be $c_e - \ell_e$. The demand of node v will be $d_v - L_v$. Now we claim

(6.43) *There is a feasible circulation in G if and only if there is a feasible circulation in G'. If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

Proof. First suppose there is a circulation f' in G'. Define a circulation f in G by $f(e) = f'(e) + \ell_e$. Then f satisfies the capacity conditions in G, and

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell_e + f'(e)) - \sum_{e \text{ out of } v} (\ell_e + f'(e)) = L_v + (d_v - L_v) = d_v,$$

so it satisfies the demand conditions in G as well.

Conversely, suppose there is a circulation f in G, and define a circulation f' in G' by $f'(e) = f(e) - \ell_e$. Then f' satisfies the capacity conditions in G', and

$$(f')^{\text{in}}(v) - (f')^{\text{out}}(v) = \sum_{e \text{ into } v} (f(e) - \ell_e) - \sum_{e \text{ out of } v} (f(e) - \ell_e) = d_v - L_v,$$

so it satisfies the demand conditions in G' as well. ■

6.9 Applications of Maximum Flows and Minimum Cuts

Many problems that arise in applications can in fact be solved efficiently by a reduction to maximum flow, but it is often difficult to discover when such a reduction is possible. In this section, we give several paradigmatic examples of such problems; the goal is to indicate what such reductions tend to look like, and to illustrate some of the most common uses of flows and cuts in the design of efficient combinatorial algorithms. One point that will emerge is the following: sometimes the solution one wants involves the computation of a maximum flow, and sometimes it involves the computation of a minimum cut — both flows and cuts are very useful algorithmic tools.

Survey Design

A major issue in the burgeoning field of *data mining* is the study of consumer preference patterns. The following *survey design* problem is a simple version of a task faced by many companies wanting to measure customer satisfaction. Consider a company that sells k products, and has a database containing the purchase histories of a large number of customers. (Those of you with "Shopper's Club" cards may be able to guess how this this data gets collected.) The company wishes to conduct a survey, sending customized questionnaires to a particular group of n of its customers, to try determining which products people like overall.

Here are the guidelines for designing the survey.

- Each customer will receive questions about certain of the products.
- A customer can only be asked about products that he or she has purchased.
- To make each questionnaire informative, but not too long so as to discourage participation, each customer i should be asked about a number of products between c_i and c_i'.
- Finally, to collect sufficient data about each product, there must be between p_j and p_j' distinct customers asked about each product j.

More formally, the input to the *survey problem* consists of a bipartite graph G whose nodes are the customers and the products, and there is an edge between customer i and product j if he or she has ever purchased product j. Further, for each customer $i = 1, \ldots, n$, we have limits $c_i \leq c_i'$ on the number of products he or she can be asked about; for each product $j = 1, \ldots, k$ we have limits $p_j \leq p_j'$ on the number of distinct customers that have to be asked it. The problem is to decide if there is a way to design a questionnaire for each customer so as to satisfy all these conditions.

We will solve this problem by reducing it to a circulation problem on a flow network G' with demands and lower bounds. To obtain graph G' from G, we orient the edges of G from customers to products, add nodes s and t with edges (s, i) for each customer $i = 1, \ldots, n$, edges (j, t) for each product $j = 1, \ldots, k$, and an edge (t, s). The circulation in this network will correspond to the way in which questions are asked. The flow on the edge (s, i) is the number of products surveyed on the questionnaire for customer i, so this edges will have a capacity of c'_i and a lower bound of c_i. The flow on the edge (j, t) will correspond to the number of customers who were asked about product j, so this edges will have a capacity of p'_j and a lower bound of p_j. Each edges (i, j) going from a customer to a product he or she bought has capacity 1, and 0 as lower bound. The flow carried by the edge (t, s) corresponds to the overall number of questions asked. We can give this edge a capacity of $\sum_i c'_i$ and a lower bound of $\sum_i c_i$. All nodes have demand 0.

(6.44) *The graph G' with 0 demand, and the capacities and lower bounds given above, has a feasible circulation if and only if there is a feasible way to design the survey.*

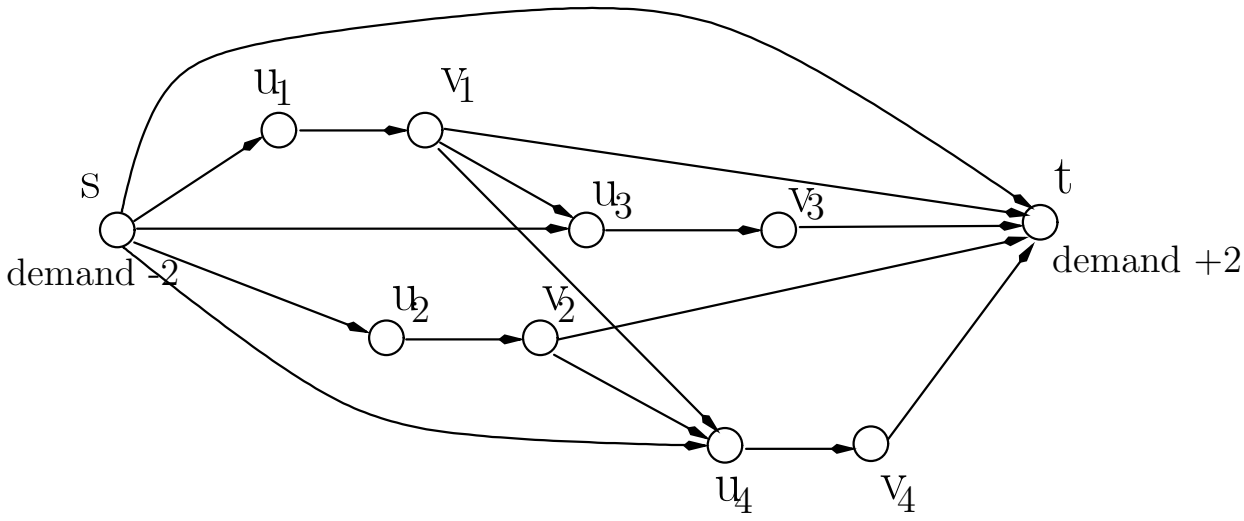
Proof. The above construction immediately suggest a way to turn a survey design into the corresponding flow. The edge (i, j) will carry 1 unit of flow if customer i is asked about product j in the survey, and will carry no flow otherwise. The flow on the edges (s, i) is the number of questions asked from customer i, the flow on the edge (j, t) is the number of customers who were asked about product j, and finally, the flow on edge (t, s) is the overall number of questions asked. This flow satisfies the 0 demand, i.e., there is flow conservation at every node. If the survey satisfies the rules above, than the corresponding flow satisfies the capacities and lower bounds.

Conversely, if the circulation problem is feasible, then by (6.43) there is a feasible circulation that is integer-valued, and such an integral valued circulation naturally corresponds to a feasible survey design. Customer i will be surveyed about product j if and only if the edge (i, j) carries a unit of flow. ■

Airline Scheduling

The computational problems faced by the nation's large airline carriers are almost too complex to even imagine — they have to produce schedules for thousands of routes each day that are efficient in terms of equipment usage, crew allocation, customer satisfaction, and a host of other factors; all in the face of unpredictable issues like weather and breakdowns. It's not surprising that they're one of the largest consumers of high-powered algorithmic techniques.

Covering these computational problems in any realistic level of detail would take us much too far afield; instead, we'll discuss a "toy" problem that captures in a very clean way some of the resource allocation issues that arise in a context like this. And, as is common in this course, the toy problem will be much more useful for our purposes than the "real" problem;



for the solution to the toy problem involves a very general technique that can be applied in a wide range of situations.

So suppose you're in charge of managing a fleet of aircraft, together with flight crews (i.e. pilots and flight attendants) to staff the flights. You have a list of cities P and a table that tells you the travel time $t(p, p')$ between cities $p, p' \in P$. On a given day, you have a list of k flights that need to be completed; the i^{th} flight is specified by the data (o_i, d_i, t_i), where $o_i \in P$ is the *origin*, $d_i \in P$ is the *destination*, and t_i is the *departure time*. (The arrival time of the flight is then $t_i + t(o_i, d_i)$.)

It is possible to use the same flight crew (i.e. pilots and flight attendants) for more than one flight during the day. For example, if flight i arrives in San Jose at 1 pm, and flight j departs from San Francisco at 4 pm, and there is a way to transport people from San Jose to San Francisco in the intervening three hours, then a single flight crew could perform flight i, be moved up to San Francisco, and be there in time to perform flight j as well. Generally, for each pair of flights i and j, we will designate j as being *reachable* from i if the arrival time of i is before the departure time of j, and it is possible to transport a single flight crew from the destination of i to the origin of j in the intervening time.

Your goal is to use as few flight crews as possible to perform all the flights; to do this, you want to re-use crews over multiple flights as efficiently as possible.

It turns out that this problem can be solved very naturally by flow techniques. First of all, we will consider a "decision" version of the problem: can all flights be performed using at most c flight crews, for a given value of c? By searching over the possible values of c from 1 to k, we will then be able to find the minimum necessary number of crews.

The solution is based on the following idea. Units of flow will correspond to flight crews. We will have an edge for each flight, and have a lower bound of 1 on these edges to require that at least one unit of flow crosses this edge. In other words, each flight must actually be

completed by some crew. If (u_i, v_i) is the edge representing flight i, and (u_j, v_j) is the edge representing flight j, and flight j is reachable from flight i, then we will have an edge from v_i to u_j — in this way, a unit of flow can traverse (u_i, v_i) and then move directly to (u_j, v_j).

Let us give the construction more precisely, in terms of the following graph $G = (V, E)$. The node set of G is defined as follows.

- For each flight i, G will have the two nodes u_i and v_i.
- G will also have a distinct source node s and sink node t.

The edge set of G is defined as follows.

- For each i, there is an edge (u_i, v_i) with a lower bound of 1 and a capacity of 1. *(Each flight must be performed.)*
- For each i and j so that flight j is reachable from flight i, there is an edge (v_i, u_j) with a lower bound of 0 and a capacity of 1. *(The same crew can perform flights i and j.)*
- For each i, there is an edge (s, u_i) with a lower bound of 0 and a capacity of 1. *(Any crew can begin the day with flight i.)*
- For each j, there is an edge (v_j, t) with a lower bound of 0 and a capacity of 1. *(Any crew can end the day with flight j.)*
- There is an edge (s, t) with lower bound 0 and capacity c. *(If we have extra flight crews, we don't need to use them for any of the flights.)*

Finally, the node s will have a demand of $-c$, and the node t will have a demand of c. All other nodes will have a demand of 0.

We now have the following fact.

(6.45) *There is a way to perform all flights using at most c crews if and only if there is a feasible circulation in the network G.*

Proof. First, suppose there is a way to perform all flights using $c' \leq c$ crews. The set of flights performed by each individual crew defines a path P in the network G, and we send one unit of flow on each such path P. To satisfy the full demands at s and t, we send $c - c'$ units of flow on the edge (s, t). The resulting circulation satisfies all demand, capacity, and lower bound conditions.

Conversely, consider a feasible circulation in the network G. By (6.43), we know that there is a feasible circulation with integer flow values. Suppose that c' units of flow are sent on edges other than (s, t). Since all other edges have a capacity bound of 1, and the circulation is integer-valued, each such edge that carries flow has exactly one unit of flow on it.

Consider an edge (s, u_i) that carries one unit of flow. It follows by conservation that (u_i, v_i) carries one unit of flow, and that there is a unique edge out of v_i that carries one unit of flow. If we continue in this way, we construct a path P from s to t, so that each edge on this path carries one unit of flow. We can apply this construction to each edge of the form (s, u_j) carrying one unit of flow; in this way, we produce c' paths from s to t, each consisting of edges that carry one unit of flow.

Now, for each path P we create in this way, we can assign a single crew to perform all the flights contained in this path. In this way, we perform all the flights using only $c' \leq c$ flight crews. ■

Flight crew scheduling consumes countless hours of CPU time in real life. We mentioned at the beginning, however, that our formulation here is really a toy problem; it ignores several obvious factors that would have to be taken into account in applications. First of all, it ignores the fact that a given flight crew can only fly a certain number of hours in a given interval of time — this is in accordance with union regulations, and the common-sense fact that we don't want exhausted pilots to be flying planes. Second, we are making up an optimal schedule for a single day as though there were no yesterday or tomorrow; in fact we also need the flight crews to be optimally positioned for the start of day $N + 1$ at the end of day N. Third, while flight crews really do migrate between different cities as passengers — many of you have presumably seen pilots sometimes sitting in coach class — this comes at a cost; these are seats that cannot be sold at regular fares. And these issues don't even begin to cover the fact that we have the option of re-working our flight schedules to improve the efficiency of our operation; that we are simultaneously trying to plan a fare structure to optimize revenue; and so forth.

Ultimately, the message is probably this: flow techniques are useful for solving problems of this type, and they are genuinely used in practice. Indeed, our solution above is a general approach to the efficient re-use of a limited set of resources. At the same time, running an airline efficiently in real life a very difficult problem.

Image Segmentation

As some of you know from other classes here, a central problem in image processing is the *segmentation* of an image into various coherent regions. For example, you may have an image representing a picture of three people standing in front of a complex background scene; a natural but difficult goal is to identify each of the three people as coherent objects in the scene.

One of the most basic problems to be considered along these lines is that of foreground/background segmentation — we wish to label each pixel in an image as either belonging to the foreground of the scene or the background. It turns out that a very natural model here leads to a problem that can be solved efficiently by a minimum cut computation.

Let V be the set of *pixels* in the underlying image that we're analyzing. We will declare certain pairs of pixels to be *neighbors*, and use E to denote the set of all pairs of neighboring pixels. In this way, we obtain an *undirected* graph $G = (V, E)$. We will be deliberately vague on what exactly we mean by a "pixel," or what we mean by the "neighbor" relation — in fact any graph G will yield an efficiently solvable problem, so we are free to define these notions in any way that we want. Of course, it is natural to picture the pixels as constituting a grid of dots, and the neighbors of a pixel to be those that are directly adjacent to it in this grid.

For each pixel i, we have a *likelihood* a_i that it belongs to the foreground, and a *likelihood* b_i that it belongs to the background. For our purposes, we will assume that these likelihood values are arbitrary non-negative numbers provided as part of the problem, and that they specify how desirable it is to have pixel i in the background or foreground; beyond this, it is not crucial precisely what physical properties of the image they are measuring, or how they were determined.

So in isolation, we would want to label pixel i as belonging to the foreground if $a_i > b_i$, and to the background otherwise. However, decisions that we make about the neighbors of i should affect our decision about i. If many of i's neighbors are labeled "background," for example, we should be more inclined to label i as "background" too; for this makes the labeling "smoother" by minimizing the amount of foreground/background boundary. Thus for each pair (i, j) of neighboring pixels, there is a *separation penalty* $p_{ij} \geq 0$ for placing one of i or j in the foreground and the other in the background.

We can now specify our segmentation problem precisely, in terms of the likelihood and separation parameters — it is to find a partition of the set of pixels into sets A and B (foreground and background respectively) so as to maximize

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1|}} p_{ij}.$$

Thus, we are rewarded for having high likelihood values, and penalized for having neighboring pairs (i, j) with one pixel in A and the other in B.

We can find an *optimal labeling* — a partition (A, B) that maximizes $q(A, B)$ — as follows. We notice right away that there is clearly a resemblance to the minimum cut problem, with three significant differences. First, we are seeking to maximize an objective function rather than minimizing one. Second, it is not clear what should correspond to the source and the sink. Third, we have an undirected graph G, whereas for the minimum cut problem we want to work with a directed graph. Let's deal with problems in order.

We deal with the first issue through the following observation. Let $Q = \sum_i (a_i + b_i)$. The sum $\sum_{i \in A} a_i + \sum_{j \in B} b_j$ is the same as the sum $Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j$, so we can write

$$q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1|}} p_{ij}.$$

Thus we see that the maximization of $q(A, B)$ is the same problem as the minimization of the quantity

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1|}} p_{ij}.$$

As for the source and the sink, we work by analogy with our constructions of the previous section: we create a new "super-source" s to represent the foreground, and a new "super-sink" t to represent the background. To take care of the undirected edges, we do something very simple: we model each neighboring pair (i, j) with *two* directed edges, (i, j) and (j, i). We will see that this works very well, since in any s-t cut, at most one of these two oppositely directed edges can cross from the s-side to the t-side of the cut. (For if one does, then the other must go from the t-side to the s-side.)

Specifically, we define the following flow network $G' = (V', E')$. The node set V' consists of the set V of pixels, together with two additional nodes s and t. For each neighboring pair of pixels i and j, we add directed edges (i, j) and (j, i), each with capacity p_{ij}. For each pixel i, we add an edge (s, i) with capacity a_i and an edge (i, t) with capacity b_i.

Now, an s-t cut (A, B) corresponds to a partition of the pixels into sets A and B. The capacity of such a cut can be written as

$$\begin{aligned} c(A, B) &= \sum_{i \notin B} b_i + \sum_{i \notin A} a_i + \sum_{i \in A, j \in B} p_{ij} \\ &= \sum_{i \in A} b_i + \sum_{i \in B} a_i + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1|}} p_{ij} \\ &= q'(A, B). \end{aligned}$$

So finding a cut of minimum capacity is the same as minimizing the quantity $q'(A, B)$; and we have already argued that this allows us to solve the problem of maximizing $q(A, B)$.

Thus, through solving this minimum cut problem, we have an optimal algorithm in our model of foreground/background segmentation.

Project Selection

Say that some of your friends have formed a start-up company, and they have to decide on a set of initial projects to take on. Some of these projects can generate revenue (e.g. creating an interactive e-commerce interface for a potential client, or designing a new Web site for the Cornell CS Department); others of these projects will cost money (e.g. upgrading their computers, or getting site licenses for some useful encryption software). Of course, being able to do certain projects may depend on having completed others — perhaps they can't start on the e-commerce opportunity until they've got the encryption software. The question is: which projects should they pursue, and which should they allow to pass them by? It's an issue of balancing costs incurred with profitable opportunities that are made possible.

Here's a very general framework for modeling a set of decisions like this. There is an underlying set P of *projects*, and each project $i \in P$ has an associated *revenue* p_i, which can either be positive or negative. Certain projects are prerequisites for other projects, and we model this by an underlying directed acyclic graph $G = (P, E)$. The nodes of G are the projects, and there is an edge (i, j) to indicate that project i can only be selected if project j is selected as well. Note that a project i can have many prerequisites; the graph can have many directed edges whose tails are all equal to i. A set of projects $A \subseteq P$ is *feasible* if the prerequisite of every project in A also belongs to A: for each $i \in A$, and each edge $(i, j) \in E$, we also have $j \in A$. We will refer to requirements of this form as *precedence constraints*. The profit of a set of project is defined to be

$$\texttt{profit}(A) = \sum_{i \in A} p_i.$$

The *project selection problem* is to select a feasible set of projects with maximum profit.

This problem has been studied in the mining literature since the early 1960's; here it has been the called the *open-pit mining problem*.[1] Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. Before the mining operation begins, the entire area is divided into a set P of *blocks*, and the net value p_i of each block is estimated — this is the value of the ore minus the processing costs, for this block considered in isolation. Some of these net values will be positive, others negative. The full set of blocks has precedence constraints that essentially prevent blocks from being extracted before others on top of them are extracted. The open-pit mining problem is to determine the most profitable set of blocks to extract, subject to the precedence constraints. This problem clearly falls into the framework of project selection — each block corresponds to a separate project.

Here we will show that the project selection problem can be solved by reducing it to a minimum cut computation on an extended graph G', defined analogously to the graph we used earlier for image segmentation. The idea is to construct G' from G in such a way that the source side of a minimum cut in G' will correspond to an optimal set of projects to select.

To form the graph G' we add a new source s and a new sink t to the graph G. For each node $i \in P$ with $p_i > 0$, we add an edge (s, i) with capacity p_i. For each node $i \in P$ with $p_i < 0$, we add an edge (i, t) with capacity $-p_i$. We will set the capacities on the edges in G later. However, we can already see that the capacity of the cut $(\{s\}, P \cup \{t\})$ is $C = \sum_{i \in P : p_i > 0} p_i$, so the maximum flow value in this network is at most C.

We want to ensure that if (A', B') is a minimum cut in this graph, then $A = A' - \{s\}$ obeys the precedence constraints; i.e., if the node $i \in A$ has an out-going edge $(i, j) \in E$, then we must have $j \in A$. The conceptually cleanest way to ensure this is to give each of the

[1] When we discussed survey design, we made reference to the whimsically-named field of "data mining." Here, on the other hand, we're talking about real mining, where you dig things out of the ground.

edges in G capacity of ∞. We haven't previously formalized what an infinite capacity would mean, but there is no problem in doing this: it is simply an edge for which the capacity condition imposes no upper bound at all. The algorithms of the previous section, as well as the Max-Flow Min-Cut theorem, carry over to handle infinite capacities. However, we can also avoid bringing in the notion of infinite capacities by simply assigning each of these edges a capacity that is "effectively infinite". In our context, giving each of these edges a capacity of $C+1$ is would accomplish this: the maximum possible flow value in G' is at most C, and so no minimum cut can contain an edge with capacity above C. In the description below, it will not matter which of these options we choose.

First consider a set of projects A that satisfy the precedence constraints. Let $A' = A \cup \{s\}$ and $B' = (P - A) \cup \{t\}$, and consider the s-t cut (A', B'). If the set A satisfies the precedence constraints then no edge $(i, j) \in E$ crosses this cut. The capacity of the cut can be expressed as follows.

(6.46) *The capacity of the cut (A', B'), as defined from a project set A satisfying the precedence constraints, is $c(A', B') = C - \sum_{i \in A} p_i$.*

Proof. Edges of G' can be divided into three categories: those corresponding to the edge set E of G, those leaving the source s, and those entering the sink t. Because A satisfies the precedence constraints, the edges in E do not cross the cut (A', B'), and hence do not contribute to its capacity. The edges entering the sink t contribute

$$\sum_{i \in A \text{ and } p_i < 0} -p_i$$

to the capacity of the cut, and the edges leaving the source s contribute

$$\sum_{i \notin A \text{ and } p_i > 0} p_i.$$

Using the definition of C, we can re-write this latter quantity as $C - \sum_{i \in A \text{ and } p_i > 0} p_i$. The capacity of the cut (A', B') is the sum of these two terms, which is

$$- \sum_{i \in A \text{ and } p_i > 0} p_i + C - \sum_{i \in A \text{ and } p_i < 0} p_i = C - \sum_{i \in A} p_i,$$

as claimed. ■

Next, recall that edges of G have capacity more than $C = \sum_{i \in P : p_i > 0} p_i$, and so these edges cannot cross a cut of capacity at most C. This implies that such cuts define feasible sets of projects.

(6.47) *If (A', B') is a cut with capacity at most C, then the set $A = A' - \{s\}$ satisfies the precedence constraints.*

Now we can prove the main goal of our construction, that the minimum cut in G' determines the optimum set of projects. Putting the previous two claims together we see that the cuts (A', B') of capacity at most C are in one-to-one correspondence with feasible sets of project $A = A' - \{s\}$. The capacity of such a cut (A', B') is

$$c(A', B') = C - \texttt{profit}(A).$$

The capacity value C is a constant, independent of the cut (A', B'), so the cut with minimum capacity corresponds to the set of projects A with maximum profit. We have therefore proved the following.

(6.48) *If (A', B') is a minimum cut in G' then the set $A = A' - \{s\}$ is an optimum solution to the project selection problem.*

Baseball Elimination

> *Over on the radio side the producer's saying, "See that thing in the paper last week about Einstein? ... Some reporter asked him to figure out the mathematics of the pennant race. You know, one team wins so many of their remaining games, the other teams win this number or that number. What are the myriad possibilities? Who's got the edge?"*
> *"The hell does he know?"*
> *"Apparently not much. He picked the Dodgers to eliminate the Giants last Friday."*
>
> — *Don DeLillo,* Underworld.

Suppose you're a reporter for the Algorithmic Sporting News, and the following situation arises late one September. There are four baseball teams trying to finish in first place in the American League Eastern Division; let's call them New York, Baltimore, Toronto, and Boston. Currently, each team has the following number of wins:

New York: 92, Baltimore: 91, Toronto: 91, Boston: 90.

There are five games left in the season: these consist of all possible pairings of the above four teams, except for New York and Boston.

The question is: can Boston finish with at least as many wins as every other team in the Division? (That is: finish in first place, possibly in a tie.)

If you think about it, you realize that the answer is "no." One argument is the following. Clearly Boston must win both its remaining games and New York must lose both its remaining games. But this means that Baltimore and Toronto will both beat New York; so then the winner of the Baltimore-Toronto game will end up with the most wins.

Here's a cleaner argument. Boston can finish with at most 92 wins. Cumulatively, the other three teams have 274 wins currently, and their three games against each other will

produce exactly three more wins, for a final total of 277. But 277 wins over three teams means that one of them must have ended up with more than 92 wins.

So now you might start wondering: (i) Is there an efficient algorithm to determine whether a team has been eliminated from first place? And (ii) whenever a team has been eliminated from first place, is there an "averaging" argument like this that proves it?

In more concrete notation, suppose we have a set S of teams, and for each $x \in S$, its current number of wins w_x. Also, for two teams $x, y \in S$, they still have to play g_{xy} games against one another. Finally, we are given a specific team z.

We can use maximum flow techniques to achieve the following two things. First, we give an efficient algorithm to decide whether z has been eliminated from first place — or, to put it in positive terms, whether it is possible to choose outcomes for all the remaining games in such a way that the team z ends with at least as many wins as every other team in S. Second, we prove the following clean characterization theorem for baseball elimination — essentially, that there is always a short "proof" when a team has been eliminated.

(6.49) *Suppose that team z has indeed been eliminated. Then there exists a "proof" of this fact of the following form:*

- *z can finish with at most m wins.*
- *There is a set of teams $T \subseteq S$ so that*

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|.$$

(And hence one of the teams in T must end with strictly more than m wins.)

We begin by constructing a flow network that provides an efficient algorithm for determining whether z has been eliminated. Then, by examining the minimum cut in this network, we will prove (6.49).

Clearly, if there's any way for z to end up in first place, we should have z win all its remaining games — suppose that this leaves it with m wins. We now want to carefully allocate the wins from all remaining games so that no other team ends with more than m wins. Allocating wins in this way can be solved by a maximum flow computation, via the following basic idea. We have a source s from which all "wins" emanate. The i^{th} win can pass through one of the two teams involved in the i^{th} game. We then impose a capacity constraint saying that at most $m - w_x$ wins can pass through team x.

More concretely: Let $S' = S - \{z\}$. Let $g^* = \sum_{x,y \in S'} g_{xy}$ — the total number of games left between all pairs of teams in S'. Let G denote the following graph. There are nodes s and t, a node v_x for each team $x \in S'$ and a node u_{xy} for each pair of teams $x, y \in S'$ with a non-zero number of games left to play against each other. We have the following edges.

- Edges (s, u_{xy}) (*wins emanate from s*);
- Edges (u_{xy}, v_x) and (u_{xy}, v_y) (*only x or y can win a game that they play against each other*); and
- Edges (v_x, t) (*wins are absorbed at t).*

Let's consider what capacities we want to place on these edges. We want g_{xy} wins to flow from s to u_{xy} at saturation, so we give (s, u_{xy}) a capacity of g_{xy}. We want to ensure that team x cannot win more than $m - w_x$ games, so we give the edge (v_x, t) a capacity of $m - w_x$. Finally, an edge of the form (u_{xy}, v_y) should have *at least* g_{xy} units of capacity, so that it has the ability to transport all the wins from u_{xy} on to v_x; in fact, our analysis will be the cleanest if we give it *infinite* capacity. (We note that the construction still works even if this edge is given only g_{xy} units of capacity, but the proof of (6.49) becomes a little more complicated.)

It is now easy to check that z is not yet eliminated from first place if and only if there is an s-t flow of value g^*; this gives a polynomial-time algorithm to test for this property.

Proof of (6.49). Suppose that z has been eliminated from first place. Then the maximum s-t flow in G has value $g' < g^*$; so there is an s-t cut (A, B) of capacity g', and (A, B) is a minimum cut. Let T be the set of teams x for which $v_x \in A$.

Now consider the node u_{xy}. Suppose one of x or y is not in T, but $u_{xy} \in A$. Then the edge (u_{xy}, v_x) would cross from A into B, and hence the cut (A, B) would have infinite capacity — this contradicts the assumption that (A, B) is minimum cut of capacity less than g^*. So if one of x or y is not in T, then $u_{xy} \in B$. On the other hand, suppose both x and y belong to T, but $u_{xy} \in B$. Consider the cut (A', B') that we would obtain by adding u_{xy} to the set A and deleting it from the set B. The capacity of (A', B') is simply the capacity of (A, B), minus the capacity g_{xy} of the edge (s, u_{xy}) — for this edge (s, u_{xy}) used to cross from A to B, and now it does not cross from A' to B'. But since $g_{xy} > 0$, this means that (A', B') has smaller capacity than (A, B), again contradicting our assumption that (A, B) is a minimum cut. So if both x and y belong to T, then $u_{xy} \in A$.

Thus we have established the following conclusion, based on the fact that (A, B) is a minimum cut: $u_{xy} \in A$ if and only if $x, y \in T$.

Let Γ be the collection of all unordered pairs of teams in T; and let $\overline{\Gamma}$ be the collection of all unordered pairs from S' in which (at least) one member does not belong to T. Note that an unordered pair of teams is in exactly one of Γ or $\overline{\Gamma}$; and our fact in the previous paragraph says that $u_{xy} \in A$ if and only if (x, y) belongs to Γ and not $\overline{\Gamma}$. So

$$\begin{aligned} c(A, B) &= \sum_{x \in T} (m - w_x) + \sum_{\{x,y\} \in \overline{\Gamma}} g_{xy} \\ &= m|T| - \sum_{x \in T} w_x + (g^* - \sum_{\{x,y\} \in \Gamma} g_{xy}). \end{aligned}$$

Since we know that $c(A, B) = g' < g^*$, this last inequality implies

$$m|T| - \sum_{x \in T} w_x - \sum_{\{x,y\} \in \Gamma} g_{xy} < 0,$$

and hence

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|.$$

■

6.10 Minimum Cost Perfect Matchings

Let's go back to the first problem we discussed in this chapter, bipartite matching. Perfect matchings in a bipartite graph formed a way to model the problem of pairing one kind of object with one another — jobs with machines, for example. But in many settings, there are a large number of possible perfect matchings on the same set of objects, and we'd like a way to express the idea that some perfect matchings may be "better" than others.

A natural way to do this is to introduce *costs*. It may be that we incur a certain cost to perform a given job on a given machine, and we'd like to match jobs with machines in a way that minimizes the total cost. Or there may be n fire trucks that must be sent to n distinct houses; each house is at a given distance from each fire station, and we'd like a matching that minimizes the average distance each truck drives to its associated house. In short, it is very useful to have an algorithm that finds a perfect matching in a bipartite *of minimum total cost*.

Formally, we consider a bipartite graph $G = (V, E)$ whose node set, as usual, is partitioned as $V = X \cup Y$ so that every edge $e \in E$ has one end in X and the other end in Y. Furthermore, each edge e has a non-negative cost c_e. For a matching M, we say that the cost of the matching is the total cost of all edges in M, i.e., $cost(M) = \sum_{e \in M} c_e$. The *Minimum-Cost Perfect Matching* problem assumes that $|X| = |Y| = n$, and the goal is to find a perfect matching of minimum cost.

We now describe an efficient algorithm to solve this problem, based on the idea of augmenting paths but adapted to take the costs into account. In developing the solution, it is useful to consider a slightly more general version: we will find a minimum cost matching using i edges, for each value of i from 1 to n. Clearly this will include the solution to the minimum cost perfect matching problem, which is the case $i = n$. The high-level structure of the algorithm is quite simple. If we have a minimum-cost matching of size i, then we seek an augmenting path to produce a matching of size $i+1$; and rather than looking for any augmenting path (as was sufficient in the case without costs), we use the cheapest augmenting path so that the larger matching will also have minimum cost.

Recall the construction of the residual graph used for finding augmenting paths. Let M be a matching. We add two new nodes s and t to the graph. We add edges (s, x) for all

nodes $x \in X$ that are unmatched, edges (y, t) for all nodes $y \in Y$ that are unmatched. An edge $e = (x, y) \in E$ is oriented from x to y if e is not in the matching M and from y to x if $e \in M$. We will use G_M to denote this residual graph. Note that all edges going from Y to X are in the matching M, while the edges going from X to Y are not. Any directed s-t path P in the graph G_M corresponds to a matching one larger than M by swapping edges along P, i.e., the edges in P from X to Y are added to M and all edges in P that go from Y to X are deleted from M. As before, we will call a path P in G_M an *augmenting path*, and we say that we *augment* the matching M using the path P.

Now, we would like to have the resulting matching have as small costs as possible. To achieve this, we will search for a cheap augmenting path with respect to the following natural costs. The edges leaving s and entering t will have cost 0; an edge e oriented from X to Y will cost c_e (as including this edge in the path means that we add the edge to M); and an edge e oriented from Y to X will cost $-c_e$ (as including this edge in the path means that we delete the edge from M). We will use $cost(P)$ to denote the cost of a path P in G_M. The following statement summarizes this construction.

(6.50) *Let M be a matching and P be a path in G_M from s to t. Let M' be the matching obtained from M by augmenting along P. Then $|M'| = |M| + 1$ and $cost(M') = cost(M) + cost(P)$.*

Given this statement, it is natural to suggest an algorithm to find a minimum cost perfect matching: we iteratively find minimum cost paths in G_M, and use the paths to augment the matchings. But how can we be sure that the perfect matching we find is of minimum cost? Or even worse, is this algorithm even meaningful? We can only find minimum cost paths if we know that the graph G_M has no negative cycles.

In fact, understanding the role of negative cycles in G_M is the key to analyzing the algorithm. First, consider the case in which M is a perfect matching. The nodes s and t have no incident edges in G_M, as our matching is perfect, but the rest of the graph and costs are still meaningful.

(6.51) *Let M be a perfect matching. If there is a negative cost directed cycle C in G_M then M is not minimum cost.*

Proof. To see this, we use the cycle C for augmentation, just the same way we used directed paths to obtain larger matchings. Augmenting M along C involves swapping edges along C in and out of M. The resulting new perfect matching M' has cost $cost(M') = cost(M)+cost(C)$; but $cost(C) < 0$, and and hence M is not of minimum cost. ■

More importantly, the converse of this statement is true as well; so in fact a perfect matching M has minimum cost precisely when there is no negative cycle in G_M.

(6.52) *Let M be a perfect matching. If there are no negative cost directed cycles C in G_M, then M is a minimum cost perfect matching.*

Proof. Suppose the statement is not true, and let M' be a perfect matching of smaller cost. Consider the set of edges that are in one of M and M' but not in both. Observe that these set of edges correspond to a set of node-disjoint directed cycles in G_M. The cost of the set of directed cycles is exactly $cost(M') - cost(M)$. Assuming M' has smaller cost than M, it must be that at least one of these cycles has negative cost. ■

Our plan is thus to iterate through matchings of larger and larger size, maintaining the property that the graph G_M has no negative cycles in any iteration. In this way, our computation of a minimum-cost path will always be well-defined; and when we terminate with a perfect matching, we can use (6.52) to conclude that it has minimum cost.

As we run the algorithm, we will in fact maintain a numerical *price* $p(v)$ associated with each node v; these will turn out to serve as a compact proof that G_M has no negative cycles. Specifically, we say that a set of numbers $\{p(v) : v \in V\}$ forms a set of *compatible prices* with respect to a matching M if

(i) for all unmatched nodes $x \in X$ we have $p(x) = 0$;

(ii) for all edges $e = (x, y)$ we have $p(x) + c_e \geq p(y)$; and

(iii) for all edges $e = (x, y) \in M$ we have $p(x) + c_e = p(y)$.

Why are such prices useful? We claim first of all that if M is any matching for which there exists a set of compatible prices, then G_M has no negative cycles. To see this, let us define the *reduced cost* of an edge e to be $c_e^p = p(v) + c_e - p(w)$. Note that for any cycle C, we have

$$cost(C) = \sum_{e \in C} c_e = \sum_{e \in C} c_e^p,$$

since all the terms on the right-hand-side corresponding to prices cancel out. But by the definition of compatible prices, we know that each term on the right-hand-side is non-negative, and so clearly $cost(C)$ is non-negative.

There is a second, algorithmic reason why it is useful to have explicit prices on the nodes. When you have a graph with negative-cost edges but no negative cycles, you can compute shortest paths using the Bellman-Ford algorithm in $O(mn)$ time. But if the graph in fact has no negative-cost edges, then you can use Dijkstra's algorithm instead, which only requires time $O(m \log n)$ — almost a full factor of n faster.

In our case, having the prices around allows us to compute shortest paths with respect to the non-negative reduced costs c_e^p, arriving at an equivalent answer. Indeed, suppose we use Dijkstra's algorithm to find the minimum cost $d_{p,M}(v)$ of a directed path from s to every node $v \in X \cup Y$ subject to the costs c_e^p. Given the minimum costs $d_{p,M}(y)$ for an unmatched nodes $y \in Y$, the (non-reduced) cost of the path from s to t through y is $d_{p,M}(y) + p(y)$, and

so we find the minimum cost in $O(n)$ additional time. In summary, we have the following fact.

(6.53) *Let M be a matching, and p be compatible prices. We can use one run of Dijkstra's algorithm and $O(n)$ extra time to find the minimum cost path from s to t.*

This handles one iteration of the algorithm, provided we have a set of compatible prices. In order to be ready for the next iteration, we need not only the minimum-cost path, but also a way to produce a set of compatible prices with respect to the new matching. Thus can be done using the fact that Dijkstra's algorithm computes the distances to all nodes.

(6.54) *Let M be a matching, p be compatible prices, and let M' be matching obtained by augmenting along the minimum cost path from s to t. Then $p'(v) = d_{p,M}(v) + p(v)$ is a compatible set of prices for M'.*

Proof. To prove compatibility, consider first an edge $e = (x, y) \in M$. The only edge entering x is the directed edge (y, x) and hence $d_{p,M}(x) = d_{p,M}(y) - c_e^p$, where $c_e^p = p(x) + c_e - p(x)$, and hence we get the desired equation on such edges. Next consider edges in $M' - M$. These edges are along the minimum cost path from s to t, and hence they satisfy $d_{p,M}(y) = d_{p,M}(y) + c_e^p$ as desired. Finally, we get the required inequality for all other edges by considering the fact that all edges $e = (x, y) \notin M$ must satisfy $d_{p,M}(y) \leq d_{p,M}(y) + c_e^p$. ■

Finally, we have to consider how to initialize the algorithm, so as to get it underway. We initialize M to be the empty set, define $p(x) = 0$ for all $x \in X$, and define $p(y)$, for $y \in Y$, to be the minimum cost of an edge entering y. Note that these prices are compatible with respect to $M = \phi$.

We summarize the algorithm below.

```
Start with M equal to the empty set
Define p(x) = 0 for x ∈ X, and p(y) = min_{e into y} c_e for y ∈ Y.
While M is not a perfect matching
  Find a minimum-cost s-t path P in G_M using  (6.53)
       with respect to the reduced costs c_e^p.
  Augment along P to produce a new matching M'.
  Produce a set of compatible prices with respect to M' via  (6.54)
.
EndWhile
```

The final set of compatible prices yields a proof that G_M has no negative cycles; and by (6.52), this implies that M has minimum cost.

To conclude our discussion of the minimum-cost perfect matching problem, we develop a natural economic interpretation of the prices. Consider the following scenario. Assume

X is a set of n people looking to buy a house, and Y is a set of n houses that they are all considering. Let $v(x, y)$ denote the value of house y to buyer y. Since each buyer wants one of the houses, one could argue that the best arrangement would be to find a perfect matching M that maximizes $\sum_{(x,y)\in M} v(x, y)$. We can find such a perfect matching by using our minimum cost perfect matching algorithm with costs $c_e = -v(x, y)$ if $e = (x, y)$.

The question we will ask now is this: Can we convince these buyers to buy the house they are allocated? On her own, each buyer x would want to buy the house y that has maximum value $v(x, y)$ to her. How can we convince her to buy instead the house that our matching M allocated? We will use prices to change the incentives of incentives of the buyers. Suppose we set a price $P(y)$ for each house y, i.e., the buyer buying the house y must pay $P(y)$. With these prices in mind, a buyer will be interested in buying the house with maximum net value, i.e., will want to chose the house y that maximizes $v(x, y) - P(y)$. We say that a perfect matching M and house prices P are in *equilibrium*, if for all edges $(x, y) \in M$ and all other houses y' we have that

$$v(x, y) - P(y) \geq v(x, y') - P(y').$$

But can we find a perfect matching and a set of prices so as to achieve this state of affairs, with every buyer ending up happy? In fact, the minimum-cost perfect matching and an associated set of compatible prices provide exactly what we're looking for.

(6.55) *Let M be a perfect matching of minimum cost, where cost$(x, y) = -v(x, y)$, and let p be compatible set of prices. Then the matching M and the set of prices $\{P(y) = -p(y) : y \in Y\}$ are in equilibrium.*

Proof. Consider an edge $e = (x, y) \in M$, and $e' = (x, y')$. We have that M and p are compatible, hence $p(x) + c_e = p(y)$ and $p(x) + c_{e'} \geq p(y')$. Subtracting these two inequalities to cancel $p(x)$, and substituting the values of P and c we get the desired inequality. ■

6.11 Exercises

1. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum s-t flow in G, defined by a flow value f_e on each edge e. The flow $\{f_e\}$ is *acyclic*: there is no cycle in G on which all edges carry positive flow.

   Now, suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m)$, where m is the number of edges in G.

2. Consider the following problem. You are given a flow network with unit-capacity edges: it consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k.

   The goal is delete k edges so as to reduce the maxmimum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

   Give a polynomial-time algorithm to solve this problem.

3. Suppose you're looking at a flow network G with source s and sink t, and you want to be able to express something like the following intuitive notion: some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

   Here's one way to divide the nodes of G into three categories of this sort.

   - We say a node v is *upstream* if for all minimum s-t cuts (A, B), we have $v \in A$ — that is, v lies on the source side of every minimum cut.
   - We say a node v is *downstream* if for all minimum s-t cuts (A, B), we have $v \in B$ — that is, v lies on the sink side of every minimum cut.
   - We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s-t cut (A, B) for which $v \in A$, and at least one minimum s-t cut (A', B') for which $v \in B'$.

   Give an algorithm that takes a flow network G, and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within in a constant factor of the time required to compute a *single* maximum flow.

4. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and non-negative edge capacities $\{c_e\}$. Give a polynomial time algorithm to decide whether G has a *unique* minimum s-t cut. (I.e. an s-t of capacity strictly less than that of all other s-t cuts.)

5. In a standard minimum s-t cut problem, we assume that all capacities are non-negative; allowing an arbitrary set of positive and negative capacities results in an NP-complete problem. (You don't have to prove this.) However, as we'll see here, it is possible to relax the non-negativity requirement a little, and still have a problem that can be solved in polynomial time.

   Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and edge capacities $\{c_e\}$. Suppose that for every edge e that has neither s nor t as an endpoint, we have

$c_e \geq 0$. Thus, c_e can be negative for edges e that have at least one end equal to either s or t. Give a polynomial-time algorithm to find an s-t cut of minimum value in such a graph. (Despite the new non-negativity requirements, we still define the value of an s-t cut (A, B) to be the sum of the capacities of all edges e for which the tail of e is in A and the head of e is in B.)

6. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A *diagonal entry* is one of the form m_{ii} for some i.

   Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, \ldots, n$. Swapping two columns is defined analogously.

   We say that M is *re-arrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of M are equal to 1.

   (a) Give an example of a matrix M which is not re-arrangeable, but for which at least one entry in each row and each column is equal to 1.

   (b) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries, is re-arrangeable.

7. You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus, there is a collection of n wireless laptops; there is also have a collection of n wireless *access points*, to which a laptop can connect when it is in range.

   The laptops are currently scattered across campus; laptop ℓ is within range of a *set* S_ℓ of access points. We will assume that each laptop is within range of at least one access point (so the sets S_ℓ are non-empty); we will also assume that every access point p has at least one laptop within range of it.

   To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points, in such a way that each laptop and each access point is involved in at least one connection. Thus, we will say that a *test set* T is a collection of ordered pairs of the form (ℓ, p), for a laptop ℓ and access point p, with the properties that

   (i) If $(\ell, p) \in T$, then ℓ is within range of p. (I.e. $p \in S_\ell$).

   (ii) Each laptop appears in at least one ordered pair in T.

   (iii) Each access point appears in at least one ordered pair in T.

   This way, by trying out all the connections specified by the pairs in T, we can be sure that each laptop and each access point have correctly functioning software.

The problem is: Given the sets S_ℓ for each laptop (i.e. which laptops are within range of which access points), and a number k, decide whether there is a test set of size at most k.

Example: Suppose that $n = 3$; laptop 1 is within range of access points 1 and 2; laptop 2 is within range of access point 2; and laptop 3 is within range of access points 2 and 3. Then the set of pairs

(laptop 1, access point 1), (laptop 2, access point 2),
(laptop 3, access point 3)

would form a test set of size three.

(a) Give an example of an instance of this problem for which there is no test set of size n. (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)

(b) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k), and decides whether there is a test set of size at most k.

8. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like *Yahoo!* was in the "eyeballs" — the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if the user has told *Yahoo!* that they're a 20-year old computer science major from Cornell University, the site can throw up a banner ad for apartments in Ithaca, NY; on the other hand, if they're a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

   But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups* $G_1, G_2, \ldots, G_k$. (These groups can overlap; for example G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like:

- For a subset $X_i \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now, consider the problem of designing a good *advertising policy* — a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i, are shown an ad provided by advertiser i.)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

9. Some of your friends have recently graduated and started a small company called WebExodus, which they are currently running out of their parents' garages in Santa Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

   They have a collection of n software applications, $\{1, 2, \ldots, n\}$, running on their old system; and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both — let's denote this expense by $x_{ij} \geq 0$.

   So if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem ...

   Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

   So this is the question they pose to you: which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

10. Consider a variation on the previous problem. In the new scenario, any application can potentially be moved, but now some of the benefits b_i for moving to the new system are in fact *negative*: if $b_i < 0$, then it is preferable (by an amount quantifed in b_i) to keep i on the old system. Again, give a polynomial-time algorithm to find a set $S \subseteq \{1, 2, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

11. Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have the total value e_{ij} of all exports from i to j; this number is always non-negative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S, minus the total value of all exports from countries in S to countries not in S, is non-negative.

    Give a polynomial-time algorithm that takes this data for a set of n countries, and decides whether it contains a non-empty free-standing subset that is not equal to the full set.

12. ($*$) In sociology, one often studies a graph G in which nodes represent people, and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

    Now, suppose we want to study this graph G, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a subset S of nodes let $e(S)$ denote the number of edges in S, i.e., the number of edges that have both ends in S. We define the *cohesiveness* of S as $e(S)/|S|$. A natural thing to search for would be the set S of people achieving the maximum cohesiveness.

    (a.) Give a polynomial time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α.

    (b.) Give a polynomial time algorithm to find a set S of nodes with maximum cohesiveness

13. Suppose we are given a directed network $G = (V, E)$ with a root node r, and a set of *terminals* $T \subseteq V$. We'd like to disconnect many terminals from r, while cutting relatively few edges.

    We make this trade-off precise as follows. For a set of edges $F \subseteq E$, let $q(F)$ denote the number of nodes $v \in T$ such that there is no r-v path in the subgraph $(V, E - F)$. Give a polynomial-time algorithm to find a set F of edges that maximizes the quantity $q(F) - |F|$. (Note that setting F equal to the empty set is an option.)

14. Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

    Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair, and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

    Here's one way to define *fairness*. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i^{th} day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; unfortunately, Δ_j may not be an integer.

    So let's say that a *driving schedule* is a choice of a driver for each day — i.e. a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ — and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

    (a) Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

    (b) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d.

    (c)$(*)$ One could expect k to be a much smaller parameter than d (e.g. perhaps $k = 5$ and $d = 365$). So it could be worth reducing the dependence of the running time on d even at the expense of a much worse dependence on k. Give an algorithm to compute a fair driving schedule whose running time has the form $O(f(k) \cdot d)$, where $f(\cdot)$ can be an arbitrary function.

15. Suppose you live in a big apartment with a bunch of friends. Over the course of a year, there are a lot of occasions when one of you pays for an expense shared by some subset of the apartment, with the expectation that everything will get balanced out fairly at the end of the year. For example, one of you may pay the whole phone bill in a given month, another will occasionally make communal grocery runs to the nearby organic food emporium; and a third might sometimes use a credit card to cover the whole bill at the local Italian-Indian restaurant, *Little Idli*.

    In any case, it's now the end of the year, and time to settle up. There are n people in the apartment; and for each ordered pair (i, j) there's an amount $a_{ij} \geq 0$ that i owes j, accumulated over the course of the year. We will require that for any two people i

and j, at least one of the quantities a_{ij} or a_{ji} is equal to 0. This can be easily made to happen as follows: if it turns out that i owes j a positive amount x, and j owes i a positive amount $y < x$, then we will subtract off y from both sides and declare $a_{ij} = x - y$ while $a_{ji} = 0$. In terms of all these quantities, we now define the *imbalance* of a person i to be the sum of the amounts that i is owed by everyone else, minus the sum of the amounts that i owes everyone else. (Note that an imbalance can be positive, negative, or zero.)

In order to restore all imbalances to 0, so that everyone departs on good terms, certain people will write checks to others; in other words, for certain ordered pairs (i, j), i will write a check to j for an amount $b_{ij} > 0$. We will say that a set of checks constitutes a *reconciliation* if for each person i, the total value of the checks received by i, minus the total value of the checks written by i, is equal to the imbalance of i. Finally, you and your friends feel it is bad form for i to write j a check if i did not actually owe j money, so we say that a reconciliation is *consistent* if, whenever i writes a check to j, it is the case that $a_{ij} > 0$.

Show that for any set of amounts a_{ij} there is always a consistent reconciliation in which at most $n - 1$ checks get written, by giving a polynomial-time algorithm to compute such a reconciliation.

16. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified its by (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

    For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a *range parameter* r — a client can only be connected to a base station that is within distance r. There is also a *load parameter* L — no more than L clients can be connected to any single base station.

    Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

17. You can tell that cellular phones are at work in rural communities, from the giant microwave towers you sometimes see sprouting out of corn fields and cow pastures. Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

We are given the locations of n *base stations*, specified as points $b_1, \ldots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \ldots, p_n$ in the plane. Finally, we are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

- Each phone is assigned to a different base station, and
- If a phone at p_i is assigned to a base station at b_j, then the straight-line distance between the points p_i and b_j is at most Δ.

Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones *fully connected*.

Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained.

You should try to make your algorithm run in $O(n^3)$ time if possible.

Example: Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. Then it is possible to keep the phones fully connected during this motion: We begin by assigning p_1 to b_1 and p_2 to b_2, and we re-assign p_1 to b_2 and p_2 to b_1 during the motion. (For example, when p_1 passes the point $(2, 0)$.)

18. Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

    The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be pre-empted and resumed from where it left off (possibly after a delay) on another processor.

    Moreover, the collection of processors is not entirely static either: you have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t'_i]$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deeadlines, or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done:

- At time 0, we start job J_1 on processor P_1.
- At time 1, we pre-empt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1.

Notice that there is no solution that does not involve pre-emption and moving of jobs.

19. In a lot of numerical computations, we can ask about the "stability" or "robustness" of the answer. This kind of question can be asked for combinatorial problems as well; here's one way of phrasing the question for the minimum spanning tree problem.

    Suppose you are given a graph $G = (V, E)$, with a cost c_e on each edge e. We view the costs as quantities that have been measured experimentally, subject to possible errors in measurement. Thus, the minimum spanning tree one computes for G may not in fact be the "real" minimum spanning tree.

    Given error parameters $\varepsilon > 0$ and $k > 0$, and a specific edge $e' = (u, v)$, you would like to be able to make a claim of the following form:

    > ($*$) Even if the cost of *each* edge were to be changed by at most ε (either increased or decreased), and the costs of k of the edges *other than* e' were further changed to arbitrarily different values, the edge e' would still not belong to any minimum spanning tree of G.

    Such a property provides a type of guarantee that e' is not likely to be belong to the minimum spanning tree, even assuming significant measurement error.

    Give a polynomial-time algorithm that takes G, e', ε, and k, and decides whether or not property ($*$) holds for e'.

20. Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

$$|\{(u, v) : (u, v) \in E)\}| = |\{(v, w) : (v, w) \in E)\}|.$$

Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x? Give a proof, or a counter-example with explanation.

21. Given a graph $G = (V, E)$, and a natural number k, we can define a relation $\xrightarrow{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \xrightarrow{G,k} y$ if there exist k mutually edge-disjoint paths from x to y in G.

Is it true that for every G and every $k \geq 0$, the relation $\xrightarrow{G,k}$ is transitive? That is, is it always the case that if $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, then we have $x \xrightarrow{G,k} z$? Give a proof or a counter-example.

22. Give a polynomial time algorithm for the following minimization analogue of the max-flow problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f, and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow, i.e., they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

    (a) Give a polynomial time algorithm that finds a feasible flow of minimum possible value.

    (b) Prove an analog of the max-flow min-cut theorem for this problem (i.e., does min-flow = max-cut?)

23. We define the *escape problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads); a certain collection of nodes $X \subset V$ are designated as *populated nodes*, and a certain other collection $S \subset V$ are designated as *safe nodes*. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

    (a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus, we change (iii) to say "the paths do not share any *nodes*."

With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

Also, provide an example with a given G, X, and S, in which the answer is "yes" to the question in (a) but "no" to the question in (b).

24. You are helping the medical consulting firm *Doctors Without Weekends* set up a system for arranging the work schedules of doctors in a large hospital. For each of the next n days, the hospital has determined the number of doctors they want on hand; thus, on day i, they have a requirement that *exactly* p_i doctors be present.

There are k doctors, and each is asked to provide a list of days on which he or she is willing to work. Thus, doctor j provides a set L_j of days on which he or she is willing to work.

The system produced by the consulting firm should take these lists, and try to return to each doctor j a list L'_j with the following properties.

(A) L'_j is a subset L_j, so that doctor j only works on days he or she finds acceptable.

(B) If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

(a) Describe a polynomial-time algorithm that implements this system. Specifically, give a polynomial-time algorithm that takes the numbers $p_1, p_2, \ldots, p_n$, and the lists $L_1, \ldots, L_k$, and does one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A) and (B).

You should prove your algorithm is correct, and briefly analyze the running time.

(b) The hospital finds that the doctors tend to submit lists that are much too restrictive, and so it often happens that the system reports (correctly, but unfortunately) that no acceptable set of lists $L'_1, L'_2, \ldots, L'_k$ exists.

Thus, the hospital relaxes the requirements as follows. They add a new parameter $c > 0$, and the system now should try to return to each doctor j a list L'_j with the following properties.

(A*) L'_j contains at most c days that do not appear on the list L_j.

(B) *(Same as before.)* If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

Describe a polynomial-time algorithm that implements this revised system. It should take the numbers $p_1, p_2, \ldots, p_n$, the lists $L_1, \ldots, L_k$, and the parameter $c > 0$, and do one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A*) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A*) and (B).

In this question, you need only describe the algorithm; you do not need to explicitly write a proof of correctness or running time analysis. (However, your algorithm must be correct, and run in polynomial time.)

25. You are consulting for an environmental statistics firm. They collect statistics, and publish the collected data in a book. The statistics is about populations of different regions in the world, and is in the millions. Examples of such statistics would look like the top table.

Country	A	B	C	Total
grownup men	11.998	9.083	2.919	24.000
grownup women	12.983	10,872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

We will assume here for simplicity that our data is such that all row and column sums are integers. The census rounding problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down without. For example a good rounding for the above data would be as follows.

Country	A	B	C	Total
grownup men	11.000	10.000	3.000	24.000
grownup women	13.000	10.000	4.000	27.000
children	2.000	2.000	0.000	4.000
Total	26.000	22.000	7.000	55.000

(a) Consider first the special case when all data is between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each

fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.

(b) Consider the census data rounding problem as defined above, where row and column sums are integers, and you want round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

(c) Prove that the rounding we are looking for in (a) and (b) always exists.

26. $(*)$ Some friends of yours have grown tired of the game "Six degrees of Kevin Bacon" (after all, they ask, isn't it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here's how it works.

    You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. P_0 names an actress $x_1 \in X$, P_1 names an actor y_1 who has appeared in a movie with x_1, P_0 names an actress x_2 who has appeared in a movie with y_1, and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \ldots$ such that each actor/actress in the sequence has co-starred with the actress/actor immediately preceding. A player P_i $(i = 0, 1)$ loses when it is P_i's turn to move, and he/she cannot name a member of his/her set who hasn't been named before.

    Suppose you are given a specific pair of such sets X and Y, with complete information on who has appeared in a movie with whom. A *strategy* for P_i, in our setting, is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \ldots$ and generates a legal next move for P_i (assuming it's P_i's turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

27. Statistically, the arrival of spring typically results in increased accidents, and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether their blood supply is sufficient.

    The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.[2]

[2]The Austrian scientist Karl Landsteiner received the Nobel Prize in 1930 for his discovery of the blood types A, B, O, and AB.

(a) Let s_O, s_A, s_B and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B and d_{AB} for the coming week. Give a polynomial time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in US patients is 45% type O, 42% type A, 10% type B, and 3% type AB. The hospital wants to know if the blood supply they have on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type:	O	A	B	AB
supply:	50	36	11	8
demand:	45	42	10	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words "flow," "cut," or "graph" in the sense we use them in CS 482.)

28. Suppose you and your friend Alanis live, together with $n-2$ other people, at a popular off-campus co-operative apartment, The Upson Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

Of course, everyone has scheduling conflicts with some of the nights (e.g. prelims, concerts, etc.) — so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people

$$\{p_1, \ldots, p_n\},$$

the nights

$$\{d_1, \ldots, d_n\};$$

and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are *not* able to cook.

A *feasible dinner schedule* is an assignment of each person in the co-op to a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Anyway, your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule, and heads off to class for the day.

Unfortunately, when you look at the schedule she created, you notice a big problem. $n-2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people — p_i and p_j — and the other two days — d_k and d_ℓ — you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_ℓ.

You want to fix Alanis's mistake, but without having to re-compute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

29. We consider the *bipartite matching* problem on a bipartite graph $G = (V, E)$. As usual, we say that V is partitioned into sets X and Y, and each edge has one end in X and the other in Y.

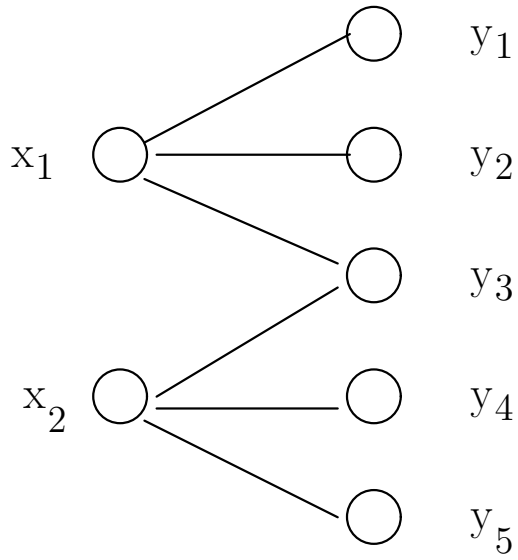


Figure 6.3: An instance of Coverage Expansion.

If M is a matching in G, we say that a node $y \in Y$ is *covered* by M if y is an end of one of the edges in M.

(a) Consider the following problem. We are given G and a matching M in G. For a given number k, we want to decide if there is a matching M' in G so that

(i) M' has k more edges than M does, *and*

(ii) every node $y \in Y$ that is covered by M is also covered by M'.

We call this the Coverage Expansion problem, with input G, M, and k. and we will say that M' is a *solution* to the instance.

Give a polynomial-time algorithm that takes an instance of Coverage Expansion and either returns a solution M' or reports (correctly) that there is no solution. (You should include an analysis of the running time, and a brief proof of why it is correct.)

Note: You may wish to also look at part (b) to help in thinking about this.

Example: Consider the accompanying figure, and suppose M is the matching consisting of the one edge (x_1, y_2). Suppose we are asked the above question with $k = 1$.

Then the answer to this instance of Coverage Expansion is "yes." We can let M' be the matching consisting (for example) of the two edges (x_1, y_2) and (x_2, y_4); M' it has 1 more edge than M, and y_2 is still covered by M'.

(b) Give an example of an instance of Coverage Expansion — specified by G, M, and k — so that the following situation happens.

> The instance has a solution; but in any solution M', the edges of M do not form a subset of the edges of M'.

(c) Let G be a bipartite graph, and let M be any matching in G. Consider the following two quantities.

- K_1 is the size of the largest matching M' so that every node y that is covered by M is also covered by M'.
- K_2 is the size of the largest matching M'' in G.

Clearly $K_1 \leq K_2$, since K_2 is obtained by considering *all possible* matchings in G.

Prove that in fact $K_1 = K_2$; that is, we can obtain a maximum matching even if we're constrained to cover all the nodes covered by our initial matching M.

30. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the set-up of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one

switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

Sometimes this is possible, and sometimes it isn't. Consider the two simple floor plans for houses in the accompanying figure. There are three light fixtures (labeled a, b, c) and three switches (labeled 1, 2, 3). It is possible to wire switches to fixtures in the example on the left so that every switch has line-of-sight to the fixture, but this is not possible in the example on the right.

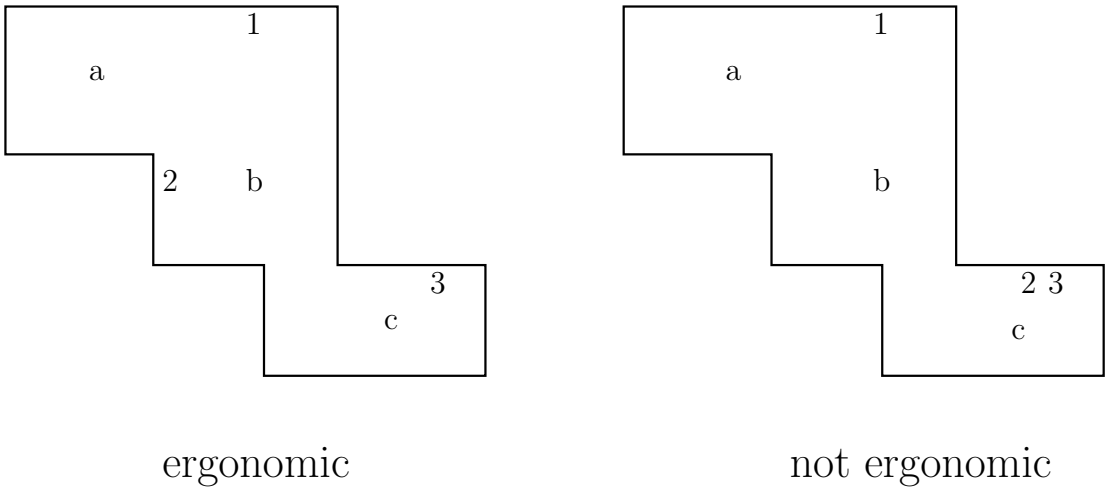


Figure 6.4: Two floor plans with lights and switches.

Let's call a floor plan — together with n light fixture locations and n switch locations — "ergonomic" if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i^{th} wall has endpoints $(x_i, y_i), (x'_i, y'_i)$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan, in the above representation, is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

31. Some of your friends are interning at the small high-tech company WebExodus. A running joke among the employees there is that the back room has less space devoted to high-end servers than it does to empty boxes of computer equipment, piled up in case something needs to be shipped back to the supplier for maintainence.

A few days ago, a large shipment of computer monitors arrived, each in its own large box; and since there are many different kinds of monitors in the shipment, the boxes do not all have the same dimensions. A bunch of people spent some time in the morning trying to figure out how to store all these things, realizing of course that less space would be taken up if some of the boxes could be *nested* inside others.

Suppose each box i is a rectangular parallelepiped with side lengths equal to (i_1, i_2, i_3); and suppose each side length is strictly between half a meter and one meter. Geometrically, you know what it means for one box to nest inside another — it's possible if you can rotate the smaller so that it fits inside the larger in each dimension. Formally, we can say that box i with dimensions (i_1, i_2, i_3) *nests* inside box j with dimensions (j_1, j_2, j_3) if there is a permutation a, b, c of the dimensions $\{1, 2, 3\}$ so that $i_a < j_1$, and $i_b < j_2$, and $i_c < j_3$. Of course, nesting is recursive — if i nests in j, and j nests in k, then by putting i inside j inside k, only box k is visible. We say that a *nesting arrangement* for a set of n boxes is a sequence of operations in which a box i is put inside another box j in which it nests; and if there were already boxes nested inside i, then these end up inside j as well. (Also notice the following: since the side lengths of i are more than half a meter each, and since the side lengths of j are less than a meter each, box i will take up more than half of each dimension of j, and so after i is put inside j, nothing else can be put inside j.) We say that a box k is *visible* in a nesting arrangement if the sequence of operations does not result in its ever being put inside another box.

So this is the problem faced by the people at WebExodus: Since only the visible boxes are taking up any space, how should a nesting arrangement be chosen so as to minimize the *number* of visible boxes?

Give a polynomial-time algorithm to solve this problem.

Example. Suppose there are three boxes with dimensions $(.6, .6, .6)$, $(.75, .75, .75)$, and $(.9, .7, .7)$. Then the first box can be put into either of the second or third boxes; but in any nesting arrangement, both the second and third boxes will be visible. So the minimum possible number of visible boxes is two, and one solution that achieves this is to nest the first box inside the second.

32. **(a)** Suppose you are given a flow network with integer capacities, together with a maximum flow f that has been computed in the network. Now, the capacity of one of the edges e out of the source is raised by one unit. Show how to compute a maximum flow in the resulting network in time $O(m + n)$, where m is the number of edges and n is the number of nodes.

(b) You're designing an interactive image segmentation tool that works as follows. You start with the image segmentation set-up described in Section 6.8, with n pixels,

a set of neighboring pairs, and parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$. We will make two assumptions about this instance. First, we will suppose that each of the parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$ is a non-negative integer between 0 and d, for some number d. Second, we will suppose that the neighbor relation among the pixels has the property that each pixel is a neighbor of at most four other pixels (so in the resulting graph, there are at most four edges out of each node).

You first perform an *initial segmentation* (A_0, B_0) so as to maximize the quantity $q(A_0, B_0)$. Now, this might result in certain pixels being assigned to the background, when the user knows that they ought to be in the foreground. So when presented with the segmentation, the user has the option of mouse-clicking on a particular pixel v_1, thereby bringing it to the foreground. But the tool should not simply bring this pixel into the foreground; rather, it should compute a segmentation (A_1, B_1) that maximizes the quantity $q(A_1, B_1)$ *subject to the condition that* v_1 *is in the foreground.* (In practice, this is useful for the following kind of operation: in segmenting a photo of a group of people, perhaps someone is holding a bag that has been accidentally labeled as part of the background. By clicking on a single pixel belonging to the bag, and re-computing an optimal segmentation subject to the new condition, the whole bag will often become part of the foreground.)

In fact, the system should allow the user to perform a sequence of such mouse-clicks $v_1, v_2, \ldots, v_t$; and after mouse-click v_i, the system should produce a segmentation (A_i, B_i) that maximizes the quantity $q(A_i, B_i)$ subject to the condition that all of $v_1, v_2, \ldots, v_i$ are in the foreground.

Give an algorithm that performs these operations so that the initial segmentation is computed within a constant factor of the time for a single maximum flow, and then the interaction with the user is handled in $O(dn)$ time per mouse-click.

(Note: Part (a) is a useful primitive for doing this. Also, the symmetric operation of forcing a pixel to belong to the background can be handled by analogous means, but you do not have to work this out here.)

33. We now consider a different variation on the image segmentation problem from Section 6.8. We will develop a solution to an *image labeling* problem, where the goal is to label each pixel with a rough estimate of its distance from the camera (rather than the simple *foreground/background* labeling used in the text). The possible labels for each pixel will be $0, 1, 2, \ldots, M$ for some integer M.

Let $G = (V, E)$ denote the graph whose nodes are pixels, and edges indicate neighboring pairs of pixels. A *labeling* of the pixels is a partition of V into sets $A_0, A_1, \ldots, A_M$, where A_k is the set of pixels that is labeled with distance k for $k = 0, \ldots, M$. We will seek a labeling of minimum *cost*; the cost will come from two types of terms.

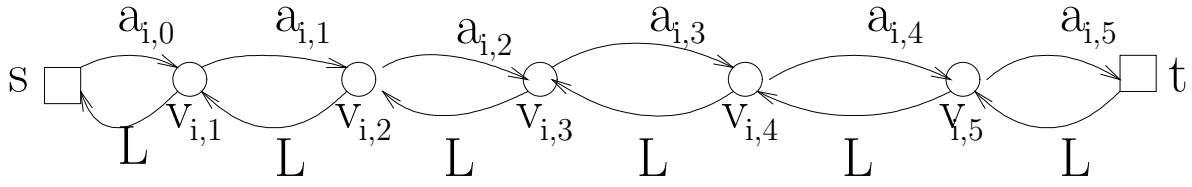
By analogy with the foreground/background segmentation problem, we will have an *assignment cost*: for each pixel i and label k, the cost $a_{i,k}$ is the cost of assigning label k to pixel i. Next, if two neighboring pixels $(i, j) \in E$ are assigned different labels, there will be a *separation* cost. In the book, we use a separation penalty p_{ij}. In our current problem, the separation cost will also depend on how far the two pixels are separated; specifically, it will be proportional to the difference in value between their two labels.

Thus, the overall cost q' of a labeling is defined as follows:

$$q'(A_0, \ldots, A_M) = \sum_{k=0}^{M} \sum_{i \in A_i} a_{i,k} + \sum_{k<\ell} \sum_{\substack{(i,j)\in E \\ i\in A_k, j\in A_\ell}} (\ell - k) p_{ij}.$$

The goal of this problem is to develop a polynomial-time algorithm that finds the optimal labeling given the graph G and the penalty parameters $a_{i,k}$ and p_{ij}. The algorithm will be based on constructing a flow network, and we will start you off on designing the algorithm by providing a portion of the construction.

The flow network will have a source s and a sink t. In addition, for each pixel $i \in V$ we will have nodes $v_{i,k}$ in the flow network for $k = 1, \ldots, M$ as shown in the accompanying figure. ($M = 5$ in the example in the figure.)



For notational convenience, the nodes $v_{i,0}$ and $v_{i,M+1}$ will refer to s and t respectively, for any choice of $i \in V$.

We now add edges $(v_{i,k}, v_{i,k+1})$ with capacity $a_{i,k}$ for $k = 0, \ldots, M$; and edges $(v_{i,k+1}, v_{i,k})$ in the opposite direction with very large capacity L. We will refer to this collection of nodes and edges as the *chain* associated with pixel i.

Notice that if we make this very large capacity L large enough, then there will be no minimum cut (A, B) so that an edge of capacity L leaves the set A. (How large do we have to make it for this to happen?). Hence, for any minimum cut (A, B), and each pixel i, there will be exactly one low-capacity edge in the chain associated with i that leaves the set A. (You should check that if there were two such edges, then a large-capacity edge would also have to leave the set A.)

Finally, here's the question: Use the nodes and edges defined so far to complete the construction of a flow network with the property that a minimum-cost labeling can be efficiently computed from a minimum (s, t)-cut. You should prove that your construction has the desired property, and show to recover the minimum-cost labeling from the cut.

34. The goal of this problem is to suggest variants of the preflow-push algorithm that speed up the practical running time without ruining its worst case complexity. Recall that the algorithm maintains the invariant that $h(v) \leq h(w) + 1$ for all edges (v, w) in the residual graph of the current preflow. We proved that if f is a flow (not just a preflow) with this invariant, then it is a maximum flow. Heights were monotone increasing and the whole running time analysis depended on bounding the number of times nodes can increase their heights. Practical experience shows that the algorithm is almost always much faster than suggested by the worst case, and that the practical bottleneck of the algorithm is relabeling nodes (and not the unsaturating pushes that lead to the worst case in the theoretical analysis). The goal of the problems below is the decrease the number of relabelings by increasing heights faster than one-by-one. Assume you have a graph G with n nodes, m edges, capacities c, source s and sink t.

(a) The preflow-push algorithm, as described in the text, starts by setting the flow equal to the capacity c_e on all edges e leaving the source, setting the flow to 0 on all other edges, setting $h(s) = n$, and setting $h(v) = 0$ for all other nodes $v \in V$. Give an $O(m)$ procedure that for initializing node heights that is better than what we had in class. Your method should set the height of each node v be as high as possible given the initial flow.

(b) In this part we will add a new step, called *gap relabeling* to preflow-push, that will increase the labels of lots of nodes by more than one at a time. Consider a preflow f and heights h satisfying the invariant. A *gap* in the heights is an integer $0 < h < n$ so that no node has height exactly h. Assume h is a gap value, and let A be the set of nodes v with heights $n > h(v) > h$. *Gap relabeling* is to change the height of all nodes in A to n. Prove that the preflow/push algorithm with Gap relabeling is a valid max-flow algorithm. Note that the only new thing that you need to prove is that gap relabeling preserves the invariant above.

(c) In Section 6.6 we proved that $h(v) \leq 2n - 1$ throughout the algorithm. Here we will have a variant that has $h(v) \leq n$ throughout. The idea is that we "freeze" all nodes when they get to height n, i.e., nodes at hight n are no longer considered active, and hence are not used for push and relabel. This way at the end of the algorithm we have a preflow and height function that satisfies the invariant above, and so that all

excess is at height n. Let B be the set of nodes v so that there is a path from v to t in the residual graph of the current preflow. Let $A = V - B$. Prove that at the end of the algorithm (A, B) is a minimum capacity $s - t$ cut.

(d) The algorithm in part (c) computes a minimum $s - t$ cut, but fails to find a maximum flow (as it ends with a preflow that has excesses). Give an algorithm that takes the preflow f at the end of the algorithm of part (c) and converts it into a max flow in at most $O(mn)$ time. Hint: consider nodes with excess and try to send the excess back to s using only edges that the flow came on.

Chapter 7

NP and Computational Intractability

We now arrive at a major transition point in the course. Up till now, we've developed efficient algorithms for a wide range of problems, and have even made some progress on informally categorizing the problems that admit efficient solutions — for example, problems expressible as minimum cuts in a graph, or problems that allow a dynamic programming formulation. But although we've often paused to take note of other problems that we don't see how to solve, we haven't yet made any attempt to actually quantify or characterize the range of problems that *can't be solved efficiently.*

Back when we were first laying out the fundamental definitions, we settled on polynomial time as our working notion of efficiency. One advantage of using a concrete definition like this, as we noted earlier, is that it gives us the opportunity to prove mathematically that certain problems cannot be solved by polynomial-time — and hence "efficient" — algorithms.

When people began investigating computational complexity in earnest, there was some initial progress in proving that certain *extremely hard* problems cannot be solved by efficient algorithms. But for many of the most fundamental discrete computational problems — arising in optimization, artificial intelligence, combinatorics, logic, and elsewhere — the question was too difficult to resolve, and it has remained open since then: we do not know of polynomial-time algorithms for these problems, and we cannot prove that no polynomial-time algorithm exists.

In the face of this formal ambiguity — which becomes increasingly hardened as years pass — people working in the study of complexity have made significant progress. A large class of problems in this "gray area" has been characterized, and it has been proved that they are equivalent in the following sense: a polynomial-time algorithm for any one of them would imply the existence of a polynomial-time algorithm for all of them. These are the *NP-complete problems*, a name that will make more sense as we proceed a little further. There are literally thousands of NP-complete problems, arising in numerous areas, and the class seems to contain a large fraction of the fundamental problems whose complexity we can't resolve. So the formulation of NP-completeness, and the proof that all these problems

are equivalent, is a powerful thing: it says that all these open questions are really a *single* open question, a single type of complexity that we don't yet fully understand.

From a pragmatic point of view, NP-completeness essentially means: "Computationally hard for all practical purposes, though we can't prove it." Discovering that a problem is NP-complete provides a compelling reason to stop searching for an efficient algorithm — you might as well search for an efficient algorithm for any of the famous computational problems already known to be NP-complete, for which many people have tried and failed to find efficient algorithms.

7.1 Computationally Hard Problems

Before starting on the formal definition of NP-completeness, we introduce some paradigmatic examples of hard problems, and then discuss some of the relations among them. All of these problems will turn out to be NP-complete; hence we suspect, but cannot prove, that there is no polynomial-time algorithm to solve any of them.

We will phrase all these problems as *decision problems* — i.e., each will have a "yes/no" answer. Many of them also have a natural optimization version, in which we are seeking to maximize or minimize a particular quantity. In all these cases, the decision and optimization versions are essentially equivalent in terms of computational difficulty — we focus on the decision versions because this will be cleaner in the mathematical developments that are coming.

We break our discussion into six basic genres.

1. Packing Problems. In a packing problem, one is trying to select as large a number of objects as possible, subject to some source of conflicts among the objects. Perhaps the cleanest example is the *Set Packing* problem:

> Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect?

In other words, we wish to "pack" a large number of sets together, with the constraint that no two of them are overlapping.

As an example of where this type of issue might arise, imagine that we have a set U of n non-shareable *resources*, and a set of m software processes. The i^{th} process requires the set $S_i \subseteq U$ of resources in order to run. Then the *Set Packing* problem seeks a large collection of these process that can be run simultaneously, with the property that none of their resource requirements overlap (i.e. represent a conflict).

The *Independent Set* problem, which we introduced at the beginning of the course, is a graph-based packing problem. Recall that in a graph $G = (V, E)$, we say a set of nodes $S \subseteq V$ is *independent* if no two nodes in S are joined by an edge.

> Given a graph G and a number k, does G contain an independent set of size at least k?

These two problems illustrate the contrast between decision and optimization: above, we have phrased each as a yes/no decision question with an extra parameter k, but each has a natural optimization version as well. We could ask: What is the largest collection of sets, no two of which intersect? And: What is the largest independent set in G?

Given a method to solve these optimization versions, we automatically solve the decision versions (for any k) as well. But there is also a slightly less obvious converse to this: if we can solve the decision version of, say, *Independent Set* for every k, then we can also find a maximum independent set. For given a graph G on n nodes, we simply solve the decision version of *Independent Set* for each k; the largest k for which the answer is "yes" is the size of the largest independent set in G. (And using binary search, we need only solve the decision version for $O(\log n)$ different values of k.)

This simple equivalence between decision and optimization will also hold in the problems we discuss below.

2. Covering Problems. Covering problems form a natural contrast to packing problems: we seek to "cover" a collection of objects with as few sets as possible. The most basic example is the *Set Cover* problem:

> Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?

Imagine, for example, that we have m available pieces of software, and a set U of n *capabilities* that we would like our system to have. The i^{th} piece of software includes the set $S_i \subseteq U$ of capabilities. In the *Set Cover* problem, we seek to include a small number of these pieces of software on our system, with the property that our system will then have all n capabilities.

There is also a natural graph-based covering problem. Given a graph $G = (V, E)$, we say that a set of nodes $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one end in S. Note the following thing about this use of terminology: in a vertex cover, it is the vertices that do the covering; the edges are the objects being covered. We formulate the *Vertex Cover* problem as follows:

> Given a graph G and a number k, does G contain a vertex cover of size at most k?

3. Partition Problems. We could view the bipartite matching problem in the following way: we are given two sets X and Y, each of size n, and a set P of pairs drawn from $X \times Y$. The question is: does there exist a set of n pairs in P so that each element in $X \cup Y$ is contained in exactly one of these pairs? The relation to bipartite matching is clear: the set P of pairs is simply the edges of the graph.

This is a fundamental partitioning problem: we seek a collection of *disjoint* sets that covers a ground set of elements. (Equivalently, we seek to *partition* a ground set into "allowable" subsets.) The disjointness constraint is crucial; otherwise, this would simply be a problem in the style of *Set Cover*.

Of course, bipartite matching is a problem we know how to solve in polynomial time. But things get much more complicated when we move from ordered pairs to ordered triples. Consider the following *3-Dimensional Matching* problem:

> Given disjoint sets X, Y, and Z, each of size n; and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

Such a set of triples is called a *perfect three-dimensional matching.*

4. Sequencing Problems. Our first three types of problems have involved searching over subsets of a collection of objects. Another type of computationally hard problem involves searching over the set of all *permutations* of a collection of objects.

Probably the most famous such sequencing problem is the *Traveling Salesman* problem. Consider a salesman who must visit n cities labeled $v_1, v_2, \ldots, v_n$. The salesman starts in city v_1, his home, and wants to find a *tour* — an order in which to visit all the other cities and return home. His goal is to find a tour that causes him to travel as little total distance as possible.

To formalize this, we will take a very general notion of distance — for each ordered pair of cities (v_i, v_j), we will specify a non-negative number $d(v_i, v_j)$ as the distance from v_i to v_j. We will not require the distance to be symmetric (so it may happen that $d(v_i, v_j) \neq d(v_j, v_i)$) nor will we require it to satisfy the triangle inequality (so it may happen that $d(v_i, v_j)$ plus $d(v_j, v_k)$ is actually less than the "direct" distance $d(v_i, v_k)$). The reason for this is to make our formulation as general as possible. Indeed, *Traveling Salesman* arises naturally in many applications where the points are not cities and the traveler is not a salesman. For example, people have used *Traveling Salesman* formulations for problems such as planning the most efficient motion of a robotic arm that drills holes in n points on the surface of a VLSI chip; or for serving I/O requests on a disk; or for sequencing the execution of n software modules to minimize the context-switching time.

Thus, given the set of distances, we ask: order the cities into a *tour* $v[i_1], v[i_2], \ldots, v[i_n]$, with $i_1 = 1$, so as to minimize the total distance $\sum_j d(v[i_j], v[i_{j+1}]) + d(v[i_n], [i_1])$. The

requirement $i_1 = 1$ simply "orients" the tour so that it starts at the home city, and the terms in the sum simply give the distance from each city on the tour to the next one. (The last term in the sum is the distance required for the salesman to return home at the end.)

Here is a decision version of the *Traveling Salesman* problem:

> Given a set of distances on n cities, and a bound D, is there a tour of length at most D?

The *Traveling Salesman* problem has a natural graph-based analogue, which forms one of the fundamental problems in graph theory. Given a directed graph $G = (V, E)$, we say that a cycle C in G is a *Hamiltonian cycle* if it visits each vertex exactly once. In other words, it constitutes a "tour" of all the vertices, with no repetitions. The *Hamiltonian Cycle* problem is then the following:

> Given a directed graph G, does it contain a Hamiltonian cycle?

5. Constraint Satisfaction Problems. We now turn to a somewhat more "abstract" set of problems, which are formulated in Boolean notation. As such, they model a wide range of problems in which we need to set decision variables so as to satisfy a given set of constraints; such formalisms are common, for example, in artificial intelligence.

Suppose we are given a set X of n *Boolean variables* $x_1, \ldots, x_n$; each can take the value 0 or 1 (equivalently, "false" or "true"). By a *term* over X, we mean one of the variables x_i or its negation $\overline{x_i}$. Finally, a *clause* is simply a disjunction of terms

$$t_1 \vee t_2 \vee \cdots \vee t_\ell.$$

(Again, each $t_i \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$.) We say the clause has length ℓ if it contains ℓ terms.

We now formalize what it means for an assignment of values to satisfy a collection of clauses. A *truth assignment* for X is an assignment of the value 0 or 1 to each x_i; in other words, it is a function $\nu : X \to \{0, 1\}$. The assignment ν implicitly gives $\overline{x_i}$ the opposite truth value from x_i. An assignment *satisfies* a clause C if it causes C to evaluate to 1 under the rules of Boolean logic; this is equivalent to requiring that at least one of the terms in C should receive the value 1. An assignment satisfies a collection of clauses $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1; in other words, if it causes the conjunction

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

to evaluate to 1. In this case, we will say that ν is a *satisfying assignment* with respect to $C_1, \ldots, C_k$; and that the set of clauses $C_1, \ldots, C_k$ is *satisfiable*.

Here is a simple example. Suppose we have the three clauses

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3}).$$

Then the truth assignment ν which sets all variables to 1 is not a satisfying assignment, because it does not satisfy the second of these clauses; but the truth assignment ν' which sets all variables to 0 is a satisfying assignment.

We can now state the *Satisfiability* problem, also referred to as *SAT*:

> Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Satisfiability is a really fundamental combinatorial search problem; it contains the basic ingredients of a hard computational problem in very "bare-bones" fashion. We have to make n independent decisions (the assignments for each x_i) so as to satisfy a set of constraints. There several ways to satisfy each constraint in isolation, but we have to arrange our decisions so that all constraints are satisfied simultaneously.

There is a special case of *SAT* that will turn out to be equivalently difficult, and is somewhat easier to think about; this is the case in which all clauses have length 3. We call this problem *3-Satisfiability*, or *3-SAT*:

> Given a set of clauses $C_1, \ldots, C_k$, *each of length 3*, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

6. Numerical Problems. Finally, we consider some computational problems that involve arithmetic operations on numbers. Our basic problem in this genre will be *Subset Sum*, which we saw before when we covered dynamic programming. We can formulate a decision version of this problem as follows:

> Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

We have already seen an algorithm to solve this problem; why are we now including it on our list of computationally hard problems? This goes back to an issue that we raised the first time we considered *Subset Sum*. The algorithm we developed has running time $O(nW)$, which is reasonable when W is small, but becomes hopelessly impractical as W (and the numbers w_i) grow large. Consider, for example, an instance with 100 numbers, each of which is 100 bits long. Then the input is only $100 \times 100 = 10000$ digits, but W is now roughly 2^{100}.

To phrase this more generally: since integers will typically be given in bit representation, or base-10 representation, the quantity W is really *exponential* in the size of the input; our algorithm was not a polynomial-time algorithm. (We referred to it as *pseudo polynomial*, to indicate that it ran in time polynomial in the magnitude of the input numbers, but not polynomial in the size of their representation.)

This is an issue that comes up in many settings; for example, we encountered it in the context of network flow algorithms, where the capacities had integer values. Other settings

may be familiar to you as well. For example, the security of a cryptosystem such as RSA is motivated by the sense that factoring a 400-bit number is difficult. But if we considered a running time of 2^{400} steps feasible, factoring such a number would not be difficult at all.

It is worth pausing here for a moment and asking: Is this notion of polynomial time for numerical operations too severe a restriction? For example, given two natural numbers w_1 and w_2 represented in base-d notation for some $d > 1$ how long does it take to add, subtract, or multiply them? Fortunately, the standard ways that kids in elementary school learn to perform these operations have (low-degree) polynomial running times. Addition and subtraction (with carries) takes $O(\log w_1 + \log w_2)$ time, while the standard multiplication algorithm runs in $O(\log w_1 \cdot \log w_2)$ time. (There are asymptotically faster multiplication algorithms that elementary schoolers are unlikely to invent on their own; but we will not be focusing on these here.)

So a basic question is: can *Subset Sum* be solved by a (genuinely) polynomial-time algorithm? We will see that the framework developed here will shed light on this question, as well as the analogous questions for all the other problems introduced above.

7.2 Polynomial-time Reductions

Our study of computationally hard problems will involve two basic ingredients: a mathematical characterization of their structure, which we will develop in the next section, and a method for quantifying their relative complexities, which we do now.

For various computational problems, we want to be able to ask questions of the form, "Suppose we could solve problem X in polynomial time. What else could we then solve in polynomial time?" Intuitively, this would be very useful for establishing relationships between problems X and Y via statements like,

> ($*$) If we could solve problem X in polynomial time, then we could also solve problem Y in polynomial time.

But it is not easy to work with an assumption phrased this way. To make the notion more manageable, we will essentially add the assumption that X can be solved in polynomial time directly to our model of computation. Suppose we had a *black box* that could solve instances of a problem X; if we write down the input for an instance of X, then in a single step, the black box will return the correct yes/no answer. We can now ask the following question:

> ($**$) Can arbitrary instances of problem Y be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?

If the answer to this question is yes, then we write $Y \leq_P X$; we read this as "Y is polynomial-time reducible to X," or "X is at least as hard as Y (with respect to polynomial time)."

Note that in this definition, we still pay for the time it takes to write down the input to the black box solving X.

This formulation of reducibility is actually very natural. When we ask about reductions to a problem X, it is as though we've supplemented our computational model with a piece of specialized hardware that solves instances of X in a single step. We can now explore the question: how much extra power does this piece of hardware give us?

An importance consequence of our definition of $\leq_P$ is the following. Suppose $Y \leq_P X$ and there actually *exists* a polynomial-time algorithm to solve X. Then our specialized black box for X is actually not so valuable; we can replace it with a polynomial-time algorithm for X. Consider what happens to our algorithm for problem Y that involved a polynomial number of steps plus a polynomial number of calls to the black box. It now becomes an algorithm that involves a polynomial number of steps, plus a polynomial number of calls to a subroutine that runs in polynomial time; in other words, it has become a polynomial-time algorithm. We have therefore proved the following fact.

(7.1) *Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.*

We've made use of precisely this fact — implicitly — at earlier points in the course. Recall that we solved the bipartite matching problem using a polynomial amount of pre-processing plus the solution of a single maximum flow problem. Since the maximum flow problem can be solved in polynomial time, we concluded that bipartite matching could as well. Similarly, we solved the foreground/background image segmentation problem using a polynomial amount of pre-processing plus the solution of a single minimum cut problem, with the same consequences. Both of these were direct applications of (7.1) . Indeed, (7.1) summarizes a great way to design polynomial-time algorithms for new problems: by reduction to a problem we already know how to solve in polynomial time.

In this part of the course, however, we will be using (7.1) to establish the computational *intractability* of various problems. For this purpose, we will really be using its contrapositive form, which is sufficiently valuable that we'll state it as a separate fact.

(7.2) *Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.*

(7.2) is transparently equivalent to (7.1) , but it emphasizes our overall plan: if we have a problem Y that is known to be hard, and we show that $Y \leq_P X$, then the hardness has "spread" to X; X must be hard or else it could be used to solve Y.

In reality, given that we don't actually know whether the problems we're studying can be solved in polynomial time or not, we'll be using $\leq_P$ to establish relative levels of difficulty among problems.

With this in mind, we now establish some reducibilities among the problems introduced in the previous section. We divide these into several categories.

Reductions from a Special Case to the General Case. Probably the most basic kind of reduction is the following: we have problems X and Y, and we establish that Y is in effect a "special case" of X. Consider, for example, the relationship between *Vertex Cover* and *Set Cover*. In the latter case, we are trying to cover an arbitrary set using arbitrary subsets; in the former case, we are specifically trying to cover edges of a graph using sets of edges incident to vertices. Indeed, we can prove

(7.3) Vertex Cover $\leq_P$ Set Cover.

Proof. Suppose we have access to a black box that can solve *Set Cover*, and consider an arbitrary instance of *Vertex Cover*, specified by a graph $G = (V, E)$ and a number k. How can we use the black box to help us?

Our goal is to cover the edges in E, so we formulate an instance of *Set Cover* in which the ground set U is equal to E. Each time we pick a vertex in the *Vertex Cover* problem, we cover all the edges incident to it; thus, for each vertex $i \in V$, we add a set $S_i \subseteq U$ to our *Set Cover* instance, consisting of all the edges in G incident to i.

We now claim that U can be covered with at most k of the sets $S_1, \ldots, S_n$ if and only if G has a vertex cover of size at most k. This can be proved very easily. For if $S_{i_1}, \ldots, S_{i_\ell}$ are $\ell \leq k$ sets that cover U, then every edge in G is incident to one of the vertices $i_1, \ldots, i_\ell$, and so the set $\{i_1, \ldots, i_\ell\}$ is a vertex cover in G of size $\ell \leq k$. Conversely, if $\{i_1, \ldots, i_\ell\}$ is a vertex cover in G of size $\ell \leq k$, then the sets $S_{i_1}, \ldots, S_{i_\ell}$ cover U.

Thus, given our instance of *Vertex Cover*, we formulate the instance of *Set Cover* described above, and pass it to our black box. We answer "yes" if and only if the black box answers "yes." ■

Here is something worth noticing about this proof. Although the definition of $\leq_P$ allows us to issue many calls to our black box for *Set Cover*, we issued only one; indeed, our algorithm for *Vertex Cover* consisted simply of encoding the problem as a single instance of *Set Cover*, and then using the answer to this instance as our overall answer. This will be true of essentially all the reductions that we consider; they will consist of establishing $Y \leq_P X$ by transforming our instance of Y to a single instance of X, invoking our black box for X on this instance, and reporting the box's answer as our answer for the instance of Y.

We can establish an analogous fact for packing problems. The proof is almost identical to that of (7.3); we will leave the details as an exercise.

(7.4) Independent Set $\leq_P$ Set Packing.

Reductions from the General Case to a Special Case. A more surprising kind of reduction goes in the opposite direction: it establishes that a general problem is reducible to one of its special cases. This is extremely useful; it shows that the true "hardness" of the general problem is already present in the special case. It is also generally more difficult to prove than the previous type of reduction, since we have to work within the confines of a black box that can only solve the special case.

A basic example of this phenomenon is the relationship between *Satisfiability* and *3-Satisfiability*. It is clear that *3-SAT* $\leq_P$ *SAT*, since every instance of *3-SAT* is also an instance of *SAT*. But the opposite reduction is true as well, for much less obvious reasons.

(7.5) SAT $\leq_P$ 3-SAT.

Proof. Consider an instance of *SAT* with variables $X = \{x_1, \ldots, x_n\}$ and clauses $C_1, \ldots, C_k$. We will construct an instance of *3-SAT* that is satisfiable if and only if the given instance of *SAT* is satisfiable. In this way, by presenting the constructed *3-SAT* instance to our black box, we can determine whether the original *SAT* instance was satisfiable.

For each clause C_i in the *SAT* instance, we will construct a set of clauses $D_{i1}, D_{i2}, \ldots$ containing the variables of C_i plus some variables that will not appear anywhere else. Any assignment that satisfies the set of all D_{ij} will also satisfy C_i; conversely, for any assignment that satisfies C_i, we will be able to choose an assignment for the extra variables as well so that all D_{ij} are satisfied. In this way, we will have completely translated the clause C_i into an ensemble of length-3 clauses.

Here is the specific construction. First, if C_i has length 2 — say $C_i = t_{i1} \vee t_{i2}$ — we create the single length-3 clause $t_{i1} \vee t_{i2} \vee t_{i2}$, which is clearly equivalent. Similarly, if $C_i = t_{i1}$ has length 1, we create the length-3 clause $t_{i1} \vee t_{i1} \vee t_{i1}$.

The interesting part is the translation of long clauses. Suppose that

$$C_i = t_{i1} \vee t_{i2} \vee \cdots \vee t_{i\ell}$$

for terms $t_{i1}, \ldots, t_{i\ell}$, with $\ell > 3$. We create the following new clauses, each of length 3:

$$\begin{aligned}
D_{i1} &= t_{i1} \vee t_{i2} \vee y_{i1} \\
D_{i2} &= \overline{y_{i1}} \vee t_{i3} \vee y_{i2} \\
\cdots &= \cdots \\
D_{i,j-1} &= \overline{y_{i,j-2}} \vee t_{ij} \vee y_{i,j-1} \\
D_{ij} &= \overline{y_{i,j-1}} \vee t_{i,j+1} \vee y_{ij} \\
\cdots &= \cdots \\
D_{i,\ell-2} &= \overline{y_{i,\ell-3}} \vee t_{i,\ell-1} \vee t_{i\ell}
\end{aligned}$$

The variables $y_{i1}, \ldots, y_{i,\ell-3}$ will appear only in these clauses. We use Y to denote the set of all new variables $\{y_{ij}\}$.

First, suppose there is a truth assignment ν that satisfies all C_i. Then we can extend ν to a truth assignment ν' on all variables in $X \cup Y$ as follows. First, ν' will agree with ν on all variables in X. Now, for clause C_i, at least one of the terms evaluates to 1; suppose that t_{ij} evaluates to 1. Then we set $\nu'(y_{im}) = 1$ for each $m \leq j-2$, and we set $\nu'(y_{im}) = 0$ for each $m \geq j-1$. By looking at the clauses $D_{i1}, \ldots, D_{i,\ell-2}$, we see that they will all be satisfied by this truth assignment. Thus, if the set of clauses $\{C_i\}$ is satisfiable, then the set of clauses $\{D_{ij}\}$ is satisfiable.

Conversely, suppose that there is an assignment ν' that satisfies all D_{ij}. Consider clause C_i. We claim that at least one of the terms t_{ij} in C_i must evaluate to 1 under the assignment ν'. For if not, then since D_{i1} is satisfied, we know that $\nu'(y_{i1}) = 1$; since D_{i2} is satisfied, it follows that $\nu'(y_{i2}) = 1$; and continuing in this way, we conclude that $\nu'(y_{ij}) = 1$ for all j. But then the last clause $D_{i,\ell-2}$ would not be satisfied, which contradicts our assumption. Thus $\nu'(t_{ij}) = 1$ for some j, and hence the clause C_i is satisfied by ν'. We have concluded that if the set of clauses $\{D_{ij}\}$ is satisfiable, then the set of clauses $\{C_i\}$ is satisfiable.

Since the original *SAT* instance is satisfiable if and only if the new *3-SAT* instance is satisfiable, the proof is complete. ■

Reductions by Simple Equivalence. In many cases, we are faced with two problems that look similar, but neither is a special case of the other. Sometimes, it is possible to prove that they are in a sense "equivalent," and this can lead to reductions from one problem to the other.

A natural example of this is the relationship between *Independent Set* and *Vertex Cover*. Both independent sets and vertex covers are subsets of the vertex set of a graph, satisfying specific properties; is there a connection between the two definitions? In fact, there is a very basic one:

(7.6) *Let $G = (V, E)$ be a graph. Then S is an independent set if and only if its complement $V - S$ is a vertex cover.*

Proof. First, suppose that S is an independent set. Consider an arbitrary edge $e = (u, v)$. Since S is independent, it cannot be the case that both u and v are in S; so one of them must be in $V - S$. It follows that every edge has at least one end in $V - S$, and so $V - S$ is a vertex cover.

Conversely, suppose that $V - S$ is a vertex cover. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in $V - S$, contradicting our assumption that $V - S$ is a vertex cover. It follows that no two nodes in S are joined by an edge, and so S is an independent set. ■

Reductions in each direction between the two problems follow immediately from this definition.

(7.7) Independent Set $\leq_P$ Vertex Cover.

Proof. If we have a black box to solve *Vertex Cover*, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most $n - k$. ■

(7.8) Vertex Cover $\leq_P$ Independent Set.

Proof. If we have a black box to solve *Independent Set*, then we can decide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least $n - k$. ■

Reductions by Encoding with "Gadgets." We have just seen that *Vertex Cover* can be reduced to *Independent Set*. This is not so surprising, given that both are problems of a similar flavor involving graphs.

We now show something considerably more surprising: that *3-SAT* $\leq_P$ *Independent Set*. The difficulty in proving a thing like this is clear; *3-SAT* is about setting Boolean variables in presence of constraints, while *Independent Set* is about selecting vertices in a graph. To solve an instance of *3-SAT* using a black box for *Independent Set*, we need a way to encode all these Boolean constraints in the nodes and vertices of a graph, so that satisfiability corresponds to the existence of a large independent set.

Doing this illustrates a general principle for designing complex reductions $Y \leq_P X$: building "gadgets" out of components in problem X to represent what is going on in problem Y.

(7.9) 3-SAT $\leq_P$ Independent Set.

Proof. We have a black box for *Independent Set* and want to solve an instance of *3-SAT* consisting of variables $X = \{x_1, \ldots, x_n\}$ and clauses $C_1, \ldots, C_k$.

The key is to picture the *3-SAT* instance as follows. Say that two terms t and t' *conflict* if one is equal to a variable x_i and the other is equal to its negation $\overline{x_i}$. In order to satisfy the instance, we must pick a term in each clause that will evaluate to 1, in such a way that we do not pick two terms that conflict.

Here is how we encode this notion using independent sets in a graph. First, construct a graph $G = (V, E)$ consisting of $3k$ nodes grouped into k triangles. That is, for $i = 1, 2, \ldots, k$, we construct three vertices v_{i1}, v_{i2}, v_{i3} joined to one another by edges. We give each of these vertices a *label*; v_{ij} is labeled with the j^{th} term from the clause C_i of the *3-SAT* instance.

Before proceeding, consider what the independent sets of size k look like in this graph: since two vertices cannot be selected from the same triangle, they consist of all ways of choosing one vertex from each of the triangles. This is implementing our goal of choosing a

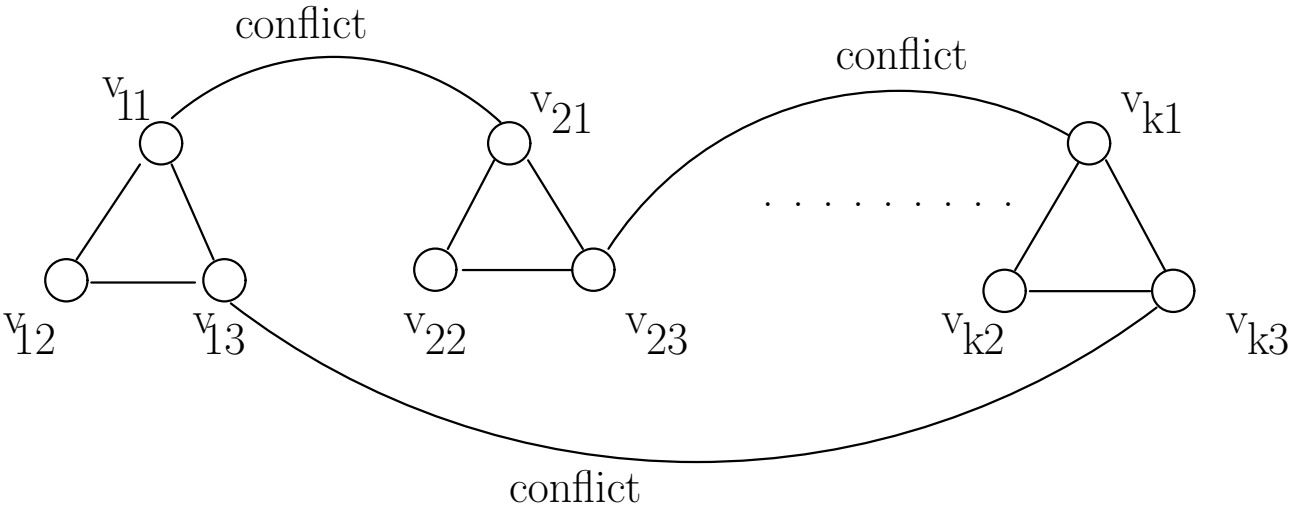


Figure 7.1: The reduction from *3-SAT* to *Independent Set*.

term in each clause that will evaluate to 1; but we have so far not prevented ourselves from choosing two terms that conflict.

We encode this notion by adding some more edges to the graph: For each pair of vertices whose labels correspond to terms that conflict, we add an edge between them. Have we now destroyed all the independent sets of size k, or does one still exist? It's not clear; it depends on whether we can still select one node from each triangle so that no conflicting pairs of vertices are chosen. But this is precisely what the *3-SAT* instance required ...

Let's claim, precisely, that the original *3-SAT* instance is satisfiable if and only if the graph G we have constructed has an independent set of size at least k. First, if the *3-SAT* instance is satisfiable, then each triangle in our graph contains at least one node whose label evaluates to 1. Let S be a set consisting of one such node from each triangle. We claim S is independent; for if there were an edge between two nodes $u, v \in S$, then the labels of u and v would have to conflict; but this is not possible, since they both evaluate to 1.

Conversely, suppose our graph G has an independent set S of size at least k. Then, first of all, the size of S is exactly k, and it must consist of one node from each triangle. Now, we claim that there is a truth assignment ν for the variables in the *3-SAT* instance with the property that the labels of all nodes in S evaluate to 1. Here is how we could construct such an assignment ν. For each variable x_i, if neither x_i nor $\overline{x_i}$ appears as a label of a node in S, then we arbitrarily set $\nu(x_i) = 1$. Otherwise, exactly one of x_i or $\overline{x_i}$ appears as a label of a node in S — for if one node in S were labeled x_i and another were labeled $\overline{x_i}$, then there would be an edge between these two nodes, contradicting our assumption that S is an independent set. Thus, if x_i appears as a label of a node in S, we set $\nu(x_i) = 1$, and otherwise we set $\nu(x_i) = 0$. By constructing ν in this way, all labels of nodes in S will

evaluate to 1.

Since G has an independent set of size at least k if and only if the original *3-SAT* instance is satisfiable, the reduction is complete. ■

Some Final Observations

We have now worked through a number of reductions between problems. We can infer a number of additional relationships using the following fact: $\leq_P$ is a *transitive* relation.

(7.10) *If $Z \leq_P Y$, and $Y \leq_P X$, then $Z \leq_P X$.*

Proof. Given a black box for X, we show how to solve an instance of Z; essentially, we just compose the two algorithms implied by $Z \leq_P Y$ and $Y \leq_P X$. We run the algorithm for Z using a black box for Y; but each time the black box for Y is called, we *simulate* it in a polynomial number of steps using the algorithm that solves instances of Y using a black box for X. ■

Transitivity can be quite useful. For example, since we have proved

$$\textit{SAT} \leq_P \textit{3-SAT} \leq_P \textit{Independent Set} \leq_P \textit{Vertex Cover} \leq_P \textit{Set Cover},$$

we can conclude that $\textit{SAT} \leq_P \textit{Set Cover}$.

7.3 Efficient Certification and the Definition of NP

Reducibility among problems was the first main ingredient in our study of computational intractability. The second ingredient is a characterization of the class of problems that we are dealing with. Combining these two ingredients, together with a powerful theorem of Cook from 1971, will yield some surprising consequences.

Recall at the beginning of the course, when we first encountered the *Independent Set* problem, we asked: Can we say anything *good* about it, from a computational point of view? And, indeed, there was something: if a graph does contain an independent set of size at least k, then we could give you an easy proof of this fact by exhibiting such an independent set. Similarly, if a *3-SAT* instance is satisfiable, we can prove this to you by revealing the satisfying assignment. It may an enormously difficult task to actually *find* such an assignment; but if we've done the hard work of finding one, it's easy for you to plug it into the clauses and check that they are all satisfied.

The issue here is the contrast between *finding* a solution and *checking* a proposed solution. For *Independent Set* or *3-SAT*, we do not know a polynomial-time algorithm to find solutions; but *checking* a proposed solution to these problems can be easily done in polynomial time. To see that this is not entirely trivial issue, consider the problem we'd face if we had to

prove that a *3-SAT* instance was *not* satisfiable. What "evidence" could we show that would convince you, in polynomial time, that the instance was unsatisfiable?

This will be the crux of our characterization; we now proceed to formalize it. The input to a computational problem will be encoded as a finite binary string s. We denote the length of a string s by $|s|$. We will identify a decision problem X with the *set* of strings on which the answer is "yes." An algorithm A for a decision problem receives an input string s and returns the value "yes" or "no" — we will denote this return value by $A(s)$. We say that A *solves* the problem X if for all strings s, $A(s) = \texttt{yes}$ if and only if $s \in X$.

As always, we say that A has a *polynomial running time* if there is a polynomial function $p(\cdot)$ so that for every input string s, A terminates on s in at most $O(p(|s|))$ steps. Thus far in the course, we have been concerned with problems solvable in polynomial time. In the above notation, we can express this as the set $\mathcal{P}$ of all problems X for which there exists an algorithm A with a polynomial running time that solves X.

Efficient Certification. Now, how should formalize the idea that a solution to a problem can be *checked* efficiently, independently of whether it can be solved efficiently? A "checking algorithm" for a problem X has a different structure from an algorithm that actually seeks to solve the problem; in order to "check" a solution we need the input string s, as well as a separate "certificate" string t that contains the evidence that s is a "yes" instance of X.

Thus, we say that B is an *efficient certifier* for a problem X if the following properties hold:

- B is a polynomial-time algorithm that takes two input arguments s and t.
- There is a polynomial function p so that for every string s, $s \in X$ if and only if there exists a string t such that $|t| \leq p(|s|)$ and $B(s,t) = \texttt{yes}$.

It takes some time to really think through what this definition is saying. One should view an efficient certifier as approaching a problem X from a "managerial" point of view. It will not actually try to decide whether an input s belongs to X on its own. Rather, it is willing to efficiently evaluate proposed "proofs" t that s belongs to X — provided they are not too long — and it is a correct algorithm in the weak sense that s belongs to X if and only there exists a proof that will convince it.

An efficient certifier B can be used as the core component of a "brute-force" algorithm for a problem X: on an input s, try all strings t of length $\leq p(|s|)$, and see if $B(s,t) = \texttt{yes}$ for any of these string. But the existence of B does not provide us with any clear way to design an efficient algorithm that actually solves X; after all, it is still up to us to *find* a string t that will cause $B(s,t)$ to say "yes" — and there are exponentially many possibilities for t.

The class NP. We define $\mathcal{NP}$ to be the set of all problems for which there exists an efficient certifier.[1] Here is one thing we can observe immediately.

(7.11) $\mathcal{P} \subseteq \mathcal{NP}$.

Proof. Consider a problem $X \in \mathcal{P}$; this means that there is a polynomial-time algorithm A that solves X. To show that $X \in \mathcal{NP}$, we must show that there is an efficient certifier B for X.

This is very easy; we design B as follows. When presented with the input pair (s, t), B simply returns the value $A(s)$. (Think of B as a very "hands-on" manager who ignores the proposed proof t and simply solves the problem on its own.) Why is B an efficient certifier for X? Clearly it has polynomial running time, since A does. If a string $s \in X$, then for *every* t of length at most $p(|s|)$, $B(s, t) =$ `yes`. On the other hand, if $s \notin X$, then for every t of length at most $p(|s|)$, $B(s, t) =$ `no`. ■

We can easily check that the problems introduced in the first section belong to $\mathcal{NP}$: it is a matter of determining how an efficient certifier for each of them will make use of a "certificate" string t. For example:

- For the *Satisfiability* problem, the certificate t is an assignment of truth values to the variables; the certifier B evaluates the given set of clauses with respect to this assignment.
- For the *Independent Set* problem, the certificate t is the identity of a set of at least k vertices; the certifier B checks that, for these vertices, no edge joins any pair of them.
- For the *Traveling Salesman* problem, the certificate t is a permutation of the cities; the certifier checks that the length of the corresponding tour is at most D.

Yet, we cannot prove that any of these problems require more than polynomial time to solve. Indeed, we cannot prove that there is any problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$. So in place of a concrete theorem, we can only ask a question:

(7.12) *Is there a problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$? Does $\mathcal{P} = \mathcal{NP}$?*

The problem of whether $\mathcal{P} = \mathcal{NP}$ is the most fundamental question in the area of algorithms, and the most famous precisely formulated problem in computer science. The general consensus is that $\mathcal{P} \neq \mathcal{NP}$ — and this is taken as a working hypothesis throughout the field — but there is not a lot of hard technical evidence for it. It is more based on

[1] The act of searching for a string t that will cause an efficient certifier to accept the input s is often viewed as a *non-deterministic search* over the space of possible proofs t; for this reason, $\mathcal{NP}$ was named as an acronym for "non-deterministic polynomial time."

the sense that $\mathcal{P} = \mathcal{NP}$ would be too amazing to be true. How could there be a general transformation from the task of *checking* a solution, to the much harder task of actually *finding* a solution? How could there be a general means for designing efficient algorithms — powerful enough to handle all these hard problems — that we have somehow failed to discover? More generally, a huge amount of effort has gone into failed attempts at designing polynomial-time algorithms for hard problems in $\mathcal{NP}$; perhaps the most natural explanation for this consistent failure is that many or all of these problems simply cannot be solved in polynomial time.

7.4 NP-complete Problems

In the absence of progress on the $\mathcal{P} = \mathcal{NP}$ question, people have turned to a related but more approachable question: What are the hardest problems in $\mathcal{NP}$? Polynomial-time reducibility gives us a way of addressing this question, and gaining insight into the structure of $\mathcal{NP}$.

Arguably the most natural way to define a "hardest" problem X is via the following two properties: (i) $X \in \mathcal{NP}$; and (ii) for all $Y \in \mathcal{NP}$, $Y \leq_P X$. In other words, we require that every problem in $\mathcal{NP}$ can be reduced to X. We will call such an X an *NP-complete* problem.

The following fact helps to further reinforce our use of the term "hardest."

(7.13) *Suppose X is an NP-complete problem. Then X is solvable in polynomial time if and only if $\mathcal{P} = \mathcal{NP}$.*

Proof. Clearly, if $\mathcal{P} = \mathcal{NP}$, then X can be solved in polynomial time since it belongs to $\mathcal{NP}$. Conversely, suppose that X can be solved in polynomial time. If Y is any other problem in $\mathcal{NP}$, then $Y \leq_P X$, and so by (7.1), it follows that Y can be solved in polynomial time. Hence $\mathcal{NP} \subseteq \mathcal{P}$; combined with (7.11), we have the desired conclusion. ■

A crucial consequence of (7.13) is the following: if there is an *any* problem in $\mathcal{NP}$ that cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.

Thus, our definition has some very nice properties. But before we get too carried away in thinking about this notion, we should stop to notice something: it is not at all obvious that NP-complete problems should even *exist*. Why couldn't there exist two incomparable problems X' and X'', so that there is no $X \in \mathcal{NP}$ with the property that $X' \leq_P X$ and $X'' \leq_P X$? To prove a problem is NP-complete, one must show how it could encode *any* problem in $\mathcal{NP}$. This is much trickier matter than what we encountered in the previous section, where we sought to encode specific, individual problems in terms of others.

In 1971, Cook and Levin independently showed how to do this for very natural problems in $\mathcal{NP}$. Cook's theorem can be stated as follows.

(7.14) Satisfiability *is NP-complete.*

This was a very fundamental result, and it opened the door to a much fuller understanding of this class of hardest problems. We will not go into the proof of (7.14) at the moment, though it is actually not so hard to follow the basic idea that underlies it. Essentially, a polynomial-time algorithm implemented in any standard model of computation maintains a polynomial amount of *state* in the form of the settings of its variables — consider a processor manipulating bits in registers and memory locations. This state completely describes the behavior of the algorithm at any point in time. At the end of a polynomial number of steps, the algorithm returns either "yes" or "no." To prove (7.14), one first shows that the state of the computation of an algorithm can modeled using Boolean variables and clauses. The proof then considers any problem in $\mathcal{NP}$, and its efficient certifier $B(\cdot, \cdot)$. It establishes that the question, "Given s, is there a t so that the algorithm $B(s, t)$ will return 'yes'?" can be encoded as a polynomial-size collection of clauses; and that these clauses have a satisfying assignment if and only if there is such a t.

The key feature of (7.14) for our purposes is this: once we have our hands on a first NP-complete problem, we can discover many more via the following simple observation.

(7.15) *If Y is an NP-complete problem, and X is a problem in $\mathcal{NP}$ with the property that $Y \leq_P X$, then X is NP-complete.*

Proof. Since $X \in \mathcal{NP}$, we need only verify property (ii) of the definition. So let Z be any problem in $\mathcal{NP}$. We have $Z \leq_P Y$, by the NP-completeness of Y, and $Y \leq_P X$ by assumption. By (7.10), it follows that $Z \leq_P X$. ■

In view of the sequence of reductions summarized at the end of the previous section, we can use (7.15) to conclude:

(7.16) *All of the following problems are NP-complete:* 3-SAT*,* Independent Set*,* Set Packing*,* Vertex Cover*, and* Set Cover*.*

Proof. Each of these problems has the property that it is in $\mathcal{NP}$, and that *SAT* can be reduced to it. ■

For most of the remainder of this chapter, we will take off in search of further NP-complete problems. In particular, we will prove that the rest of the problems introduced in the first section are also NP-complete. As we suggested initially, there is a very practical motivation in doing this: since it is widely believed that $\mathcal{P} \neq \mathcal{NP}$, the discovery that a problem is NP-complete can be taken as a strong indication that it cannot be solved in polynomial time.

Given a new problem X, here is the basic strategy for proving it is NP-complete.

1. Prove that $X \in \mathcal{NP}$.
2. Choose a problem Y that is known to be NP-complete.
3. Prove that $Y \leq_P X$.

We noticed earlier that most of our reductions $Y \leq_P X$ consist of transforming a given instance of Y into a *single* instance of X with the same answer. This is a particular way of using a black box to solve X; in particular, it requires only a single invocation of the black box. When we use this style of reduction, we can refine the strategy above to the following outline of an NP-completeness proof:

1. Prove that $X \in \mathcal{NP}$.
2. Choose a problem Y that is known to be NP-complete.
3. Consider an arbitrary instance s_Y of problem Y, and show how to construct, in polynomial time, an instance s_X of problem X that satisfies the following properties:
   (a) If s_Y is a "yes" instance of Y, then s_X is a "yes" instance of X.
   (b) If s_X is a "yes" instance of X, then s_Y is a "yes" instance of Y.

   In other words, this establishes that s_Y and s_X have the same answer.

7.5 Sequencing and Partitioning Problems

We now prove the NP-completeness of the sequencing and partitioning problems that we discussed in the opening section. There will be two main results here, proving the NP-completeness of *Hamiltonian Cycle* and of *3-Dimensional Matching*. Fundamentally, they will follow exactly the style outlined above; we identify known NP-complete problems, and reduce them to these two problems. In the details, however, the reductions will be distinctly more complicated than what we have encountered so far. In both cases, we will reduce from *3-SAT*, constructing "gadgets" that encode the variables and clauses.

Sequencing Problems.

(7.17) Hamiltonian Cycle *is NP-complete.*

Proof. We first show that *Hamiltonian Cycle* is in $\mathcal{NP}$. Given a directed graph $G = (V, E)$, a certificate that there is a solution would be the ordered list of the vertices on a Hamiltonian cycle. We could then check, in polynomial time, that this list of vertices does contain each vertex exactly once, and that each consecutive pair in the ordering is joined by an edge; this would establish the ordering defines a Hamiltonian cycle.

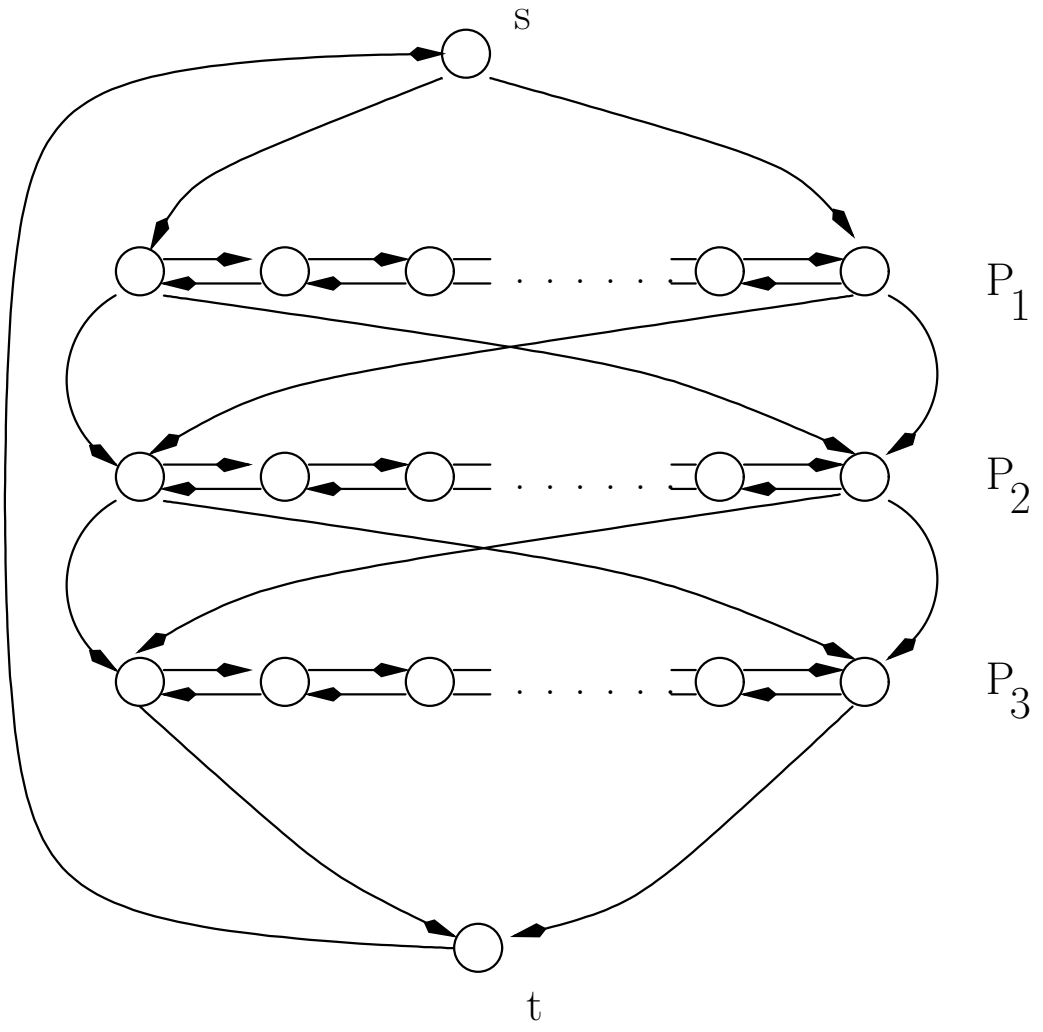


Figure 7.2: The reduction from *3-SAT* to *Hamiltonian Cycle*: Part 1.

We now show that *3-SAT* $\leq_P$ *Hamiltonian Cycle*. Why are we reducing from *3-SAT*? Essentially, faced with *Hamiltonian Cycle*, we really have no idea *what* to reduce from; it's sufficiently different from all the problems we've seen so far that there's no real basis for choosing. In such a situation, one strategy is to go all the way back to *3-SAT*, since its combinatorial structure is very basic. Of course, this strategy guarantees at least a certain level of complexity in the reduction, since we need to encode variables and clauses in the language of graphs.

So consider an arbitrary instance of *3-SAT*, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$. We must show to solve it, given the ability to detect Hamiltonian cycles in directed graphs. As always, it helps to focus on the essential ingredients of *3-SAT*: We can set the values of the variables however we want, and we are given three chances to satisfy each clause.

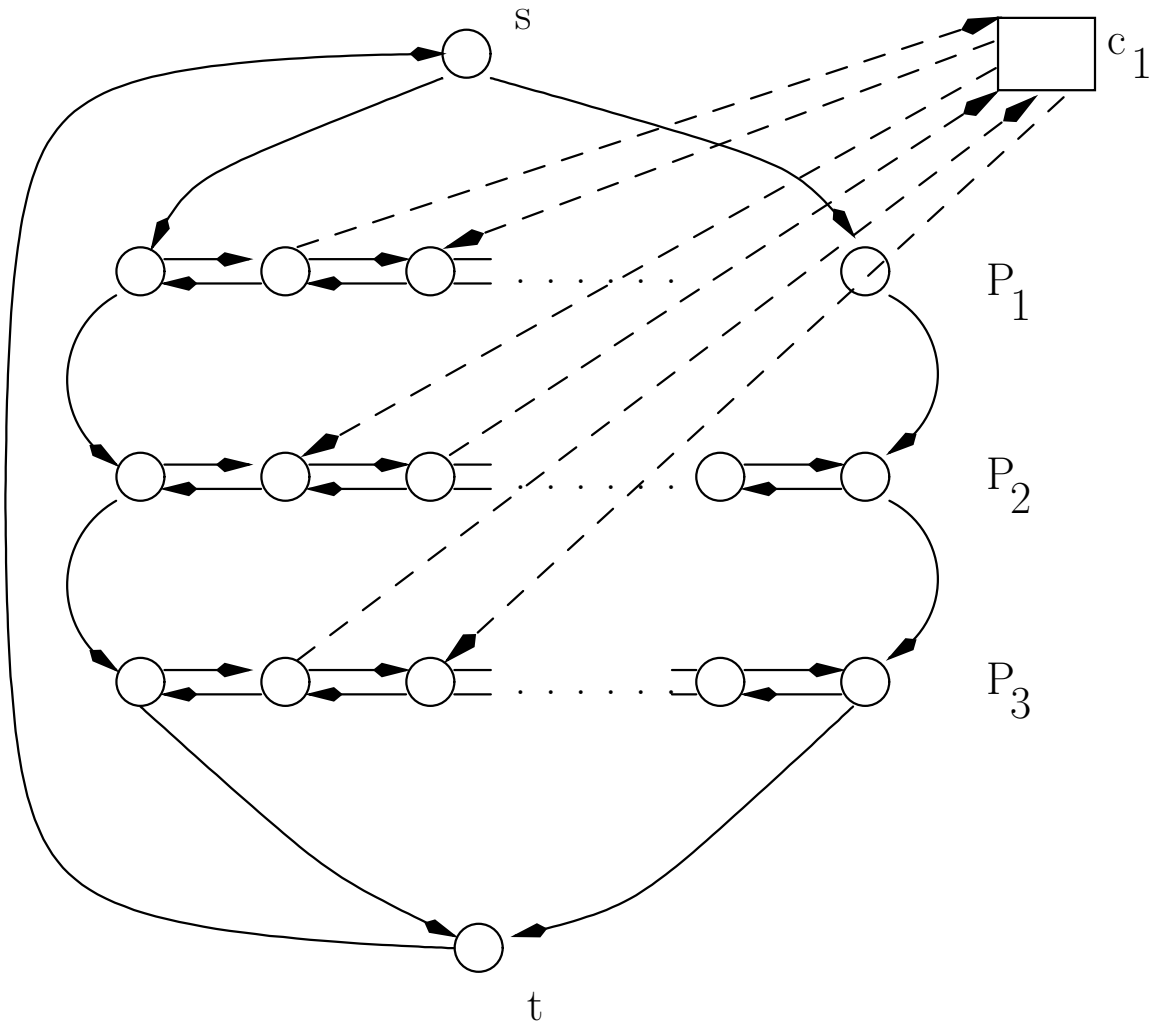


Figure 7.3: The reduction from *3-SAT* to *Hamiltonian Cycle*: Part 2.

We begin by describing a graph that contains 2^n different Hamiltonian cycles that correspond very naturally to the 2^n possible truth assignments to the variables. After this, we will add nodes to model the constraints imposed by the clauses.

We construct n paths $P_1, \ldots, P_n$, where P_i consists of nodes $v_{i1}, v_{i2}, \ldots, v_{ib}$ for a quantity b that we take to be somewhat larger than k; say, $b = 3k + 3$. There are edges from v_{ij} to $v_{i,j+1}$ and in the other direction from $v_{i,j+1}$ to v_{ij}. Thus, P_i can be traversed "left-to-right", from v_{i1} to v_{ib}, or "right-to-left", from v_{ib} to v_{i1}.

We hook these paths together as follows. For each $i = 1, 2, \ldots, n-1$, we define edges from v_{i1} to $v_{i+1,1}$ and to $v_{i+1,b}$. We also define edges from v_{ib} to $v_{i+1,1}$ and to $v_{i+1,b}$. We add two extra nodes s and t; we define edges from s to v_{11} and v_{1b}; from v_{n1} and v_{nb} to t; and from t to s.

The construction up to this point is pictured in Figure 7.2. It's important to pause here, and consider what the Hamiltonian cycles in our graph look like. Since only one edge leaves t, we know that any Hamiltonian cycle $\mathcal{C}$ must use the edge (t, s). After entering s, $\mathcal{C}$ can then traverse P_1 either left-to-right or right-to-left; regardless of what it does here, it can then traverse P_2 either left-to-right or right-to-left; and so forth, until it finishes traversing P_n and enters t. In other words, there are exactly 2^n different Hamiltonian cycles, and they correspond to the n independent choices of how to traverse each P_i.

This naturally models the n independent choices of how to set each variables $x_1, \ldots, x_n$ in the *3-SAT* instance. Thus, we will identify each Hamiltonian cycle uniquely with a truth assignment as follows: if $\mathcal{C}$ traverses P_i left-to-right, then x_i is set to 1; otherwise, x_i is set to 0.

Now we add nodes to model the clauses; the *3-SAT* instance will turn out to be satisfiable if and only if any Hamiltonian cycle survives. Let's consider, as a concrete example, a clause

$$C_1 = x_1 \vee \overline{x_2} \vee x_3.$$

In the language of Hamiltonian cycles, this clause says, "The cycle should traverse P_1 left-to-right; or it should traverse P_2 right-to-left; or it should traverse P_3 left-to-right." So we add a node c_1 as in Figure 7.3 that does just this. (Note that certain edges have been eliminated from this drawing, for the sake of clarity.) For some value of ℓ, node c_1 will have edges *from* $v_{1\ell}$, $v_{2,\ell+1}$, and $v_{3\ell}$; it will have edges *to* $v_{1,\ell+1}$, $v_{2,\ell}$, and $v_{3,\ell+1}$. Thus it can be easily spliced into any Hamiltonian cycle that traverses P_1 left-to-right by visiting node c_1 between $v_{1\ell}$ and $v_{1,\ell+1}$; similarly c_1 can be spliced into any Hamiltonian cycle that traverses P_2 right-to-left, or P_3 left-to-right. It cannot be spliced into a Hamiltonian cycle that does not do any of these things.

More generally, we will define a node c_j for each clause C_j. We will reserve node positions $3j$ and $3j + 1$ in each path P_i for variables that participate in clause C_j. Suppose clause C_j contains a term t. Then if $t = x_i$, we will add edges $(v_{i,3j}, c_j)$ and $(c_j, v_{i,3j+1})$; if $t = \overline{x_i}$, we will add edges $(v_{i,3j+1}, c_j)$ and $(c_j, v_{i,3j})$.

This completes the construction of the graph G. Now, following our generic outline for NP-completeness proofs, we claim that the *3-SAT* instance is satisfiable if and only if G has a Hamiltonian cycle.

First, suppose there is a satisfying assignment for the *3-SAT* instance. Then we define a Hamiltonian cycle following our informal plan above. If x_i is assigned 1 in the satisfying assignment, then we traverse the path P_i left-to-right; otherwise we traverse P_i right-to-left. For each clause C_j, since it is satisfied by the assignment, there will be at least path P_i in which we will be going in the "correct" direction relative to the node c_j, and we can splice it into the tour there via edges incident on $v_{i,3j}$ and $v_{i,3j+1}$.

Conversely, suppose that there is a Hamiltonian cycle $\mathcal{C}$ in G. The crucial thing to observe is the following. If $\mathcal{C}$ enters a node c_j on an edge from $v_{i,3j}$, it must depart on an edge to

$v_{i,3j+1}$. For if not, then $v_{i,3j+1}$ will have only one unvisited neighbor left, namely $v_{i,3j+2}$, and so the tour will not be able to visit this node and still maintain the Hamiltonian property. Symmetrically, if it enters from $v_{i,3j+1}$, it must depart immediately to $v_{i,3j}$. Thus, for each node c_j, the nodes immediately before and after c_j in the cycle $\mathcal{C}$ are joined by an edge e in G; thus, if we remove c_j from the cycle and insert this edge e for each j, then we obtain a Hamiltonian cycle $\mathcal{C}'$ on the subgraph $G - \{c_1, \ldots, c_k\}$. This is our original subgraph, before we added the clause nodes; as we noted above, any Hamiltonian cycle in this subgraph must traverse each P_i fully in one direction or the other. We thus use $\mathcal{C}'$ to define the following truth assignment for the *3-SAT* instance. If $\mathcal{C}'$ traverses P_i left-to-right, then we set $x_i = 1$; otherwise, we set $x_i = 0$. Since the larger cycle $\mathcal{C}$ was able to visit each clause node c_j, at least one of the paths was traversed in the "correct" direction relative to the node c_j, and so the assignment we have defined satisfies all the clauses.

Having established that the *3-SAT* instance is satisfiable if and only if G has a Hamiltonian cycle, our proof is complete. ■

It is sometimes useful to think about a variant of *Hamiltonian Cycle* in which it is not necessary to return to one's starting point. Thus, given a directed graph $G = (V, E)$, we say that a path P in G is a *Hamiltonian path* if it contains each vertex exactly once. (It is allowed to start at any node and end at any node, provided it respects this constraint.) Thus, such a path consists of distinct nodes $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ in order, such that they collectively constitute the entire vertex set V; by way of contrast with a Hamiltonian cycle, it is not necessary for there to be an edge from v_{i_n} back to v_{i_1}. Now, the *Hamiltonian Path* problem asks:

Given a directed graph G, does it contain a Hamiltonian path?

Using the intractability of *Hamiltonian Cycle*, it is not hard to show

(7.18) Hamiltonian Path *is NP-complete.*

Proof. First of all, *Hamiltonian Path* is in $\mathcal{NP}$: a certificate could be a path in G, and a certifier could then check that it is indeed a path and that it contains each node exactly once.

One way to show that *Hamiltonian Path* is NP-complete is to use a reduction from *3-SAT* that is almost identical to the one we used for *Hamiltonian Cycle*: we construct the same graph that appears in Figure 7.2, *except* that we do not include an edge from t to s. If there is any Hamiltonian path in this modified graph, it must begin at s (since s has no incoming edges) and end at t (since t has no outgoing edges). With this one change, we can adapt the argument used in the *Hamiltonian Cycle* reduction more or less word-for-word to argue that there is a satisfying assignment for the instance of *3-SAT* if and only if there is a Hamiltonian path.

An alternate way to show that *Hamiltonian Path* is NP-complete is to prove that *Hamiltonian Cycle* $\leq_P$ *Hamiltonian Path*. Given an instance of *Hamiltonian Cycle*, specified by a directed graph G, we construct a graph G' as follows. We choose an arbitrary node v in G, and replace it with two new nodes v' and v''. All edges out of v in G are now out of v'; and all edges into v in G are now into v''. More precisely, each edge (v, w) in G is replaced by an edge (v', w); and each edge (u, v) in G is replaced by an edge (u, v''). This completes the construction of G'.

We claim that G' contains a Hamiltonian path if and only if G contains a Hamiltonian cycle. Indeed, suppose C is a Hamiltonian cycle in G, and consider traversing it beginning and ending at node v. It is easy to see that the same ordering of nodes forms a Hamiltonian path in G' that begins at v' and ends at v''. Conversely, suppose P is a Hamiltonian cycle in G'. Clearly P must begin at v' (since v' has no incoming edges) and end at v'' (since v'' has no outgoing edges). If we replace v' and v'' with v, then this ordering of nodes forms a Hamiltonian cycle in G. ■

Armed with our basic hardness results for sequencing problems on directed graphs, we can move on to show the intractability of *Traveling Salesman*.

(7.19) Traveling Salesman *is NP-complete.*

Proof. We have argued earlier that *Traveling Salesman* is in $\mathcal{NP}$: the certificate is a permutation of the cities, and a certifier checks that the length of the corresponding tour is at most the given bound.

We now show that *Hamiltonian Cycle* $\leq_P$ *Traveling Salesman*. Given a directed graph $G = (V, E)$, we define the following instance of *Traveling Salesman*. We have a city v'_i for each node v_i of the graph G. We define $d(v'_i, v'_j)$ to be 1 if there is an edge (v_i, v_j) in G, and we define it to be 2 otherwise.

Now we claim that G has a *Hamiltonian Cycle* if and only if there is tour of length at most n in our *Traveling Salesman* instance. For if G has a *Hamiltonian Cycle*, then this ordering of the corresponding cities defines a tour of length n. Conversely, suppose there is a tour of length at most n. The expression for the length of this tour is a sum of n terms, each of which is at least 1; thus, it must be the case that all the terms are equal to 1. Hence, each pair of nodes in G that correspond to consecutive cities on the tour must be connected by an edge; it follows that the ordering of these corresponding nodes must form a Hamiltonian cycle. ■

Note that allowing *asymmetric* distances in the *Traveling Salesman* problem ($d(v'_i, v'_j) \neq d(v'_j, v'_i)$) played a crucial role; since the graph in the *Hamiltonian Cycle* instance is directed, our reduction yielded a *Traveling Salesman* instance with asymmetric distances.

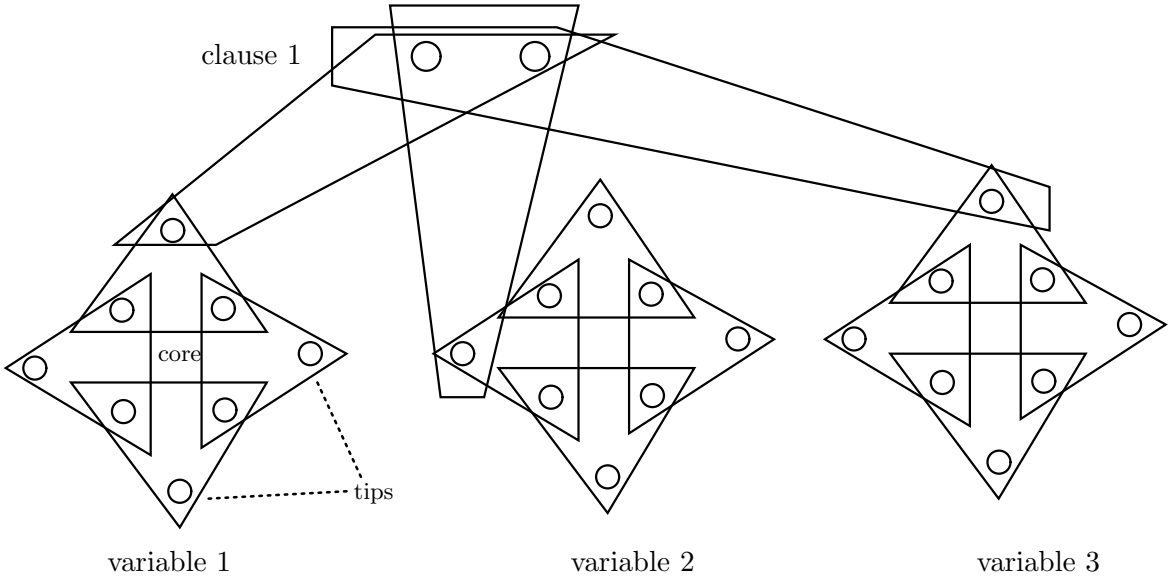


Figure 7.4: The reduction from *3-SAT* to *3-Dimensional Matching*.

In fact, the analogue of the *Hamiltonian Cycle* problem for undirected graphs is also NP-complete; although we will not prove this here, it follows via a not-too-difficult reduction from directed *Hamiltonian Cycle*. Using this undirected *Hamiltonian Cycle* problem, an exact analogue of (7.19) can be used to prove that the *Traveling Salesman* problem with symmetric distances is also NP-complete.

Of course, the most famous special case of the *Traveling Salesman* problem is the one in which the distances are defined by a set of n points in the plane. It is possible to reduce *Hamiltonian Cycle* to this special case as well, though this is much trickier.

Partitioning Problems.

(7.20) 3-Dimensional Matching *is NP-complete.*

Proof. Not surprisingly, it is easy to prove that *3-Dimensional Matching* is in $\mathcal{NP}$. Given a collection of triples $T \subset X \times Y \times Z$, a certificate that there is a solution could be a collection of triples $T' \subseteq T$. In polynomial time one could verify that each element in $X \cup Y \cup Z$ belongs to exactly one of the triples in T'.

For the reduction, we again return all the way to *3-SAT*. This is perhaps a little more curious than in the case of *Hamiltonian Cycle*, since *3-Dimensional Matching* bears some resemblance to both *Set Packing* and *Set Cover*— but the partitioning requirement is hard to encode using either of these problems.

Thus, consider an arbitrary instance of *3-SAT*, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$. We will show to solve it, given the ability to detect perfect three-dimensional

matchings.

The overall strategy in this reduction will be similar — at a very high level — to the approach we followed in the reduction from *3-SAT* to *Hamiltonian Cycle*. We will first design gadgets that encode the independent choices involved in the truth assignment to each variable; we will then add gadgets that encode the constraints imposed by the clauses. In performing this construction, we will initially describe all the elements in the *3-Dimensional Matching* instance simply as "elements," without trying to specify for each one whether it comes from X, Y, or Z. At the end, we will observe that they naturally decompose into these three sets.

Here is the basic gadget associated with variable x_i. We define elements $A_i = \{a_{i1}, a_{i2}, \ldots, a_{i,2k}\}$ that constitute the *core* of the gadget; we define elements $B_i = \{b_{i1}, \ldots, b_{i,2k}\}$ at the *tips* of the gadget. For each $j = 1, 2, \ldots, 2k$, we define a triple $t_{ij} = (a_{ij}, a_{i,j+1}, b_{ij})$, where we interpret addition modulo $2k$. Three of these gadgets are pictured in Figure 7.4. In gadget i, we will call a triple t_{ij} "even" if j is even, and "odd" if j is odd. In an analogous way, we will refer to a tip b_{ij} as being either "even" or "odd."

These will be the only triples that contain the elements in A_i, so we can already say something about how they must be covered in any perfect matching: we must either use all the even triples in gadget i, or all the odd triples in gadget i. This will be our basic way of encoding the idea that x_i can be set to either 0 or 1; if we select all the even triples, this will represent setting $x_i = 0$, and if we select all the odd triples, this will represent setting $x_i = 1$.

Here is another way to view the odd/even decision, in terms of the tips of the gadget. If we decide to use the even triples, we cover the even tips of the gadget, and leave the odd tips free. If we decide to use the odd triples, we cover the odd tips of the gadget, and leave the even tips free. Thus, our decision of how to set x_i can be viewed as follows: leaving the odd tips free corresponds to 0, while leaving the even tips free corresponds to 1. This will actually be the more useful way to think about things in the remainder of the construction.

So far, we can make this even/odd choice independently for each of the n variable gadgets. We now add elements to model the clauses, and constrain the assignments we can choose. As in the proof of (7.17), let's consider the example of a clause

$$C_1 = x_1 \vee \overline{x_2} \vee x_3.$$

In the language of three-dimensional matchings, it tells us, "The matching on the cores of the gadgets should leave the even tips of the first gadget free; or it should leave the odd tips of the second gadget free; or it should leave the even tips of the third gadget free." So we add a *clause gadget* that does precisely this. It consists of a set of two *core* elements $P_1 = \{p_1, p_1'\}$, and three triples that contain them. One has the form (p_1, p_1', b_{1j}) for an even tip b_{1j}; another includes p_1, p_1', and an odd tip $b_{2,j'}$; and a third includes p_1, p_1', and an even

tip $b_{3,j''}$. These are the only three triples that cover P_1, so we know that one of them must be used; this enforces the clause constraint exactly.

In general, for clause C_j, we create a gadget with two core elements $P_j = \{p_j, p'_j\}$, and we define three triples containing P_j as follows. Suppose clause C_j contains a term t. If $t = x_i$, we define a triple $(p_j, p'_j, b_{i,2j})$; if $t = \overline{x_i}$, we define a triple $(p_j, p'_j, b_{i,2j-1})$. Note that only clause gadget j makes use of tips b_{im} with $m = 2j$ or $m = 2j - 1$; thus, the clause gadgets will never "compete" with each other for free tips.

We are almost done with the construction, but there's still one problem. Suppose the set of clauses has a satisfying assignment. Then we make the corresponding choices of odd/even for each variable gadget; this leaves at least one free tip for each clause gadget, and so all the core elements of the clause gadgets get covered as well. The problem is: *we haven't covered all the tips.* We started with $n \cdot 2k = 2nk$ tips; the triples $\{t_{ij}\}$ covered nk of them; and the clause gadgets covered an additional k of them. This leaves $(n-1)k$ tips left to be covered.

We handle this problem with a very simple trick: we add $(n-1)k$ "cleanup gadgets" to the construction. Cleanup gadget i consists of two core elements $Q_i = \{q_i, q'_i\}$, and there is a triple (q_i, q_i, b) for *every* tip b in every variable gadget. This is the final piece of the construction.

Thus, if the set of clauses has a satisfying assignment, then we make the corresponding choices of odd/even for each variable gadget; as before, this leaves at least one free tip for each clause gadget. Using the cleanup gadgets to cover the remaining tips, we see that all core elements in the variable, clause, and cleanup gadgets have been covered, and all tips have been covered as well.

Conversely, suppose there is a perfect three-dimensional matching in the instance we have constructed. Then, as we argued above, in each variable gadget the matching chooses either all the even $\{t_{ij}\}$ or all the odd $\{t_{ij}\}$. In the former case, we set $x_i = 0$ in the *3-SAT* instance; and in the latter case, we set $x_i = 1$. Now, consider clause C_j; has it been satisfied? Because the two core elements in P_j have been covered, at least one of the three variable gadgets corresponding to a term in C_j made the "correct" odd/even decision, and this induces a variable assignment that satisfies C_j.

This concludes the proof, except for one last thing to worry about: Have we really constructed an instance of *3-Dimensional Matching*? We have a collection of elements, and triples containing certain of them, but can the elements really be partitioned into appropriate sets X, Y, and Z of equal size?

Fortunately, the answer is yes. We can define X to be set of all a_{ij} with j even, the set of all p_j, and the set of all q_i. We can define Y to be set of all a_{ij} with j odd, the set of all p'_j, and the set of all q'_i. Finally, we can define Z to be the set of all tips b_{ij}. It is now easy to check that each triple consists of one element from each of X, Y, and Z. ■

7.6 The Hardness of Numerical Problems

We now consider problems that involve arithmetic operations on integers. Up till now, there has not been much ambiguity about how to define the "size" of the input to a problem — of the many standard representations for graphs or sets, for example, all are related by polynomial factors. But when we work with numbers, we have to resolve a fundamental issue: given an input number w, is its *size* $O(\log w)$ or $O(w)$. In other words, do we consider a 100-bit number as having size roughly 100 or roughly 2^{100}?

In most settings, the standard is that the number w has input size $O(\log w)$ — numbers are represented in base-d notation, for some $d > 1$. As a result, algorithms that run in time $O(w)$ are not polynomial-time.

With this in mind, we consider the *Subset Sum* problem. We have seen an $O(nW)$ algorithm that uses dynamic programming; we now ask: Could there be an algorithm with running time polynomial in n and $\log W$? Or polynomial in n alone?

The following result suggests that this is not likely to be the case.

(7.21) Subset Sum *is NP-complete.*

Proof. We first show that *Subset Sum* is in $\mathcal{NP}$. Given natural numbers $w_1, \ldots, w_n$, and a target W, a certificate that there is a solution would be the subset $w_{i_1}, \ldots, w_{i_k}$ that is purported to add up to W. In polynomial time, we can compute the sum of these numbers, and verify that it is equal to W.

We now reduce a known NP-complete problem to *Subset Sum*. Since we are seeking a set that adds up to *exactly* a given quantity (as opposed to being bounded above or below by this quantity), we look for a combinatorial problem that is based meeting an *exact* bound. The *3-Dimensional Matching* problem is a natural choice; we show that *3-Dimensional Matching* $\leq_P$ *Subset Sum*. The trick will be to encode the manipulation of sets via the addition of integers.

So consider an instance of *3-Dimensional Matching* specified by sets X, Y, Z, each of size n, and a set of m triples $T \subseteq X \times Y \times Z$. A common way to represent sets is via *bit-vectors*: each entry in the vector corresponds to a different element, and it holds a 1 if and only if the set contains that element. We adopt this type of approach for representing each triple $t = (x_i, y_j, z_k) \in T$: we construct a number w_t with $3n$ digits that has a 1 in positions i, $n + j$, and $2n + k$, and a 0 in all other positions. In other words, for some base $d > 1$, $w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}$.

Note how taking the union of triples *almost* corresponds to integer addition: the 1's fill in the places where there is an element in any of the sets. But we say "almost" because addition includes *carries*: too many 1's in the same column will "roll over" and produce a non-zero entry in the next column. This has no analogue in the context of the union operation.

In the present situation, we handle this problem by a simple trick. We have only m numbers in all, and each has digits equal to 0 or 1; so if we assume that our numbers are written in base $d = m + 1$, then there will be no carries at all.

Thus, we construct the following instance of *Subset Sum*. For each triple $t = (x_i, y_j, z_k) \in T$, we construct a number w_t in base $m+1$ as defined above. We define W to be the number in base $m + 1$ with $3n$ digits, each of which is equal to 1, i.e., $W = \sum_{i=0}^{3n-1}(m + 1)^i$.

We claim that the set T of triples contains a perfect three-dimensional matching if and only if there is a subset of the numbers $\{w_t\}$ that adds up to W. For suppose there is a perfect three-dimensional matching consisting of triples $t_1, \ldots, t_n$. Then in the sum $w_{t_1} + \cdots + w_{t_n}$, there is a single 1 in each of the $3n$ digit positions, and so the result is equal to W.

Conversely, suppose there exists a set of numbers $w_{t_1}, \ldots, w_{t_k}$ that adds up to W. Then since each w_{t_i} has three 1's in its representation, and there are no carries, we know that $k = n$. It follows that for each of the $3n$ digit positions, exactly one of the w_{t_i} has a 1 in that position. Thus, $t_1, \ldots, t_k$ constitute a perfect three-dimensional matching. ■

The hardness of *Subset Sum* can be used to establish the hardness of a range of scheduling problems — including some that do not obviously involve the addition of numbers. Here is a nice example, a natural (but much harder) generalization of a scheduling problem we solved earlier in the course using a greedy algorithm.

Suppose we are given a set of n jobs that must be run on a single machine. Each job i has a *release time* r_i, a *deadline* d_i, and a processing duration t_i. We will assume that all of these parameters are natural numbers. In order to be completed, job i must be allocated a contiguous slot of t_i time units somewhere in the interval $[r_i, d_i]$. The machine can only run one job at a time. The question is: can we schedule all jobs so that each completes by its deadline? We will call this an instance of *Scheduling with Release Times and Deadlines*.

(7.22) Scheduling with Release Times and Deadlines *is NP-complete.*

Proof. Given an instance of the problem, a certificate that it is solvable would be a specification of the starting time for each job. We could then check that each job runs for a distinct interval of time, between its release time and deadline. Thus, the problem is in $\mathcal{NP}$.

We now show that *Subset Sum* is reducible to this scheduling problem. Thus, consider an instance of *Subset Sum* with numbers $w_1, \ldots, w_n$ and a target W. In constructing an equivalent scheduling instance, one is struck initially by the fact that we have so many parameters to manage — release times, deadlines, durations. The key is to sacrifice most of this flexibility, producing a "skeletal" instance of the problem that still encodes the *Subset Sum* problem.

Let $S = \sum_{i=1}^{n} w_i$. We define jobs $1, 2, \ldots, n$; job i has a release time of 0, a deadline of $S + 1$, and a duration of w_i. For this set of jobs, we have the freedom to arrange them in any order, and they will all finish on time.

We now further constrain the instance so that the only way to solve it will be to group together a subset of the jobs whose durations add up to precisely W. To do this, we define an $(n+1)^{\text{st}}$ job; it has a release time of W, a deadline of $W+1$, and a duration of 1.

Now, consider any feasible solution to this scheduling instance. The $(n+1)^{\text{st}}$ job must be run in the interval $[W, W+1]$. This leaves S available time units between the common release time and the common deadline; and there are S time units worth of jobs to run. Thus, the machine must not have any "idle time," when no jobs are running. In particular, if jobs $i_1, \ldots, i_k$ are the ones that run before time W, then the corresponding numbers $w_{i_1}, \ldots, w_{i_k}$ in the *Subset Sum* instance add up to exactly W.

Conversely, if there are numbers $w_{i_1}, \ldots, w_{i_k}$ that add up to exactly W, then we can schedule these before job $n+1$, and the remainder after job $n+1$; this is a feasible solution to the scheduling instance. ■

Caveat: Subset Sum with Polynomially Bounded Numbers. There is a very common source of pitfalls involving the *Subset Sum* problem, and while it is closely connected to the issues we have been discussing already, we feel it is worth discussing explicitly. The pitfall is the following:

> Consider the special case of *Subset Sum*, with n input numbers, in which W is bounded by a polynomial function of n. Assuming $\mathcal{P} \neq \mathcal{NP}$, this special case is *not* NP-complete.

It is not NP-complete for the simple reason that it can be solved in time $O(nW)$, by our dynamic programming algorithm from earlier in the course; when W is bounded by a polynomial function of n, this is a polynomial-time algorithm.

All this is very clear; so you may ask: Why dwell on it? The reason is that there is a genre of problem that is often wrongly claimed to be NP-complete (even in published papers) via reduction to this special case of *Subset Sum*. Here is a basic example of such a problem, which we will call *Component Grouping*:

> Given a graph G that is not connected, and a number k, does there exist a subset of its connected components whose union has size exactly k?

Incorrect Claim: Component Grouping *is NP-complete.*

Incorrect Proof: Component Grouping is in $\mathcal{NP}$, and we'll skip the proof of this. We show that *Subset Sum* $\leq_P$ *Component Grouping*. Given an instance of *Subset Sum* with numbers $w_1, \ldots, w_n$ and target W, we construct an instance of *Component Grouping* as follows. For each i, we construct a path P_i of length w_i. The graph G will be the union of the paths $P_1, \ldots, P_n$, each of which is a separate connected component. We set $k = W$. It is clear that

G has a set of connected components whose union has size k if and only if some subset of the numbers $w_1, \ldots, w_n$ adds up to W. ■

The error here is subtle; in particular, the claim in the last sentence is correct. The problem is that the construction described above does not establish that *Subset Sum* $\leq_P$ *Component Grouping*, because it requires more than polynomial time. In constructing the input to our black box that solves *Component Grouping*, we had to build the encoding of a graph of size $w_1 + \cdots + w_n$, and this takes time exponential in the size of the input to the *Subset Sum* instance. In effect, *Subset Sum* works with the numbers $w_1, \ldots, w_n$ in a very compact representation, but *Component Grouping* does not accept "compact" encodings of graphs.

The problem is more fundamental than the incorrectness of this proof; in fact, *Component Grouping* is a problem that can be solved in polynomial time. If $n_1, n_2, \ldots, n_c$ denote the sizes of the connected components of G, we simply use our dynamic programming algorithm for *Subset Sum* to decide whether some subset of these numbers $\{n_i\}$ adds up to k. The running time required for this is $O(nk)$; and since $k \leq n$, this is $O(n^2)$ time.

Thus, we have discovered a new polynomial-time algorithm by reducing in the other direction, to a polynomial-time solvable special case of *Subset Sum*.

7.7 co-NP and the Asymmetry of NP.

To conclude this chapter, let's return to the definitions underlying the class $\mathcal{NP}$. We've seen that the notion of an efficient certifier doesn't suggest a concrete algorithm for actually solving the problem that's better than brute-force search.

Now, here's another thing to notice: the definition of efficient certification, and hence of $\mathcal{NP}$, is fundamentally *asymmetric*. An input string s is a "yes' instance if and only if there exists a short t so that $B(s, t) =$ `yes`. Negating this statement, we see that an input string s is a "no" instance if and only if *for all* short t, it's the case that $B(s, t) =$ `no`.

This relates closely to our intuition about $\mathcal{NP}$: when we have a "yes" instance, we can provide a short proof of this fact. But when we have a "no" instance, no correspondingly short proof is guaranteed by the definition; the answer is "no" simply because there is no string that will serve as a proof. In concrete terms, recall our question from earlier in this chapter: given an unsatisfiable set of clauses, what evidence could we show to quickly convince you that there is no satisfying assignment?

For every problem X, there is a natural *complementary* problem $\overline{X}$: for all input strings s, we say $s \in \overline{X}$ if and only if $s \notin X$. Note that if $X \in \mathcal{P}$, then $\overline{X} \in \mathcal{P}$, since from an algorithm A that solves X, we can simply produce an algorithm $\overline{A}$ that runs A and then flips its answer.

But it is far from clear that if $X \in \mathcal{NP}$, it should follow that $\overline{X} \in \mathcal{NP}$. $\overline{X}$, rather, has a different property: for all s, we have $s \in \overline{X}$ if and only if for all t of length at most $p(|s|)$, $B(s,t) = \texttt{no}$. This is a fundamentally different definition, and it can't be worked around by simply "inverting" the output of the efficient certifier B to produce $\overline{B}$. The problem is that the "exists t" in the definition of $\mathcal{NP}$ has become a "for all t", and this is a serious change.

There is a class of problems parallel to $\mathcal{NP}$ that is designed to model this issue; it is called, naturally enough, co$-\mathcal{NP}$. A problem X belongs to co$-\mathcal{NP}$ if and only if the complementary problem $\overline{X}$ belongs to $\mathcal{NP}$. We do not know for sure that $\mathcal{NP}$ and co$-\mathcal{NP}$ are different; we can only ask

(7.23) *Does* $\mathcal{NP} = \text{co}-\mathcal{NP}$*?*

Again, the widespread belief is that $\mathcal{NP} \neq \text{co}-\mathcal{NP}$: just because the "yes" instances of a problem have short proof, it should not automatically follow that the "no" instances have short proofs as well.

Proving $\mathcal{NP} \neq \text{co}-\mathcal{NP}$ would be an even bigger step than proving $\mathcal{P} \neq \mathcal{NP}$, for the following reason:

(7.24) *If* $\mathcal{NP} \neq \text{co}-\mathcal{NP}$*, then* $\mathcal{P} \neq \mathcal{NP}$

Proof. We'll actually prove the contrapositive statement: $\mathcal{P} = \mathcal{NP}$ implies $\mathcal{NP} = \text{co}-\mathcal{NP}$. Essentially, the point is that $\mathcal{P}$ is closed under complementation; so if $\mathcal{P} = \mathcal{NP}$, then $\mathcal{NP}$ would be closed under complementation as well. More formally, starting from the assumption $\mathcal{P} = \mathcal{NP}$, we have

$$X \in \mathcal{NP} \Longrightarrow X \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow X \in \text{co}-\mathcal{NP}$$

and

$$X \in \text{co}-\mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow X \in \mathcal{P} \Longrightarrow X \in \mathcal{NP}.$$

Hence it would follow that $\mathcal{NP} \subseteq \text{co}-\mathcal{NP}$ and co$-\mathcal{NP} \subseteq \mathcal{NP}$, whence $\mathcal{NP} = \text{co}-\mathcal{NP}$. ■

Good Characterizations: The class $\mathcal{NP} \cap$ co$-\mathcal{NP}$. If a problem X belongs to both $\mathcal{NP}$ and co$-\mathcal{NP}$, then it has the following nice property: when the answer is "yes," there is a short proof; and when the answer is "no," there is also a short proof. Thus, problems that belong to this intersection $\mathcal{NP} \cap \text{co}-\mathcal{NP}$ are said to have a *good characterization*, since there is always a nice certificate for the solution.

This notion corresponds directly to some of the results we have seen earlier in the course. For example, consider the problem of determining whether a flow network contains a flow of value at least ν, for some quantity ν. To prove that the answer is "yes", we could simply exhibit a flow that achieves this value; this is consistent with the problem belonging to $\mathcal{NP}$.

But we can also prove the answer is "no": we can exhibit a cut whose capacity is strictly less than ν. This duality between "yes" and "no" instances is the crux of the Max-flow Min-Cut Theorem.

Similarly, Hall's Theorem for matchings proved that the bipartite perfect matching problem is in $\mathcal{NP} \cap \text{co}-\mathcal{NP}$: we can either exhibit a perfect matching, or a set of vertices $A \subseteq X$ such that the total number of neighbors of A is strictly less than $|A|$.

Now, if a problem X is in $\mathcal{P}$, then it belongs to both $\mathcal{NP}$ and co$-\mathcal{NP}$; thus, $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}-\mathcal{NP}$. Interestingly, both our proof of the Max-flow Min-cut Theorem and our proof of Hall's Theorem came hand-in-hand with proofs of the stronger results that maximum flow and bipartite matching are problems in $\mathcal{P}$. Nevertheless, the good characterizations themselves are so clean that formulating them separately still gives us a lot of conceptual leverage in reasoning about these problems.

Naturally, one would like to know whether there's a problem that has a good characterization but no polynomial-time algorithm. But this too is an open question:

(7.25) *Does $\mathcal{P} = \mathcal{NP} \cap \text{co}-\mathcal{NP}$?*

Unlike questions (7.12) and (7.23), general opinion seems fairly mixed on this one. In part, this is because there are many cases in which a problem was found to have a non-trivial good characterization; and then — sometimes many years later — it was also discovered to have a polynomial-time algorithm. We simply lack a large set of good candidates for problems in $(\mathcal{NP} \cap \text{co}-\mathcal{NP}) - \mathcal{P}$.

7.8 Exercises

1. You want to get a break from all the homework and projects that you're doing, so you're planning a large party over the weekend. Many of your friends are involved in group projects, and you are worried that if you invite a whole group to your party, the group might start discussing their project instead of enjoying your party.

   Before you realize it, you've formulated the Party Selection Problem. You have a set of F friends whom you're considering to invite, and you're aware of a set of k project groups, $S_1, \ldots, S_k$, among these friends. The problem is to decide if there is a set of n of your friends whom you could invite so that not all members of any one group are invited.

   Prove that the Party Selection Problem is NP-complete.

2. Given an undirected graph $G = (V, E)$, a *feedback set* is a set $X \subseteq V$ with the property that $G - X$ has no cycles. The undirected feedback set problem asks: given G and k, does G contain a feedback set of size at most k? Prove that undirected feedback set is NP-complete.

3. Consider a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. (That is, $B_i \subseteq A$ for each i.)

   We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i. (So H "hits" all the sets B_i.)

   We now define the HITTING SET problem as follows. We are given a set $A = \{a_1, \ldots, a_n\}$, a collection $B_1, B_2, \ldots, B_m$ of subsets of A, and a number k. We are asked: is there a hitting set $H \subseteq A$ for $B_1, B_2, \ldots, B_m$ so that the size of H is at most k?

   Prove that HITTING SET is NP-complete.

4. Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who's skilled at each of the n sports covered by the camp (baseball, volleyball, and so on). They have received job applications from m potential counselors. For each of the n sports, there is some subset of the m applicants that is qualified in that sport. The question is: for a given number $k < m$, is it possible to hire at most k of the counselors and have at least one counselor qualified in each of the n sports? We'll call this the EFFICIENT RECRUITING PROBLEM.

   Show that EFFICIENT RECRUITING PROBLEM is NP-complete.

5. We've seen the Interval Scheduling problem in class; here we consider a computationally much harder version of it that we'll call MULTIPLE INTERVAL SCHEDULING. As before, you have a processor that is available to run jobs over some period of time. (E.g. 9 AM to 5 PM.)

   People submit jobs to run on the processor; the processor can only work on one job at any single point in time. Jobs in this model, however, are more complicated than we've seen in the past: each job requires a *set* of intervals of time during which it needs to use the processor. Thus, for example, a single job could require the processor from 10 AM to 11 AM, and again from 2 PM to 3 PM. If you accept this job, it ties up your processor during those two hours, but you could still accept jobs that need any other time periods (including the hours from 11 to 2).

   Now, you're given a set of n jobs, each specified by a set of time intervals, and you want to answer the following question: For a given number k, is it possible to accept at least k of the jobs so that no two of the accepted jobs have any overlap in time?

   Show that MULTIPLE INTERVAL SCHEDULING is NP-complete.

6. Since the 3-Dimensional Matching problem is NP-complete, it is natural to expect that the corresponding 4-Dimensional Matching problem is at least as hard. Let us define 4-Dimensional Matching as follows. Given sets W, X, Y, and Z, each of size n, and a collection C of ordered 4-tuples of the form (w_i, x_j, y_k, z_ℓ), do there exist n 4-tuples from C so that no two have an element in common?

   Prove that 4-Dimensional Matching is NP-complete.

7. The following is a version of the Independent Set problem. You are given a graph $G = (V, E)$ and an integer k. For this problem, we will call a set $I \subset V$ *strongly independent* if for any two nodes $v, u \in I$, the edge (v, u) does not belong to E, and there is also no path of 2 edges from u to v, i.e., there is no node w such that both $(u, w) \in E$ and $(w, v) \in E$. The Strongly Independent Set problem is to decide whether G has a strongly independent set of size k.

   Prove that the Strongly Independent Set problem is NP-complete.

8. Consider the problem of reasoning about the identity of a set from the size of its intersections with other sets. You are given a finite set U of size n, and a collection $A_1, \ldots, A_m$ of subsets of U. You are also given numbers $c_1, \ldots, c_m$. The question is: does there exist a set $X \subset U$ so that for each $i = 1, 2, \ldots, m$, the cardinality of $X \cap A_i$ is equal to c_i? We will call this an instance of the *Intersection Inference* problem, with input U, $\{A_i\}$, and $\{c_i\}$.

   Prove that *Intersection Inference* is NP-complete.

9. You're consulting for a small high-tech company that maintains a high-security computer system for some sensitive work that it's doing. To make sure this system is not being used for any illicit purposes, they've set up some logging software that records the IP addresses that all their users are accessing over time. We'll assume that each user accesses at most one IP address in any given minute; the software writes a log file that records, for each user u and each minute m, a value $I(u, m)$ that is equal to the IP address (if any) accessed by user u during minute m. It sets $I(u, m)$ to the null symbol $\perp$ if u did not access any IP address during minute m.

   The company management just learned that yesterday, the system was used to launch a complex attack on some remote sites. The attack was carried out by accessing t distinct IP addresses over t consecutive minutes: in minute 1, the attack accessed address i_1; in minute 2, it accessed address i_2; and so on, up to address i_t in minute t.

   Who could have been responsible for carrying out this attack? The company checks the logs, and finds to its surprise that there's no single user u who accessed each of the IP addresses involved at the appropriate time; in other words, there's no u so that $I(u, m) = i_m$ for each minute m from 1 to t.

So the question becomes: what if there were a small *coalition* of k users that collectively might have carried out the attack? We will say a subset S of users is a *suspicious coalition* if for each minute m from 1 to t, there is at least one user $u \in S$ for which $I(u, m) = i_m$. (In other words, each IP address was accessed at the appropriate time by at least one user in the coalition.)

The *Suspicious Coalition* problem asks: given the collection of all values $I(u, m)$, and a number k, is there a suspicious coalition of size at most k?

10. As some people remember, and many have been told, the idea of hypertext predates the World Wide Web by decades. Even hypertext fiction is a relatively old idea — rather than being constrained by the linearity of the printed page, you can plot a story that consists of a collection of interlocked virtual "places" joined by virtual "passages."[2] So a piece of hypertext fiction is really riding on an underlying directed graph; to be concrete (though narrowing the full range of what the domain can do) we'll model this as follows.

    Let's view the structure of a piece of hypertext fiction as a directed graph $G = (V, E)$. Each node $u \in V$ contains some text; when the reader is currently at u, they can choose to follow any edge out of u; and if they choose $e = (u, v)$, they arrive next at the node v. There is a start node $s \in V$ where the reader begins, and an end node $t \in V$; when the reader first reaches t, the story ends. Thus, any path from s to t is a valid *plot* of the story. Note that, unlike a Web browser, there is not necessarily a way to go back; once you've gone from u to v, you might not be able to ever return to u.

    In this way, the hypertext structure defines a huge number of different plots on the same underlying content; and the relationships among all these possibilities can grow very intricate. Here's a type of problem one encounters when reasoning about a structure like this. Consider a piece of hypertext fiction built on a graph $G = (V, E)$ in which there are certain crucial *thematic elements* — love; death; war; an intense desire to major in computer science; and so forth. Each thematic element i is represented by a set $T_i \subseteq V$ consisting of the nodes in G at which this theme appears. Now, given a particular set of thematic elements, we may ask: is there a valid plot of the story in which each of these elements is encountered? More concretely, given a directed graph G, with start node s and end node t, and thematic elements represented by sets $T_1, T_2, \ldots, T_k$, the *Plot Fulfillment* problem asks: is there a path from s to t that contains at least one node from each of the sets T_i?

    Prove that *Plot Fulfillment* is NP-complete.

11. *(We thank Maverick Woo for the idea of the Star Wars theme.)* There are those who

[2] See e.g. http://www.eastgate.com

insist that the initial working title for Episode XXVII of the Star Wars series was "P = NP" — but this is surely apocryphal. In any case, if you're so inclined, it's easy to find NP-complete problems lurking just below the surface of the original Star Wars movies.

Consider the problem faced by Luke, Leia, and friends as they tried to make their way from the Death Star back to the hidden Rebel base. We can view the galaxy as an undirected graph $G = (V, E)$, where each node is a star system and an edge (u, v) indicates that one can travel directly from u to v. The Death Star is represented by a node s, the hidden Rebel base by a node t. Certain edges in this graph represent longer distances than others; thus, each edge e has an integer *length* $\ell_e \geq 0$. Also, certain edges represent routes that are more heavily patrolled by evil Imperial spacecraft; so each edge e also has an integer *risk* $r_e \geq 0$, indicating the expected amount of damage incurred from special-effects-intensive space battles if one traverses this edge.

It would be safest to travel through the outer rim of the galaxy, from one quiet upstate star system to another; but then one's ship would run out of fuel long before getting to its destination. Alternately, it would be quickest to plunge through the cosmopolitan core of the galaxy; but then there would be far too many Imperial spacecraft to deal with. In general, for any path P from s to t, we can define its *total length* to be the sum of the lengths of all its edges; and we can define its *total risk* to be the sum of the risks of all its edges.

So Luke, Leia, and company are looking at a complex type of shortest-path problem in this graph: they need to get from s to t along a path whose total length and total risk are *both* reasonably small. In concrete terms, we can phrase the *Galactic Shortest Path* problem as follows: given a set-up as above, and integer bounds L and R, is there a path from s to t whose total length is at most L, *and* whose total risk is at most R?

Prove that *Galactic Shortest Path* is NP-complete.

12. The mapping of genomes involves a large array of difficult computational problems. At the most basic level, each of an organism's chromosomes can be viewed as an extremely long string (generally containing millions of symbols) over the four-letter alphabet $\{a, c, g, t\}$. One family of approaches to genome mapping is to generate a large number of short, overlapping snippets from a chromosome, and then to infer the full long string representing the chromosome from this set of overlapping substrings.

    While we won't be able to go into these string assembly problems in full detail, here's a simplified problem that suggests some of the computational difficulty one encounters in this area. Suppose we have a set $S = \{s_1, s_2, \ldots, s_n\}$ of short DNA strings over a q-letter alphabet; and each string s_i has length 2ℓ, for some number $\ell \geq 1$. We also have a library of additional strings $T = \{t_1, t_2, \ldots, t_m\}$ over the same alphabet;

each of these also has length 2ℓ. In trying to assess whether the string s_b might come directly after the string s_a in the chromosome, we will look to see whether the library T contains a string t_k so that the first ℓ symbols in t_k are equal to the last ℓ symbols in s_a, and the last ℓ symbols in t_k are equal to the first ℓ symbols in s_b. If this is possible, we will say that t_k *corroborates* the pair (s_a, s_b). (In other words, t_k could be a snippet of DNA that straddled the region in which s_b directly followed s_a.)

Now, we'd like to concatenate all the strings in S in some order, one after the other with no overlaps, so that each consecutive pair is corroborated by some string in the library T. That is, we'd like to order the strings in S as $s_{i_1}, s_{i_2}, \ldots, s_{i_n}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $\{1, 2, \ldots, n\}$, so that for each $j = 1, 2, \ldots, n-1$, there is a string t_k that corroborates the pair $(s_{i_j}, s_{i_{j+1}})$. (The same string t_k can be used for more than one consecutive pair in the concatenation.) If this is possible, we will say that the set S has a *perfect assembly*.

Given sets S and T, the *Perfect Assembly* problem asks: does S have an assembly with respect to T? Prove that *Perfect Assembly* is NP-complete.

Example. Suppose the alphabet is $\{a, c, g, t\}$, the set $S = \{ag, tc, ta\}$, and the set $T = \{ac, ca, gc, gt\}$. (So each string has length $2\ell = 2$.) Then the answer to this instance of *Perfect Assembly* is "yes" — we can concatanate the three strings in S in the order *tcagta*. (So $s_{i_1} = s_2$, $s_{i_2} = s_1$, and $s_{i_3} = s_3$.) In this order, the pair (s_{i_1}, s_{i_2}) is corroborated by the string *ca* in the library T, and the pair (s_{i_2}, s_{i_3}) is corroborated by the string *gt* in the library T.

13. Suppose you're consulting for a company that's setting up a Web site. They want to make the site publically accessible, but only in a limited way; people should be allowed to navigate through the site provided they don't become too "intrusive."

    We'll model the site as a directed graph $G = (V, E)$, in which the nodes represent Web pages and the edges represent directed hyperlinks. There is a distinguished *entry node* $s \in V$. The company plans to regulate the flow of traffic as follows. They'll define a collection of *restricted zones* $Z_1, \ldots, Z_k$, each of which is a subset of V. These zones Z_i need not be disjoint from one another. The company has a mechanism to track the path followed by any user through the site; if the user visits a single zone more than once, an *alarm* is set off and the user's session is terminated.

    The company wants to be able to answer a number of questions about its monitoring system; among them is the following EVASIVE PATH problem: Given G, $Z_1, \ldots, Z_k$, $s \in V$, and a *destination node* $t \in V$, is there an s-t path in G that does not set off an alarm? (I.e. it passes through each zone at most once.) Prove that EVASIVE PATH is NP-complete.

14. Consider an instance of the *Satisfiability* problem, specified by clauses $C_1, \ldots, C_k$ over a set of Boolean variables $x_1, \ldots, x_n$. We say that the instance is *monotone* if each term in each clause consists of a non-negated variable; that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$. Monotone instances of *Satisfiability* are very easy to solve: they are always satisfiable, by setting each variable equal to 1.

    For example, suppose we have the three clauses

    $$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

    This is monotone, and indeed the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

    Given a monotone instance of *Satisfiability*, together with a number k, the problem of *Monotone Satisfiability with Few True Variables* asks: is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

15. Suppose you're consulting for a group that manages a high-performance real-time system in which asynchronous process make use of shared resources. Thus, the system has a set of n *processes* and a set of m *resources*. At any given point in time, each process specifies a set of resources that it requests to use. Each resource might be requested by many processes at once; but it can only be used by a single process at a time.

    Your job is to allocate resources to processes that request them. if a process is allocated all the resources it requests, then it is *active*; otherwise it is *blocked*. You want to perform the allocation so that as many processes as possible are active. Thus, we phrase the Resource Reservation problem as follows: given a set of process and resources, the set of requested resources for each process, and a number k, is it possible to allocate resources to processes so that at least k processes will be active?

    Show that Resource Reservation is NP-complete.

16. You're configuring a large network of workstations, which we'll model as an undirected graph G — the nodes of G represent individual workstations and the edges represent direct communication links. The workstations all need access to a common *core database*, which contains data necessary for basic operating system functions.

    You could replicate this database on each workstation — then lookups would be very fast from any workstation, but you'd have to manage a huge number of copies. Alternately, you could keep a single copy of the database on one workstation and have the remaining workstations issue requests for data over the network G; but this could

result in large delays for a workstation that's many hops away from the site of the database.

So you decide to look for the following compromise: you want to maintain a small number of copies, but place them so that any workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database.

Thus, we phrase the Server Placement problem as follows: Given the network G, and a number k, is there a way to place k copies of the database at k different workstations so that every workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database?

17. Three of your friends work for a large computer-game company, and they've been working hard for several months now to get their proposal for a new game, *Droid Trader!*, approved by higher management. In the process, they've had to endure all sorts of discouraging comments, ranging from, "You're really going to have to work with Marketing on the name," to, "Why don't you emphasize the parts where people get to kick each other in the head?"

    At this point, though, it's all but certain that the game is really heading into production, and your friends come to you with one final issue that's been worrying them: What if the overall premise of the game is too simple, so that players get really good at it and become bored too quickly?

    It takes you a while, listening to their detailed description of the game, to figure out what's going on; but once you strip away the space battles, kick-boxing interludes, and Stars-Wars-inspired-pseudo-mysticism, the basic idea is as follows. A player in the game controls a spaceship and is trying to make money buying and selling droids on different planets. There are n different types of droids, and k different planets. Each planet p has the following properties: there are $s(j,p) \geq 0$ droids of type j available for sale, at a fixed price of $x(j,p) \geq 0$ each, for $j = 1, 2, \ldots, n$; and there is a demand for $d(j,p) \geq 0$ droids of type j, at a fixed price of $y(j,p) \geq 0$ each. (We will assume that a planet does not simultaneously have both a positive supply and a positive demand for a single type of droid; so for each j, at least one of $s(j,p)$ or $d(j,p)$ is equal to 0.)

    The player begins on planet s with z units of money, and must end at planet t; there is a directed acyclic graph G on the set of planets, such that s-t paths in G correspond to valid routes by the player. (G is chosen to be acyclic to prevent arbitrarily long games.) For a given s-t path P in G, the player can engage in transactions as follows: whenever the player arrives at a planet p on the path P, she can buy up to $s(j,p)$ droids of type j for $x(j,p)$ units of money each (provided she has sufficient money on hand) and/or sell up to $d(j,p)$ droids of type j for $y(j,p)$ units of money each (for

$j = 1, 2, \ldots, n$). The player's *final score* is the total amount of money she has on hand when she arrives at planet t. (There are also bonus points based on space battles and kick-boxing, which we'll ignore for the purposes of formulating this question ...)

So basically, the underlying problem is to achieve a high score. In other words, given an instance of this game, with a directed acyclic graph G on a set of planets, all the other parameters described above, and also a target bound B, is there an path P in G and a sequence of transactions on P so that the player ends with a final score that is at least B? We'll call this an instance of the *High-Score-on-Droid-Trader!* problem, or *HSoDT!* for short.

Prove that *HSoDT!* is NP-complete, thereby guaranteeing (assuming $P \neq NP$) that there isn't a simple strategy for racking up high scores on your friends' game.

18. ($*$) Suppose you're consulting for one of the many companies in New Jersey that designs communication networks, and they come to you with the following problem. They're studying a specific n-node communication network, modeled as a directed graph $G = (V, E)$. For reasons of fault-tolerance they want to divide up G into as many virtual "domains" as possible: a *domain* in G is a set X of nodes, of size at least 2, so that for each pair of nodes $u, v \in X$ there are directed paths from u to v and v to u that are contained entirely in X.

    Show that the following Domain Decomposition problem is NP-complete. Given a directed graph $G = (V, E)$ and a number k, can V be *partitioned* into at least k sets, each of which is a domain?

19. You and a friend have been trekking through various far-off parts of the world, and have accumulated a big pile of souvenirs. At the time you weren't really thinking about which of these you were planning to keep, and which your friend was going to keep, but now the time has come to divide everything up.

    Here's a way you could go about doing this. Suppose there are n objects, labeled $1, 2, \ldots, n$, and object i has an agreed-upon *value* x_i. (We could think of this, for example, as a monetary re-sale value; the case in which you and your friend don't agree on the value is something we won't pursue here.) One reasonable way to divide things would be to look for a *partition* of the objects into two sets, so that the total value of the objects in each set is the same.

    This suggests solving the following *Number Partitioning* problem. You are given positive integers $x_1, \ldots, x_n$; you want to decide whether the numbers can be partitioned into two sets S_1 and S_2 with the same sum:

$$\sum_{x_i \in S_1} x_i = \sum_{x_j \in S_2} x_j.$$

Show that *Number Partitioning* is NP-complete.

20. Consider the following problem. You are given positive integers $x_1, \ldots, x_n$, and numbers k and B. You want to know whether it is possible to *partition* the numbers $\{x_i\}$ into k sets $S_1, \ldots, S_k$ so that the squared sums of the sets add up to at most B:

$$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B.$$

Show that this problem is NP-complete.

21. You are given a graph $G = (V, E)$ with weights w_e on its edges $e \in E$. The weights can be negative or positive. The ZERO-WEIGHT-CYCLE problem is to decide if there is a simple cycle in G so that the sum of the edge weights on this cycle is exactly 0. Prove that this problem is NP-complete.

22. Consider the following version of the Steiner tree problem, which we'll refer to as GRAPHICAL STEINER TREE. You are given an undirected graph $G = (V, E)$, a set $X \subseteq V$ of vertices, and a number k. You want to decide whether there is a set $F \subseteq E$ of at most k edges so that in the graph (V, F), X belongs to a single connected component.

Show that GRAPHICAL STEINER TREE is NP-complete.

23. The *Directed Disjoint Paths* problem is defined as follows. We are given a directed graph G and k pairs of nodes $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$. The problem is to decide whether there exist node-disjoint paths $P_1, P_2, \ldots, P_k$ so that P_i goes from s_i to t_i.

Show that *Directed Disjoint Paths* is NP-complete.

24. Given a directed graph G a *cycle cover* is a set of node-disjoint cycles so that each node of G belongs to a cycle. The *Cycle Cover* problem asks whether a given directed graph has a cycle cover.

(a) Show that the cycle cover problem can be solved in polynomial time. (Hint: use bipartite matching.)

(b)$(*)$ Suppose we require each cycle to have at most 3 edges. Show that determining whether a graph G has such a cycle cover is NP-complete.

25. Given two undirected graphs G and H, we say that H is a *subgraph* of G if we can obtain a copy of the graph H by starting from G and deleting some of the vertices and edges.

The SUBGRAPH CONTAINMENT problem is defined as follows: given G and H, is H a subgraph of G? Show that SUBGRAPH CONTAINMENT is NP-complete.

26. Let $G = (V, E)$ be a graph with n nodes and m edges. If $E' \subset E$, we say that the *coherence* of E' is equal to n minus the number of connected components of the graph (V, E'). (So a graph with no edges has coherence 0, while a connected graph has coherence $n - 1$.)

    We are given subsets $E_1, E_2, \ldots, E_m$ of E. Our goal is to choose k of these sets — $E_{i_1}, \ldots, E_{i_k}$ — so that the *coherence* of the union $E_{i_1} \cup \cdots \cup E_{i_k}$ is as large as possible.

    Given this input, and a bound C, prove that it is NP-complete to decide whether it is possible to achieve a coherence of at least C.

27. Let $G = (V, E)$ be a bipartite graph; suppose its nodes are partitioned into sets X and Y so that each edge has one end in X and the other in Y. We define an (a, b)-*skeleton* of G to be a set of edges $E' \subseteq E$ so that *at most* a nodes in X are incident to an edge in E', and *at least* b nodes in Y are incident to an edge in E'.

    Show that, given a bipartite graph G and numbers a and b, it is NP-complete to decide whether G has an (a, b)-skeleton.

28. After a few too many days immersed in the popular entrepreneurial self-help book *Mine Your Own Business*, you've come to the realization that you need to upgrade your office computing system. This, however, leads to some tricky problems ...

    In configuring your new system, there are k *components* that must be selected: the operating system, the text editing software, the e-mail program, and so forth; each is a separate component. For the j^{th} component of the system, you have a set A_j of options; and a *configuration* of the system consists of a selection of one element from each of the sets of options $A_1, A_2, \ldots, A_k$.

    Now, the trouble arises because certain pairs of options from different sets may not be compatible: we say that option $x_i \in A_i$ and option $x_j \in A_j$ form an *incompatible pair* if a single system cannot contain them both. (For example, Linux (as an option for the operating system) and Word (as an option for the text-editing software) form an incompatible pair.) We say that a configuration of the system is *fully compatible* if it consists of elements $x_1 \in A_1, x_2 \in A_2, \ldots x_k \in A_k$ such that none of the pairs (x_i, x_j) is an incompatible pair.

    We can now define the *Fully Compatible Configuration* (FCC) problem. An instance of FCC consists of the sets of options $A_1, A_2, \ldots, A_k$, and a set P of *incompatible pairs* (x, y), where x and y are elements of different sets of options. The problem is to decide whether there exists a fully compatible configuration: a selection of an element from each option set so that no pair of selected elements belongs to the set P.

Example. Suppose $k = 3$, and the sets A_1, A_2, A_3 denote options for the operating system, the text editing software, and the e-mail program respectively. We have

$$A_1 = \{\texttt{Linux}, \texttt{Windows NT}\},$$

$$A_2 = \{\texttt{emacs}, \texttt{Word}\},$$

$$A_3 = \{\texttt{Outlook}, \texttt{Eudora}, \texttt{rmail}\},$$

with the set of incompatible pairs equal to

$$P = \{(\texttt{Linux}, \texttt{Word}), (\texttt{Linux}, \texttt{Outlook}), (\texttt{Word}, \texttt{rmail})\}.$$

Then the answer to the decision problem in this instance of FCC is "yes" — for example, the choices $\texttt{Linux} \in A_1, \texttt{emacs} \in A_2, \texttt{rmail} \in A_3$ is a fully compatible configuration according to the above definitions.

Prove that *Fully Compatible Configuration* is NP-complete.

29. For functions $g_1, \ldots, g_\ell$, we define the function $\max(g_1, \ldots, g_\ell)$ via

$$[\max(g_1, \ldots, g_\ell)](x) = \max(g_1(x), \ldots, g_\ell(x)).$$

Consider the following problem. You are given n piecewise linear, continuous functions $f_1, \ldots, f_n$ defined over the interval $[0, t]$ for some integer t. You are also given an integer B. You want to decide: do there exist k of the functions $f_{i_1}, \ldots, f_{i_k}$ so that

$$\int_0^t [\max(f_{i_1}, \ldots, f_{i_k})](x)\, dx \geq B?$$

Prove that this problem is NP-complete.

30. Consider the following *Broadcast Time* problem. We are given a directed graph $G = (V, E)$, with a designated node $r \in V$ and a designated set of "target nodes" $T \subseteq V - \{r\}$. Each node v has a *switching time* s_v, which is a positive integer.

At time 0, the node r generates a message that it would like every node in T to receive. To accomplish this, we want to find a scheme whereby r tells some of its neighbors (in sequence), who in turn tell some of their neighbors, and so on, until every node in T has received the message. More formally, a *broadcast scheme* is defined as follows. Node r may send a copy of the message to one of its neighbors at time 0; this neighbor will receive the message at time 1. In general, at time $t \geq 0$, any node v that has already received the message may send a copy of the message to one of its neighbors, provided it has not sent a copy of the message in any of the time steps $t - s_v + 1, t - s_v + 2, \ldots, t - 1$. (This reflects the role of the *switching time*; v needs a pause of $s_v - 1$ steps between

successive sendings of the message. Note that if $s_v = 1$, then no restriction is imposed by this.)

The *completion time* of the broadcast scheme is the minimum time t by which all nodes in T have received the message. The *Broadcast Time* problem is the following: given the input described above, and a bound b, is there a broadcast scheme with completion time at most b?

Prove that *Broadcast Time* is NP-complete.

Example. Suppose we have a directed graph $G = (V, E)$, with $V = \{r, a, b, c\}$; edges (r, a), (a, b), (r, c); the set $T = \{b, c\}$; and switching time $s_v = 2$ for each $v \in V$. Then a broadcast scheme with minimum completion time would be as follows: r sends the message to a at time 0; a sends the message to b at time 1; r sends the message to c at time 2; and the scheme completes at time 3 when c receives the message. (Note that a can send the message as soon as it receives it at time 1, since this is its first sending of the message; but r cannot send the message at time 1 since $s_r = 2$ and it sent the message at time 0.)

31. Suppose that someone gives you a black-box algorithm $\mathcal{A}$ that takes an undirected graph $G = (V, E)$, and a number k, and behaves as follows.

    - If G is not connected, it simply returns "G is not connected."
    - If G is connected and has an independent set of size at least k, it returns "yes."
    - If G is connected and does not have an independent set of size at least k, it returns "no."

    Suppose that the algorithm $\mathcal{A}$ runs in time polynomial in the size of G and k.

    Show how, using calls to $\mathcal{A}$, you could then solve the Indepedent Set problem in polynomial time: Given an arbitrary undirected graph G, and a number k, does G contain an independent set of size at least k?

32. Given a set of finite binary strings $S = \{s_1, \ldots, s_k\}$, we say that a string u is a *concatenation* over S if it is equal to $s_{i_1} s_{i_2} \cdots s_{i_t}$ for some indices $i_1, \ldots, i_t \in \{1, \ldots, k\}$.

    A friend of yours is considering the following problem: Given two sets of finite binary strings, $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$, does there exist any string u so that u is both a concatenation over A and a concatenation over B?

    You friend announces, "At least the problem is in NP, since I would just have to exhibit such a string u in order to prove the answer is 'yes.' " You point out — politely, of course — that this is a completely inadequate explanation; how do we know that the

shortest such string u doesn't have length exponential in the size of the input, in which case it would not be a polynomial-size certificate?

However, it turns out that this claim can be turned into a proof of membership in NP. Specifically, prove the following statement:

> If there is a string u that is a concatenation over both A and B, then there is such a string whose length is bounded by a polynomial in the sum of the lengths of the strings in $A \cup B$.

33. Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites — contractual issues prevent them from saying which ones — and they've come across the following Strategic Advertising problem.

A company comes to them with the map of a Web site, which we'll model as a directed graph $G = (V, E)$. The company also provides a set of t *trails* typically followed by users of the site; we'll model these trails as directed paths $P_1, P_2, \ldots, P_t$ in the graph G. (I.e. each P_i is a path in G.)

The company wants WebExodus to answer the following question for them: Given G, the paths $\{P_i\}$, and a number k, is it possible to place advertisements on at most k of the nodes in G, so that each path P_i includes at least one node containing an advertisement? We'll call this the Strategic Advertising problem, with input G, $\{P_i : i = 1, \ldots, t\}$, and k.

Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple ...

(a) Prove that Strategic Advertising is NP-complete.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input G, $\{P_i\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes, *or*
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most a polynomial number of steps, together with at most a polynomial number of calls to the algorithm $\mathcal{S}$.

34. Consider the following problem. You are managing a communication network, modeled by a directed graph $G = (V, E)$. There are c *users* who are interested in making use of this network. User i (for each $i = 1, 2, \ldots, c$) issues a *request* to reserve a specific path P_i in G on which to transmit data.

    You are interested in accepting as many of these path requests as possible, subject to the following restriction: if you accept both P_i and P_j, then P_i and P_j cannot share any nodes.

    Thus, the *Path Selection* problem asks: given a directed graph $G = (V, E)$, a set of requests $P_1, P_2, \ldots, P_c$ — each of which must be a path in G — and a number k, is it possible to select at least k of the paths so that no two of the selected paths share any nodes?

    Prove that *Path Selection* is NP-complete.

35. A store trying to analyze the behavior of its customers will often maintain a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer i.

    Here's a tiny example of such an array A.

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

    One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is *diverse* if no two of the of the customers in S have ever bought the same product. (I.e. for each product, at most one of the customers in S has ever bought it.) A diverse set of customers can be useful, for example, as a target pool for market research.

    We can now define the DIVERSE SUBSET problem as following: given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?

    Show that DIVERSE SUBSET is NP-complete.

36. A *combinatorial auction* is a particular mechanism developed by economists for selling a collection of items to a collection of potential buyers. (The Federal Communications Commission has studied this type of auction for assigning stations on the radio spectrum to broadcasting companies.)

    Here's a simple type of combinatorial auction. The are n items for sale, labeled $I_1, \ldots, I_n$. Each item is indivisible, and can only be sold to one person. Now, m different people place *bids*: the i^{th} bid specifies a subset S_i of the items, and an *offering price* x_i that the offeror of this bid is willing to pay for the items in the set S_i, as a single unit. (We'll represent this bid as the pair (S_i, x_i).)

    An auctioneer now looks at the set of all m bids; she chooses to *accept* some of these bids, and *reject* the others. The offeror of an accepted bid gets to take all the items in S_i. Thus, the rule is that no two accepted bids can specify sets that contain a common item, since this would involve giving the same item to two different people.

    The auctioneer collects the sum of the offering prices of all accepted bids. (Note that this is a "one-shot" auction; there is no opportunity to place further bids.) The auctioneer's goal is to collect as much money as possible.

    Thus, the *Combinatorial Auction* problem asks: Given items $I_1, \ldots, I_n$, bids $(S_1, x_1), \ldots, (S_m, x_m)$, and a bound B, is there a collection of bids that the auctioneer can accept so as to collect an amount of money that is at least B?

    Example. Suppose an auctioneer decides to use this method to sell some excess computer equipment. There are four items labeled "`PC`," "`monitor`," "`printer`", and "`modem`"; and three people place bids. Define

    $$S_1 = \{\texttt{PC}, \texttt{monitor}\}, S_2 = \{\texttt{PC}, \texttt{printer}\}, S_3 = \{\texttt{monitor}, \texttt{printer}, \texttt{modem}\}$$

    and

    $$x_1 = x_2 = x_3 = 1.$$

    The bids are $(S_1, x_1), (S_2, x_2), (S_3, x_3)$ and the bound B is equal to 2.

    Then the answer to this instance is "no": The auctioneer can accept at most one of the bids (since any two bids have a desired item in common), and this results in a total monetary value of only 1.

    Prove that *Combinatorial Auction* is NP-complete.

Chapter 8

PSPACE

8.1 PSPACE

Throughout the course, one of the main issues has been the notion of *time* as a computational resource. It was this notion that formed the basis for adopting *polynomial time* as our working definition of efficiency; and, implicitly, it underlies the distinction between $\mathcal{P}$ and $\mathcal{NP}$. To some extent, we have also been concerned with the *space* — i.e. memory — requirements of algorithms. In this chapter, we investigate a class of problems defined by treating space as the fundamental computational resource. In the process, we develop a natural class of problems that appear to be even harder than $\mathcal{NP}$ and co$-\mathcal{NP}$.

The basic class we study is PSPACE, the set of all problems that can be solved by an algorithm that uses an amount of *space* that is polynomial in the size of the input.

We begin by considering the relationship of PSPACE to classes of problems we have considered earlier. First of all, in polynomial time, an algorithm can consume only a polynomial amount of space; so we can say

(8.1) $\mathcal{P} \subseteq$ PSPACE.

But PSPACE is much broader than this. Consider, for example, an algorithm that just counts from 0 to $2^n - 1$ in base-2 notation. It simply needs to implement an n-bit counter, which it maintains in exactly the same way one increments an odometer in a car. Thus, this algorithm runs for an exponential amount of time, and then halts; in the process, it has used only a polynomial amount of space. Although this algorithm is not doing anything particularly interesting, it illustrates an important principle: Space can be re-used during a computation in ways that time, by definition, cannot.

Here is a more striking application of this principle.

(8.2) *There is an algorithm that solves* 3-SAT *using only a polynomial amount of space.*

Proof. We simply use a brute-force algorithm that tries all possible truth assignments; for each assignment, it plugs it into the set of clauses and sees if it satisfies them. The key is to implement this all in polynomial space.

To do this, we increment an n-bit counter from 0 to $2^n - 1$ just as described above. The values in counter correspond to truth assignments in the following way: when the counter holds a value q, we interpret it as a truth assignment ν that sets x_i to be the value of the i^{th} bit of q.

Thus, we devote a polynomial amount of space to enumerating all possible truth assignment ν. For each truth assignment, we need only polynomial space to plug it into the set of clauses and see if it satisfies them. If it does satisfy the clauses, we can stop the algorithm immediately. If it doesn't, we delete the intermediate work involved in this "plugging in" operation, and *re-use* this space for the next truth assignment. Thus, we spend only polynomial space cumulatively in checking all truth assignments; this completes the bound on the algorithm's space requirements. ■

Since *3-SAT* is an NP-complete problem, (8.2) has a significant consequence.

(8.3) $\mathcal{NP} \subseteq$ PSPACE.

Proof. Consider an arbitrary problem Y in $\mathcal{NP}$. Since $Y \leq_P$ *3-SAT*, there is an algorithm that solves Y using a polynomial number of steps plus a polynomial number of calls to a black box for *3-SAT*. Using the algorithm in (8.2) to implement this black box, we obtain an algorithm for Y that uses only polynomial space. ■

Just as with the class $\mathcal{P}$, a problem X is in PSPACE if and only if its complementary problem $\overline{X}$ is in PSPACE as well. Thus, we can conclude that co$-\mathcal{NP} \subseteq$ PSPACE. We draw what is known about the relationships among these classes of problems in Figure 8.1.

Given that PSPACE is an enormously large class of problems, containing both $\mathcal{NP}$ and co$-\mathcal{NP}$, it is very likely that it contains problems that cannot be solved in polynomial time. But despite this widespread belief, it has, amazingly, not been proven that $\mathcal{P} \neq$ PSPACE. Nevertheless, the nearly universal conjecture is that PSPACE contains problems that are not even in $\mathcal{NP}$ or co$-\mathcal{NP}$.

8.2 Some Hard Problems in PSPACE

We now survey some natural examples of problems in PSPACE that are not known — and not believed — to belong to $\mathcal{NP}$ or co$-\mathcal{NP}$.

As was the case with $\mathcal{NP}$, we can try understanding the structure of PSPACE by looking for *complete problems* — the hardest problems in the class. We will say that a problem X is

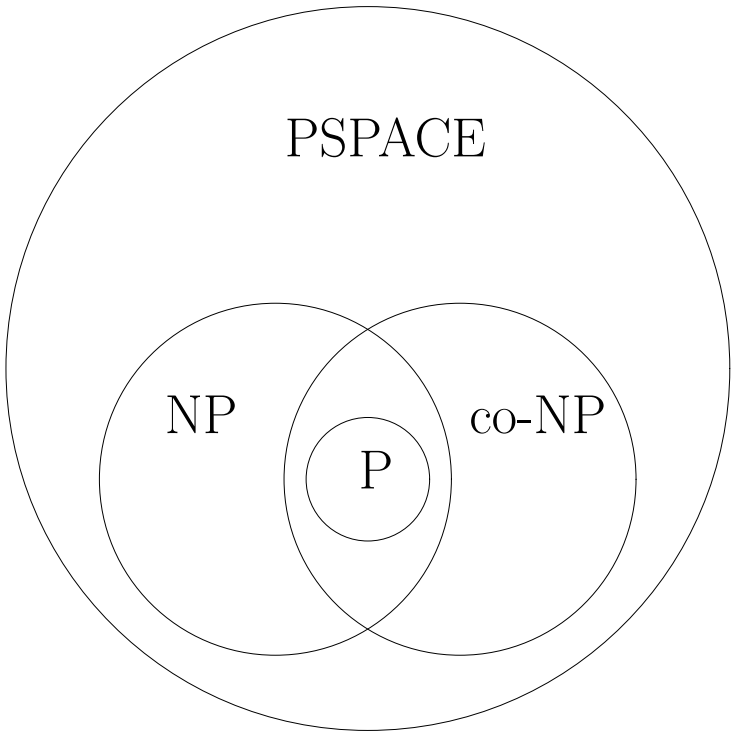


Figure 8.1: The relationships among various classes of problems.

PSPACE-*complete* if (i) it belongs to PSPACE; and (ii) for all problems Y in PSPACE, we have $Y \leq_P X$. Note that we continue to use $\leq_P$ as our notion of reduction.

It turns out — analogously to the case of $\mathcal{NP}$ — that a wide range of natural problems are PSPACE-complete. Indeed, a number of the basic problems in artificial intelligence are PSPACE-complete, and we describe three genres of these here.

1. Planning. *Planning problems* seek to capture, in a clean way, the problem of interacting with a complex environment to achieve a desired set of goals. A common toy example used in AI is the *fifteen-puzzle* — a 4×4 grid with fifteen movable tiles labeled $1, 2, \ldots, 15$ and a single *hole*. One wishes to push the tiles around so that the numbers end up in ascending order. In the same vein, one could picture peg-jumping solitaire games or Rubik's Cube.

If we wanted to search for a solution to such a problem, we could enumerate the conditions we are trying to achieve, and the set of allowable *operators* that can we apply to achieve these conditions. We model the environment by a set $\mathcal{C} = \{C_1, \ldots, C_n\}$ of *conditions*: a given state of the world is specified by the subset of the conditions that currently hold. We interact with the environment through a set $\{\mathcal{O}_1, \ldots, \mathcal{O}_k\}$ of *operators*. Each operator $\mathcal{O}_i$ is specified by a *prerequisite list*, containing a set of conditions that must hold for $\mathcal{O}_i$ to be invoked; an *add list*, containing a set of conditions that will become true after $\mathcal{O}_i$ is invoked;

and a *delete list*, containing a set of conditions that will cease to hold after $\mathcal{O}_i$ is invoked.

The problem we face is the following. Given a set $\mathcal{C}_0$ of *initial conditions*, and a set $\mathcal{C}^*$ of *goal conditions*, is it possible to apply a sequence of operators beginning with $\mathcal{C}_0$ so that we reach a situation in which precisely the conditions in $\mathcal{C}^*$ (and no others) hold? We will call this an instance of the *Planning* problem.

2. Quantification. We have seen, in the *3-SAT* problem, some of the difficulty in determining whether a set of disjunctive clauses can be simultaneously satisfied. When we add quantifiers to such a formula, the problem appears to become even more difficult.

Let $\Phi(x_1, \ldots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each C_i is a disjunction of three terms. (In other words, it is an instance of *3-SAT*.) Assume for simplicity that n is an odd number, and suppose we ask

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

That is, we wish to know whether there is a choice for x_1, so that for both choices of x_2, there is a choice for x_3 — and so on — so that Φ is satisfied. We will refer to this decision problem as *Quantified 3-SAT* (or, briefly, *QSAT*).

Problems of this type arise naturally as a form of *contingency planning* — we wish to know whether there is a decision we can make (the choice of x_1) so that for all possible responses (the choice of x_2) there is a decision we can make (the choice of x_3), and so forth ...

The original *3-SAT* problem, by way of comparison, simply asked

$$\exists x_1 \exists x_2 \cdots \exists x_{n-2} \exists x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

In other words, in *3-SAT* it was sufficient to look for a single setting of the Boolean variables.

3. Games. In 1996 and 1997, world chess champion Garry Kasparov was billed by the media as the defender of the human race, as he faced IBM's program *Deep Blue* in two chess matches. We needn't look further than this picture to convince ourselves that computational game-playing is one of the most visible successes of contemporary artificial intelligence.

A large number of combinatorial two-player games fit naturally into the following framework. Players alternate moves, and the first one to achieve a specific goal wins. Moreover, there is often a natural, polynomial, upper bound on the maximum possible length of a game.

The *Competitive Facility Location* problem that we introduced at the beginning of the course fits naturally within this framework. We are given a graph G, with a non-negative

value b_i attached to each node i. Two players alternately select nodes of G, so that the set of selected nodes at all times forms an independent set. Player 2 wins if she ultimately selects a set of nodes of total value at least B, for a given bound B; Player 1 wins if he prevents this from happening. The question is, given the graph G and the bound B, is there a strategy by which Player 1 can force a win?

We now discuss how to solve all of these problems in polynomial space. As we will see this will be trickier — in one case, a lot trickier — than the (simple) task we faced in showing that problems like *3-SAT* and *Independent Set* belong to $\mathcal{NP}$.

8.3 Solving Problems in Polynomial Space

Quantifiers and Games

First let's show that *QSAT* can be solved in polynomial space. As was the case with *3-SAT*, the idea will be to run a brute-force algorithm that re-uses space carefully as the computation proceeds.

Here is the basic brute-force approach. To deal with the first quantifier $\exists x_1$, we consider both possible values for x_1 in sequence. We first set $x_1 = 0$ and see, recursively, whether the remaining portion of the formula evaluates to 1. We then set $x_1 = 1$ and see, recursively, whether the remaining portion of the formula evaluates to 1. The full formula evaluates to 1 if and only if *either* of these recursive calls yields a 1 — that's simply the definition of the $\exists$ quantifier.

If we were to save all the work done in both these recursive call, our space usage $S(n)$ would look something like

$$S(n) = 2S(n-1) + \text{poly}(n),$$

where $\text{poly}(n)$ is a polynomial function, since each level of the recursion generates two new calls. This would result in an exponential bound, which is too large.

Fortunately, we can perform a simple optimization that greatly reduces the space usage: When we're done with the case $x_1 = 0$, all we really need to save is the single bit that represents the outcome of the recursive call; we can throw away all the other intermediate work. This is another example of "re-use" — we're re-using the space from the computation for $x_1 = 0$ in order to compute the case $x_1 = 1$.

Here is a compact description of the algorithm.

```
If the first quantifier is ∃x_i then
  Set x_i = 0 and recursively evaluate quantified expression
                    over remaining variables.
  Save the result (0 or 1) and delete all other intermediate work.
```

```
  Set x_i = 1 and recursively evaluate quantified expression
                    over remaining variables.
  If either outcome yielded an evaluation of 1, then
     return 1;
  Else return 0
  Endif
If the first quantifier is ∀x_i then
  Set x_i = 0 and recursively evaluate quantified expression
                    over remaining variables.
  Save the result (0 or 1) and delete all other intermediate work.
  Set x_i = 1 and recursively evaluate quantified expression
                    over remaining variables.
  If both outcomes yielded an evaluation of 1, then
     return 1;
  Else return 0
  Endif
Endif
```

Since the recursive calls for the cases $x_1 = 0$ and $x_1 = 1$ over-write the same space, our space requirement $S(n)$ for an n-variable problem is simply a polynomial in n plus the space requirement for one recursive call on an $(n-1)$-variable problem:

$$S(n) \leq \text{poly}(n) + S(n-1). \tag{8.1}$$

Thus, $S(n) \leq n \cdot \text{poly}(n)$, which is a polynomial bound.

We can determine which player has a forced win in a game such as *Competitive Facility Location* by a very similar type of algorithm.

Suppose Player 1 moves first. We consider all of his possible moves in sequence. For each of these moves, we see who has a forced win in the resulting game, with Player 2 moving first. If Player 1 has a forced win in any of them, then Player 1 has a forced win from the initial position. The crucial point, as in the *QSAT* algorithm, is that we can re-use the space from one candidate move to the next — we need only store the single bit representing the outcome. In this way, we only consume a polynomial amount of space plus the space requirement for one recursive call on a graph with fewer nodes. As before, we get the recurrence

$$S(n) \leq \text{poly}(n) + S(n-1).$$

Planning

Now we consider how to solve the basic *Planning* problem in polynomial space. The issues here will look quite different, and it will turn out to be a more difficult task.

Recall that we have a set of *conditions* $\mathcal{C} = \{C_1, \ldots, C_n\}$ and a set of *operators* $\{\mathcal{O}_1, \ldots, \mathcal{O}_k\}$. Each operator $\mathcal{O}_i$ has a *prerequisite list* P_i, an *add list* A_i, and a *delete list* D_i. Note that $\mathcal{O}_i$ can still be applied even if conditions other than those in P_i are present; and it does not affect conditions that are not in A_i or D_i.

We define a *configuration* to be a subset $\mathcal{C}' \subseteq \mathcal{C}$; the state of the planning problem at any given time can be identified with a unique configuration $\mathcal{C}'$ consisting precisely of the conditions that hold at that time. For an initial configuration $\mathcal{C}_0$ and a goal configuration $\mathcal{C}^*$, we wish to determine whether there is a sequence of operators that will take us from $\mathcal{C}_0$ to $\mathcal{C}^*$.

We can view our planning instance in terms of a giant, implicitly defined, directed graph $\mathcal{G}$. There is a node of $\mathcal{G}$ for each of the 2^n possible configurations; and there is an edge of $\mathcal{G}$ from configuration $\mathcal{C}'$ to configuration $\mathcal{C}''$ if in one step, one of the operators can convert $\mathcal{C}'$ to $\mathcal{C}''$. In terms of this graph, the planning problem has a very natural formulation: Is there a path in $\mathcal{G}$ from $\mathcal{C}_0$ to $\mathcal{C}^*$? Such a path corresponds precisely to a sequence of operators leading from $\mathcal{C}_0$ to $\mathcal{C}^*$.

The first thing to appreciate is that there need not always be a short solutions to instances of *Planning*, i.e., there need not always be short path in $\mathcal{G}$. This should not be so surprising, since $\mathcal{G}$ has an exponential number of nodes. But we must be careful in applying this intuition, since $\mathcal{G}$ has a special structure: it is defined very compactly in terms of the n conditions and k operators.

(8.4) *There are instances of the* Planning *problem with n conditions and k operators for which there exists a solution, but the shortest solution has length $2^n - 1$.*

Proof. We give a simple example of such an instance; it essentially encodes the task of incrementing an n-bit counter from the all-zeroes state to the all-ones state.

- We have conditions $C_1, C_2, \ldots, C_n$.
- We have operators $\mathcal{O}_i$ for $i = 1, 2, \ldots, n$.
- $\mathcal{O}_1$ has no pre-requisites or delete list; it simply adds C_1.
- For $i > 1$, $\mathcal{O}_i$ requires C_j for all $j < i$ as pre-requisites. When invoked, it adds C_i and deletes C_j for all $j < i$.

Now we ask: is there a sequence of operators that will take us from $\mathcal{C}_0 = \phi$ to $\mathcal{C}^* = \{C_1, C_2, \ldots, C_n\}$?

We claim the following, by induction on i:

> (†) From any configuration that does not contain C_j for any $j \leq i$, there exists a sequence of operators that reaches a configuration containing C_j for all $j \leq i$; but any such sequence has at least $2^i - 1$ steps.

This is clearly true for $i = 1$. For larger i, here's one solution.

- By induction, achieve conditions $\{C_{i-1}, \ldots, C_1\}$ using operators $\mathcal{O}_1, \ldots, \mathcal{O}_{i-1}$.
- Now invoke operator $\mathcal{O}_i$, adding C_i but deleting everything else.
- Again, by induction, achieve conditions $\{C_{i-1}, \ldots, C_1\}$ using operators $\mathcal{O}_1, \ldots, \mathcal{O}_{i-1}$. Note that condition C_i is preserved throughout this process.

Now we take care of the other part of the inductive step — that *any* such sequence requires at least $2^i - 1$ steps. So consider the first moment when C_i is added. At this step, $C_{i-1}, \ldots, C_1$ must have been present, and by induction, this must have taken at least $2^{i-1} - 1$ steps. C_i can only be added by $\mathcal{O}_i$, which deletes all C_j for $j < i$. Now we have to achieve conditions $\{C_{i-1}, \ldots, C_1\}$ again; this will take another $2^{i-1} - 1$ steps, by induction, for a total of at least $2(2^{i-1} - 1) + 1 = 2^i - 1$ steps.

The overall bound now follows by applying this claim with $i = n$. ■

Of course, if every "yes" instance of *Planning* had a polynomial-length solution, then *Planning* would be in $\mathcal{NP}$ — we could just exhibit the solution. But (8.4) shows that the shortest solution is not necessarily a good certificate for a *Planning* instance, since it can have a length that is exponential in the input size.

However, (8.4) describes essentially the worst case, for we have the following matching upper bound. The graph $\mathcal{G}$ has 2^n nodes, and if there is a path from $\mathcal{C}_0$ to $\mathcal{C}^*$, then the shortest such path does not visit any node more than once. As a result, the shortest path can take at most $2^n - 1$ steps after leaving $\mathcal{C}_0$.

(8.5) *If a* Planning *instance with n conditions has a solution, then it has one using at most $2^n - 1$ steps.*

Now we return to the main issue: how can we solve an arbitrary *Planning* instance using only polynomial space? Here is a brute-force algorithm to solve the *Planning* instance. We build the graph $\mathcal{G}$, and use any graph connectivity algorithm — depth-first search or breadth-first search — to decide whether there is a path from $\mathcal{C}_0$ to $\mathcal{C}^*$.

But this algorithm is too "brute-force" for our purposes; it takes exponential space to construct the graph $\mathcal{G}$. We could try an approach in which we never actually build $\mathcal{G}$, and just simulate the behavior of depth-first search or breadth-first search on it. But this too is not feasible. Depth-first search crucially requires us to maintain a list of all the nodes in the current path we are exploring — and this can grow to exponential size. Breadth-first requires a list of all nodes in the current "frontier" of the search — and this too can grow to exponential size.

We seem stuck. Our problem is transparently equivalent to finding a path in $\mathcal{G}$, and all the standard path-finding algorithms we know are too lavish in their use of space. Could there really be a fundamentally different path-finding algorithm out there?

In fact, there is. The basic idea, proposed by Savitch in 1970, is a clever use of the divide-and-conquer principle. It subsequently inspired the trick for reducing the space requirements in the sequence alignment problem; so the overall approach may remind you of what we discussed there. Our plan, as before, is to find a clever way to re-use space, admittedly at the expense of increasing the time spent. Neither depth-first search nor breadth-first search are nearly aggressive enough in their re-use of space; they need to maintain a large history at all times. We need a way to solve half the problem, throw away almost all the intermediate work, and then solve the other half of the problem.

The key is a procedure that we will call $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$ — it determines whether there is a sequence of operators, *consisting of at most L steps*, that leads from configuration $\mathcal{C}_1$ to configuration $\mathcal{C}_2$. So our initial problem is to determine the result (yes or no) of $\texttt{Path}(\mathcal{C}_0, \mathcal{C}^*, 2^n)$. Breadth-first search can be viewed as the following dynamic programming implementation of this procedure: to determine $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$ we first determine all $\mathcal{C}'$ for which $\texttt{Path}(\mathcal{C}_1, \mathcal{C}', L-1)$ holds; we then see, for each such $\mathcal{C}'$, whether any operator leads directly from $\mathcal{C}'$ to $\mathcal{C}_2$.

This indicates some of the wastefulness — in terms of space — that breadth-first search entails. We are generating a huge number of intermediate configurations just to reduce the parameter L by one. More effective would be to try determining whether there is any configuration $\mathcal{C}'$ that could serve as the *midpoint* of a path from $\mathcal{C}_1$ to $\mathcal{C}_2$. We could first generate all possible midpoints $\mathcal{C}'$. For each $\mathcal{C}'$, we then check recursively whether we can get from $\mathcal{C}_1$ to $\mathcal{C}'$ in at most $L/2$ steps; and also whether we can get from $\mathcal{C}'$ to $\mathcal{C}_2$ in at most $L/2$ steps. This involves two recursive calls, but we only care about the yes/no outcome of each; other than this, we can re-use space from one to the next.

Does this really reduce the space usage to a polynomial amount? We first write down the procedure carefully, and then analyze it. We will assume that L is a power of 2.

```
Path(C1, C2, L)
  If L = 1 then
    If there is an operator O converting C1 to C2 then
      return ‘‘yes’’
    Else
      return ‘‘no’’
    Endif
  Else (L > 1)
    Enumerate all configurations C′ using an n-bit counter.
    For each C′ do the following:
       Compute x = Path(C1, C′, L/2)
```

```
      Delete all intermediate work, saving only the return value x.
      Compute y = Path(C', C_2, L/2)
      Delete all intermediate work, saving only the return value y.
      If both x and y are equal to ``yes'', then return ``yes.''
    Endfor
    If ``yes'' was not returned for any of the C' then
      Return ``no''
    Endif
  Endif
```

Again, note that this procedure solves a generalization of our original question, which simply asked for $\texttt{Path}(\mathcal{C}_0, \mathcal{C}^*, 2^n)$. This does mean, however, that we should remember to view L as an exponentially large parameter: $\log L = n$.

The following claim therefore implies that *Planning* can be solved in polynomial space.

(8.6) $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$ *returns "yes" if and only if there is a sequence of operators of length at most L leading from $\mathcal{C}_1$ to $\mathcal{C}_2$. Its space usage is polynomial in n, k, and* $\log L$.

Proof. The correctness follows by induction on L. It clearly holds when $L = 1$, since all operators are considered explicitly. Now consider a larger value of L. If there is a sequence of operators from $\mathcal{C}_1$ to $\mathcal{C}_2$, of length $L' \leq L$, then there is a configuration $\mathcal{C}'$ that occurs at position $\lfloor L'/2 \rfloor$ in this sequence. By induction, $\texttt{Path}(\mathcal{C}_1, \mathcal{C}', L/2)$ and $\texttt{Path}(\mathcal{C}', \mathcal{C}_2, L/2)$ will both return "yes", and so $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$ will return "yes." Conversely, if there is a configuration $\mathcal{C}'$ so that $\texttt{Path}(\mathcal{C}_1, \mathcal{C}', L/2)$ and $\texttt{Path}(\mathcal{C}', \mathcal{C}_2, L/2)$ both return "yes," then the induction hypothesis implies that there exist corresponding sequences of operators; concatenating these two sequences, we obtain a sequence of operators from $\mathcal{C}_1$ to $\mathcal{C}_2$ of length at most L.

Now we consider the space requirements. Aside from the space spent inside recursive calls, each invocation of `Path` involves an amount of space polynomial in n, k, and $\log L$. But at any given point in time, only a single recursive call is active, and the intermediate work from all other recursive calls has been deleted. Thus, for a polynomial function p, the space requirement $S(n, k, L)$ satisfies the recurrence

$$\begin{aligned} S(n, k, L) &\leq p(n, k, \log L) + S(n, k, L/2). \\ S(n, k, 1) &\leq p(n, k, 1). \end{aligned}$$

Unwinding the recurrence for $\log L$ levels, we obtain the bound $S(n, k, L) = O(\log L \cdot p(n, k, \log L))$, which is a polynomial in n, k, and $\log L$. ■

If dynamic programming has an opposite, this is it. Back when we were solving problems by dynamic programming, the fundamental principle was: save all the intermediate work, so

that you don't have to re-compute it. Now that conserving space is our goal, we have just the opposite priorities: throw away all the intermediate work, since it's just taking up space and it can always be re-computed.

As we saw when we designed the space-efficient sequence alignment algorithm, the sensible strategy often lies somewhere in between, motivated by these two approaches: throw away some of the intermediate work, but not so much that you blow up the running time.

8.4 Proving Problems PSPACE-complete

When we studied $\mathcal{NP}$, we had to prove a *first* problem NP-complete directly from the definition of $\mathcal{NP}$. After Cook and Levin did this for *Satisfiability* in 1971, many other NP-complete problems could follow by reduction.

A similar sequence of events followed for PSPACE, shortly after the results for $\mathcal{NP}$. The natural analogue of *Satisfiability* here is played by *QSAT*, and one can prove directly from the definitions that it is PSPACE-complete. This basic PSPACE-complete problem can then serve as a good "root" from which to discover other PSPACE-complete problems. By strict analogy with the case of $\mathcal{NP}$, it's easy to see from the definition that if a problem Y is PSPACE-complete, and a problem X in PSPACE has the property that $Y \leq_P X$, then X is PSPACE-complete as well.

Our goal in this section is to prove the PSPACE-completeness of problems from the two other genres we have been considering: Planning and Games. We first consider *Competitive Facility Location*, then *Planning*; we describe a way to reduce *QSAT* to each of these problems.

Games. It is actually not surprising at all that there should be a close relation between quantifiers and games. Indeed, we could have equivalently defined *QSAT* as the problem of deciding whether the first player has a forced win in the following *Competitive 3-SAT* game. Suppose we fix a formula $\Phi(x_1, \ldots, x_n)$ consisting, as in *QSAT*, of a conjunction of length-3 clauses. Two players alternate turns picking values for variables: the first player picks the value of x_1, then the second player picks the value of x_2, then the first player picks the value of x_3, and so on. We will say that the first player wins if $\Phi(x_1, \ldots, x_n)$ ends up evaluating to 1, and the second player wins if it ends up evaluating to 0.

When does the first player have a forced win in this game? (I.e. when does our instance of *Competitive 3-SAT* have a "yes" answer?) Precisely when there is a choice for x_1 so that for all choices of x_2 there is a choice for x_3 so that ... and so on, resulting in $\Phi(x_1, \ldots, x_n)$ evaluating to 1. That is, the first player has a forced win if and only if (assuming n is an odd number)

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n).$$

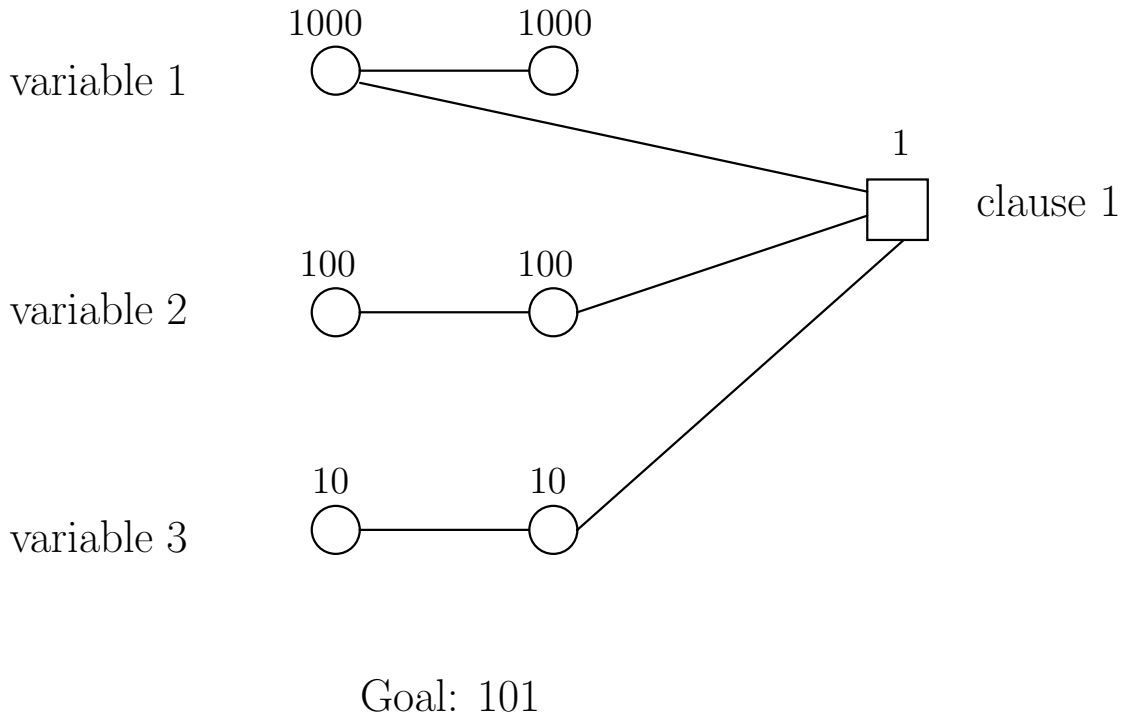


Figure 8.2: The reduction from *Competitive 3-SAT* to *Competitive Facility Location*.

In other words, our *Competitive 3-SAT* game is directly equivalent to the instance of *QSAT* defined by the same Boolean formula Φ, and so we have proved the following

(8.7) QSAT $\leq_P$ Competitive 3-SAT.

This result moves us nicely into the world of games. We now establish the PSPACE-completeness of *Competitive Facility Location* by proving the following.

(8.8) Competitive 3-SAT $\leq_P$ Competitive Facility Location.

Proof. We are given an instance of *Competitive 3-SAT*, defined by a formula Φ. Φ is the conjunction of clauses

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k;$$

each C_j has length 3, and can be written $C_j = t_{j1} \vee t_{j2} \vee t_{j3}$. As before, we will assume that there is an odd number n of variables. We will also assume, quite naturally, that no clause contains both a term and its negation — after all, such a clause would be automatically satisfied by any truth assignment. We must show how to encode this Boolean structure in the graph that underlies *Competitive Facility Location*.

We can picture the instance of *Competitive 3-SAT* as follows. The players alternately select values in a truth assignment, beginning and ending with Player 1; at the end, Player

2 has won if she can select a clause C_j in which none of the terms has been set to 1. Player 1 has won if Player 2 cannot do this.

It is this notion that we would like to encode in an instance of *Competitive Facility Location* — that the players alternately make a fixed number of moves, in a highly constrained fashion, and then there's a final chance by Player 2 to win the whole thing. But in its general form, *Competitive Facility Location* looks much more wide-open than this — whereas the players in *Competitive 3-SAT* must set one variable at a time, in order, the players in *Competitive Facility Location* can jump all over the graph, choosing nodes wherever they want.

Our fundamental trick, then, will be to use the *values* b_i on the nodes to tightly constrain where the players can move, under any "reasonable" strategy. In other words, we will set things up so that if the either of the players deviates from a particular narrow course, he or she will lose instantly.

As with our complicated NP-completeness reductions, the construction will have gadgets to represent assignments to the variables, and further gadgets to represent the clauses. Here is how we encode the variables. For each variable x_i, we define two nodes v_i, v_i' in the graph G, and include an edge (v_i, v_i'). The selection of v_i will represent setting $x_i = 1$; v_i' will represent $x_i = 0$. The constraint that the chosen variables must form an independent set naturally prevents both v_i and v_i' from being chosen. At this point, we do not define any other edges.

How do we get the players to set the variables in order — first x_1, then x_2, and so forth? We place values on v_1 and v_1' so high that Player 1 will lose instantly if he does not choose them. We place somewhat lower values on v_2 and v_2', and continue in this way. Specifically, for a value $c \geq k+2$ we define the node values b_{v_i} and $b_{v_i'}$ to be c^{1+n-i}. We define the bound that Player 2 is trying to achieve to be

$$B = c^{n-1} + c^{n-3} + \cdots + c^2 + 1.$$

Let's pause, before worrying about the clauses, to consider the game played on this graph. In the opening move of the game, Player 1 must select one of v_1 or v_1' (thereby obliterating the other one) — for if not, then Player 2 will immediately select one of them on her next move, winning instantly. Similarly, in the second move of the game, Player 2 must select one of v_2 or v_2'. For otherwise, Player 1 will select one on his next move; and then, even if Player 2 acquired all the remaining nodes in the graph, she would not be able to meet the bound B. Continuing by induction in this way, we see that to avoid an immediate loss, the player making the i^{th} move must select one of v_i or v_i'. Note that our choice of node values has achieved precisely what we wanted — the players must set the variables in order. And what is the outcome on this graph? Player 2 ends up with a total of value of $c^{n-1} + c^{n-3} + \cdots + c^2 = B - 1$ — she has lost by 1 unit!

We now complete the analogy with *Competitive 3-SAT* by giving Player 2 one final move on which she can try to win. For each clause C_j, we define a node c_j with value $b_{c_j} = 1$ and an edge associated with each of its terms as follows. If $t = x_i$, we add an edge (c_j, v_i); if $t = \overline{x_i}$, we add an edge (c_j, v_i'). In other words, we join c_j to the node that represents the term t.

This now defines the full graph G. We can verify that, because their values are so small, the addition of the clause nodes did not change the property that the players will begin by selecting the variable nodes $\{v_i, v_i'\}$ in the correct order. However, after this is done, Player 2 will win if and only if she can select a clause node c_j that is not adjacent to any selected variable node — in other words, if and only the truth assignment defined alternately by the players failed to satisfy some clause.

Thus, Player 2 can win the *Competitive Facility Location* instance we have defined if and only if she can win the original *Competitive 3-SAT* instance. The reduction is complete. ■

Planning. We now establish the PSPACE-completeness of *Planning* by proving the following result.

(8.9) QSAT $\leq_P$ Planning.

Proof. We are given an instance of *Competitive 3-SAT*, defined by a formula Φ. Φ is the conjunction of clauses

$$K_1 \wedge K_2 \wedge \cdots \wedge K_k;$$

each K_j has length 3, and can be written $K_j = t_{j1} \vee t_{j2} \vee t_{j3}$.[1] As before, we will assume that there is an odd number n of variables.

The key to the reduction to consider the *Planning* formalism as a very abstract type of "programming language." We will not simply express the *QSAT* instance in this language — we will actually construct a *Planning* instance to describe the full execution of the polynomial-space *QSAT* algorithm that we covered earlier in this chapter. To this end, we will have conditions that model the variable assignments this algorithm makes, and the control flow as it is running. Indeed, if one looks at the reduction to follow from a sufficiently high level, it does look like a schematic piece of code that implements the *QSAT* algorithm; and our *Planning* instance is to decide whether this piece of code will output 0 or 1, after its exponentially long execution.

For $i = 1, 2, \ldots, n$, we have a condition L_i, indicating that the execution of the algorithm is currently at "level i" of the recursion, and hence working on variable x_i. For $i = 1, 2, \ldots, n$, and $b = 0, 1$, we have the following conditions:

- $C[x_i = b : *]$, indicating that the algorithm is currently in the middle of the recursive call with $x_i = b$.

[1] We use "K" to denote the clauses, since "C" will reserved for the conditions in the *Planning* problem.

- $C[x_i = b :?]$, indicating that it has not yet started the recursive call with $x_i = b$.
- For $c = 0, 1$, conditions $C[x_i = b : c]$ indicating that the recursive call with $x_i = b$ recursive call has returned with result c.

Finally, we have a condition $C[1]$ to indicate that the full algorithm has returned 1 and $C[0]$ to indicate that the full algorithm has returned 0. The initial configuration is $\mathcal{C}_0 = \{L_1, C[x_1 = 0 : *], C[x_1 = 1 :?]\}$. The goal configuration is $\mathcal{C}^* = \{C[0]\}$.

Here are the operators that model the control flow, enforcing these interpretations of the conditions. We also give the intended meaning of each.

- Let $i < n$. Given L_i and $C[x_i = b : *]$, we can add $\{L_{i+1}, C[x_{i+1} = 0 : *], C[x_{i+1} = 1 :?]\}$ and delete L_i. *(x_i has just been set to b, so we move to level $i + 1$.)*
- Let $i > 1$ be odd, so the associated quantifier is $\exists x_i$.
  - Given L_i, $C[x_i = 0 : 0]$, and $C[x_i = 1 :?]$, we can add $C[x_i = 1 : *]$ and delete $C[x_i = 1 :?]$. *(The option $x_i = 0$ has returned 0, so we move on to the case $x_i = 1$.)*
  - Given L_i, $C[x_i = b : 1]$, and $C[x_{i-1} = b' : *]$, we can add $\{L_{i-1}, C[x_{i-1} = b' : 1]\}$ and delete all conditions associated with levels $j \geq i$. *(The option $x_i = b$ has returned 1, so the $\exists$ evaluates to 1 and we pop up to level $i - 1$.)*
  - Given L_i, $C[x_i = 0 : 0]$, $C[x_i = 1 : 0]$, and $C[x_{i-1} = b' : *]$, we can add $\{L_{i-1}, C[x_{i-1} = b' : 0]\}$ and delete all conditions associated with levels $j \geq i$. *(Both settings of x_i have returned 0, so the $\exists$ evaluates to 0 and we pop up to level $i - 1$.)*
- Let i be even, so the associated quantifier is $\forall x_i$.
  - Given L_i, $C[x_i = 0 : 1]$, and $C[x_i = 1 :?]$, we can add $C[x_i = 1 : *]$ and delete $C[x_i = 1 :?]$. *(The option $x_i = 1$ has returned 0, so we move on to the case $x_i = 1$.)*
  - Given L_i, $C[x_i = 0 : 1]$, $C[x_i = 1 : 1]$, and $C[x_{i-1} = b' : *]$, we can add $\{L_{i-1}, C[x_{i-1} = b' : 1]\}$ and delete all conditions associated with levels $j \geq i$. *(Both settings of x_i have returned 1, so the $\forall$ evaluates to 1 and we pop up to level $i - 1$.)*
  - Given L_i, $C[x_i = b : 0]$, and $C[x_{i-1} = b' : *]$, we can add $\{L_{i-1}, C[x_{i-1} = b' : 0]\}$ and delete all conditions associated with levels $j \geq i$. *(The option $x_i = b$ has returned 0, so the $\forall$ evaluates to 0 and we pop up to level $i - 1$.)*
- Finally, here are some operators that apply to the first level.

- Given L_1, $C[x_1 = 0 : 0]$, and $C[x_1 = 1 :?]$, we can add $C[x_1 = 1 : *]$ and delete $C[x_1 = 1 :?]$. *(The option $x_i = 0$ has returned 0, so we move on to the case $x_i = 1$.)*
- Given L_1, $C[x_1 = b : 1]$, and $C[x_{i-1} = b' : *]$, we can add $C[1]$ and delete all other conditions. *(The option $x_i = b$ has returned 1, so the $\exists$ evaluates to 1 and we have reached the goal configuration.)*
- Given L_1, $C[x_1 = 0 : 0]$, and $C[x_1 = 1 : 0]$, we can add $C[0]$ and delete all other conditions. *(Both settings of x_i have returned 0, so the $\exists$ evaluates to 0 and we declare failure. No operators will be valid from this configuration.)*

We now have to model level n, when the recursion "bottoms out" with a full assignment to all variables. Here, we must plug this assignment into the clauses and see if they can all be satisfied. Thus, for each clause K_j, we have a condition $C[K_j]$ that will express the notion that the current assignment satisfies K_j. These conditions will be associated with level n, so they are deleted when the other level-n conditions are deleted. Here are the operators that model the evaluation of the clauses.

- Given L_n and $C[x_i = 1 : *]$, where x_i is a term of K_j, we can add $C[K_j]$. *(The current assignment satisfies K_j.)*
- Given L_n and $C[x_i = 0 : *]$, where $\overline{x_i}$ is a term of K_j, we can add $C[K_j]$. *(The current assignment satisfies K_j.)*
- Given L_n, $C[x_{n-1} = b : *]$, and $C[K_1], C[K_2], \ldots, C[K_k]$, we can add $\{L_{n-1}, C[x_{n-1} = b : 1]\}$ and delete all conditions associated with level n. *(All clauses are satisfied, so this level evaluates to 1 and we pop up to level $n - 1$.)*
- Given L_n and $C[x_n = 0 : *]$, we can add $\{C[x_n = 0 : 0], C[x_n = 1 : *]\}$ and delete $\{C[x_n = 0 : *], C[K_1], \ldots, C[K_k]\}$. *(We cannot satisfy all clauses with $x_n = 0$, so we set $x_n = 1$.)*
- Given L_n, $C[x_n = 0 : 0]$, $C[x_n = 1 : *]$, and $C[x_{n-1} = b : *]$, we can add $\{L_{n-1}, C[x_{n-1} = b : 0]\}$ and delete all conditions associated with level n. *(We cannot satisfy all clauses with $x_n = 0$ or with $x_n = 1$, so this level returns 0 and we pop up to level $n - 1$.)*

These operators directly model the execution of the polynomial-space *QSAT* algorithm; it is easy to check that if this algorithm has an execution that returns 1, then the corresponding sequence of operators leads to the goal configuration. Conversely, one can show inductively that each condition has its intended meaning relative to the algorithm, though we will not write out the full details of this here. As a result, if the configuration $\mathcal{C}^* = \{C[1]\}$ is reachable, then the *QSAT* algorithm must have an execution that returns 1.

Since we have seen earlier that the *QSAT* algorithm correctly evaluates the formula Φ, we see that Φ evaluates to 1 if and only if the goal configuration $\mathcal{C}^*$ is reachable from $\mathcal{C}_0$. ■

8.5 Exercises

1. *(Based on a problem proposed by Maverick Woo and Ryan Williams.)* Let's consider a special case of *Quantified 3-SAT* in which the underlying Boolean formula is *monotone* (in the sense of Problem Set 5). Specifically, let $\Phi(x_1, \ldots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each C_i is a disjunction of three terms. We say Φ is *monotone* if each term in each clause consists of a non-negated variable — i.e. each term is equal to x_i, for some i, rather than $\overline{x_i}$.

We define *Monotone QSAT* to be the decision problem

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

where the formula Φ is monotone.

Do one of the following two things: (a) prove that *Monotone QSAT* is PSPACE-complete; or (b) give an algorithm to solve arbitrary instances of *Monotone QSAT* that runs in time polynomial in n. (Note that in (b), the goal is polynomial *time*, not just polynomial space.)

2. *Self-avoiding walks* are a basic object of study in the area of statistical physics; they can be defined as follows. Let $\mathcal{L}$ denote the set of all points in $\mathbf{R}^2$ with integer coordinates. A *self-avoiding walk* W of length n is a sequence of points $(p_1, p_2, \ldots, p_n) \in \mathcal{L}^n$ so that

   (i) $p_1 = (0, 0)$. *(The walk starts the origin.)*

   (ii) No two of the points are equal. *(The walk "avoids" itself.)*

   (iii) For each $i = 1, 2, \ldots, n-1$, the points p_i and p_{i+1} are at distance 1 from each other. *(The walk moves between neighboring points in $\mathcal{L}$.)*

   Self-avoiding walks (in both two and three dimensions) are used in physical chemistry as a simple geometric model for the possible conformations of long-chain polymer molecules. Such molecules can be viewed as a flexible chain of beads that flop around in solution, adopting different geometric layouts, and self-avoiding walks are a simple combinatorial abstraction for these layouts.

   A famous unsolved problem in this area is the following. For a natural number $n \geq 1$, let $A(n)$ denote the number of distinct self-avoiding walks of length n. Note that we

view walks as *sequences* of points rather than sets; so two walks can be distinct even if they pass through the same set of points, provided that they do so in different orders. (Formally, the walks $(p_1, p_2, \ldots, p_n)$ and $(q_1, q_2, \ldots, q_n)$ are distinct if there is some i $(1 \leq i \leq n)$ for which $p_i \neq q_i$.) See the figure below for an example. In polymer models based on self-avoiding walks, $A(n)$ is directly related to the *entropy* of a chain molecule, and so it appears in theories concering the rates of certain metabolic and organic synthesis reactions.

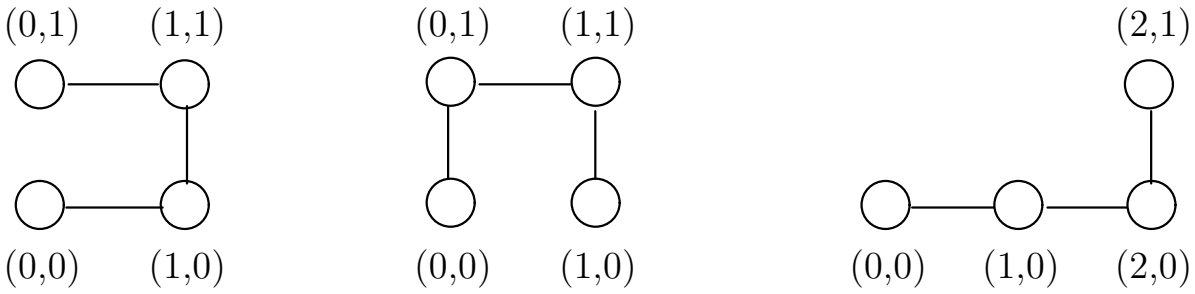


Figure 8.3: Three distinct self-avoiding walks of length 4. Note that although walks (a) and (b) involve the same set of points, they are considered different walks because they pass through them in a different order.

Despite its importance, no simple formula is known for the value $A(n)$. Indeed, no algorithm is known for computing $A(n)$ that runs in time polynomial in n.

(a) Show that $A(n) \geq 2^{n-1}$ for all natural numbers $n \geq 1$.

(b) Give an algorithm that takes a number n as input, and outputs $A(n)$ as a number in binary notation, using space (i.e. memory) that is polynomial in n.

(Thus the running time of your algorithm can be exponential, as long as its space usage is polynomial. Note also that "polynomial" here means "polynomial in n," not "polynomial in $\log n$". Indeed, by part (a), we see that it will take at least $n - 1$ bits to write the value of $A(n)$, so clearly $n - 1$ is a lower bound on the amount of space you need for producing a correct answer.)

Chapter 9

Extending the Limits of Tractability

Although we started the course by studying a number of techniques for solving problems efficiently, we've been looking for a while at classes of problems — $\mathcal{NP}$-complete and PSPACE-complete problems — for which no efficient solution is believed to exist. And because of the insights we've gained this way, we've implicitly arrived at a two-pronged approach to dealing with new computational problems we encounter: try for a while to develop an efficient algorithm; and if this fails, then try to prove it $\mathcal{NP}$-complete (or even PSPACE-complete). Assuming one of the two approaches works out, you end up either with a solution to the problem (an algorithm), or a potent "reason" for its difficulty: it is as hard as many of the famous problems in computer science.

Unfortunately, this strategy will only get you so far. If this is a problem that someone really wants your help in solving, they won't be particularly satisfied with the resolution that it's NP-hard[1] and so they should give up on it. They'll still want a solution that's as good as possible, even if it's not the exact, or optimal, answer. For example, in the independent set problem, even if we can't find the largest independent set in a graph, it's still natural to want to compute for as much time as we have available, and output as large an independent set as we can find.

The next few topics in the course will be focused on different aspects of this notion. In subsequent lectures, we'll look at algorithms that provide approximate answers with guaranteed error bounds in polynomial time; we'll also consider *local search heuristics* that are often very effective in practice, even when we are not able to establish any provable guarantees about their behavior.

But to start, we explore some situations in one can exactly solve instances of NP-complete problems with reasonable efficiency. How do these situations arise? The point is to recall the basic message of NP-completeness — the worst-case instances of these problems are very difficult, and not likely to be solvable in polynomial time. On a *particular* instance,

[1] We use the term *NP-hard* to mean, "At least as hard as an NP-complete problem." We avoid referring to optimization problems as "NP-complete," since technically this term applies only to decision problems.

however, it's possible that we are not really in the "worst case" — maybe, in fact, the instance we're looking at has some special structure that makes our task easier. Thus, the crux of this chapter is to look at situations in which it is possible to quantify some precise senses in which an instance may be easier than the worst case, and to take advantage of these situations when they occur.

We'll look at this principle in two concrete settings. First we'll consider the *Vertex Cover* problem, in which there are two natural "size" parameters for a problem instance — the size of the graph, and the size of the vertex cover being sought. The NP-completeness of *Vertex Cover* suggests that we will have to be exponential in (at least) one of these parameters; but judiciously choosing which one can have an enormous affect on the running time.

Next, we'll explore the idea that many NP-complete graph problems become polynomial-time solvable if we require the input to be a tree — this is a concrete illustration of the way in which an input with "special structure" can help us avoid many of the difficulties can make the worst case intractable. Armed with this insight, one can generalize the notion of a tree to a more general class of graphs — those with small *tree-width* — and show that many NP-complete problems are tractable on this more general class as well.

Having said this, we should stress that our basic point remains the same as it has always been — *Exponential algorithms scale very badly.* The current chapter represents ways off staving off this problem that can be effective in various settings, but there is clearly no way around it in the fully general case. This will motivate our focus on approximation algorithms and local search in subsequent chapters.

9.1 Finding Small Vertex Covers

Let us briefly recall the *Vertex Cover* problem, which we saw when we covered NP-completeness. Given a graph $G = (V, E)$ and an integer k, we would like to find a vertex cover of size at most k, i.e., a set of nodes $S \subseteq V$ of size $|S| \leq k$, such that every edge $e \in E$ has at least one end in S.

Like many NP-complete decision problems, *Vertex Cover* comes with two parameters: n, the number of nodes in the graph, and k, the allowable size of a vertex cover. This means that the range of possible running time bounds is much richer, since it involves the interplay between these two parameters.

First of all, we notice that if k is a fixed constant (e.g. $k = 2$ or $k = 3$) then we can solve *Vertex Cover* in polynomial time: we simply try all subsets of V of size k, and see whether any of them constitutes a vertex cover. There are $\binom{n}{k}$ subsets, and each takes time $O(kn)$ to check whether each is a vertex cover, for a total time of $O(kn\binom{n}{k}) = O(kn^{k+1})$. So from this we see that the intractability of *Vertex Cover* only sets in for real once k grows as a function of n.

However, even for moderately small values of k, a running time of $O(kn^{k+1})$ is quite impractical. For example, if $n = 1000$ and $k = 10$, then on our computer executing a million high-level instructions per second, it would take at least 10^{24} seconds to decide if G has a k-node vertex cover — this is several orders of magnitude larger than the age of the universe. And this was for a small value of k, where the problem was supposed to be more tractable! It's natural to start asking whether we can do something that is practically viable when k is a small constant.

It turns out that a much better algorithm can be developed, with a running time bound of $O(2^k \cdot kn)$. There are two things worth noticing about this. First, plugging in $n = 1000$ and $k = 10$, we see that our computer should be able to execute the algorithm in a few seconds. Second, we see that as k grows, the running time is still increasing very rapidly; it's simply that the exponential dependence on k has been moved out of the exponent on n and into a separate function. From a practical point of view, this is much more appealing.

Designing the Improved Algorithm. As a first observation, we notice that if a graph has a small vertex cover, then it cannot have very many edges. Recall that the *degree* of a node is the number of edges that are incident to it.

(9.1) *If $G = (V, E)$ has n nodes, the maximum degree of any node is at most d, and there is a vertex cover of size at most k, then G has at most kd edges.*

Proof. Let S be a vertex cover in G of size $k' \leq k$. Every edge in G has at least one end in S; but each node in S can cover at most d edges. Thus, there can be at most $k'd \leq kd$ edges in G. ■

Since the degree of any node in a graph can be at most $n - 1$, we have the following simple consequence of (9.1).

(9.2) *If $G = (V, E)$ has n nodes and a vertex cover of size k, then G has at most $k(n - 1) \leq kn$ edges.*

So as a first step in our algorithm, we can check if G contains more than kn edges; if it does, then we know that the answer to the decision problem — is there a vertex cover of size at most k? — is "no." Having done this, we will assume that G contains at most kn edges.

The idea behind the algorithm is conceptually very clean. We begin by considering any edge $e = (u, v)$ in G. In any k-node vertex cover S of G, one of u or v must belong to S. Suppose that u belongs to such a vertex cover S. Then if we delete u and all its incident edges, it must be possible to cover the remaining edges by at most $k - 1$ nodes. That is, defining $G-\{u\}$ to be the graph obtained by deleting u and all its incident edges, there must be a vertex cover of size at most $k - 1$ in $G-\{u\}$. Similarly, if v belongs to S, this would imply there is a vertex cover of size at most $k - 1$ in $G-\{v\}$.

Here is a concrete way to formulate this idea.

(9.3) *Let $e = (u, v)$ be any edge of G. G has a vertex cover of size at most k if and only if at least one of the graphs $G-\{u\}$ and $G-\{v\}$ has a vertex cover of size at most $k-1$.*

Proof. First, suppose G has a vertex cover S of size at most k. Then S contains at least one of u or v; suppose that it contains u. Then the set $S-\{u\}$ must cover all edges that have neither end equal to u. Therefore $S-\{u\}$ is a vertex cover of size at most $k-1$ for the graph $G-\{u\}$.

Conversely suppose that one of $G-\{u\}$ and $G-\{v\}$ has a vertex cover of size at most $k-1$ — suppose in particular that $G-\{u\}$ has such a vertex cover T. Then the set $T \cup \{u\}$ covers all edges in G, so it is a vertex cover for G of size at most k. ■

(9.3) directly establishes the correctness of the following recursive algorithm for deciding whether G has a k-node vertex cover.

```
To search for a k-node vertex cover in G:
  If G contains no edges then the empty set is a vertex cover.
  If G contains ≥ k|V| edges then it has no k-node vertex cover
  Else let e = (u, v) be an edge of G.
    Recursively check if either of G−{u} or G−{v}
                has a vertex cover of size k − 1.
    If neither of them does, then G has no k-node vertex cover
    Else, one of them (say G−{u}) has a (k−1)-node vertex cover T.
       In this case, T ∪ {u} is a k-node vertex cover of G.
    Endif
  Endif
```

Now we bound the running time of this algorithm. Intuitively, we are searching a "tree of possibilities"; we can picture the recursive execution of the algorithm as giving rise to a tree, in which each node corresponds to a different recursive call. A node corresponding to a recursive call with parameter k has, as children, two nodes corresponding to recursive calls with parameter $k-1$. Thus the tree has a total of at most 2^{k+1} nodes. In each recursive call, we spend $O(kn)$ time.

Thus, we can prove the following.

(9.4) *The running time of the* Vertex Cover *algorithm on an n-node graph, with parameter k, is $O(2^k \cdot kn)$.*

We could also prove this by a recurrence as follows. If $T(n, k)$ denotes the running time on an n-node graph with parameter k, then $T(\cdot, \cdot)$ satisfies the following recurrence, for some absolute constant c:

$$\begin{aligned} T(n, 1) &\leq ckn, \\ T(n, k) &\leq 2T(n, k-1) + ckn, \end{aligned}$$

By induction on $k \geq 1$, it is easy to prove that $T(n,k) \leq c \cdot 2^k kn$. Indeed, if this is true for $k-1$, then

$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1) + ckn \\ &\leq 2c \cdot 2^{k-1}(k-1)n + ckn \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn \\ &\leq c \cdot 2^k kn \end{aligned}$$

In summary, this algorithm is a powerful improvement on the simple brute-force approach. However, no exponential algorithm can scale well for very long, and that includes this one. Suppose we want to know whether there is a vertex cover with at most 30 nodes, rather than 10; then on the same machine as before, our algorithm will take several years to terminate.

9.2 Solving NP-hard Problem on Trees

In the previous lecture we designed an algorithm for the *Vertex Cover* problem that works well when the size of the desired vertex cover is not too large. We saw that finding a relatively small vertex cover is much easier than the *Vertex Cover* problem in its full generality.

In this lecture we consider special cases of NP-complete graph problems with a different flavor — not when the natural "size" parameters are small, but when the input graph is structurally "simple." Perhaps the simplest types of graphs are *trees* — recall that an undirected graph is a tree if it is connected and has no cycles. Not only are they structurally easy to understand, but it has been found that many NP-hard graph problems can be solved efficiently when the underlying graph is a tree. Here we will see why this is true for variants of the *Independent Set* problem; however, it is important to keep in mind that this principle is quite general, and we could equally well have considered other NP-complete graph problems on trees.

First, we will see that the *Independent Set* problem itself can be solved by a greedy algorithm on a tree. Then we will consider the a generalization called the *Maximum-Weight Independent Set* problem, in which nodes have weight, and we seek an independent set of maximum weight. We'll see that the *Maximum-Weight Independent Set* problem can be solved on trees via dynamic programming.

A Greedy Algorithm for *Independent Set* on Trees

The starting point of our algorithm is to consider the way a solution looks from the perspective of a single edge — this is a variant on an idea from the previous lecture. Specifically, consider an edge $e = (u,v)$ in G. In any independent set S of G, at most one of u or v can

belong to S. We'd like to find an edge e for which we can greedily decide which of the two ends to place in our independent set.

For this we exploit a crucial property of trees that we've seen earlier in the course: every tree has at least one *leaf* — a node of degree 1. Consider a leaf v, and let (u, v) be the unique edge incident to v. How might we "greedily" evaluate the relative benefits of including u or v in our independent set S? If we include v, the only other node that is directly "blocked" from joining the independent set is u. If we include u, it blocks not only v but all the other nodes joined to u as well. So if we're trying to maximize the size of the independent set, it seems that including v should be better than — or at least as good as — including u.

(9.5) *If $T = (V, E)$ is a tree and v is a leaf of the tree, then there exists a maximum-size independent set that contains v.*

Proof. Consider a maximum-size independent set S, and let $e = (u, v)$ be the unique edge incident to node v. Clearly at least one of u or v is in S; for if neither is present, then we could add v to S, thereby increasing its size. Now, if $v \in S$, then we are done; and if $u \in S$, then we can obtain another independent set S' of the same size by deleting u from S and inserting v. ■

We will use (9.5) repeatedly to identify and delete nodes that can be placed in the independent set. As we do this deletion, the tree T may become disconnected. So to handle things more cleanly, we actually describe our algorithm for the more general case in which the underlying graph is a *forest* — a graph in which each connected component is a tree. We can view the problem of finding a maximum-size independent set for a forest as really being the same as the problem for trees: an optimal solution for a forest is simply the union of optimal solutions for each tree component, and we can still use (9.5) to think about the problem in any component.

Specifically, suppose we have a forest F; then (9.5) allows us to make our first decision in the following greedy way. Consider again an edge $e = (u, v)$, where v is a leaf. We will include node v in our independent set S, and not include node u. Given this decision, we can delete the node v — since it's already been included — and the node u — since it cannot be included — and obtain a smaller forest. We continue recursively on this smaller forest to get a solution.

```
To find a maximum-size independent set in a forest F:
  Let S be the independent set to be constructed (initially empty).
  While F has at least one edge
     Let e = (u, v) be an edge of F such that v is a leaf.
     Add v to S
     Delete from F nodes u and v, and all edges incident to them.
  Endwhile
  Return S
```

(9.6) *The above algorithm finds a maximum-size independent set in forests (and hence in trees as well).*

Although (9.5) was a very simple fact, it really represents an application of one of the design principles for greedy algorithms that we saw earlier in the course: an *exchange argument.* In particular, the crux of our independent set algorithm is the observation that any solution not containing a particular leaf can be "transformed" into a solution that is just as good and contains the leaf.

To implement this algorithm so it runs quickly, we need to maintain the current forest F in a way that allows us to find an edge incident to a leaf efficiently. It is not hard to implement this algorithm in linear time: we need to maintain the forest in a way that allows us to do on one iteration of the `While` loop in time proportional to the number of edges deleted when u and v are removed.

The greedy algorithm on more general graphs. The greedy algorithm specified above is not guaranteed to work on general graphs, because we cannot be guaranteed to find a leaf in every iteration. However, (9.5) *does* apply to any graph: if we have an arbitrary graph G with an edge (u, v) such that u is the only neighbor of v, then it's always safe to put v in the independent set, delete u and v, and iterate on the smaller graph.

So if, by repeatedly deleting degree-1 nodes and their neighbors, we're able to eliminate the entire graph, then we're guaranteed to have found an independent set of maximum size — even if the original graph was not a tree. And even if we don't manage to eliminate the whole graph, we may succeed in running a few iterations of the algorithm in succession, thereby shrinking the size of the graph and making other approaches more tractable. Thus, our greedy algorithm is a useful heuristic to try "opportunistically" on arbitrary graphs, in the hope of making progress toward finding a large independent set.

Maximum-Weight Independent Set on Trees

Next we turn to the more complex problem of finding a maximum-weight independent set. As before we assume that our graph is a tree $T = (V, E)$. Now we also have a positive *weight* w_v associated with each node $v \in V$. The *Maximum-Weight Independent Set* problem is to find an independent set S in the graph $T = (V, E)$ so that the total weight $\sum_{v \in S} w_v$ is as large as possible.

First we try the idea we used before to build a greedy solution for the case without weights. Consider an edge $e = (u, v)$, such that v is a leaf. Including v blocks fewer nodes from entering the independent set; so if the weight of v is at least as large as the weight of u, then we can indeed make a greedy decision just as we did in the case without weights. However, if $w_v < w_u$ we face a dilemma: we acquire more weight by including u, but we

retain more options down the road if we include v. There seems to be no easy way to resolve this locally, without considering the rest of the graph. However, there is still something we can say. If node u has many neighbors $v_1, v_2, \ldots$ that are leaves, than we should make the same decision for all of them: Once we decide not to include u in the independent set, we may as well go ahead an include all its adjacent leaves. So for the sub-tree consisting of u and its adjacent leaves, we really have only two "reasonable" solutions to consider: including u, or including all the leaves.

We will use the ideas above to design a polynomial time algorithm using dynamic programming. As we recall, dynamic programming allows us to record a few different solutions, build these up through a sequence of sub-problems, and thereby decide only at the end which of these possibilities will be used in the final solution.

The first thing to decide for a dynamic programming algorithm is what our sub-problems will be. For *Maximum-Weight Independent Set*, we will construct sub-problems by *rooting* the tree T at an arbitrary node r; recall that this is the operation of "orienting" all the tree's edges away from r. Specifically, for any node $u \neq r$, the parent $p(u)$ of u is the node adjacent to u along the path to the root r. The other neighbors of u are the children, and we will use $children(u)$ to denote the set of children of u. The node u and all its descendents form a sub-tree T_u whose root is u.

We will base our sub-problems on these sub-trees T_u. The tree T_r is our original problem. If $u \neq r$ is a leaf, then T_u consists of a single node. For a node u all of whose children are leaves, we observe that T_u is the kind of sub-tree discussed above.

To solve the problem by dynamic programming, we will start at the leaves and gradually work our way up the tree. For a node u we want to solve the sub-problem associated with the tree T_u after we have solved the sub-problems for all its children. To get a maximum-weight independent set S for the tree T_u we will consider two cases: we either include the node u in S or we do not. If we include u, then we cannot include any of its children; if we do not include u, then we have the freedom to include or omit these children. This suggests that we should define two sub-problems for each sub-tree T_u: $OPT_{in}(u)$ will denote the maximum weight of an independent set of T_u that includes u, and $OPT_{out}(u)$ will denote the maximum weight of an independent set of T_u that does not include u.

Now that we have our sub-problems, it is not hard to see how to compute these values recursively. For a leaf $u \neq r$ we have that $OPT_{out}(u) = 0$ and $OPT_{in}(u) = w_u$. For all other nodes u we get the following recurrence that defines $OPT_{out}(u)$ and $OPT_{in}(u)$ using the values for u's children.

(9.7) *For a node u that has children, the following recurrence defines the values of the sub-problems:*

- $OPT_{in}(u) = w_u + \sum_{v \in children(u)} OPT_{out}(v)$

- $OPT_{out}(u) = \sum_{v \in children(u)} \max(OPT_{out}(v), OPT_{in}(v))$.

Using this recurrence, we get a dynamic programming algorithm by building up the optimal solutions over larger and larger sub-trees. We define arrays $M_{out}[u]$ and $M_{in}[u]$, which hold the values $OPT_{out}(u)$ and $OPT_{in}(u)$ respectively. For building up solutions, we need to process all the children of a node before we process the node itself; in the terminology of tree traversal, we visit the nodes in *post-order*.

```
To find a maximum-weight independent set of a tree T:
    Root the tree at a node r.
    For all nodes u of T in post-order
        If u is a leaf then set the values:
            M_out[u] = 0
            M_in[u] = w_u
        Else set the values:
            M_out[u] = Σ_{v∈children(u)} max(M_out[u], M_in[u])
            M_in[u] = w_u + Σ_{v∈children(u)} M_out[u].
        Endif
    Endfor
    Return max(M_out[r], M_in[r]).
```

This gives us the value of the maximum-weight independent set. Now, as is standard in the dynamic programming algorithms we've seen before, it's easy to recover an actual independent set of maximum weight by recording the decision we make for each node, and then tracing back through these decisions to decide which nodes should be included. Thus we have

(9.8) *The above algorithm finds a maximum-weight independent set in trees in linear time.*

9.3 Tree Decompositions of Graphs

Although *Maximum-Weight Independent Set* is NP-complete, we now have a polynomial-time algorithm that solves the problem on trees. When you find yourself in this situation — able to solve an NP-complete problem in a reasonably natural special case — it's worth asking why the approach doesn't work in general. Our algorithm in the previous section was indeed taking advantage of a special property of (rooted) trees: once we decide whether or not to include a node u in the independent set, the sub-problems in each sub-tree become completely separated; we can solve each as though the others did not exist. We don't

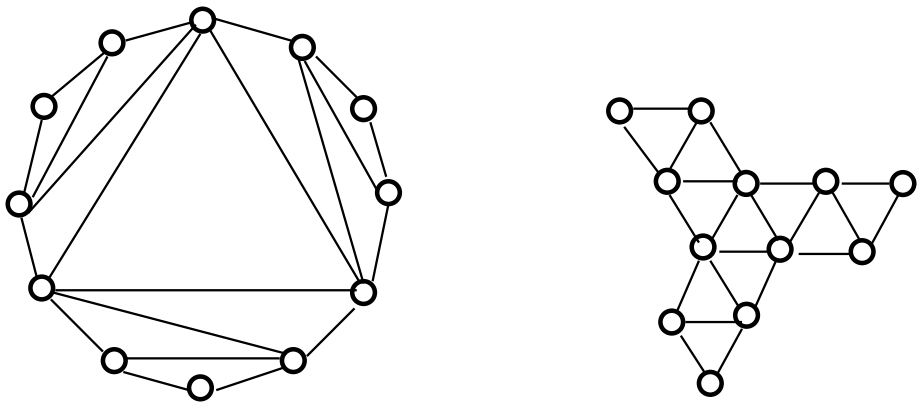


Figure 9.1: Two different drawings of a tree-like graph.

encounter such a nice situation in general graphs, where there might not be a node that "breaks the communication" between sub-problems in the rest of the graph. Rather, for the independent set problem in general graphs, decisions we make in one place seem to have complex repercussions all across the graph.

So we can ask a weaker version of our question instead: is the algorithm from the previous section useful only on trees, or can we generalize it to a larger class of graphs? All we really needed, after all, was this notion of breaking the communication between sub-problems — we needed a recursive way to delete a small set of nodes, check all possibilities for these nodes exhaustively, and end up with a set of sub-problems that can be solved completely independently.

In fact, there is a larger class of graphs that supports this type of algorithm; they are essentially "generalized trees," and for reasons that will become clear shortly, we will refer to them as *graphs of bounded tree-width*. Just as with trees, many NP-complete problems are tractable on graphs of bounded tree-width; and the class of graphs of bounded tree-width turns out to have considerable practical value, since it includes many real-world networks on which NP-complete graph problems arise. So in a sense, this type of graph serves as a nice example of finding the "right" special case of a problem that simultaneously allows for efficient algorithms and also includes graphs that arise in practice.

Defining Tree-Width

We now give a precise definition for this class of graphs that is designed to generalize trees. The definition is motivated by two considerations. First, we want to find graphs that we can decompose into disconnected pieces by removing a small number of nodes; this allows us to implement dynamic programming algorithms of the type we discussed above. Second, we want to make precise the intuition conveyed by Figure 9.1.

We want to claim that the graph G pictured in this figure is decomposable in a "tree-like" way, along the lines that we've been considering. If we were to encounter G as it drawn on the left, it might not be immediately clear why this is so. In the drawing on the right, however, we see that G is really composed of ten interlocking triangles; and seven of the ten triangles have the property that if we delete them, then the remainder of G falls apart into disconnected pieces that recursively have this interlocking-triangle structure. The other three triangles are attached at the extremities, and deleting them is sort of like deleting the leaves of a tree.

So G is "tree-like" if we view it not as being composed of twelve nodes, as we usually would, but instead as being composed of ten triangles. Although G clearly contains many cycles, it seems — intuitively — to lack cycles when viewed at the level of these ten triangles; and based on this, it inherits many of the nice decomposition properties of a tree. How can we make this precise?

We do this by introducing the idea of a *tree decomposition* of a graph G, so named because we will seek to decompose G according to a tree-like pattern. Formally, a tree decomposition of $G = (V, E)$ consists of a tree T (on a different node set), and a subset $V_t \subseteq V$ associated with each node t of T. (We will call these subsets V_t the "pieces" of the tree decomposition.) We will sometimes write this as the ordered pair $(T, \{V_t : t \in T\})$. The tree T and the collection of pieces $\{V_t : t \in T\}$ must satisfy the following three properties.

- *(Node Coverage.)* Every node of G belongs to at least one piece V_t.
- *(Edge Coverage.)* For every edge e of G, there is some piece V_t containing both ends of e.
- *(Coherence.)* Let t_1, t_2, and t_3 be three nodes of T such that t_2 lies on the path from t_1 to t_3. Then if a node v of G belongs to both V_{t_1} and V_{t_3}, then it also belongs to V_{t_2}.

It's worth checking that there is a tree decomposition of the graph in Figure 9.1 using a tree T with ten nodes, and using the ten triangles as the pieces.

If we consider the definition more closely, we see that the Node Coverage and Edge Coverage properties simply ensure that the collection of pieces corresponds to the graph G in a minimal way. The crux of the definition is in the Coherence property. While it is not obvious from its statement that Coherence leads to tree-like separation properties, in fact it does so quite naturally.

It we think about, trees have two nice separation properties, closely related to each other, that get used all the time. One says that if we delete an edge e from a tree, it falls apart into exactly two connected components. The other says that if we delete a node t from a tree, then this is like deleting all the incident edges, and so the tree falls apart into a number of components equal to the degree of t. The central fact about tree decompositions is that

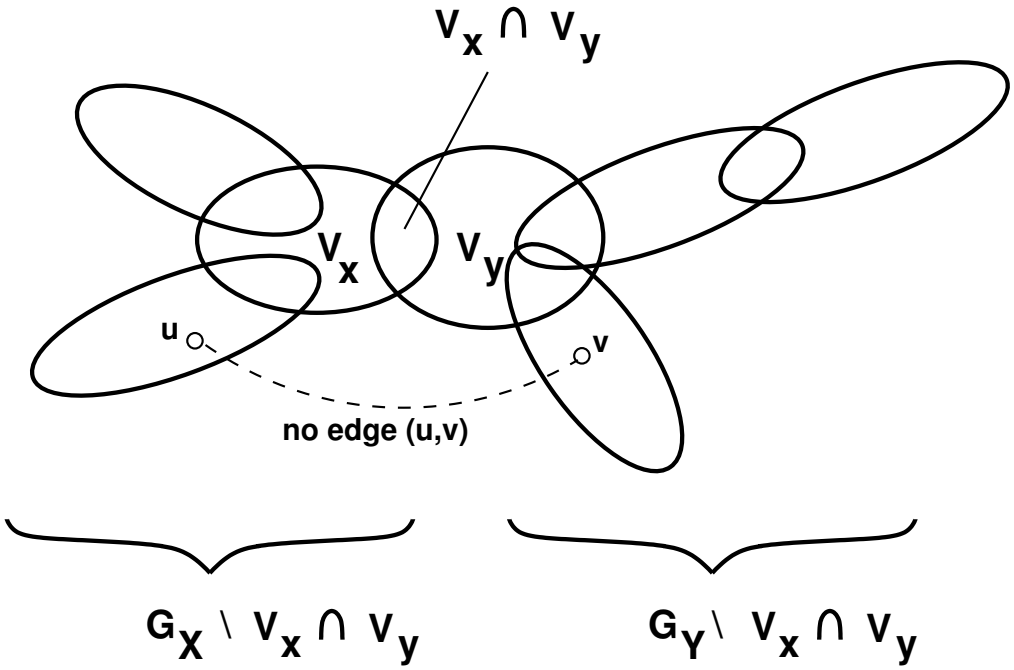


Figure 9.2: Separations of the tree T translate to separations of the graph G.

separations of T, of both these types, correspond naturally to separations of G as well. If T' is a subgraph of T, we use $G_{T'}$ to denote the subgraph of G induced by the nodes in all pieces associated with nodes of T', i.e. the set $\cup_{t \in T'} V_t$.

Now, if we delete an edge (x, y) from T, then T falls apart into two components: X, containing x, and Y, containing y. Let's establish the corresponding way in which G is separated by this operation.

(9.9) *As above, let X and Y be the two components of T after the deletion of the edge (x, y). Then deleting the set $V_x \cap V_y$ from V disconnects G into the two subgraphs $G_X - (V_x \cap V_y)$ and $G_Y - (V_x \cap V_y)$ More precisely, these two subgraphs do not share any nodes, and there is no edge with one end in each of them.*

Proof. We refer to Figure 9.2 for a general view of what the separation looks like. We first prove that the two subgraphs $G_X - (V_x \cap V_y)$ and $G_Y - (V_x \cap V_y)$ do not share any nodes. Indeed, any such node v would need to belong to both G_X and G_Y. So such a node v belongs to some piece $V_{x'}$ with $x' \in X$, and to some piece $V_{y'}$ with $y' \in Y$. Since both x and y lie on the x'-y' path in T, it follow from the Coherence property that v lies in both V_x and V_y. Thus v belongs to $V_x \cap V_y$, and hence belongs to neither of $G_X - (V_x \cap V_y)$ nor $G_Y - (V_x \cap V_y)$.

Now we must show that there is no edge $e = (u, v)$ in G with one end u in $G_X - (V_x \cap V_y)$ and the other end v in $G_Y - (V_x \cap V_y)$. If there were such an edge, then by the Edge Coverage property, there would need to be some piece V_z containing both u and v. Suppose by

symmetry that $z \in X$. Then since the node v belongs to both V_y and V_z, and since x lies on the y-z path in T, it follows that v also belongs to V_x. Hence $v \in V_x \cap V_y$, and so it does not lie in $G_Y - (V_x \cap V_y)$ as required. ■

It follows from (9.9) that if t is a node of T, then deleting V_t from G has a similar separating effect.

(9.10) *Suppose that $T-t$ has components $T_1, \ldots, T_d$. Then the subgraphs*

$$G_{T_1} - V_t, G_{T_2} - V_t, \ldots, G_{T_d} - V_t$$

have no nodes in common, and there are no edges between them.

So tree decompositions are useful in that the separation properties of T carry over to G. At this point, one might think that the key question is: "Which graphs have tree decompositions?" But this is not the point, for if we think about it, we see that of course every graph has a tree decomposition; given any G, we can let T be a tree consisting of a single node t, and let the single piece V_t be equal to the entire node set of G. This easily satisfies the three properties required by the definition; and such a tree decomposition is no more useful to us than the original graph.

The crucial point, therefore, is to look for a tree decomposition in which all the pieces are *small* — this is really what we're trying to carry over from trees, by requiring that the deletion of a very small set of nodes break apart the graph into disconnected subgraphs. So we define the *width* of a tree decomposition $(T, \{V_t\})$ to be one less than the maximum size of any piece V_t:

$$\text{width}(T, \{V_t\}) = \max_t |V_t| - 1.$$

We then define the *tree-width* of G to the minimum width of any tree decomposition of G.

Thus we can talk about the set of all graphs of tree-width 1, the set of all graphs of tree-width 2, and so forth. The following fact establishes that our definitions here indeed generalize the notion of a tree; the proof also provides a good way for us to exercise some of the basic properties of tree decompositions.

(9.11) *A connected graph G has tree-width 1 if and only if it is a tree.*

Proof. First, if G is a tree, then we can build a tree decomposition of it as follows. The underlying tree T has a node t_v for each node v of G, and a node t_e for each edge e of G. T has an edge (t_v, t_e) when v is an end of e. Finally, if v is a node, then we define the piece $V_{t_v} = \{v\}$; and if $e = (u, v)$ is an edge, then we define the piece $V_{t_e} = \{u, v\}$. One can now check that the three properties in the definition of a tree decomposition are satisfied.

To prove the converse, we first establish the following useful fact: if H is a subgraph of G, then the tree-width of H is at most the tree-width of G. This is simply because, given a

tree decomposition $(T, \{V_t\})$ of G, we can define a tree-decomposition of H by keeping the same underlying tree T, and replacing each piece V_t with $V_t \cap H$. It is easy to check that the required three properties still hold. (The fact that certain pieces may now be equal to the empty set does not pose a problem.)

Now, suppose by way of contradiction that G is a connected graph of tree-width 1 that is not a tree. Since G is not a tree, it has a subgraph consisting of a simple cycle C. By our argument from the previous paragraph, it is now enough for us to argue that the graph C does not have tree-width 1. Indeed, suppose it had a tree decomposition $(T, \{V_t\})$ in which each piece had size at most 2. Choose any two edges (u, v) and (u', v') of C; by the Edge Coverage property, there are pieces V_t and $V_{t'}$ containing them. Now, on the path in T from t to t' there must be an edge (x, y) such that the pieces V_x and V_y are unequal. It follows that $|V_x \cap V_y| \leq 1$. We now invoke (9.9): defining X and Y to be the components of $T - (x, y)$ containing x and y respectively, we see that deleting $V_x \cap V_y$ separates C into $C_X - (V_x \cap V_y)$ and $C_Y - (V_x \cap V_y)$. Neither of these two subgraphs can be empty, since one contains $\{u, v\} - (V_x \cap V_y)$ and the other contains $\{u', v'\} - (V_x \cap V_y)$. But it is not possible to disconnect a cycle into two non-empty subgraphs by deleting a single node, and so this yields a contradiction. ■

In fact, the somewhat puzzling "-1" in the definition of the width of a tree-decomposition is precisely so that trees would turn out to have tree-width 1, rather than 2. We also observe that the graph in Figure 9.1 is thus, according to the notion of tree-width, a member of the next "simplest" class of graphs after trees — it is a graph of tree-width 2.

When we use tree decompositions in the context of dynamic programming algorithms, we would like — for the sake of efficiency — that they not have too many pieces. Here is a simple way to do this. If we are given a tree decomposition $(T, \{V_t\})$ of a graph G, and we see an edge (x, y) of T such that $V_x \subseteq V_y$, then we can contract the edge (x, y) (folding the piece V_x into the piece V_y) and obtain a tree decomposition of G based on a smaller tree. Repeating this process as often as necessary, we end up with a *non-redundant tree decomposition*: there is no edge (x, y) of the underlying tree such that $V_x \subseteq V_y$.

Once we've reached such a tree decomposition, we can be sure that it does not have too many pieces:

(9.12) *Any non-redundant tree decomposition of an n-node graph has at most n pieces.*

Proof. We prove this by induction on n, the case $n = 1$ being clear. Let's consider the case in which $n > 1$. Given a non-redundant tree decomposition $(T, \{V_t\})$ of an n-node graph, we first identify a leaf t of T. By the non-redundancy condition, there must be at least one node in V_t that does not appear in the neighboring piece, and hence (by the Coherence property) does not appear in any other piece. Let U be the set of all such nodes. We now observe that by deleting t from T, and removing V_t from the collection of pieces, we

obtain a non-redundant tree decomposition of $G-U$. By our inductive hypothesis, this tree decomposition has at most $n - |U| \leq n - 1$ pieces, and so $(T, \{V_t\})$ has at most n pieces. ■

While (9.12) is very useful for making sure one has a small tree decomposition, it is often easier in the course of analyzing a graph to start by building a redundant tree decomposition, and only later "condensing" it down to a non-redundant one. For example, the proof of (9.11) started by constructing a redundant tree decomposition; it would not have been as simple to directly describe a non-redundant one.

Having thus laid the groundwork, we now turn to the algorithmic uses of tree decompositions.

Dynamic Programming over a Tree Decomposition

We began by claiming that the *Maximum-Weight Independent Set* could be solved efficiently on any graph for which the tree-width was bounded; now it's time to deliver on this promise. Specifically, we will develop an algorithm that closely follows the linear-time algorithm for trees; given an n-node graph with an associated tree decomposition of width w, it will run in time $O(f(w) \cdot n)$, where $f(\cdot)$ is an exponential function that depends only on the width w, not on the number of nodes n. And as in the case of trees, although we are focusing on *Maximum-Weight Independent Set*, the approach here is useful for many NP-complete problems.

So in a very concrete sense, the complexity of the problem has been pushed off of the size of the graph and into the tree-width, which may be much smaller. As we mentioned earlier, large networks in the real world often have very small tree-width; and often this is not coincidental, but a consequence of the structured or modular way in which they are designed. So if we encounter a 1000-node network with a tree decomposition of width 4, the approach discussed here takes a problem that would have been hopelessly intractable and makes it potentially quite manageable.

Of course, this is all somewhat reminiscent of the *Vertex Cover* algorithm from the first section of this chapter. There we pushed the exponential complexity into the parameter k, the size of the vertex cover being sought. Here we did not have an obvious parameter other than n lying around, so we were forced to invent a fairly non-obvious one — the tree-width.

To design the algorithm, we recall what we did for the case of a tree T. After rooting T, we built the independent set by working our way up from the leaves. At each internal node u, we enumerated the possibilities for what to do with u — include it or not include it — since once this decision was fixed, the problems for the different sub-trees below u became independent.

The generalization for a graph G with a tree decomposition $(T, \{V_t\})$ of width w looks very similar. We root the tree T, and build the independent set by considering the pieces

V_t from the leaves upward. At an internal node t of T, we confront the following basic question: the optimal independent set intersects the piece V_t in some subset U, but we don't know which set U it is. So we enumerate all the possibilities for this subset U — i.e., all possibilities for which nodes to include from V_t and which to leave out. Since V_t may have size up to $w + 1$, this may be 2^{w+1} possibilities to consider. But we now can exploit two key facts: first, that the quantity 2^{w+1} is a lot more reasonable than 2^n when w is much smaller than n; and second, that once we fix a particular one of these 2^{w+1} possibilities — once we've decided which nodes in the piece V_t to include — the separation properties (9.9) and (9.10) ensure that the problems in the different subtrees of T below t can be solved independently. So while we settle for doing brute-force search at the level of a *single* piece, we have an algorithm that is quite efficient at the global level when the individual pieces are small.

Defining the Sub-Problems. More precisely, we root the tree T at a node r. For any node t, let T_t denote the sub-tree rooted at t. Recall that G_{T_t} denotes the subgraph of G induced by the nodes in all pieces associated with nodes of T_t; for notational simplicity, we will also write this subgraph as G_t. For a subset U of V, we use $w(U)$ to denote the total weight of nodes in U; that is, $w(U) = \sum_{u \in U} w_u$.

We define a set of sub-problems for each sub-tree T_t, one corresponding to each possible subset U of V_t that may represent the intersection of the optimal solution with V_t. Thus, for each independent set $U \subseteq V_t$, we write $f_t(U)$ to denote the maximum weight of an independent set S in G_t, subject to the requirement that $S \cap V_t = U$. The quantity $f_t(U)$ is undefined if U is not an independent set, since in this case we know that U cannot represent the intersection of the optimal solution with V_t.

There are at most 2^{w+1} sub-problems associated with each node t of T, since this is the maximum possible number of independent subsets of V_t. By (9.12), we can assume we are working with a tree decomposition that has at most n pieces, and hence there are a total of at most $2^{w+1}n$ sub-problems overall. Clearly, if we have the solutions to all these sub-problems, we can determine the maximum weight of an independent set in G by looking at the sub-problems associated with the root r: we simply take the maximum, over all independent sets $U \subseteq V_r$, of $f_r(U)$.

Building up Solutions. Now we must show how to build up the solutions to these sub-problems via a recurrence. It's easy to get started: When t is a leaf, $f_t(U)$ is equal to $w(U)$ for each independent set $U \subseteq V_t$.

Now suppose that t has children $t_1, \ldots, t_d$, and we have already determined the values of $f_{t_i}(W)$ for each child t_i and each independent set $W \subseteq V_{t_i}$. How do we determine the value of $f_t(U)$ for an independent set $U \subseteq V_t$?

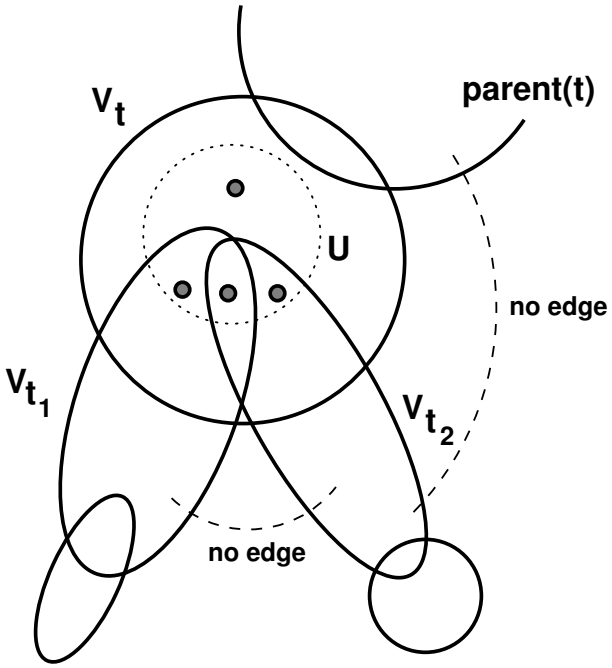


Figure 9.3: The sub-problem $f_t(U)$ in the subgraph G_t.

Let S be the maximum-weight independent set in G_t subject to the requirement that $S \cap V_t = U$; i.e. $w(S) = f_t(U)$. The key is to understand how this set S looks when intersected with each of the subgraphs G_{t_i}, as suggested in Figure 9.3. We let S_i denote the intersection of S with the nodes of G_{t_i}.

(9.13) *S_i is a maximum-weight independent set of G_{t_i}, subject to the constraint that $S_i \cap V_t = U \cap V_{t_i}$.*

Proof. Suppose there were an independent set S_i' of G_{t_i} with the property that $S_i' \cap V_t = U \cap V_{t_i}$ and $w(S_i') > w(S_i)$. Then consider the set $S' = (S - S_i) \cup S_i'$. Clearly $w(S') > w(S)$. Also, it is easy to check that $S' \cap V_t = U$.

We claim that S' is an independent set in G; this will contradict our choice of S as the maximum-weight independent set in G_t subject to $S \cap V_t = U$. For suppose S' is not independent, and let $e = (u, v)$ be an edge with both ends in S'. It cannot be that u and v both belong to S, or that they both belong to S_i', since these are both independent sets. Thus we must have $u \in S - S_i'$ and $v \in S_i' - S$, from which it follows that u is not a node of G_{t_i} while $v \in G_{t_i} - (V_t \cap V_{t_i})$. But then, by (9.9), there cannot be an edge joining u and v. ■

(9.13) is exactly what we need to design a recurrence relation for our sub-problems — it says that the information needed to compute $f_t(U)$ is implicit in the values already computed

for the subtrees. Specifically, for each child t_i, we need simply determine the value of the maximum-weight independent set S_i of G_{t_i}, subject to the constraint that $S_i \cap V_t = U \cap V_{t_i}$. This constraint does not completely determine what $S_i \cap V_{t_i}$ should be; rather, it says that it can be any independent set $U_i \subseteq V_{t_i}$ such that $U_i \cap V_t = U \cap V_{t_i}$. Thus, the weight of the optimal S_i is equal to

$$\max w(U) + \{f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}.$$

Finally, the value of $f_t(U)$ is simply $w(U)$ plus these maxima added over the d children of t — except that to avoid over-counting the nodes in U, we exclude them from the contribution of the children. Thus we have

(9.14) *The value of $f_t(U)$ is given by the following recurrence:*

$$f_t(U) = \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U_i \cap U) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}.$$

The overall algorithm now just builds up the values of all the sub-problems from the leaves of T upward.

```
To find a maximum-weight independent set of G,
given a tree decomposition (T, {V_t}) of G:
    Modify the tree decomposition if necessary so it is non-redundant.
    Root T at a node r.
    For each node t of T in post-order
        If t is a leaf then
            For each independent set U of V_t
                f_t(U) = w(U)
        Else
            For each independent set U of V_t
                f_t(U) is determined by the recurrence in (9.14)
        Endif
    Endfor
    Return max{f_r(U) : U ⊆ V_r is independent}.
```

An actual independent set of maximum weight can be found, as usual, by tracing back through the execution.

We can determine the time required for computing $f_t(U)$ as follows: for each of the d children t_i, and for each independent set U_i in V_{t_i}, we spend time $O(w)$ determining whether it should be considered in the computation of (9.14). This is a total time of $O(2^{w+1}wd)$ for $f_t(U)$; since there are at most 2^{w+1} sets U associated with t, the total time spent on node t is $O(4^{w+1}wd)$. Finally, we sum this over all nodes t to get the total running time. We observe that the sum, over all nodes t, of the number of children of t is $O(n)$ — since each node is counted as a child once. Thus the total running time is $O(4^{w+1}wn)$.

Constructing a Tree Decomposition

There is still a crucial missing piece in our algorithmic use of tree-width — thus far, we have simply provided an algorithm for *Maximum-Weight Independent Set* on a graph G, *provided we have been given a low-width tree decomposition of* G. What if we simply encounter G "in the wild," and no one has been kind enough to hand us a good tree decomposition of it? Can we compute one on our own, and then proceed with the dynamic programming algorithm?

The answer is basically "yes," with some caveats. First, we must warn that given a graph G, it is NP-hard to determine its tree-width. However, the situation for us is not actually so bad, because we are only interested here in graphs for which the tree-width is a small constant. And in this case, we will describe an algorithm with the following guarantee: given a graph G of tree-width w, it will produce a tree decomposition of G of width at most $4w$ in time $O(f(w) \cdot mn)$, where m and n are the number of edges and nodes of G, and $f(\cdot)$ is a function that depends only on w. So essentially, when the tree-width is small, there's a reasonably fast way to produce a tree decomposition whose width is almost as small as possible.

An obstacle to low tree-width. The first step in designing an algorithm for this problem is to work out a reasonable "obstacle" to a graph G having low tree-width. In other words, as we try to construct a tree decomposition of low width for $G = (V, E)$, might there be some "local" structure we could discover that will tell us the tree-width must in fact be large?

The following idea turns out to provide us with such an obstacle. First, given two sets $Y, Z \subseteq V$ of the same size, we say they are *separable* if some strictly smaller set can completely disconnect them — specifically, if there is a set $S \subseteq V$ such that $|S| < |Y| = |Z|$ and there is no path from $Y-S$ to $Z-S$ in $G-S$. (In this definition, Y and Z need not be disjoint.) Next, we say that a set X of nodes in G is *w-linked* if $|X| \geq w$ and X does not contain separable subsets Y and Z, such that $|Y| = |Z| \leq w$.

For later algorithmic use of w-linked sets, we make note of the following fact.

(9.15) *Let $G = (V, E)$ have m edges, let X be a set of k nodes in G, and let $w \leq k$ be a given parameter. Then we can determine whether X is w-linked in time $O(f(k) \cdot m)$, where $f(\cdot)$ depends only on k. Moreover, if X is not w-linked, we can return a proof of this in the form of sets $Y, Z \subseteq X$ and $S \subseteq V$ such that $|S| < |Y| = |Z| \leq w$ and there is no path from $Y-S$ to $Z-S$ in $G-S$.*

Proof. We are trying to decide whether X contains separable subsets Y and Z such that $|Y| = |Z| \leq w$. We can first enumerate all pairs of sufficiently small subsets Y and Z; since X only has 2^k such subsets, there are at most 4^k such pairs.

Now, for each pair of subsets Y, Z, we must determine whether they are separable. Let $\ell = |Y| = |Z| \leq w$. But this is exactly the Max-Flow Min-Cut theorem when we have an

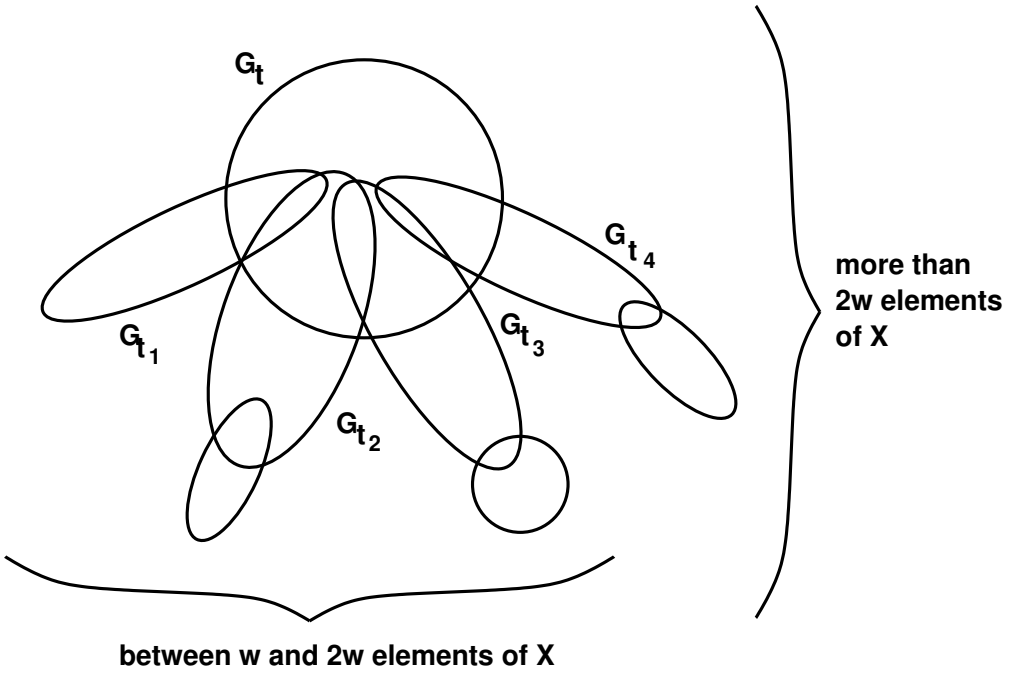


Figure 9.4: The final step in the proof of (9.16).

undirected graph with capacities on the nodes: Y and Z are separable if and only there do not exist ℓ node-disjoint paths, each with one end in Y and the other in Z. We can determine whether such paths exist using an algorithm for flow with (unit) capacities on the nodes; this takes time $O(\ell m)$. ■

One should imagine a w-linked set as being highly self-entwined — it has no two small parts that can be easily split off from one another. At the same time, a tree decomposition cuts up a graph using very small separators; and so it is intuitively reasonable that these two structures should be in opposition to one another.

(9.16) *If G contains a $(w+1)$-linked set of size at least $3w$, then G has tree-width at least w.*

Proof. Suppose by way of contradiction that G has a $(w+1)$-linked set X of size at least $3w$, and it also has a tree decomposition $(T, \{V_t\})$ of width less than w; in other words, each piece V_t has size at most w. We may further assume that $(T, \{V_t\})$ is non-redundant.

The idea of the proof is to find a piece V_t that is "centered" with respect to X, so that when some part of V_t is deleted from G, one small subset of X is separated from another. Since V_t has size at most w, this will contradict our assumption that X is $(w+1)$-linked.

So how do we find this piece V_t? We first root the tree T at a node r; using the same notation as before, we let T_t denote the sub-tree rooted at a node t, and write G_t for G_{T_t}.

Now, let t be a node that is far from the root r as possible subject to the condition that G_t contains more than $2w$ nodes of X.

Clearly, t is not a leaf (or else G_t could contain at most w nodes of X); so let $t_1, \ldots, t_d$ be the children of t. Note that since each t_i is farther than t from the root, each subgraph G_{t_i} contains at most $2w$ nodes of X. If there is a child t_i so that G_{t_i} contains at least w nodes of X, then we can define Y to be w nodes of X belonging to G_{t_i}, and Z to be w nodes of X belonging to $G - G_{t_i}$. Since $(T, \{V_t\})$ is non-redundant, $S = V_{t_i} \cap V_t$ has size at most $w - 1$; but by (9.9), deleting S disconnects $Y - S$ from $Z - S$. This contradicts our assumption that X is $(w+1)$-linked.

So we consider the case in which there is no child t_i such that G_{t_i} contains at least w nodes of X; Figure 9.4 suggests the structure of the argument in this case. We begin with the node set of G_{t_1}, combine it with G_{t_2}, then G_{t_3}, and so forth, until we first obtain a set of nodes containing more than w members of X. This will clearly happen by the time we get to G_{t_d}, since G_t contains more than $2w$ nodes of X, and at most w of them can belong to V_t. So suppose our process of combining $G_{t_1}, G_{t_2}, \ldots$ first yields more than w members of X once we reach index $i \leq d$. Let W denote the set of nodes in the subgraphs $G_{t_1}, G_{t_2}, \ldots, G_{t_i}$. By our stopping condition, we have $|W \cap X| > w$. But since G_{t_i} contains fewer than w nodes of X, we also have $|W \cap X| < 2w$. Hence we can define Y to be $w+1$ nodes of X belonging to W, and Z to be $w+1$ nodes of X belonging to $V - W$. By (9.10), the piece V_t is now a set of size at most w whose deletion disconnects $Y - V_t$ from $Z - V_t$. Again this contradicts our assumption that X is $(w+1)$-linked, completing the proof. ■

An algorithm to search for a low-width tree decomposition. Building on these ideas, we now give a greedy algorithm for constructing a tree decomposition of low width. The algorithm will not precisely determine the tree-width of the input graph $G = (V, E)$; rather, given a parameter w, it will either produce a tree decomposition of width at most $4w$, or it will discover a $(w+1)$-linked set of size at least $3w$. In the latter case, this constitutes a proof that the tree-width of G is at least w, by (9.16); so our algorithm is essentially capable of narrowing down the true tree-width of G to within a factor of 4. As discussed above, the running time will have the form $O(f(w) \cdot mn)$, where m and n are the number of edges and nodes of G, and $f(\cdot)$ depends only on w.

Having worked with tree decompositions for a little while now, one can start imagining what might be involved in constructing one for an arbitrary input graph G. The process is depicted at a high level in Figure 9.5. Our goal is to make G fall apart into tree-like portions; we begin the decomposition by placing the first piece V_t anywhere. Now, hopefully, $G - V_t$ consists of several disconnected components; we recursively move into each of these components, placing a piece in each so that it partially overlaps the piece V_t that we've already defined. We hope that these new pieces cause the graph to break up further, and

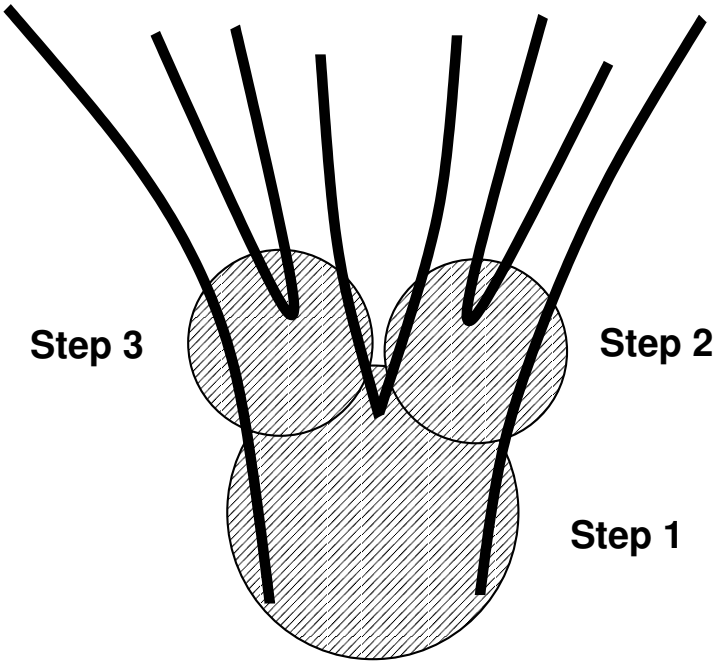


Figure 9.5: Constructing a tree decomposition: a schematic view.

we thus continue in this way, pushing forward with small sets while the graph breaks apart in front of us. The key to making this algorithm work is to argue the following: if at some point we get stuck, and our small sets don't cause the graph to break up any further, then we can extract a large $(w+1)$-linked set that proves the tree-width was in fact large.

Given how vague this intuition is, the actual algorithm follows it more closely than you might think. We start by assuming that there is no $(w+1)$-linked set of size at least $3w$; our algorithm will produce a tree decomposition provided this holds true, and otherwise we can stop with a proof that the tree-width of G is at least w. We grow the underlying tree T of the decomposition, and the pieces V_t, in a greedy fashion. At every intermediate stage of the algorithm, we will maintain the property that we have a *partial tree decomposition*: by this we mean that if $U \subseteq V$ denotes the set of nodes of G that belong to at least one of the pieces already constructed, then our current tree T and pieces V_t should form a tree decomposition of the subgraph of G induced on U.

If C is a connected component of $G-U$, we say that $u \in U$ is a *neighbor* of C if there is some node $v \in C$ with an edge to u. The key behind the algorithm is to not simply maintain a partial tree decomposition of width at most $4w$, but to also make sure the following invariant is enforced the whole time:

> $(*)$ At any stage in the execution of the algorithm, each component C of $G-U$ has at most $3w$ neighbors, and there is a single piece V_t that contains all of them.

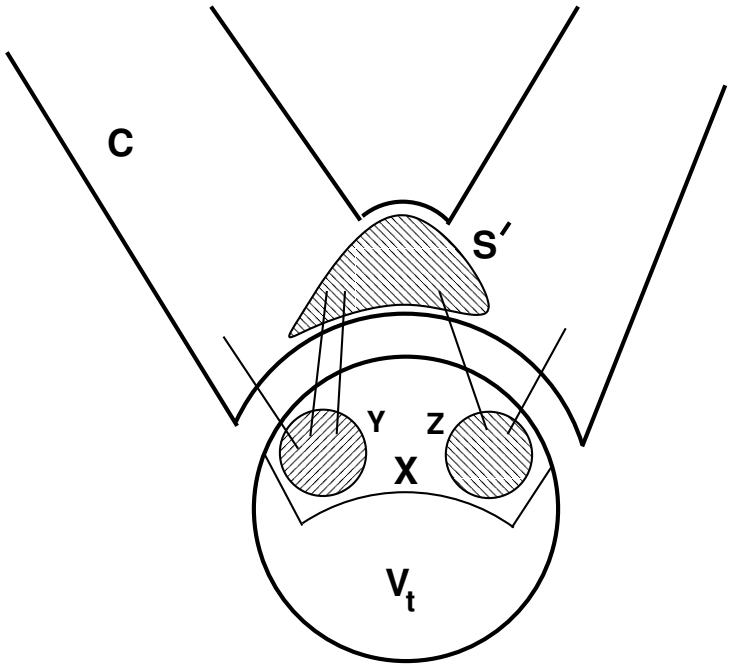


Figure 9.6: Adding a new piece to the partial tree decomposition.

Why is this invariant so useful? It's useful because it will let us add a new node s to T, and grow a new piece V_s in the component C, with the confidence that s can be a leaf hanging off t in the larger partial tree decomposition. Moreover, $(*)$ requires there be at most $3w$ neighbors, while we are trying to produce a tree decomposition of width at most $4w$; this extra w gives our new piece "room" to expand by a little as it moves into C.

Specifically, we now describe how to add a new node and a new piece so that we still have a partial tree decomposition, the invariant $(*)$ is still maintained, and the set U has grown strictly larger. In this way, we make at least one node's worth of progress, and so the algorithm will terminate in at most n iterations with a tree decomposition of the whole graph G.

Let C be any component of $G-U$, let X be the set of neighbors of U, and let V_t be a piece that, as guaranteed by $(*)$, contains all of X. We know, again by $(*)$, that X contains at most $3w$ nodes. If X in fact contains strictly fewer than $3w$ nodes, we can make progress right away: for any node $v \in C$ we define a new piece $V_s = X \cup \{v\}$, making s a leaf of t. Since all the edges from v into U have their ends in X, it is easy to confirm that we still have a partial tree decomposition obeying $(*)$, and U has grown.

Thus, let's suppose that X has exactly $3w$ nodes. In this case, it is less clear how to proceed; for example, if we try to create a new piece by arbitrarily adding a node $v \in C$ to X, we may end up with a component of $C-\{v\}$ (which may be all of $C-\{v\}$) whose neighbor set includes all $3w+1$ nodes of $X \cup \{v\}$, and this would violate $(*)$.

There's no simple way around this; for one thing, G may not actually have a low-width tree decomposition. So this is precisely the place where it makes sense to ask whether X poses a genuine obstacle to the tree decomposition or not: we test whether X is a $(w+1)$-linked set. By (9.15) , we can determine the answer to this in time $O(f(w) \cdot m)$, since $|X| = 3w$. If it turns out that X is $(w+1)$-linked, then we are all done; we can halt with the conclusion that G has tree-width at least w, which was one acceptable outcome of the algorithm. On the other hand, if X is not $(w+1)$-linked, then we end up with $Y, Z \subseteq X$ and $S \subseteq V$ such that $|S| < |Y| = |Z| \leq w+1$ and there is no path from $Y-S$ to $Z-S$ in $G-S$. The sets Y, Z, and S will now provide us with a means to extend the partial tree decomposition.

Let S' consist of the nodes of S that lie in $Y \cup Z \cup C$. The situation is now as pictured in Figure 9.6. We observe that S' cannot be empty: Y and Z each have edges into C, and so if $S' = \phi$ then there would be a path from $Y-S$ to $Z-S$ in $G-S$ that started in Y, jumped immediately into C, traveled through C, and finally jumped back into Z. At the same time, $|S'| \leq |S| \leq w$.

We define a new piece $V_s = X \cup S'$, making s a leaf of t. All the edges from S' into U have their ends in X, and $|X \cup S'| \leq 3w + w = 4w$, so we still have a partial tree decomposition. Moreover, the set of nodes covered by our partial tree decomposition has grown since S' is not empty. So we will be done if we can show that the invariant $(*)$ still holds. This brings us exactly the intuition we tried to capturing when discussing Figure 9.5 — as we add the new piece $X \cup S'$, we are hoping that the component C breaks up into further components in a nice way.

Concretely, our partial tree decomposition now covers $U \cup S'$; and where we previously had a component C of $G-U$, we now may have several components $C' \subseteq C$ of $G-(U \cup S')$. Each of these components C' has all its neighbors in $X \cup S'$; but we must additionally make sure there are at most $3w$ such neighbors, so that the invariant $(*)$ continues to hold. So consider one of these components C'. We claim that all its neighbors in $X \cup S'$ actually belong to one of the two subsets $(X-Z) \cup S'$ or $(X-Y) \cup S'$ — and each of these sets has size at most $|X| \leq 3w$. For if this did not hold, then C' would have a neighbor in both $Y-S$ and $Z-S$, and hence there would be a path, through C', from $Y-S$ to $Z-S$ in $G-S$. But we know there cannot be such a path. This establishes that $(*)$ still holds after the addition of the new piece, and completes the argument that the algorithm works correctly.

Finally, what is the running time of the algorithm? The time to add a new piece to the partial tree decomposition is dominated by the time required to check whether X is $(w+1)$-linked, which is $O(f(w) \cdot m)$. We do this for at most n iterations, since we increase the number of nodes of G that we cover in each iteration. So the total running time is $O(f(w) \cdot mn)$.

9.4 Exercises

1. Earlier, we considered an exercise in which we claimed that the *Hitting Set* problem was NP-complete. To recap the definitions, Consider a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i. (So H "hits" all the sets B_i.)

   Now, suppose we are given an instance of this problem, and we'd like to determine whether there is a hitting set for the collection of size at most k. Furthermore, suppose that each set B_i has at most c elements, for a constant c. Give an algorithm that solves this problem with a running time of the form $O(f(c,k) \cdot p(n,m))$, where $p(\cdot)$ is a polynomial function, and $f(\cdot)$ is an arbitrary function that depends only on c and k, not on n or m.

2. Consider a network of workstations modeled as an undirected graph G, where each node is a workstation, and the edges represent direct communication links. We'd like to place copies of a database at nodes in G, so that each node is close to at least one copy.

   Specifically, assume that each node v in G has a cost c_v charged for placing a copy of the database at node v. The Min-Cost Server Placement problem is as follows. Given the network G, and costs $\{c_v\}$, find a set of nodes $S \subseteq V$ of minimum total cost $\sum_{v \in S} c_v$, so that if we place copies of a database at each node in S, then every workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database.

   Give a polynomial time algorithm for the special case of the Min-Cost Server Placement where the graph G is a tree.

   Note the difference between Server Placement and Vertex Cover. If the graph G is a path of consisting of 6 nodes, then Vertex Cover needs to select at least 3 of the 6 nodes, while the second and the 5th node form a valid solution of the Min-Cost Server Placement problem, requiring only two nodes.

3. Suppose we are given a directed graph $G = (V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$, and we want to decide whether G has a Hamiltonian path from v_1 to v_n. (That is, is there a path in G that goes from v_1 to v_n, passing through every other vertex exactly once?)

   Since the Hamiltonian path problem is NP-complete, we do not expect that there is a polynomial-time solution for this problem. However, this does not mean that all non-polynomial-time algorithms are equally "bad." For example, here's the simplest brute-force approach: for each permutation of the vertices, see if it forms a Hamiltonian

path from v_1 to v_n. This takes time roughly proportional to $n!$, which is about 3×10^{17} when $n = 20$.

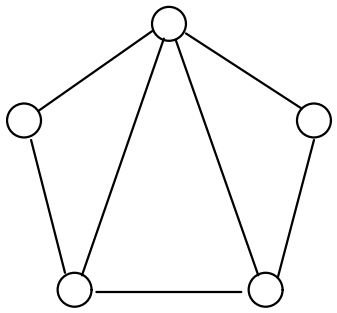
Show that the Hamiltonian path problem can in fact be solved in time $O(2^n \cdot p(n))$, where $p(n)$ is a polynomial function of n. This is a much better algorithm for moderate values of n; 2^n is only about a million when $n = 20$.

4. ($*$) Give a polynomial time algorithm for the following problem. We are given a binary tree $T = (V, E)$ with an even number of nodes, and a non-negative cost on each edge. Find a partition of the nodes V into two sets of *equal* size so that the weight of the cut between the two sets is as large as possible. (I.e. the total weight of edges with one end in each set is as large as possible.) Note that the restriction that the graph is a tree crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

5. We say that a graph $G = (V, E)$ is a *triangulated cycle graph* if it consists of the vertices and edges of a triangulated convex n-gon in the plane — in other words, if it can be drawn in the plane as follows.

   The vertices are all placed on the boundary of a convex set in the plane (we may assume on the boundary of a circle), with each pair of consecutive vertices on the circle joined by an edge. The remaining edges are then drawn as straight line segments through the interior of the circle, with no pair of edges crossing in the interior. If we let S denote the set of all points in the plane that lie on vertices or edges of the drawing, then each bounded component of $\mathbf{R}^2 - S$ is bordered by exactly three edges. (This is the sense in which the graph is a "triangulation.")

   A triangulated cycle graph is pictured below.



   Prove that every triangulated cycle graph has a tree decomposition of width at most 2, and describe an efficient algorithm to construct such a decomposition.

6. The *Minimum-cost Dominating Set Problem* is specified by an undirected graph $G = (V, E)$ and costs $c(v)$ on the nodes $v \in V$. A subset $S \subset V$ is said to be a *dominating*

set if all nodes $u \in V-S$ have an edge (u, v) to a node v in S. (Note the difference between dominating sets and vertex covers: in a dominating set, it is fine to have an edge (u, v) with neither u nor v in the set S as long as both u and v have neighbors in S.)

(a.) Give a polynomial time algorithm for the Dominating Set problem for the special case in which G is a tree.

(b.) Give a polynomial time algorithm for the Dominating Set problem for the special case in which G has tree-width 2, and we are also given a tree-decomposition of G with width 2.

7. The *Node-Disjoint Paths Problem* is given by an undirected graph G and k pairs of nodes (s_i, t_i) for $i = 1, \ldots, k$. The problem is to decide whether there are a node-disjoint paths P_i so that path P_i connects s_i to t_i. Give a polynomial time algorithm for the Node-Disjoint Paths Problem for the special case in which G has tree-width 2, and we are also given a tree-decomposition T of G with width 2.

8. A *k-coloring* of an undirected graph $G = (V, E)$ is an assignment of one of the numbers $\{1, 2, \ldots, k\}$ to each node, so that if two nodes are joined by an edge, then they are assigned different numbers. The *chromatic number* of G is the minimum k such that it has a k-coloring. For $k \geq 3$, it is NP-complete to decide whether a given input graph has chromatic number $\leq k$. (You don't have to prove this.)

(a) Show that for every natural number $w \geq 1$, there is a number $k(w)$ so that the following holds. If G is a graph of tree-width at most w, then G has chromatic number at most $k(w)$. (The point is that $k(w)$ depends only on w, not on the number of nodes in G.)

(b) Given an undirected n-node graph $G = (V, E)$ of tree-width at most w, show how to compute the chromatic number of G in time $O(f(w) \cdot p(n))$, where $p(\cdot)$ is a polynomial but $f(\cdot)$ can be an arbitrary function.

Chapter 10

Approximation Algorithms

Following our encounter with NP-completeness — and the idea of computational intractability in general — we've been dealing with a fundamental question: How should we design algorithms for problems where polynomial time is probably an unattainable goal?

In this chapter, we focus on a new theme related to this question — *approximation algorithms*, which run in polynomial time and find solutions that are guaranteed to be close to optimal. There are two key words to notice in this definition: "close," and "guaranteed." Unlike our approach via improved exponential algorithms, we will not be seeking the optimal solutions; as a result, it becomes feasible to aim for a polynomial running time.

10.1 Load Balancing Problems: Bounding the Optimum

The first problem we consider here from the perspective of approximation is the following load balancing problem. We are given a set of m machines $M_1, \ldots, M_m$, and a set of n jobs; each job j has a processing time t_j. We seek to assign each job to one of the machines.

In any assignment, we can let $A(i)$ denote the set of jobs assigned to machine M_i; under this assignment, machine M_i needs to work for a total time of

$$T_i = \sum_{j \in A(i)} t_j.$$

Our goal will be to assign the jobs to the machines that all these *loads* T_j are roughly the same.

There are many objective functions that we can use to formalize this goal. Here we will focus on a particularly natural one that we call the *makespan*; it is defined as the latest time T at which any job finishes. Notice that $T = \max_i T_i$. We will look for an assignment with a makespan that is as small as possible. Although we will not prove this, the problem of finding the minimum possible makespan is an NP-hard optimization problem.

We first consider a very simple greedy algorithm for the problem. The algorithm makes one pass through the jobs in any order; when it come to job j, it assigns j to the machine whose load is smallest so far.

```
Greedy-Balance
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
For j = 1, ..., n
  Let M_i be a machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
Endfor
```

Let T denote the makespan of the resulting schedule; we want to show that T is not much larger than the minimum possible makespan T^*. Of course, in trying to do this, we immediately encounter an obvious problem: We need to compare ourselves to the optimal value T^*, even though we don't what this value is. For the analysis, therefore, we will need a *lower bound* on the optimum — a quantity with the guarantee that no matter how good the optimum is, it cannot be less than this.

There are many possible lower bounds on the optimum. One idea for a lower bound is based on considering the total processing time $\sum_j t_j$. One of the m machines must do at least a $1/m$ fraction of the total work, and so we have

(10.1) *The optimal makespan is at least*

$$T^* \geq \frac{1}{m} \sum_j t_j.$$

There is a particular kind of case in which this lower bound is much too weak to be useful. Suppose we have one job that is extremely long relative to the sum of all processing times. In a sufficiently extreme version of this, the optimal solution will place this job on a machine by itself, and it will be the last one to finish. In such a case, our greedy algorithm would actually produce the optimal solution; but the lower bound in (10.1) isn't strong enough to establish this.

This suggests the following additional lower bound on T^*.

(10.2) *The optimal makespan is at least* $T^* \geq \max_j t_j$.

Now we are ready to evaluate the schedule obtained by our greedy algorithm. Let T_i denote the processing time on machine M_i, so that $T = \max_i T_i$ is the makespan of our schedule.

(10.3) *Algorithm* `Greedy-Balance` *produces an assignment of jobs to machines with makespan* $T \leq 2T^*$.

Proof. Here is the overall plan for the proof. In analyzing an approximation algorithm, one compares the solution obtained to what one knows about the optimum — in this case, our lower bounds (10.1) and (10.2). We consider a machine M_i that works for the full T units of time in our schedule, and we ask: what was the last job j to be placed on M_i? If t_j is not too large relative to most of the other jobs, then we are not too far above the lower bound (10.1). And if t_j is a very large job, then we can use (10.2).

Here is how we can quantify this. When we assigned job j to M_i, M_i had the smallest load of any machine — this is the key property of our greedy algorithm. Its load just before this assignment was $T_i - t_j$, and so it follows that $T_k \geq T_i - t_j$ for all k, including $k = i$. If we add up these inequalities over all m machines, we get $\sum_k T_k \geq m(T_i - t_j)$. Equivalently,

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k.$$

But the value $\sum_k T_k$ is just the total load of all jobs, and so the quantity on the right-hand-side of this inequality is exactly our lower bound on the optimal value, from (10.1). Thus,

$$T_i - t_j \leq T^*.$$

But we also know that $t_j \leq T^*$ — here we use the other lower bound from (10.2). Adding up these two inequalities, we see that

$$T_i = (T_i - t_j) + t_j \leq 2T^*.$$

Since our makespan T is equal to T_i, this is the result we want. ■

It is not hard to give an example in which the solution is indeed close to a factor of 2 away from optimal. Suppose we have m machines and $n = m(m-1) + 1$ jobs. The first $m(n-1) = n - 1$ jobs each require time $t_j = 1$. The last job is much larger; it requires time $t_n = m - 1$. What does our greedy algorithm do with this sequence of jobs? It evenly balances the first $n-1$ jobs, and then has to add the giant job n to one of them; the resulting makespan is $T = 2(m-1)$.

What does the optimal solution look like in this example? It assigns the large job to one of the machines, say M_1, and evenly spreads the remaining jobs over the other $m - 1$ machines. This results in a makespan of m. Thus the ratio between the greedy algorithm's solution and the optimal solution is $2(m-1)/m = 2 - 2/m$, which is close to a factor of 2 when m is large. (In fact, with a little care, we could improve the analysis in (10.3) to show that the greedy algorithm with m machines is within exactly this factor of $2 - 2/m$ on every instance; the example above is really as bad as possible.)

An Improved Approximation

Now let's think about how we might develop a better approximation algorithm — in other words, one for which we are always within a factor of strictly less than 2 compared to the optimum. To do this, it helps to think about the worst cases for our current approximation algorithm. Essentially, the bad example above had the following flavor: we spread everything out very evenly across the machines, and then one last, giant, job arrives to mess everything up. Intuitively, it looks like it would help to get the largest jobs arranged nicely first, with the idea that later, small jobs can only do so much damage. And in fact, this idea does lead to a measurable improvement.

Thus, we now analyze the variant of the greedy algorithm which first sorts the jobs in decreasing order of processing time, and then proceeds as before. We will prove that the resulting assignment has a makespan that is at most 1.5 times the optimum.

```
Sorted-Balance
Start with no jobs assigned,
Set T_i = 0 and A(i) = ∅ for all machines M_i
Sort jobs in decreasing order of processing times t_j.
Assume that t_1 ≥ t_2 ≥ ... ≥ t_n
For j = 1, ..., n
  Let M_i be the machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
Endfor
```

The improvement comes from the following observation. If we have fewer than m jobs, then the greedy solution will clearly be optimal, since it puts each job on its own machine. And if we have more than m jobs, then the lower bound from (10.2) can strengthened.

(10.4) *If there are more than m jobs, then $T^* \geq 2t_{m+1}$.*

Proof. Consider only the first $m+1$ jobs in the sorted order. They each take at least t_{m+1} time. There are $m+1$ jobs and only m machines, so there must be a machine that gets assigned two of these jobs. This machine will have processing time at least $2t_{m+1}$. ■

(10.5) *Algorithm* `Sorted-Balance` *produces an assignment of jobs to machines with makespan $T \leq \frac{3}{2}T^*$.*

Proof. The proof will be very similar to the analysis of the previous algorithm. As before we will consider a machine M_i that has the maximum load. If M_i only holds a single job, then the schedule is optimal.

So let's assume that machine M_i has at least two jobs, and let t_j be the last job assigned to the machine. Note that $j \geq m+1$, since the algorithm will assign the first m jobs to m distinct machines. Thus, $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, where the second inequality is (10.4).

We now proceed as in the proof of (10.3), with the following single change. At the end of that proof, we had inequalities $T_i - t_j \leq T^*$ and $t_j \leq T^*$, and we added them up to get the factor of 2. But in our case here, the second of these inequalities is in fact $t_j \leq \frac{1}{2}T^*$; so adding the two inequalities gives us the bound

$$T_i \leq \frac{3}{2}T^*.$$

■

10.2 The Center Selection Problem

Consider the following scenario. We have a set S of n *sites*, say n little towns in upstate New York. We want to select k *centers* for building large shopping malls We expect that all of the n towns will shop at one of these malls, and we want to select the sites of the k malls to be central.

Let us start by defining the input to our problem more formally. We are given an integer k, a set S of n sites, and a distance function. What do we mean by distances? Distance $dist(s, z)$ can mean the travel time from point s to point z, or the physical distance, or driving distance (i.e., distance along the roads), or even the cost of traveling. We will assume that the distances satisfy the following natural properties:

- $dist(s, s) = 0$ for all $s \in S$.
- the distance is symmetric: $dist(s, z) = dist(z, s)$ for all sites $s, z \in S$.
- the triangle inequality: $dist(s, z) + dist(z, h) \geq dist(s, h)$.

The first and third of these properties are naturally satisfied by all notions of distance. Although there are applications with asymmetric distances (e.g., one-way streets), most applications also satisfy the second property. Our greedy algorithm will apply to any distance function that satisfies these three properties, though it will depend on all three.

Next we still have to clarify what we mean by the goal of wanting the centers to be central. Let C be a set of centers. Naturally, people of town s would shop at the closest mall. This suggests we define the distance of a site s to the centers as $dist(s, C) = \min_{c \in C} dist(s, c)$. We say that C forms an *r-cover* if all sites are within distance at most r from all from one of the centers, i.e., if

$$r \geq dist(s, C)$$

for all sites $s \in S$. The minimum r for which C is an r-cover will be called the *covering radius* of C and will be denoted by $r(C)$. Our goal will be to select a set of k centers that we call C with $r(C)$ as small as possible.

Next we discuss a greedy algorithm for this problem. As before, our meaning of "greedy" is necessarily a little fuzzy; we mean that we will select sites one-by-one in a myopic fashion, i.e., not think about where the remaining sites would go as we choose each one. We would put the first center at the best possible location for a single center, then add a new center always at the best location without any consideration of where future centers might go. We have not exactly clarified what "best" means here; but before spending too much time on this direction, we stop to show that this simple greedy approach is not a good idea: it can lead to really bad solutions.

Simple Greedy Algorithm Can Be Bad. To show that the simple greedy approach can be really bad, consider an example with only two sites s and z, and $k = 2$. Assume that s and z are at distance d from each other. The best location for a single center c_1 is half way between s and z, and the covering radius of this one center is $r(\{c_1\}) = d/2$. The greedy algorithm would start with c_1 as the first center. No matter where we add a second center, either s of z will have the center c_1 as closest, and so the covering radius of the set of two centers will still be $d/2$. Note that the optimum solution with $k = 2$ is to select s and z themselves as the centers. This will lead to a covering radius of 0. A more complex example that shows the same problem can be obtained by having two dense "clusters" of sites, one around s and one around z. Here, our proposed greedy algorithm would start by opening a center halfway between the clusters, while the optimum solution would open a separate center for each cluster.

Knowing the Optimal Radius Helps. Assume for a minute that someone told us what the optimum radius r is. Would this information help? Suppose we *know* that there is a set of k centers C' that have radius $r(C') \leq r$, and our job is to find some set of k centers C whose radius is not much more than r. It turns out that finding a set of k centers with radius at most $2r$ is can be done relatively easily.

Here is the idea: we can use the existence of this solution C' in our algorithm even though we do not know what C' is. Here is how. Consider any site $s \in S$. There must be a center $c' \in C'$ that covers site s, hence center c' is at distance at most r from s. Now our idea would be to take this site s as a center in our solution instead of c', as we have no idea what c' is. We would like to make s cover all the sites that c' used to cover in the unknown solution C'. This is accomplished by expanding the ratio from r to $2r$. All the sites that were at distance at most r from center c' are at distance at most $2r$ from s.

Let S' be the active sites, and set $S' = S$.

```
Let C = ∅
While S ≠ ∅
   Select any site s ∈ S' and add s to C.
   Delete all sites from S' that are at distance at most 2r from s.
EndWhile
If |C| ≤ k then
   Return C as the selected set of sites
Else
   Return claim "k centers cannot have covering radius at most r"
EndIf
```

If this algorithm returns a set of centers, than we have what we wanted

(10.6) *The set of centers C returned by the algorithm has covering radius $r(C) \leq 2r$.*

Next we argue that if the algorithm fails to return a set of centers, than its conclusion that no set can have covering radius at most r is indeed correct.

(10.7) *If the algorithm selects a set $|C| > k$ than for any set C' of size at most k the covering radius is $r(C') > r$.*

Proof. Assume the opposite, that there is a set C' of centers with covering radius $r(C') \leq r$. Each center $c \in C$ is one of the original sites in S, so there must be a center $c' \in C'$ that is at most r distance from c, i.e., $dist(c, c') \leq r$. We want to claim that each center in C has a different center in C' at most r away. This will imply that $|C'| \geq |C|$, so if there is a set of at most k centers C' with covering radius at most r, that our algorithm will select at most k centers.

To see that each center $c \in C$ has a separate center in $c' \in C'$ at most r away we notice that each pair is centers in C are at least $2r$ away from each other, as all closer sites were deleted after the first of the two was selected. So there cannot be any location c' that is at most r from two of the selected centers. ■

Greedy Algorithm That Works. Now we return to the original question: how do we select a good set of k centers *without* knowing what the optimal covering radius might be. Surprisingly, we can in essence run the above algorithm *without* knowing what r is. What the algorithm does is to select one of the original sites s as the new center, making sure that it is at least $2r$ away from all previously selected sites. We can do essentially this without knowing what r is: we select the site that is furthest away from all previously selected centers: if there are any active sites in S' in the original algorithm, than this furthest away site s is one of them. Here is the resulting algorithm, which is in essence a greedy algorithm.

```
Assume k ≤ |S| (else define C = S).
```

```
Select any site s and let C = {s}.
While |C| < k
   Select a site s ∈ S that maximizes dist(s, C)
   Add site s to C
EndWhile
Return C as the selected set of sites
```

(10.8) *The above greedy algorithm returns a set C of k points such that $r(C) \leq 2r(C')$, where C' is any other set of k points.*

Proof. Let $r = r(C')$ denote the minimum possible radius of a set of k centers. The proof is by contradiction. Assume that we obtained a set of k centers C with $r(C) > 2r$. Let s be a site that is more than $2r$ away from C.

Consider an iteration when we add new center c' to the set of previously selected centers C'. We claim that c' is at least $2r$ away from all sites in C'. This follows as site s is more than $2r$ away from all sites in the larger set C, and we select a site c that is the furthest site from all previously selected centers. More formally, we used the following chain of inequalities

$$dist(c', C') \geq dist(s, C') \geq dist(s, C) > 2r.$$

We got that our greedy algorithm is a correct implementation of the first k iteration of the while loop of the previous algorithm with parameter r. The previous algorithm would have $S' \neq \emptyset$ after selecting k centers, as we would have $s \in S'$ at this point, and so it would go on and select more than k centers, and eventually conclude that k centers cannot have covering radius at most r. This is contradicts to our choice of r, and the contradiction proves that $r(C) \leq 2r$. ■

Our greedy algorithm found a solution with covering radius at most twice the minimum possible. We call such an algorithm a 2-*approximation algorithm.*

Note the surprising fact that our final greedy 2-approximation algorithm is a very simple modification of the first greedy algorithm that did not work. Maybe the most important change is that our algorithm always selects the sites as centers (i.e., every mall will be built in one of the little towns, and not half way between two of them).

10.3 Set Cover: A General Greedy Heuristic

As our second topic in approximation algorithms, we consider a version of the *Set Cover* problem. Recall from our discussion of NP-completeness that the *Set Cover* problem is based on a set U of n elements, and a list $S_1, \ldots, S_m$ of subsets of U; we say that a *set cover* is a collection of these sets whose union is equal to all of U.

In the version of the problem we consider here, each set S_i has an associated *weight* $w_i \geq 0$. The goal is to find a set cover $\mathcal{C}$ so that the total weight

$$\sum_{S_i \in \mathcal{C}} w_i$$

is minimized. Note that this problem is at least as hard as the decision version of *Set Cover* we encountered earlier; if we set all $w_i = 1$, then the minimum weight of a set cover is at most k if and only if there is a collection of at most k sets that covers U.

We will develop and analyze a greedy algorithm for this problem. The algorithm will have the property that it builds the cover one set at a time; to choose its next set, it looks for one that seems to make the most progress toward the goal. What is a natural way to define "progress" in this setting? Desirable sets have two properties: they have small weight w_i, and they cover lots of elements. It is natural to combine these two criteria into the single measure $w_i/|S_i|$ — by selecting S_i, we cover $|S_i|$ elements at a cost of w_i, and so this ratio gives the "cost per element covered," a very reasonable thing to use as a guide.

Of course, once some sets have already been selected, we are only concerned with the how we are doing on the elements still left uncovered. So we will maintain the set R of remaining uncovered elements, and choose the set S_i that minimizes $w_i/|S_i \cap R|$.

```
Greedy-Set-Cover
Start with R = U and no sets selected
While R ≠ ∅
  Select set S_i that minimizes w_i/|S_i ∩ R|
  Delete set S_i from R
Endwhile
Return the selected sets
```

The sets selected by the algorithm clearly form a set cover. The question we want to address is: How much larger is the weight of this set cover than the weight w^* of an optimal set cover?

As in the previous section, our analysis will require a good lower bound on this optimum. In the case of the load balancing problem we used lower bounds that emerged naturally from the statement of the problem: the average load, and the maximum job size. The *Set Cover* cover problem will turn out to be more subtle; "simple" lower bounds are not very useful, and instead we will use a lower bound that the greedy algorithm implicitly constructs as a by-product.

Recall the intuitive meaning of the ratio $w_i/|S_i \cap R|$ used by the algorithm; it is the "cost paid" for covering each new element. Let's record this cost paid for element s in the quantity c_s. We add the following line to the code immediately after selecting the set S_i.

```
Set c_s = w_i/|S_i ∩ R| for all s ∈ S_i ∩ R
```

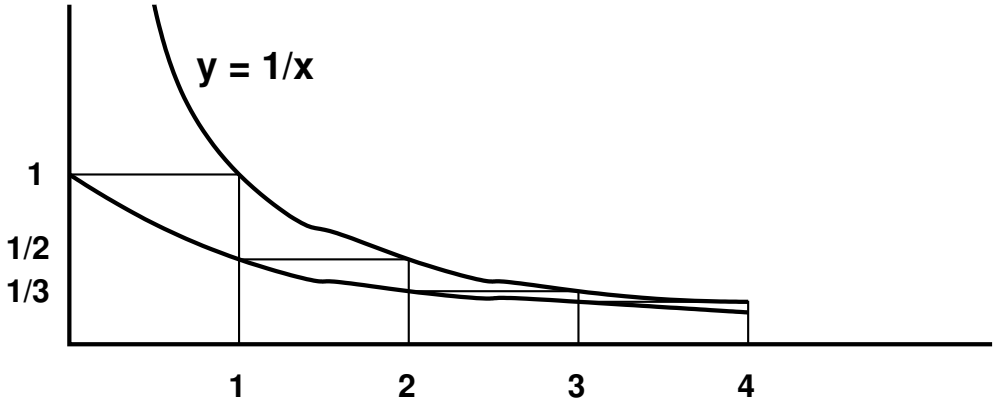


Figure 10.1: Upper and lower bounds for the Harmonic Function $H(n)$.

The values c_s do not affect the behavior of the algorithm at all; we view them as a bookkeeping device to help in our comparison to the optimum w^*. As each set S_i is selected, its weight is distributed over the costs c_s of the elements that are newly covered. Thus, these costs completely account for the total weight of the set cover, and so we have

(10.9) *If $\mathcal{C}$ is the set cover obtained by* `Greedy-Set-Cover` *then $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$.*

The key to the analysis is to ask how much total cost any single set S_k can account for — in other words, to give a bound on $\sum_{s \in S_k} c_s$ relative to the weight w_k of the set, even for sets not selected by the greedy algorithm. Giving an upper bound on the ratio

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

that holds for every set says, in effect, "To cover a lot of cost, you must use a lot of weight." We know that the optimum solution must cover the full cost $\sum_{s \in U} c_s$ via the sets it selects; so this type of bound will establish that it needs to use at least a certain amount of weight. This is a lower bound, just as we need for the analysis.

Our analysis will use the *Harmonic Function*

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}.$$

To understand its asymptotic size as a function of n, we can interpret it as a Riemann sum approximating the area under the curve $y = 1/x$. Figure 10.1 shows how it is naturally bounded above by $1 + \int_1^n dx/x = 1 + \ln n$, and bounded below by $\int_1^{n+1} dx/x = \ln(n+1)$. Thus we see that $H(n) = \Theta(\ln n)$.

Here is the key to establishing a bound on the performance of the algorithm.

(10.10) *For every set S_k, the sum $\sum_{s \in S_k} c_s$ is at most $H(|S_k|) \cdot w_k$.*

Proof. To simplify the notation, we will assume that the elements of S_k are the first $d = |S_k|$ elements of the set U; that is, $S_k = \{s_1, \ldots, s_d\}$. Furthermore, let us assume that these elements are labeled in the order in which they are assigned a cost c_{s_j} by the greedy algorithm (with ties broken arbitrarily). There is no loss of generality in doing this, since it simply involves a renaming of the elements in U.

Now consider the iteration in which element s_j is covered by the greedy algorithm, for some $j \leq d$. At the start of this iteration, $s_j, s_{j+1}, \ldots, s_d \in R$ by our labeling of the elements. This implies that $|S_k \cap R|$ is at least $d - j + 1$, and so the average cost of the set S_k is at most

$$\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

Note that this is not necessarily an equality, since s_j may be covered in the same iteration as some of the other elements $s_{j'}$ for $j' < j$. In this iteration, the greedy algorithm selected a set S_i of minimum average cost; so this set S_i has average cost at most that of S_k. It is the average cost of S_i that gets assigned to s_j, and so we have

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

We now simply add up these inequalities for all elements $s \in S_k$:

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \ldots + \frac{w_k}{1} = H(d) \cdot w_k.$$

■

We now continue with our plan for using the bound in (10.10) to compare the greedy algorithm's set cover to the optimal one. The optimum solution pays w_i for including certain sets S_i. The greedy algorithm pays at most a factor of $H(|S_i|)$ more than this to cover all the elements in S_i. Letting $d^* = \max_i |S_i|$ denote the maximum size of any set, we have the following approximation result.

(10.11) *The set cover $\mathcal{C}$ selected by* `Greedy-Set-Cover` *has weight at most $H(d^*)$ times the optimal weight w^*.*

Proof. Let $\mathcal{C}^*$ denote the optimum set cover. We have that $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$. For each of these sets, (10.10) implies

$$w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s,$$

and because they form a set cover, we have

$$\sum_{S_i \in \mathcal{C}^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s.$$

Combining these with (10.9), we obtain the desired bound:

$$w^* = \sum_{S_i \in \mathcal{C}^*} w_i \geq \sum_{S_i \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i.$$

■

10.4 Vertex Cover: An Application of Linear Programming

Finally, we consider a version of the *Vertex Cover* problem. As we develop an approximation algorithm, we will focus on two additional issues. First, *Vertex Cover* is easily reducible to *Set Cover*, and so we have to consider some of the subtle ways in which approximation results interact with polynomial-time reductions. Second, we will use this opportunity to introduce a powerful technique from operations research — *linear programming*. Linear programming is the subject of entire courses, and we will not be attempting to provide any kind of comprehensive overview of it here. (You can learn more about linear programming and its applications in courses in the OR&IE department.) In this section, we will introduce some of the basic ideas underlying linear programming, and show how these can be used to approximate NP-hard optimization problems.

Recall that a *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S. In the version of the problem we consider here, each vertex $i \in V$ has a *weight* $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a vertex cover S for which $w(S)$ is minimum. Even when all weights are equal to 1, deciding if there is a vertex cover of weight at most k is the standard decision version of *Vertex Cover*.

Approximations via Reductions? First consider the special case in which all weights are equal to 1 — i.e., we are looking for a vertex cover of minimum size. We will call this the *unweighted case*. Recall that we showed *Set Cover* to be NP-complete using a reduction from the decision version of unweighted *Vertex Cover*. That is,

Vertex Cover $\leq_P$ *Set Cover*

This reduction says: "If we had a polynomial time algorithm that solves the *Set Cover* problem, then we could use this algorithm to solve the *Vertex Cover* problem in polynomial time". Now we have a polynomial time algorithm for the *Set Cover* problem that approximates the solution. Does this imply that we can use it to formulate an approximation algorithm for *Vertex Cover*?

(10.12) *One can use the* Set Cover *approximation algorithm to give an $H(d)$-approximation algorithm for the unweighted* Vertex Cover *problem, where d is the maximum degree of the graph.*

Proof. The proof is based on the reduction that showed *Vertex Cover* $\leq_P$ *Set Cover*. Consider an instance of the unweighted *Vertex Cover* problem, specified by a graph $G = (V, E)$. We define an instance of *Set Cover* as follows. The underlying set U is equal to E. For each node i, we define a set S_i consisting of all edges incident to node i. Collections of sets that cover U now correspond precisely to vertex covers. Note that the maximum size of any S_i is precisely the maximum degree d.

Hence we can use the approximation algorithm for *Set Cover* to find a vertex cover whose size is within a factor of $H(d)$ of minimum. ■

This $H(d)$-approximation is quite good when d is small; but it gets worse as d gets larger, approaching a bound that is logarithmic in the number of vertices. Below, we will develop a stronger approximation algorithm that comes within a factor of 2 of optimal.

Before turning to the 2-approximation algorithm, we pause to make the following observation: One has to be careful when trying to use reductions for designing approximation algorithms. Here is a cautionary example. We used *Independent Set* to prove that the *Vertex Cover* problem is NP-complete. Specifically, we proved

Independent Set $\leq_P$ *Vertex Cover*,

which states that "if we had a polynomial time algorithm that solves the *Vertex Cover* problem then we could use this algorithm to solve the *Independent Set* problem in polynomial time". Can we use an approximation algorithm for the minimum-size vertex cover to design a comparably good approximation algorithm for the maximum-size independent set?

The answer is no. Recall that a set I of vertices is independent if an only if its complement $S = V - I$ is a vertex cover. Given a minimum-size vertex cover S^*, we obtain a maximum-size independent set by taking the complement $I^* = V - S$. Now suppose we use an approximation algorithm for the vertex cover problem to get an approximately minimum vertex cover S. The complement $I = V - S$ is indeed an independent set — there's no problem there. The trouble is when we try to determine our approximation factor for the maximum independent set problem; I can be very far from optimal. Suppose, for example, that the optimal vertex cover S^* and the optimal independent set I^* both have size $|V|/2$. If we invoke a 2-approximation algorithm for the vertex cover problem, we may perfectly well get back the set $S = V$. But in this case, our "approximately maximum independent set' $I = V - S$ has no elements.

Linear Programming as a General Technique

Our 2-approximation algorithm for the weighted version of *Vertex Cover* will be based on linear programming. As suggested above, we describe linear programming here not just to give the approximation algorithm, but also to illustrate its power as a very general technique.

So what is linear programming? To answer this, it helps to first recall, from linear algebra, the problem of simultaneous linear equations. Using matrix-vector notation, we

have a vector x of unknown real numbers, a given matrix A, and a given vector b; and we want to solve the equation $Ax = b$. Gaussian elimination is a well-known efficient algorithm for this problem.

The basic *Linear Programming* problem can be viewed as a more complex version of this, with inequalities in place of equations. Specifically, consider the problem of determining a vector x that satisfy $Ax \geq b$. By this notation, we mean that each coordinate of the vector Ax should be greater than or equal to the corresponding coordinate of the vector b. Such systems of inequalities define regions in space. For example, suppose $x = (x_1, x_2)$ is a 2-dimensional vector, and we have the three inequalities $x_1 \geq 0$, $x_2 \geq 0$ and $x_1 + x_2 \leq 1$. Then set the of solutions is a triangle in the plane, with corners at $(1, 0), (0, 1)$ and $(0, 0)$.

Given a region defined by $Ax \geq b$, linear programming seeks to minimize a linear combination of the coordinates of x, over all x belonging to the region. Such a linear combination can be written $c^t x$, where c is a vector of coefficients, and $c^t x$ denotes the inner product of two vectors. Thus our standard form for *Linear Programming*, as an optimization problem, will be the following.

> Given an m by n matrix A, and vectors $b \in R^m$ and $c \in R^n$, find a vector $x \in R^n$ to solve the following optimization problem:
>
> $$\min(c^t x \text{ such that } x \geq 0; Ax \geq b).$$

$c^t x$ is often called the *objective function* of the linear program, and $Ax \geq b$ is called the set of *constraints*. We can phrase *Linear Programming* as a decision problem in the following way.

> Given a matrix A, vectors b and c, and a bound γ, does there exist x so that $x \geq 0, Ax \geq b$ and $c^t x \leq \gamma$?

To avoid issues related to how we represent real numbers, we can assume that the coordinates of the vectors and matrices involved are integers.

The decision version of *Linear Programming* is in $\mathcal{NP}$. This is intuitively very believable — we just have to exhibit a vector x satisfying the desired properties. The one concern is that even if all the input numbers are integers, x may contain rational numbers requiring very large precision — how do we know that we'll be able to read and manipulate it in polynomial time? But in fact, one can show that if there is a solution, then there is one that needs only a polynomial number of bits to write down; so this is not a problem.

Linear Programming was also known to be in co$-\mathcal{NP}$ for a long time, though this is not so easy to see. Students who have taken a linear programming course before may notice that this fact follows from linear programming duality. For a long time, indeed, *Linear Programming* was the most famous example of a problem in both $\mathcal{NP}$ and in co$-\mathcal{NP}$ that

was not known to have a polynomial-time solution. In 1981 Leonid Khachiyan, who at a time was a young researcher in the Soviet Union, gave a polynomial-time algorithm for the problem. His initial algorithm was quite slow and impractical; but since then practical polynomial-time algorithms — so-called *interior point methods* — have also been developed following the work of Narendra Karmarkar in 1984.

Linear programming is an interesting example also for another reason. The most widely used algorithm for this problem is the *simplex method.* It works very well in practice, and is competitive with polynomial-time interior methods on real-world problems. Yet its worst-case running time is known to be exponential — apparently this exponential behavior does not show up in practice.

In summary, linear programming problems can be solved in polynomial time. You can learn a lot more about all this in OR&IE courses on linear programming. The question we ask here is this: How can linear programming help us when we want to solve combinatorial problems like *Vertex Cover*?

Vertex Cover as an Integer Program

We now try to formulate a linear program that is in close correspondence with the vertex cover problem. Thus, we consider a graph $G = (V, E)$ with a weight $w_i \geq 0$ on each node i. Linear programming is based on the use of vectors of variables; in our case, we will have a *decision variable* x_i for each node $i \in V$ to model the choice of whether to include node i in the vertex cover. $x_i = 0$ will indicate that node i is not in the vertex cover, and $x_i = 1$ will indicate that node i is in the vertex cover. We can create a single n-dimensional vector x in which the i^{th} coordinate corresponds to the i^{th} decision variable x_i.

We use linear inequalities to encode the requirement that the selected nodes form a vertex cover; we use the objective function to encode the goal of minimizing the total weight. For each edge $(i, j) \in E$, it must have one end in the vertex cover, and we write this as the inequality $x_i + x_j \geq 1$. Finally, to express the minimization problem, we write the set of node weights as an n-dimensional vector w, with the i^{th} coordinate corresponding to w_i; we then seek to minimize $w^t x$. In summary, we have formulated the *Vertex Cover* problem as follows.

$$\begin{array}{lllll} \text{(VCIP)} & \text{Min} & \sum_{i \in V} w_i x_i & & \\ & \text{s.t.} & x_i + x_j \geq 1 & & (i,j) \in E \\ & & x_i \in \{0,1\} & & i \in V. \end{array}$$

We claim that the vertex covers of G are in one-to-one correspondence with the solutions x to this system of linear inequalities in which all coordinates are equal to 0 or 1.

(10.13) *S is a vertex cover in G if and only if the vector x defined as $x_i = 1$ for $i \in S$, and $x_i = 0$ for $i \notin S$ satisfies the constraints in (VCIP). Further, we have that $w(S) = w^t x$.*

We can put this system into the matrix form we used for linear programming as follows. We define a matrix A whose columns correspond to the nodes in V and whose rows correspond to the edges in E; entry $A[e, i] = 1$ if node i is an end of the edge e, and 0 otherwise. (Note that each row has exactly two non-zero entries.) If we use $\vec{1}$ to denote the vector of with all coordinates equal to 1, and $\vec{0}$ to denote the vector of with all coordinates equal to 0, then the above system of inequalities can be written as

$$Ax \geq \vec{1}$$

$$\vec{1} \geq x \geq \vec{0}.$$

But keep in mind that this is not just a linear programming problem — we have crucially required that all coordinates in the solution vector be either 0 or 1. So our formulation suggests that we should solve the problem

$$\min(w^t x \text{ subject to } \vec{1} \geq x \geq \vec{0}, Ax \geq \vec{1},\ x \text{ has integer coordinates.})$$

This is an instance of a linear programming problem in which we require the coordinates of x to take integer values; without this extra constraint, the coordinates of x could be arbitrary real numbers. We call this problem *Integer Programming*, as we are looking for integer-valued solutions to a linear program.

Integer Programming is considerably harder than *Linear Programming*; indeed, the discussion above really constitutes a reduction from *Vertex Cover* to the decision version of *Integer Programming*. In other words, we have proved

(10.14) Vertex Cover $\leq_P$ Integer Programming.

To show the NP-completeness of *Integer Programming*, we would still have to establish that the decision version is in $\mathcal{NP}$. There is a complication here, as with *Linear Programming*, since we need to establish that there is always a solution x that can be written using a polynomial number of bits. But this can indeed be proven. Of course, for our purposes, the integer program we are dealing with is explicitly constrained to have solutions in which each coordinate is either 0 or 1. Thus it is clearly in $\mathcal{NP}$, and our reduction from *Vertex Cover* establishes that even this special case is NP-complete.

Using Linear Programming for Vertex Cover

We have yet to resolve whether our foray into linear and integer programming will turn out be useful, or simply a dead end. Trying to solve the integer programming problem (VCIP) optimally is clearly not the right way to go, as this is NP-hard.

The way to make progress is to exploit the fact that linear programming is not as hard as integer programming. Suppose we take (VCIP) and modify it, dropping the requirement

that each $x_i \in \{0, 1\}$ and reverting to the constraint that each x_i is an arbitrary real number between 0 and 1. This gives us a linear programming problem that we could call (VCLP), and we can solve it in polynomial time: we can find a set of values $\{x_i^*\}$ between 0 and 1 so that $x_i^* + x_j^* \geq 1$ for each edge (i, j), and $\sum_i w_i x_i^*$ is minimized. Let x^* denote this vector, and $w_{LP} = w^t x^*$ denote the value of the objective function.

We note the following basic fact.

(10.15) *Let S^* denote a vertex cover of minimum weight. Then $w_{LP} \leq w(S^*)$.*

Proof. Vertex covers of G correspond to integer solutions of (VCIP), so the minimum of $\min(w^t x : \vec{1} \geq x \geq 0, Ax \geq 1)$ over all integer x vectors is exactly the minimum weight vertex cover. To get the minimum of the linear program (VCLP) we allow x to take arbitrary real-number values — i.e., we minimize over many more choices of x — and so the minimum of (VCLP) is no larger than that of (VCIP). ■

Note that (10.15) is one of the crucial ingredients we need for an approximation algorithm — a good lower bound on the optimum, in the form of the efficiently computable quantity w_{LP}.

However, w_{LP} can definitely be smaller than $w(S^*)$. For example, if the graph G is a triangle and all weights are 1, then the minimum vertex cover has a weight of 2. But in a linear programming solution we can set $x_i = \frac{1}{2}$ for all three vertices, and so get a linear programming solution of weight only $\frac{3}{2}$. As a more general example, consider a graph on n nodes, with each pair of nodes connected by an edge. Again, all weights are 1. Then the minimum vertex cover has weight $n - 1$, but we can find a linear programming solution of value $n/2$ by setting $x_i = \frac{1}{2}$ for all vertices i.

So the question is: how can solving this linear program help us actually *find* a near-optimal vertex cover. The idea is to work with the values x_i^*, and infer a vertex cover S from them. It is natural that if $x_i^* = 1$ for some node i, then we should put it in the vertex cover S; and if $x_i^* = 0$, then we should leave it out of S. But what should we do with fractional values in between? What should we do if $x_i^* = .4$ or $x_i^* = .5$? The natural approach here is to *round*. Given a fractional solution $\{x_i^*\}$, we define $S = \{i \in V : x_i^* \geq \frac{1}{2}\}$ — that is, we round values at least $\frac{1}{2}$ up, and those below $\frac{1}{2}$ down.

(10.16) *The set S defined in this way is a vertex cover, and $w(S) \leq 2w^t x^*$.*

Proof. First we argue that S is a vertex cover. Consider an edge $e = (i, j)$. We claim that at least one of i and j must be in S. Recall that one of our inequalities is $x_i + x_j \geq 1$. So in any solution x^* that satisfies this inequality, either $x_i^* \geq \frac{1}{2}$ or $x_j^* \geq \frac{1}{2}$. Thus, at least one of these two will be rounded up, and i or j will placed in S.

Now we consider the weight $w(S)$ of this vertex cover. The set S only has vertices with $x_i^* \geq \frac{1}{2}$; thus the linear program "paid" at least $\frac{1}{2}w_i$ for node i, and we only pay w_i — at most twice as much. More formally, we have the following chain of inequalities.

$$w^t x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S).$$

■

Thus, we have a produced a vertex cover S of weight at most $2w_{LP}$. The lower bound in (10.15) showed that the optimal vertex cover has weight at least w_{LP}, and so we have the following result.

(10.17) *The algorithm described above produces a vertex cover S of at most twice the minimum possible weight.*

10.5 Arbitrarily Good Approximations for the Knapsack Problem

Often, when you talk to someone faced with an NP-complete optimization problem, they're hoping you can give them something that will produce a solution within, say, 1% of the optimum — or at least, within a small percentage of optimal. Viewed from this perspective, the approximation algorithms we've seen thus far come across as quite weak: solutions within a factor of 2 of the minimum for *Center Selection* and *Vertex Cover*(i.e. 100% more than optimal). The set-cover algorithm in Section 10.3 is even worse: its cost is not even within a fixed constant factor of the minimum possible!

Here is an important point underlying this state of affairs: NP-complete problems, as you well know, are all equivalent with respect to polynomial-time solvability; but assuming $\mathcal{P} \neq \mathcal{NP}$, they differ considerably in the extent to which their solutions can be efficiently approximated. In some cases, it is actually possible to prove limits on approximability. For example, if $\mathcal{P} \neq \mathcal{NP}$, then the guarantee provided by our *Center Selection* algorithm is the best possible for any polynomial-time algorithm. Similarly, the guarantee provided by the *Set Cover* algorithm, however bad it may seem, is very close to the best possible, unless $\mathcal{P} = \mathcal{NP}$. For other problems, such as the Vertex Cover problem, the approximation algorithm we gave is the best known, but it is an open question whether there could be polynomial-time algorithms with better guarantees. We will not discuss the topic of lower bounds on approximability in this course; while some lower bounds of this type are not so difficult to prove (such as for *Center Selection*), many are extremely technical.

In this lecture, we discuss an NP-complete problem for which it is possible to design a polynomial-time algorithm providing a very strong approximation. We will consider a

slightly more general version of the *Knapsack* (or *Subset Sum*) problem. Suppose you have n items that you consider packing in a knapsack. Each item $i = 1, \ldots, n$ has two integer parameters: a weight w_i and a value v_i. Given a knapsack capacity W, the goal of the *Knapsack Problem* is to find a subset S of items of maximum value subject to the restriction that the total weight of the set should not exceed W. In other words, we wish to maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leq W$.

How strong an approximation can we hope for? Our algorithm will take as input the weights and values defining the problem, and will also take an extra parameter ϵ, the desired precision. It will find a subset S whose total weight does not exceed W, with value $\sum_{i \in S} v_i$ at most an $(1+\epsilon)$ factor below the maximum possible. The algorithm will run in polynomial time for any *fixed* choice of $\epsilon > 0$; however, the dependence on ϵ will not be polynomial. We call such an algorithm a *polynomial time approximation scheme*.

You may ask: how could such a strong kind of approximation algorithm be possible in polynomial time when the *Knapsack* problem is NP-hard? With integer values, if we get close enough to the optimum value, we must reach the optimum itself! The catch is in the non-polynomial dependence on the desired precision: for any fixed choice of ϵ, such as $\epsilon = .5$, $\epsilon = .2$ or even $\epsilon = .01$ the algorithm runs in polynomial time, but as we change ϵ to smaller and smaller values, the running time gets larger. By the time we make ϵ small enough to make sure we get the optimum value, it is no longer a polynomial time algorithm.

The Approximation Algorithm

Earlier in this class we considered the subset sum problem, the special case of the *Knapsack* problem when $v_i = w_i$ for all items i. We gave a dynamic programming algorithm for this special case that ran in $O(nW)$ time assuming the weights are integers. This algorithm naturally extends to the more general *Knapsack* problem (see the end of Section 5.4 for this extension). The algorithm given in Section 5.4 works well when the weights are small (even if the values may be big). It is also possible to extend our dynamic programming algorithm for the case when the values are small, even if the weights may be big. At the end of this handout we give a dynamic programming algorithm for that case running in time $O(n^2 v^*)$, where $v^* = \max_i v_i$. Note that this algorithm does not run in polynomial time: it is only pseudo-polynomial, because of its dependence on the size of the values v_i.

Algorithms that depend on the values in a pseudo-polynomial way can often be used to design polynomial time approximation schemes, and the algorithm we develop here is a very clean example of the basic strategy. In particular, we will use the dynamic programming algorithm with running time $O(n^2 v^*)$ to design a polynomial time approximation scheme; the idea is as follows. If the values are small integers, then v^* is small and the problem can be solved in polynomial time already. On the other hand, if the values are large, then we do not have to deal with them exactly, as we only want an approximately optimum solution.

We will use a rounding parameter v (whose value we'll set later), and will consider the values rounded to an integer multiple of v. We will use our dynamic programming algorithm to solve the problem with the rounded values. More precisely, for each item i let its rounded value be $\tilde{v}_i = \lceil v_i/v \rceil v$. Note that the rounded and the original value are quite close to each other.

(10.18) *For each item i we have that $v_i \leq \tilde{v}_i \leq v_i + v$. For the item j of maximum value $v_j = \max_i v_i$ we have $v_j = \tilde{v}_j$.*

What did we gain by the rounding? If the values were big to start with, we did not make them any smaller. However, the rounded values are all integer multiples of a common value v. So instead of solving the problem with the rounded values $\tilde{v}_i$ we can change the units; we can divide all values by v and get an equivalent problem. Let $\hat{v}_i = \tilde{v}_i/v = \lceil v_i/v \rceil$ for $i = 1, \ldots, n$.

(10.19) *The* Knapsack *problem with values $\tilde{v}_i$ and the scaled problem with values $\hat{v}_i$ has the same optimum solution, the optimum value differs exactly by a v factor, and the scaled values are integral.*

Now we are ready to state our approximation algorithm. We will assume that all items have weight at most W (as items with weight $w_i > W$ are not in any solution, and hence can be deleted).

```
Knapsack-Approx(ε).
  Set v = (ε/n) max_i v_i.
  Solve the Knapsack problem with values v̂_i (or equivalently ṽ_i).
  Return the set S of items found.
```

First note that the solution found is at least feasible; that is, $\sum_{i \in S} w_i \leq W$. This is true as we have rounded only the values and not the weights. This is why we had to consider the more general *Knapsack* problem in this section, rather than the simpler subset sum problem where the weights and values are equal.

(10.20) *The set it items S returned by the algorithm has total weight at most W, $\sum_{i \in S} w_i \leq W$.*

Next we'll prove that this algorithm runs in polynomial time.

(10.21) *The algorithm Knapsack-Approx runs in polynomial time for any fixed $\epsilon > 0$.*

Proof. Setting v and rounding item values can clearly be done in polynomial time. The time-consuming part of this algorithm is the dynamic programming to solve the rounded problem. Recall that for problem with integer values, the dynamic programming algorithm we use runs in time $O(n^2 v^*)$, where $v^* = \max_i v_i$.

Now, we are applying this algorithms for an instance in which each item i has weight w_i and value $\hat{v}_i$. To determine the running time, we need to determine $\max_i \hat{v}_i$. The item j with maximum value $v_j = \max_i v_i$ also has maximum value in the rounded problem, so $\max_i \hat{v}_i = \hat{v}_j = \lceil v_j/v \rceil = n\epsilon^{-1}$. Hence, the overall running time of the algorithm is $O(n^3\epsilon^{-1})$. Note that this is polynomial time for any fixed $\epsilon > 0$ as claimed; but the dependence on the desired precision ϵ is not polynomial as the running time includes ϵ^{-1} rather than $\log \epsilon^{-1}$. ■

Finally, we need to consider the key issue: How good is the solution obtained by this algorithm? (10.18) shows that the values $\tilde{v}_i$ we used are close to the real values v_i, and this suggests that the solution obtained may not be far from optimal.

(10.22) *If S is the solution found by the* `Knapsack-Approx` *algorithm, and S^* is any other solution satisfying $\sum_{i \in S^*} w_i \leq W$, then we have $(1+\epsilon)\sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.*

Proof. Let S^* be any set satisfying $\sum_{i \in S^*} w_i \leq W$. Our algorithm finds the optimal solution with values $\tilde{v}_i$, so we know that

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{i \in S^*} \tilde{v}_i.$$

The rounded values $\tilde{v}_i$ and the real values v_i are quite close by (10.18) so we get the following chain of inequalities.

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} (v_i + v) \leq nv + \sum_{i \in S} v_i,$$

showing that the value $\sum_{i \in S} v_i$ of the solution we obtained is at most nv smaller than the maximum value possible. We wanted to obtain a relative error showing that the value obtained, $\sum_{i \in S} v_i$ is at most an $(1+\epsilon)$ factor less than the maximum possible, so we need to compare nv to the value $\sum_{i \in S} v_i$. Recall from (10.18) that the item j with largest value satisfies $v_j = \tilde{v}_j$. By our assumption that each item alone fits in the knapsack (that is, $w_i \leq W$ for all items i), we get that $\sum_{i \in S} v_i \geq \tilde{v}_j = \max_i v_i$, and hence $nv \leq \epsilon \sum_{i \in S} v_i$. Using this bound, we conclude with

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + nv \leq (1+\epsilon) \sum_{i \in S} v_i,$$

as claimed. ■

The new dynamic programming algorithm

To solve a problem by dynamic programming we have to define a polynomial set of subproblems. The dynamic programming algorithm we defined when we studied the Knapsack problem earlier uses subproblems of the form $OPT(i, w)$: the subproblem of finding the maximum value of any solution using a subset of the items $1, \ldots, i$ and a knapsack of weight w. When the weights are large, this is a large set of problems. We need a set of subproblems that work well when the values are reasonably small; this suggests that should use subproblems associated with values, not weights. We define our subproblems as follows. The subproblem is defined by i and a target value v, and $\overline{OPT}(i, V)$ is the smallest knapsack weight W so that one can obtain a solution using a subset of items $\{1, \ldots, i\}$ with value at least V. We will have a subproblem for all $i = 0, \ldots, n$ and values $V = 0, \ldots, \sum_{j=1}^{i} v_j$. If v^* denotes $\max_i v_i$, then we see that the largest V can get in a sub-problem is $\sum_{j=1}^{n} v_j \leq nv^*$. Thus, assuming the values are integral, there are at most $O(n^2 v^*)$ subproblems. None of these subproblems is the original instance of *Knapsack* but if we have the values of all subproblems $\overline{OPT}(n, V)$ for $V = 0, \ldots, \sum_i v_i$, then the value of the original problem can be obtained easily: it is the largest value V such that $\overline{OPT}(n, V) \leq W$.

It is not hard to give a recurrence for solving these subproblems. By analogy with the dynamic programming algorithm for *Subset Sum*, we consider cases depending whether or not the last item n is included in the optimal solution $\mathcal{O}$.

- If $n \notin \mathcal{O}$ then $\overline{OPT}(n, V) = \overline{OPT}(n-1, V)$.
- If $n \in \mathcal{O}$ is the only item in $\mathcal{O}$ then $\overline{OPT}(n, V) = w_n$
- If $n \in \mathcal{O}$ is not the only item in $\mathcal{O}$ then $\overline{OPT}(n, V) = w_n + \overline{OPT}(n-1, V - v_n)$.

These last two options can be summarized more compactly as

- If $n \in \mathcal{O}$ then $\overline{OPT}(n, V) = w_n + \overline{OPT}(n-1, \max(0, V - v_n))$.

This implies the following analogue of the recurrence (5.10) in the course packet.

(10.23) *If $V > \sum_{i=1}^{n-1} v_i$ then $\overline{OPT}(n, V) = w_n + \overline{OPT}(n-1, V - v_n)$. Otherwise,*

$$\overline{OPT}(n, V) = \min(\overline{OPT}(n-1, V), w_n + \overline{OPT}(n-1, \max(0, V - v_n))).$$

We can then write down an analogous dynamic programming algorithm.

```
Knapsack(n, V)
  Array M[0...n, 0...V]
  For i = 0, ..., n
    M[i, 0] = 0
  Endfor
```

```
For i = 1, 2, ..., n
  For v = 0, ..., Σ_{j=1}^{i} v_j
    If v > Σ_{j=1}^{i-1} v_j then
      M[i, v] = w_i + M[i − 1, v]
    Else
      M[i, v] = min(M[i − 1, v], w_i + M[i − 1, max(0, v − v_i)])
    Endif
  Endfor
Endfor
Return the maximum value v such that M[n, V] ≤ W
```

(10.24) `Knapsack`(n, V) *takes* $O(nV)$ *time, and correctly computes the optimal values of the subproblems.*

As was done before we can trace back through the table M containing the optimal values of the subproblems, to find an optimal solution in $O(n)$ time.

10.6 Exercises

1. Consider the following *greedy algorithm* for finding a matching in a bipartite graph:

   > As long as there is an edge whose endpoints are unmatched, add it to the current matching.

   (a) Give an example of a bipartite graph G for which this greedy algorithm does not return the maximum matching.

   (b) Let M and M' be matchings in a bipartite graph G. Suppose that $|M'| > 2|M|$. Show that there is an edge $e' \in M'$ such that $M \cup \{e'\}$ is a matching in G.

   (c) Use (b) to conclude that any matching constructed by the greedy algorithm in a bipartite graph G is at least *half* as large as the maximum matching in G.

2. Recall the *Shortest-First* greedy algorithm for the Interval Scheduling problem: Given a set of intervals, we repeatedly pick the shortest interval I, delete all the other intervals I' that intersect I, and iterate.

   In class, we saw that this algorithm does *not* always produce a maximum-size set of non-overlapping intervals. However, it turns out to have the following interesting approximation guarantee. If s^* is the maximum size of a set of non-overlapping intervals, and s is the size of the set produced by the *Shortest-First* algorithm, then $s \geq \frac{1}{2}s^*$. (That is, *Shortest-First* is a 2-approximation.)

   Prove this fact.

3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B.$$

The sum of the numbers in S will be called the *total sum* of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

(a) Here is an algorithm for this problem.

```
Initially S = φ
Define T = 0
For i = 1, 2, ..., n
  If T + a_i ≤ B then
    S ← S ∪ {a_i}
    T ← T + a_i
  Endif
Endfor
```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$.

You should give a proof that your algorithm has this property, and give a brief analysis of its running time.

(c) In this part, we want you to give an algorithm with a *stronger* guarantee than what you produced in part (b). Specifically, give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least $(2/3)$ times as large as the maximum total sum of any feasible set $S' \subseteq A$.

You should give a proof that your algorithm has this property, and give a brief analysis of its running time.

4. In the *bin packing problem*, we are given a collection of n items with *weights* $w_1, w_2, \ldots, w_n$. We are also given a collection of *bins*, each of which can hold a total of W units of weight. (We will assume that W is at least as large as each individual w_i.)

You want to pack each item in a bin; a bin can hold multiple items, as long as the total of weight of these items does not exceed W. The goal is to pack all the items using as few bins as possible.

Doing this optimally turns out to be NP-complete, though you don't have to prove this.

Here's a *merging heuristic* for solving this problem: We start with each item in a separate bin and then repeatedly "merge" bins if we can do this without exceeding the weight limit. Specifically:

```
Merging Heuristic:
  Start with each item in a different bin
  While there exist two bins so that the union of
        their contents has total weight ≤ W
     Empty the contents of both bins
     Place all these items in a single bin.
  Endwhile
  Return the current packing of items in bins.
```

Notice that the merging heuristic sometimes has the freedom to choice several possible pairs of bins to merge. Thus, on a given instance, there are multiple possible executions of the heuristic.

Example. Suppose we have four items with weights $1, 2, 3, 4$, and $W = 7$. Then in one possible execution of the merging heuristic, we start with the items in four different bins; then we merge the bins containing the first two items; then we merge the bins containing the latter two items. At this point we have a packing using two bins, which cannot be merged. (Since the total weight after merging would be 10, which exceeds $W = 7$.)

(a) Let's declare the size of the input to this problem to be proportional to

$$n + \log W + \sum_{i=1}^{n} \log w_i.$$

(In other words, the number of items plus the number of bits in all the weights.)

Prove that the merging heuristic always terminates in time polynomial in the size of the input. (In this question, as in NP-complete number problems from class, you should account for the time required to perform any arithmetic operations.)

(b) Give an example of an instance of the problem, and an execution of the merging heuristic on this instance, where the packing returned by the heuristic does not use the minimum possible number of bins.

(c) Prove that in any execution of the merging heuristic, on any instance, the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

5. Here's a way in which a different heuristic for bin packing can arise. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

   Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

   A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck.

   (a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

   (b) Show that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

6. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has a set of 10 machines, and each job can be processed on any of these 10 machines.

   At the moment the business is running the simple *Greedy-Balance* algorithm we discussed in class. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: they have been running it each day for the last month, and they have not observed it to produce a makespan more than 20% above the average load, $\frac{1}{10}\sum_i t_i$.

To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, i.e., $1 \leq t_i \leq 50$ for all jobs i; and the total load $\sum_i t_i$ is quite high each day: it is always at least 3000.

Prove that on the type of inputs the company sees, the *Greedy-Balance* algorithm will always find a solution whose makespan is at most 20% above the average load.

7. Consider an optimization version of the *Hitting Set* problem defined as follows. We are given a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. Also, each element $a_i \in A$ has a *weight* $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in H, $\sum_{a_i \in H} w_i$, is as small as possible. (Recall from Problem Set 7 that H is a hitting set if $H \cap B_i$ is not empty for each i). Let $b = \max_i |B_i|$ denote the maximum size of any of the sets $B_1, B_2, \ldots, B_m$. Give a polynomial time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.

8. Consider the following maximization version of the *three-dimensional matching* problem. Given disjoint sets X, Y, and Z, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subset T$ is a *three-dimensional matching* if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *maximum three-dimensional matching* problem is to find a three-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

   Give a polynomial time algorithm that finds a three-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

9. At a lecture in a computational biology conference one of us attended about a year ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set, and thereby learn (by inference) about all the proteins in the full collection.

   To be useful, the representative set must have two properties.

   - It should be relatively small, so that it will not be too expensive to study it.

- Every protein in the full collection should be "similar" to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by a *distance function* d — given two proteins p and q, it returns a number $d(p, q) \geq 0$. In fact, the function $d(\cdot, \cdot)$ most typically used is the *edit distance*, or *sequence alignment* measure, which we looked at when we studied dynamic programming. We'll assume this is the distance being used here. There is a pre-defined distance cut-off Δ that's specified as part of the input to the problem; two proteins p and q are deemed to be "similar" to one another if and only if $d(p, q) \leq \Delta$.

We say that a subset of P is a *representative set* if for every protein p, there is a protein q in the subset that is similar to it — i.e. for which $d(p, q) \leq \Delta$. Our goal is to find a representative set that is as small as possible.

(a) Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: if the minimum possible size of a representative set is s^*, your algorithm should return a representative set of size at most $O(s^* \log n)$.

(b) Note the close similarity between this problem and the Center Selection problem — a problem for which we considered approximation algorithms in the text. Why doesn't the algorithm described there solve the current problem?

10. Suppose you are given an $n \times n$ *grid graph* G, as in the figure below.

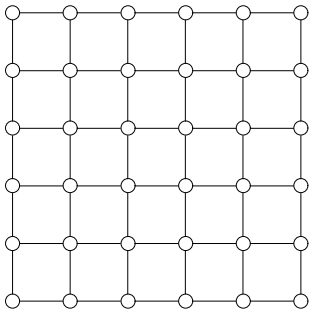


Figure 10.2: A grid graph.

Associated with each node v is a *weight* $w(v)$, which is a non-negative integer. You may assume that the weights of all nodes are distinct. Your goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight*.)

Consider the following greedy algorithm for this problem:

```
The "heaviest-first" greedy algorithm:
  Start with S equal to the empty set.
  While some node remains in G
    Pick a node v_i of maximum weight.
    Add v_i to S.
    Delete v_i and its neighbors from G.
  end while
  Return S
```

(a) Let S be the indepedent set returned by the "heaviest-first" greedy algorithm, and let T be any other independent set in G. Show that for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G.

(b) Show that the "heaviest-first" greedy algorithm returns an independent set of total weight at least $1/4$ times the maximum total weight of any independent set in the grid graph G.

Chapter 11

Local Search

Throughout this course we developed efficient algorithms techniques. In the previous two topics we developed algorithms that extend the limits of tractability by running in polynomial time on special cases of NP-complete problems; and algorithms that provide approximate answers with guaranteed error bounds in polynomial time.

As a final topic of this course, we consider a very general family of techniques: *local search algorithms*. Heuristics such as local search can be useful in dealing with NP-hard problem, even though they are not able to provide any guarantees on the quality of the solutions they find; indeed, for many examples, there are instances on which they perform very badly. However, despite the bad examples, the approach has turned out to be very useful in practice. We use the term *local search* to describe any algorithm that "explores" the space of possible solutions in a sequential fashion, moving in one step from a current solution to a "nearby" one.

Local search algorithms are generally heuristics designed to find good, but not necessarily optimal, solutions to computational problems. Thus we seek to understand the quality of the solutions found by a local search algorithm, and the running time it requires to find good solutions. However, it is often very difficult to actually prove properties of these algorithms, and much about their behavior remains an open question at a mathematical level. As a result, some of our discussion in the next few lectures will have a somewhat different flavor from what we've become familiar with in this course; we'll introduce some algorithms, discuss them qualitatively, but admit quite frankly that we can't prove very much about them.

One useful intuitive motivation of local search algorithms arises from connections with energy minimization principles in physics, and we explore this issue first.

11.1 The Landscape of an Optimization Problem

Potential Energy. Much of the core of local search was developed by people thinking in terms of analogies with physics. Looking at the wide range of hard computational problems

that require the minimization of some quantity, they reasoned as follows. Physical systems are performing minimization all the time, when they seek to minimize their potential energy. What can learn from the ways in which nature performs minimization? Does it suggest new kinds of algorithms?

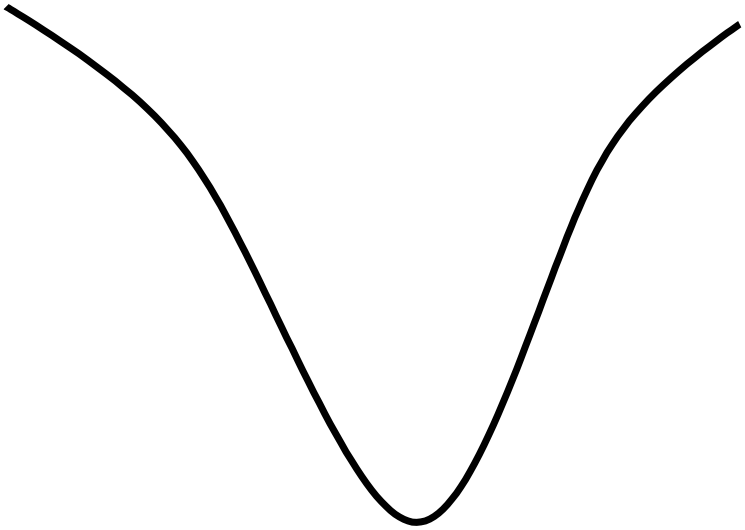


Figure 11.1: A funnel.

If the world really looked like a freshman mechanics class suggests, it seems that it would consist entirely of hockey pucks sliding on ice and balls rolling down inclined surfaces. Hockey pucks usually slide because you push them; but why do balls roll downhill? One perspective that we learn from Newtonian mechanics is that the ball is trying to minimize its *potential energy*. In particular, if the ball has mass m and falls a distance of h, it loses an amount of potential energy proportional to mh. So if we release a ball from the top of the funnel-shaped landscape in Figure 11.1, its potential energy will be minimized at the lowest point.

If we make the landscape a little more complicated, some extra issues creep in. Consider the "double funnel" in Figure 11.2. Point A is lower than point B, and so is a more desirable place for the ball to come to rest. But if we start the ball rolling from point C, it will not be able to get over the barrier between the two funnels, and it will end up at B. We say that the ball has become trapped in a *local minimum*: it is at the lowest point if one looks in the neighborhood of its current location; but stepping back and looking at the whole landscape, we see that it has missed the *global minimum*.

Of course, enormously large physical systems must also try to minimize their energy. Consider, for example, taking a few kilograms of some homogeneous substance, heating it up, and studying its behavior over time. To capture the potential energy exactly, we would in principle need to represent the behavior of each atom in the substance, as it interacts with its nearby atoms. But it is also useful to speak of the potential energy of the system

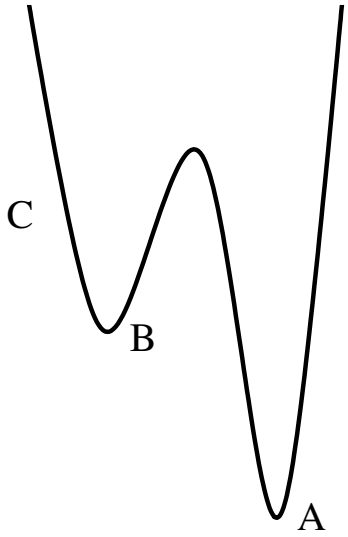


Figure 11.2: A double funnel.

as a whole — as an aggregate — and this is the domain of statistical mechanics. We will come back to statistical mechanics in a little while, but for now we simply observe that our notion of an "energy landscape" provides useful visual intuition for the process by which even a large physical system minimizes its energy. Thus, while it would in reality take a huge number of dimensions to draw the true "landscape" that constrains the system, we can use one-dimensional "cartoon" representations to discuss the distinction between local and global energy minima, the "funnels" around them, and the "height" of the energy barriers between them.

Figure 11.3: A jagged funnel.

Taking a molten material and trying to cool it to a perfect crystalline solid is really the process of trying to guide the underlying collection of atoms to its global potential energy minimum. This can be very difficult, and the large number of local minima in a typical energy landscape represent the pitfalls that can lead the system astray in its search for the global minimum. Thus, rather than the simple example of Figure 11.2, which simply contains a single wrong choice, we should be more worried about landscapes with the schematic cartoon representation depicted in Figure 11.3. This can be viewed as a "jagged funnel," in which there are local minima waiting to trap the system all the way along its journey to the bottom.

The Connection to Optimization. This perspective on energy minimization has really been based on the following core ingredients: the physical system can be in one of a large number of possible states; its energy is a function of its current state; and from a given state, a small perturbation leads to a "neighboring" state. The way in which these neighboring states are linked together, along with the structure of the energy function on them, defines the underlying energy landscape.

It's from this perspective that we again start to think about computational minimization problems. In a typical such problem we have a large (typically exponential-size) set $\mathcal{C}$ of possible solutions. We also have a *cost function* $c(\cdot)$ that measures the quality of each solution; for a solution $S \in \mathcal{C}$, we write its cost as $c(S)$. The goal is to find a solution $S^* \in \mathcal{C}$ for which $c(S^*)$ is as small as possible.

So far, this is just the way we've thought about it all along. We now add to this the notion of a *neighbor relation* on solutions, to capture the idea that one solution S' can be obtained by a small modification of another solution S. We write $S \sim S'$ to denote that S' is a neighboring solution of S, and we use $N(S)$ to denote the *neighborhood* of S, the set $\{S' : S \sim S'\}$. We will primarily be considering symmetric neighbor relations here, though the basic points we discuss will apply to asymmetric neighbor relations as well. A crucial point is that, while the set $\mathcal{C}$ of possible solutions and the cost function $c(\cdot)$ are provided by the specification of the problem, we have the freedom to make up any neighbor relation that we want.

A *local search algorithm* takes this set-up, including a neighbor relation, and works according to the following high-level scheme. At all times it maintains a current solution $S \in \mathcal{C}$. In a given step, it chooses a neighbor S' of S, declares S' to be the new current solution, and iterates. Throughout the execution of the algorithm, it remembers the minimum-cost solution that it has seen thus far; so as it runs, it gradually finds better and better solutions. The crux of a local search algorithm is in the choice of the neighbor relation, and in the design of the rule for choosing a neighboring solution at each step.

Thus, one can think of a neighbor relation as defining a (generally undirected) graph on the set of all possible solutions, with edges joining neighboring pairs of solutions. A local

search algorithm can then be viewed as performing a walk on this graph, trying to move toward a good solution.

Example: The Vertex Cover Problem. This is still all somewhat vague without a concrete problem to think about; so we'll use the *Vertex Cover* problem as a running example here and in much of what follows. It's important to keep in mind that, while *Vertex Cover* makes for a good example, there are many other optimization problems that would work just as well for this illustration.

Thus, we are given a graph $G = (V, E)$; the set $\mathcal{C}$ of possible solutions consists of all subsets S of V that form vertex covers. Hence, for example, we always have $V \in \mathcal{C}$. The *cost* $c(S)$ of a vertex cover S will simply be its size; in this way, minimizing the cost of a vertex cover is the same as finding one of minimum size. Finally, we will focus our examples on local search algorithms that use a particularly simple neighbor relation: we say that $S \sim S'$ if S' can be obtained from S by adding or deleting a single node. Thus, our local search algorithms will be walking through the space of possible vertex covers, adding or deleting a node to their current solution in each step, and trying to find as small a vertex cover as possible.

One useful fact about this neighbor relation is the following.

(11.1) *Each vertex cover S has at most n neighboring solutions.*

The reason is simply that each neighboring solution of S is obtained by adding or deleting a distinct node. A consequence of (11.1) is that we can efficiently examine all possible neighboring solutions of S in the process of choosing which to select.

Let's think first about a very simple local search algorithm, which we'll term *gradient descent*. Gradient descent starts with the full vertex set V, and uses the following rule for choosing a neighboring solution:

> Let S denote the current solution. If there is a neighbor S' of S with strictly lower cost, then choose the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.

So gradient descent moves strictly "downhill" as long as it can; once this is no longer possible, it stops.

We can see that gradient descent terminates precisely at solutions that are *local minima*: solutions S such that for all neighboring S', we have $c(S) \leq c(S')$. This definition corresponds very naturally to our notion of local minima in energy landscapes: they are points from which no one-step perturbation will improve the cost function.

How can we visualize the behavior of a local search algorithm in terms of the kinds of energy landscapes we drew above? Let's think first about gradient descent. The easiest

instance of *Vertex Cover* is surely an n-node graph with no edges. The empty set is the optimal solution (since there are no edges to cover), and gradient does exceptionally well at finding this solution: it starts with the full vertex set V, and keeps deleting nodes until there are none left. Indeed, the set of vertex covers for this edge-less graph corresponds naturally to the funnel we drew in Figure 11.1: the unique local minimum is the global minimum, and there is a downhill path to it from any point.

When can gradient descent go astray? Consider a "star graph" G, consisting of nodes $x_1, y_1, y_2, \ldots, y_{n-1}$, with an edge from x_1 to each y_i. The minimum vertex cover for G is the singleton set $\{x_1\}$, and gradient descent can reach this solution by successively deleting $y_1, \ldots, y_{n-1}$ in any order. But if gradient descent deletes the node x_1 first, then it is immediately stuck: no node y_i can be deleted without destroying the vertex cover property, so the only neighboring solution is the full node set V, which has higher cost. Thus, the algorithm has become trapped in the local minimum $\{y_1, y_2, \ldots, y_{n-1}\}$, which has very high cost relative to the global minimum.

Pictorially, we see that we're in a situation corresponding to the "double funnel" of Figure 11.2. The deeper funnel corresponds to the optimal solution $\{x_1\}$, while the shallower funnel corresponds to the inferior local minimum $\{y_1, y_2, \ldots, y_{n-1}\}$. Sliding down the wrong portion of the slope at the very beginning can send one into the wrong minimum. We can easily generalize this situation to one in which the two minima have any relative depths we want. Consider, for example, a bipartite graph G with nodes $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_\ell$, where $k < \ell$, and there is an edge from every node of the form x_i to every node of the form y_j. Then there are two local minima, corresponding to the vertex covers $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_\ell\}$. Which one is discovered by a run of gradient descent is entirely determined by whether it first deletes an element of the form x_i or y_j.

With more complicated graphs, it's often a useful exercise to think about the kind of landscape they induce; and conversely, one sometimes may look at a landscape and consider whether there's a graph that gives rise to something like it.

For example, what kind of graph might yield a *Vertex Cover* instance with a landscape like the "jagged funnel" in Figure 11.3? One such graph is simply an n-node path, where n is an odd number, with nodes labeled $v_1, v_2, \ldots, v_n$ in order. The unique minimum vertex cover S^* consists of all nodes v_i where i is even. But there are many local optima. For example, consider the vertex cover $\{v_2, v_3, v_5, v_6, v_8, v_9, \ldots\}$ in which every third node is omitted. This is a vertex cover that is significantly larger than S^*; but there's no way to delete any node from it while still covering all edges. Indeed, it's very hard for gradient descent to find the minimum vertex cover S^* starting from the full vertex set V — once it's deleted just a single node v_i with an even value of i, it's lost the chance to find the global optimum S^*. Thus, the even/odd parity distinction in the nodes captures a plethora of different wrong turns in the local search, and hence gives the overall funnel its "jagged" character. Of course, there

is not a direct correspondence between the ridges in the drawing and the local optima; as we warned above, Figure 11.3 is ultimately just a "cartoon" rendition of what's going on.

We see here that even for graphs that are structurally very simple, gradient descent is much too straightforward a local search algorithm. We now look at some more refined local search algorithms that use the same type of neighbor relation, but include a method for "escaping" from local minima.

11.2 The Metropolis Algorithm and Simulated Annealing

The first idea for an improved local search algorithm comes from work of Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller. They considered the problem of simulating the behavior of a physical system according to principles of statistical mechanics. A basic model from this field asserts that the probability of finding a physical system in a state with energy E is proportional to the *Gibbs-Boltzmann function* $e^{-E/(kT)}$, where $T > 0$ is the temperature and $k > 0$ is a constant. Let's look at this function. For any temperature T, the function is monotone decreasing in the energy E, so this states that a physical system is more likely to be in a lower energy state than in high energy states. Now, let's consider the effect of the temperature T. When T is small, the probability for a small energy state is significantly smaller than the probability for a large energy state. However, if the temperature is high, then the difference between these two probabilities is very small, and the system is almost equally likely to be in any state.

Metropolis et al. proposed the following method for performing step-by-step simulation of a system at a fixed temperature T. At all times, the simulation maintains a current state of the system, and tries to produce a new state by applying a perturbation to this state. We'll assume that we're only interested in states of the system that are "reachable" from some fixed initial state by a sequence of small perturbations, and we'll assume that there is only a finite set $\mathcal{C}$ of such states. In a single step, we first generate a small random perturbation to the current state S of the system, resulting in a new state S'. Let $E(S)$ and $E(S')$ denote the energies of S and S' respectively. If $E(S') \leq E(S)$, then we update the current state to be S'. Otherwise, let $\Delta E = E(S') - E(S) > 0$. We update the current state to be S' with probability $e^{-\Delta E/(kT)}$, and otherwise leave the current state at S.

Metropolis et al. proved that their simulation algorithm has the following property. To prevent too long a digression, we omit the proof; it is actually a direct consequence of some basic facts about random walks.

(11.2) *Let*

$$Z = \sum_{S \in \mathcal{C}} e^{-E(S)/(kT)}.$$

For a state S, let $f_S(t)$ denote the fraction of the first t steps in which the state of the simulation is in S. Then the limit of $f_S(t)$ as t approaches ∞ is almost surely $\frac{1}{Z} \cdot e^{-E(S)/(kT)}$.

This is exactly the sort of fact one wants, since it says that the simulation spends roughly the correct amount of time in each state, according to the Gibbs-Boltzmann equation.

If we want to use this overall scheme to design a local search algorithm for minimization problems, we use the analogies of the previous section in which states of the system are candidate solutions, with energy corresponding to cost. We then see that the operation of the Metropolis algorithm has a very desirable pair of features in a local search algorithm: it is biased toward "downhill" moves, but will also accept "uphill" moves with smaller probability. In this way, it is able to make progress even when situated in a local minimum. Moreover, as expressed in (11.2), it is globally biased toward lower-cost solutions.

Here is a concrete formulation of the Metropolis algorithm for a minimization problem.

```
Start with an initial solution S_0, and constants k and T.
In one step:
   Let S be the current solution.
   Let S' be chosen uniformly at random from the neighbors of S.
   If c(S') ≤ c(S) then
      Update S ← S'.
   Else
      With probability e^{-(c(S')-c(S))/(kT)},
         Update S ← S'.
      Otherwise
         Leave S unchanged.
   Endif
```

Thus, on the *Vertex Cover* instance consisting of the star graph in the previous section, in which x_1 is joined to each of $y_1, \ldots, y_{n-1}$, we see that the Metropolis algorithm will quickly bounce out of the local minimum that arises when x_1 is deleted: the neighboring solution in which x_1 is put back in will be generated, and will be accepted with positive probability. Even on the more complex graphs we considered, like the union of k disjoint stars, the Metropolis algorithm is able to correct the wrong choices it makes as it proceeds.

At the same time, the Metropolis algorithm does not always behave the way one would want, even in some very simple situations. Let's go back to the very first graph we considered, a graph G with no edges. Gradient descent solves this instance with no trouble, deleting nodes in sequence until none are left. But while the Metropolis algorithm will start out this way, it begins to go astray as it nears the global optimum. Consider the situation in which the current solution contains only c nodes, where c is much smaller than the total number of nodes, n. With very high probability, the neighboring solution generated by the Metropolis

algorithm will have size $c+1$, rather than $c-1$, and with reasonable probability this uphill move will be accepted. Thus, it gets harder and harder to shrink the size of the vertex cover as the algorithm proceeds; it is exhibiting a sort of "flinching" reaction near the bottom of the funnel.

This behavior shows up in more complex examples as well, and in more complex ways; but it is certainly striking for it to show up here so simply. In order to figure out how we might fix this behavior, we return to the physical analogy that motivated the Metropolis algorithm, and ask: what's the meaning of the temperature parameter T in the context of optimization?

We can think of T as a one-dimensional knob that we're able to turn, and it controls the extent to which the algorithm is willing to accept uphill moves. As we make T very large, the probability of accepting an uphill move approaches 1, and the Metropolis algorithm behaves like a random walk that is basically indifferent to the cost function. As we make T very close to 0, on the other hand, uphill moves are almost never accepted, and the Metropolis algorithm behaves almost identically to gradient descent.

Neither of these extremes is an effective way to solve minimization problems in general, and we can see this in physical settings as well. If we take a solid and heat it to a very high temperature, we do not expect it to maintain a nice crystal structure, even if this is energetically favorable; and this can be explained by the large value of kT in the expression $e^{-E(S)/(kT)}$, which makes the enormous number of less favorable states too probable. This is a way in which we can view the "flinching" behavior of the Metropolis algorithm on an easy *Vertex Cover* instance: it's trying to find the lowest-energy state at too high a temperature, when all the competing states have too high a probability. On the other hand, if we take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either; rather, we get a deformed crystal structure with many imperfections. This is because, with T very small, we've come too close to the realm of gradient descent, and the system has become trapped in one of the numerous ridges of its jagged energy landscape. It is interesting to note that when T is very small, then the statement (11.2) above shows that in the limit, the random walk spends most of its time in the lowest energy state. The problem is that the random walk will take an enormous amount of time before getting anywhere near this limit.

In the early 1980's, as people were considering the connection between energy minimization and combinatorial optimization, Kirkpatrick, Gelatt, and Vecchi thought about the issues we've been discussing, and they asked the following question: How do we solve this problem for physical systems, and what sort of algorithm does this suggest? In physical systems, one guides a material to a crystalline state by a process known as *annealing*: the material is cooled very gradually from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures. In this way, it is able to escape from the energy minima that it encounters all the way through the cooling process,

eventually arriving at the global optimum.

We can thus try to mimic this process computationally, arriving at an algorithmic technique known as *simulated annealing*. Simulated annealing works by running the Metropolis algorithm while gradually decreasing the value of T over the course of the execution. The exact way in which T is updated is called, for natural reasons, a *cooling schedule*, and a number of considerations go into the design of the cooling schedule. Formally, a cooling schedule is a function τ from $\{1, 2, 3, \ldots\}$ to the positive real numbers; in iteration i of the Metropolis algorithm, we use the temperature $T = \tau(i)$ in our definition of the probability.

Qualitatively, we can see that simulated annealing allows for large changes in the solution in the early stages of its execution, when the temperature is high; as the search proceeds, the temperature is lowered so that we are less likely to undo progress that has already been made. We can also view simulated annealing as trying to optimize a trade-off that is implicit in (11.2). According to (11.2), values of T arbitrarily close to 0 put the highest probability on minimum-cost solutions; *however*, (11.2) by itself says nothing about the rate of convergence of the functions $f_S(t)$ that it uses. It turns out that these functions converge, in general, much more rapidly for large values of T; and so to find minimum-cost solutions quickly, it is useful to speed up convergence by starting the process with T large, and then gradually reducing it so as to raise the probability on the optimal solutions. While we believe that physical systems reach a minimum energy state via annealing, the simulated annealing method has no guarantee of finding an optimal solution. To see why, consider the "double funnel" of Figure 11.2. If the two funnels take equal area, than at high temperatures the system is essentially equally likely to be in either funnel. Once we cool the temperature, it will become harder and harder to switch between the two funnels. There appears to be no guarantee that at the end of annealing, we will be at the bottom of the lower funnel.

There are many open problems associated with simulated annealing, both in proving properties of its behavior and in determining the range of settings for which it works well in practice. Some of the general questions that come up here involve probabilistic issues that are beyond the scope of this course. In the next lectures we will turn to some fairly clean settings in which one can prove properties of locally optimal solutions, and discuss some of the issues that are important in making simulated annealing successful for a particular application.

11.3 Application: Hopfield Neural Networks

Thus far we have been discussing local search as a method for trying to find the global optimum in a computational problem. There are some cases, however, in which by examining the specification of the problem carefully, we discover that it is really just an arbitrary *local* optimum that is required. We now consider a problem that illustrates this phenomenon.

The problem is that of finding *stable configurations* in *Hopfield neural networks*. Hopfield networks were proposed as a simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states. Concretely, a Hopfield network can be viewed as an undirected graph $G = (V, E)$, with an integer-valued weight w_e on each edge e; each weight may be positive or negative. A *configuration* S of the network is an assignment of the value -1 or $+1$ to each node u; we will refer to this value as the *state* s_u of the node u. The meaning of a configuration is that each node u, representing a unit of the neural network, is trying to choose between one of two possible states ('on' or 'off'; 'yes' or 'no'); and its choice is influenced by those of its neighbors as follows. Each edge of the network imposes a *requirement* on its endpoints: If u is joined to v by an edge of negative weight, then u and v want to have the same state, while if u is joined to v by an edge of positive weight, then u and v want to have opposite states. The absolute value $|w_e|$ will indicate the *strength* of this requirement, and we will refer to $|w_e|$ as the *absolute weight* of edge e.

Unfortunately, there may be no configuration that respects the requirements imposed by all the edges. For example, consider three nodes a, b, c all mutually connected to one another by edges of weight 1. Then no matter what configuration we choose, two of these nodes will have the same state, and thus will be violating the requirement that they have opposite states.

In view of this, we ask for something weaker. With respect to a given configuration, we say that an edge $e = (u, v)$ is *good* if the requirement it imposes is satisfied by the states of its two endpoints: either $w_e < 0$ and $s_u = s_v$, or $w_e > 0$ and $s_u \neq s_v$. Otherwise, we say e is *bad*. Note that we can express the condition that e is good very compactly as follows: $w_e s_u s_v < 0$. Next, we say that a node u is *satisfied* in a given configuration if the total absolute weight of all good edges incident to u is at least as large as the total absolute weight of all bad edges incident to u. We can write this as

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0.$$

Finally, we call a configuration *stable* if all nodes are satisfied.

Why do we use the term "stable" for such configurations? This is based on viewing network from the perspective of an individual node u. On its own, the only choice u has is whether to take the state -1 or $+1$; and like all nodes, it wants to respect as many edge requirements as possible (as measured in absolute weight). Suppose u asks: should I flip my current state? We see that if u does flip its state (while all other nodes keep their states the same), then all the good edges incident to u become bad, and all the bad edges incident to u become good. So to maximize the amount of good edge weight under its direct control, u should flip its state if and only if it is not satisfied. In other words, a stable configuration is one in which no individual node has an incentive to flip its current state.

A basic question now becomes: does a Hopfield network always have a stable configuration, and if so, how can we find one? Below, we will prove the following fact.

(11.3) *Every Hopfield network has a stable configuration, and such a configuration can be found in time polynomial in n and $W = \sum_e |w_e|$.*

We will see that stable configurations in fact arise very naturally as the local optima of a certain local search procedure on the Hopfield network.

To see that the statement of (11.3) is not entirely trivial, we note that it fails to remain true if one changes the model in certain natural ways. For example, suppose we were to define a *directed Hopfield network* exactly as above, except that each edge is directed, and each node determines whether or not it is satisfied by looking only at edges for which it is the tail. Then in fact, such a network need not have a stable configuration. Consider, for example, a directed version of the three-node network we discussed above: there are nodes a, b, c, with directed edges $(a, b), (b, c), (c, a)$, all of weight 1. Then if all nodes have the same state, they will all be unsatisfied; and if one node has a different state from the other two, then the node directly in front of it will be unsatisfied. Thus, there is no configuration of this directed network in which all nodes are satisfied.

It is clear that our proof will need to rely somewhere on the undirected nature of the network.

Proof of (11.3) . We consider performing the following iterative procedure to search for a stable configuration. As long as the current configuration is not stable, there is an unsatisfied node; so we choose one such node u, flip its state, and iterate. Clearly, if this process terminates, we will have a stable configuration. What is not as obvious is whether it must in fact terminate. Indeed, in the directed example above, this process will simply cycle through the three nodes, flipping their states sequentially forever.

The key to proving this process terminates is an idea we've used in several previous situations: to look for a measure of *progress*; a quantity that strictly increases with every flip and has an absolute upper bound. This can be used to bound the number of iterations.

Probably the most natural progress measure would be the number of satisfied nodes: if this increased every time we flipped an unsatisfied node, the process would run for at most n iterations before terminating with a stable configuration. Unfortunately, this does not turn out to work. When we flip an unsatisfied node v, it's true that it has now become satisfied — but a potentially large number of its previously satisfied neighbors could now become unsatisfied, resulting in a net decrease in the number of satisfied nodes.

However, there is a more subtle progress measure that *does* increase with each flip of an unsatisfied node. Specifically, for a given configuration S, we define $\Phi(S)$ to be the total absolute weight of all good edges in the network. That is,

$$\Phi(S) = \sum_{\text{good } e} |w_e|.$$

Clearly, for any configuration S, we have $\Phi(S) \geq 0$ (since $\Phi(S)$ is a sum of positive integers), and $\Phi(S) \leq W = \sum_e |w_e|$ (since at most, every edge is good).

Now, suppose that in a non-stable configuration S we choose a node u that is unsatisfied and flip its state, resulting in a configuration S'. What can we say about the relationship of $\Phi(S')$ to $\Phi(S)$? Recall that when u flips its state, all good edges incident to u become bad, all bad edges incident to u become good, and all edges that don't have u as an endpoint remain the same. So if we let g_u and b_u denote the total absolute weight on good and bad edges incident to u, respectively, then we have

$$\Phi(S') = \Phi(S) - g_u + b_u.$$

But since u was unsatisfied in S, we also know that $b_u > g_u$; and since b_u and g_u are both integers, we in fact have $b_u \geq g_u + 1$. Thus,

$$\Phi(S') \geq \Phi(S) + 1.$$

Hence, the value of Φ begins at some non-negative integer, increases by at least 1 on every flip, and cannot exceed W. Thus, our process runs for at most W iterations, and when it terminates, we must have a stable configuration. Moreover, in each iteration we can identify an unsatisfied node using a number of arithmetic operations that is polynomial in n; thus, the running time bound follows as well. ■

So we see that, in the end, the existence proof for stable configurations was really about local search. We first set up an objective function Φ that we sought to maximize. Configurations were the possible solutions to this maximization problem, and we defined what it meant for two configurations S and S' to be neighbors: S' should be obtainable from S by flipping a single state. We then studied the behavior of a simple iterative improvement algorithm for local search (the upside-down form of gradient descent, since we have a maximization problem); and we found that any local maximum corresponds to a stable configuration.

It's worth noting that while our algorithm proves the existence of a stable configuration, the running time leaves something to be desired when the absolute weights are large. Specifically, as we saw in the Subset Sum problem, and in our first algorithm for maximum flow, the algorithm we obtain here is polynomial only in the actual magnitude of the weights, not in the size of their binary representation. For very large weights, this can lead to running times that are quite infeasible.

However, no simple way around this situation is currently known. It turns out to be an open question to find an algorithm that constructs stable states in time polynomial in n and $\log W$ (rather than n and W) — or in a number of primitive arithmetic operations that is polynomial in n alone, independent of the value of W.

11.4 Choosing a Neighbor Relation

We began by saying that a local search algorithm is really based on two fundamental ingredients: the choice of the neighbor relation, and the the rule for choosing a neighboring solution at each step. Thus far, we've spent more time thinking about the second of these: both the Metropolis algorithm and simulated annealing took the neighbor relation as given, and modified the way in which a neighboring solution should be chosen.

What are some of the issues that should go into our choice of the neighbor relation? This can turn out to be quite subtle, though at a high level the trade-off is a basic one:

(i) The neighborhood of a solution should be rich enough that we do not tend to get stuck in bad local optima; but

(ii) the neighborhood of a solution should not be too large, since we want to be able to efficiently search the set of neighbors for possible local moves.

If the first of these points were the only concern, then it would seem that we should simply make all solutions neighbors of one another — after all, then there would be no local optima, and the global optimum would always be just one step away! The second point exposes the (obvious) problem with doing this: if the neighborhood of the current solution consists of every possible solution, then the local search paradigm gives us no leverage whatsoever; it reduces simply to brute-force search of this neighborhood.

Actually, we've already encountered one case in which choosing the right neighbor relation had a profound effect on the tractability of a problem, though we did not explicitly take note of this at the time — this was in the bipartite matching problem. Probably the simplest neighbor relation on matchings would be the following: M' is a neighbor of M if M' can be obtained by the insertion or deletion of a single edge in M. Under this definition, we get "landscapes" that are quite jagged, quite like the *Vertex Cover* examples we saw earlier; and we can get locally optimal matchings under this definition that have only half the size of the maximum matching.

But suppose we try defining a more complicated (indeed, asymmetric) neighbor relation: we say that M' is a neighbor of M if, when we set up the corresponding flow network, M' can be obtained from M by a single augmenting path. What can we say about a matching M if it is a local maximum under this neighbor relation? In this case there is no augmenting path, and so M must in fact be a (globally) maximum matching. In other words, with this neighbor relation, the only local maxima are global maxima, and so direct gradient ascent will produce a maximum matching. If we reflect on what the Ford-Fulkerson algorithm is doing in our reduction from bipartite matching to maximum flow, this makes sense: the size of the bipartite matching strictly increases in each step, and we never need to "back out" of a local maximum. By choosing the neighbor relation very carefully, we've turned a jagged optimization landscape into a simple, tractable, funnel.

Of course, we do not expect that things will always work out this well; for example, since *Vertex Cover* is NP-complete, it would be surprising if it allowed for a neighbor relation that simultaneously produced "well-behaved" landscapes and neighborhoods that could be searched efficiently. We now look at several possible neighbor relations in the context of a different NP-complete problem — *Maximum Cut* — that is representative of a family of computationally hard graph partitioning problems.

The Maximum Cut Problem

In the *Maximum Cut* problem, we are given an undirected graph $G = (V, E)$, with a positive integer weight w_e on each edge e. For a partition (A, B) of the vertex set, we use $w(A, B)$ to denote the total weight of edges with one end in A and the other in B:

$$w(A, B) = \sum_{\substack{e=(u,v) \\ u \in A, v \in B}} w_e.$$

The goal is to find a partition (A, B) of the vertex set so that $w(A, B)$ is maximized. *Maximum Cut* is NP-complete, in the sense that given a weighted graph G and a bound B, it is NP-complete to decide whether there is a partition (A, B) of the vertices of G with $w(A, B) \geq B$. At the same time, of course, *Maximum Cut* resembles the polynomially-solvable minimum s-t cut problem for flow networks; the crux of its intractability comes from the fact that we are seeking to maximize the edge weight across the cut, rather than minimize it.

Although the problem of finding a stable configuration of a Hopfield network was not an optimization problem *per se*, we can see that *Maximum Cut* is closely related to it. In the language of Hopfield networks, *Maximum Cut* is an instance in which all edge weights are positive (rather than negative), and configurations of nodes states S correspond naturally to partitions (A, B) — nodes have state -1 if and only if they are in the set A, and state $+1$ if and only if they are in the set B. The goal is to assign states so that as much weight as possible is on *good edges* — those whose endpoints have opposite states. Phrased this way, *Maximum Cut* seeks to maximize precisely the quantity $\Phi(S)$ that we used in the proof of (11.3), in the case when all edge weights are positive.

The "state-flipping" algorithm used in that proof provides a local search algorithm to approximate this objective function $\Phi(S) = w(A, B)$. In terms of partitions, it says the following: if there exists a node u so that the total weight of edges from u to nodes in its own side of the partition exceeds the total weight of edges from u to nodes on the other side of the partition, then u itself should be moved to the other side of the partition.

We'll call this the "single-flip" neighborhood on partitions: partitions (A, B) and (A', B') are neighboring solutions if (A', B') can be obtained from (A, B) by moving a single node from one side of the partition to the other. Let's ask two basic questions. First, can we say

anything concrete about the quality of the local optima under the single-flip neighborhood? And second, since the single-flip neighborhood is about as simple as one could imagine, what other neighborhoods might yield stronger local search algorithms for *Maximum Cut*?

The following result addresses the first of these questions, showing that local optima under the single-flip neighborhood provide solutions achieving a guaranteed approximation bound.

(11.4) *Let (A, B) be a partition that is a local optimum for* Maximum Cut *under the single-flip neighborhood. Let (A^*, B^*) be a globally optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.*

Proof. Let $W = \sum_e w_e$. We also extend our notation a little: for two nodes u and v, we use w_{uv} to denote w_e if there is an edge e joining u and v, and 0 otherwise.

For any node $u \in A$, we must have

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv},$$

since otherwise u should be moved to the other side of the partition, and (A, B) would not be locally optimal. Suppose we add up these inequalities for all $u \in A$; any edge that has both ends in A will appear on the left-hand side of exactly two of these inequalities, while any edge that has one end in A and one end in B will appear on the right-hand side of exactly one of these inequalities. Thus, we have

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B). \tag{11.1}$$

We can apply the same reasoning to the set B, obtaining

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B). \tag{11.2}$$

If we add together inequalities (11.1) and (11.2), and divide by 2, we get

$$\sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B). \tag{11.3}$$

The left-hand side of inequality (11.3) accounts for all edge weight that does not cross from A to B; so if we add $w(A, B)$ to both sides of (11.3), the left-hand side becomes equal to W. The right-hand side becomes $2w(A, B)$, so we have $W \leq 2w(A, B)$, or $w(A, B) \geq \frac{1}{2} W$.

Since the globally optimal partition (A^*, B^*) clearly satisfies $w(A^*, B^*) \leq W$, we have $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$. ■

Notice that we never really thought much about the optimal partition (A^*, B^*) in the proof of (11.4); we really showed the stronger statement that in any locally optimal solution

under the single-flip neighborhood, at least half the total edge weight in the graph crosses the partition.

Next let's consider neighbor relations that produce larger neighborhoods than the single-flip rule, and consequently attempt to reduce the prevalence of local optima. Perhaps the most natural generalization is the *k-flip neighborhood*, for $k \geq 1$: we say that partitions (A, B) and (A', B') are neighbors under the k-flip rule if (A', B') can be obtained from (A, B) by moving at most k nodes from one side of the partition to the other.

Now, clearly if (A, B) and (A', B') are neighbors under the k-flip rule, then they are also neighbors under the k'-flip rule for every $k' > k$. Thus, if (A, B) is a local optimum under the k'-flip rule, it is also a local optimum under the k-flip rule for every $k < k'$. But reducing the set of local optima by raising the value of k comes at a steep computational price: to examine the set of neighbors of (A, B) under the k-flip rule, we must consider all $\Theta(n^k)$ ways of moving up to k nodes to the opposite side of the partition. This becomes prohibitive even for small values of k.

Kernighan and Lin proposed an alternate method for generating neighboring solutions; it is computationally much more efficient, but still allows large-scale transformations of solutions in a single step. Their method, which we'll call the K-L heuristic, defines the neighbors of a partition (A, B) according the following n-phase procedure.

- In phase 1, we choose a single node to flip, in such a way that the value of the resulting solution is as large as possible. We perform this flip even if the value of the solution decreases relative to $w(A, B)$. We *mark* the node that has been flipped, and let (A_1, B_1) denote the resulting solution.

- At the start of phase k, for $k > 1$, we have a partition (A_{k-1}, B_{k-1}); and $k - 1$ of the nodes are marked. We choose a single unmarked node to flip, in such a way that the value of the resulting solution is as large as possible. (Again, we do this even if the value of the solution decreases as a result.) We mark the node we flip, and let (A_k, B_k) denote the resulting solution.

- After n phases, each node is marked, indicating that it has been flipped precisely once. Consequently, the final partition (A_n, B_n) is actually the "mirror image" of the original partition (A, B): we have $A_n = B$ and $B_n = A$.

- Finally, the K-L heuristic defines the $n - 1$ partitions $(A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$ to be the neighbors of (A, B). Thus, (A, B) is a local optimum under the K-L heuristic if and only if $w(A, B) \geq w(A_i, B_i)$ for $1 \leq i \leq n - 1$.

So we see that the K-L heuristic tries a very long sequence of flips, even while it appears to be making things worse, in the hope that some partition (A_i, B_i) generated along the

way will turn out better than (A, B). But even though it generates neighbors very different from (A, B), it only performs n flips in total, and each takes only $O(n)$ time to perform. Thus, it is computationally much more reasonable than the k-flip rule for larger values of k. Moreover, the K-L heuristic has turned out to be very powerful in practice, despite the fact that rigorous analysis of its properties has remained largely an open problem.

11.5 Exercises

1. Consider the load balancing problem from the chapter on approximation algorithms. Some friends of yours are running a collection of Web servers, and they've designed a local search heuristic for this problem, different from the algorithms described in that chapter.

   Recall that we have m machines $M_1, \ldots, M_m$, and we must assign each job to a machine. The load of the i^{th} job is denoted t_i. The *makespan* of an assignment is the *maximum load* on any machine:

   $$\max_{\text{machines } M_i} \sum_{\text{jobs } j \text{ assigned to } M_i} t_j.$$

   Your friends' local search heuristic works as follows. They start with an arbitrary assignment of jobs to machines, and they then repeatedly try to apply the following type of "swap move":

   > Let $A(i)$ and $A(j)$ be the jobs assigned to machines M_i and M_j respectively. To perform a swap move on M_i and M_j, choose subsets of jobs $B(i) \subseteq A(j)$ and $B(j) \subseteq A(j)$, and "swap" these jobs between the two machines. That is, update $A(i)$ to be $A(i) \cup B(j) - B(i)$ and update $A(j)$ to be $A(j) \cup B(i) - B(j)$. (One is allowed to have $B(i) = A(i)$, or to have $B(i)$ be the empty set; and analogously for $B(j)$.)

   Consider a swap move applied to machines to machines M_i and M_j. Suppose the loads on M_i and M_j before the swap are T_i and T_j respectively, and the loads after the swap are T_i' and T_j'. We say that the swap move is *improving* if $\max(T_i', T_j') < \max(T_i, T_j)$; in other words, the larger of the two loads involved has strictly decreased. We say that an assignment of jobs to machines is *stable* if there does not exist an improving swap move, beginning with the current assignment.

   Thus, the local search heuristic simply keeps executing improving swap moves until a stable assignment is reached; at this point, the resulting stable assignment is returned as the solution.

Example: Suppose there are two machines: in the current assignment, the machine M_1 has jobs of sizes $1, 3, 5, 8$, and machine M_2 has jobs of sizes $2, 4$. Then one possible improving swap move would be to define $B(1)$ to consist of the job of size 8, and define $B(2)$ to consist of the job of size 2. After these two sets are "swapped," the resulting assignment has jobs of size $1, 2, 3, 5$ on M_1, and jobs of size $4, 8$ on M_2. This assignment is stable. (It also has an optimal makespan of 12.)

(a) As specified, there is no explicit guarantee that this local search heuristic will always terminate — i.e., what if it keeps cycling forever through assignments that are not stable?

Prove that in fact the local search heuristic terminates in a finite number of steps, with a stable assignment, on any instance.

(b) Show that any stable assignment has a makespan that is within a factor of 2 of the minimum possible makespan.

2. Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

   You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

3. $(*)$ Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

   We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Chapter 12

Randomized Algorithms

The idea that a process can be "random" is not a modern one; one can trace the notion far back into the history of human thought, and certainly see its reflections in gambling and the insurance business — each of which reach into ancient times. Yet while similarly intuitive subjects like geometry and logic have been treated mathematically for several thousand years, the mathematical study of probability is surprisingly young; the first known attempts to seriously formalize it came about in the 1600's. Of course, the history of computer science plays out on a much shorter time scale, and the idea of randomization has been with it since its early days.

Randomization and probabilistic analysis are themes that cut across many areas of computer science, including algorithm design, and when one thinks about random processes in the context of computation, it is usually in one of two distinct ways. One view is to consider the world as behaving randomly: one can consider traditional algorithms that confront randomly generated input. This approach is often termed *average-case analysis*, since we are studying the behavior of an algorithm on an "average" input (subject to some underlying random process), rather than a worst-case input.

A second view is to consider algorithms that behave randomly: the world provides the same worst-case input as always, but we allow our algorithm to make random decisions as it processes the input. Thus, the role of randomization in this approach is purely internal to the algorithm, and does not require new assumptions about the nature of the input; it is this notion of a *randomized algorithm* that we will be considering in this chapter.

Why might it be useful to design an algorithm that is allowed to make random decisions? A first answer would be to observe that by allowing randomization, we've made our underlying model more powerful — efficient deterministic algorithms that always yield the correct answer are a special case of efficient randomized algorithms that only need to yield the correct answer with high probability; they are also a special case of randomized algorithms that are always correct, and run efficiently *in expectation*. Even in a worst-case world, an algorithm that does its own "internal" randomization may be able to offset certain worst-

case phenomena. So problems that may not have been solvable by efficient deterministic algorithms may still be amenable to randomized algorithms.

But this is not the whole story, and in fact we'll be looking at randomized algorithms for a number of problems where there exist comparably efficient deterministic algorithms. Nevertheless a randomized approach often exhibits considerable power for further reasons: it may be conceptually much simpler; or it may allow the algorithm to function while maintaining very little internal state or memory of the past. The advantages of randomization seem to increase further as one considers larger computer systems and networks, with many loosely interacting processes — in other words, a *distributed system*. Here, random behavior on the part of individual processes can reduce the amount of explicit communication or synchronization that is required; it is often valuable as a tool for *symmetry-breaking* among processes, reducing the danger of contention and "hot spots." A number of our examples will come from settings like this: regulating access to a shared resource, balancing load on multiple processors, or routing packets through a network. A small level of comfort with randomized heuristics can give one considerable leverage in thinking about large systems.

A natural worry in approaching the topic of randomized algorithms is that it requires an extensive knowledge of probability. Of course, it's always better to know more rather than less, and some algorithms are indeed based on complex probabilistic ideas. But one further goal of this chapter is to try illustrating *how little* underlying probability is really needed in order to understand many of the well-known algorithms in this area. We will see that there is a small set of useful probabilistic tools that recur frequently, and this chapter will try to develop the tools alongside the algorithms — ultimately, facility with these tools is as valuable as an understanding of the specific algorithms themselves.

12.1 A First Application: Contention Resolution

We begin with a first application of randomized algorithm — contention resolution in a distributed system — that illustrates the general style of analysis we will be using for many of the algorithms that follow. In particular, it is a chance to work through some basic manipulations involving *events* and their probabilities, analyzing intersections of events using *independence* and unions of events using a simple *Union Bound*. For the sake of completeness, we give a brief summary of these concepts in the appendix to this chapter.

A contention-resolution protocol. Suppose we have n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. We imagine time as being divided into discrete *rounds*. The database has the property that it can be accessed by at most one process in a single round; if two or more processes attempt to access it simultaneously, then all processes are "locked out" for the duration of that round. So while each process wants

to access the database as often as possible, it's pointless for all of them to try accessing it in every round; then everyone will be perpetually locked out. What's needed is a way to divide up the rounds among the processes in an equitable fashion, so that all processes get through to the database on a regular basis.

If it is easy for the processes to communicate with each other, than one can imagine all sorts of direct means for resolving the contention. But suppose that the processes can't communicate with each other at all; how then can they work out a protocol under which they manage to "take turns" in accessing the database?

Randomization provides a natural protocol for this problem, which we can specify simply as follows. For some number $p > 0$ that we'll determine shortly, each process will attempt to access the database in each round with probability p, independently of the decisions of the other processes. So if exactly one process decides to make the attempt in a given round, it will succeed; if two or more try, then they will all be locked out; and if none try, then the round is in a sense "wasted." This type of strategy, in which each of a set of identical processes randomizes its behavior, is the core of the *symmetry-breaking* paradigm that we mentioned initially: if all the processes operated in lock-step, repeatedly trying to access the database at the same time, there'd be no progress; but by randomizing, they "smooth out" the contention.

Some basic events. When confronted with a probabilistic system like this, a good first step it to write down some basic events and think about their probabilities. Here's a first event to consider: for a given process P_i and a given round t, let $\mathcal{A}[i,t]$ denote the event that P_i attempts to access the database in round t. We know that each process attempts an access in each round with probability p, so the probability of this event, for any i and t, is $\Pr[\mathcal{A}[i,t]] = p$. For every event, there is also a *complementary event*, indicating that the event did not occur; here we have the complementary event $\overline{\mathcal{A}[i,t]}$ that P_i does not attempt to access the database in round t, with probability

$$\Pr\left[\overline{\mathcal{A}[i,t]}\right] = 1 - \Pr[\mathcal{A}[i,t]] = 1 - p.$$

Our real concern is whether a process *succeeds* in accessing the database in a given round. Let $\mathcal{S}[i,t]$ denote this event. Clearly the process P_i must attempt an access in round t in order to succeed. Indeed, succeeding is equivalent to the following: process P_i attempts to access the database in round t, and each other process *does not* attempt to access the database in round t. Thus, $\mathcal{S}[i,t]$ is equal to the intersection of the event $\mathcal{A}[i,t]$ with the all the complementary events $\overline{\mathcal{A}[j,t]}$, for $j \neq i$:

$$\mathcal{S}[i,t] = \mathcal{A}[i,t] \cap \left(\bigcap_{j \neq i} \overline{\mathcal{A}[j,t]}\right).$$

All the events in this intersection are independent, by the definition of the contention-resolution protocol. Thus, to get the probability of $\mathcal{S}[i,t]$, we can multiply the probabilities of all the events in the intersection:

$$\Pr[\mathcal{S}[i,t]] = \Pr[\mathcal{A}[i,t]] \cdot \prod_{j \neq i} \Pr\left[\overline{\mathcal{A}[j,t]}\right] = p(1-p)^{n-1}.$$

We now have a nice, closed-form expression for the probability that P_i succeeds in accessing the database in round t; we can now ask how to set p so that this success probability is maximized. Observe first that the success probability is 0 for the extreme cases $p = 0$ and $p = 1$ (these correspond to the extreme case in which processes never bother attempting, and the opposite extreme case in which every process tries accessing the database in every round, so that everyone is locked out). The function $f(p) = p(1-p)^{n-1}$ is positive for values of p strictly between 0 and 1, and its derivative $f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$ has a single zero at the value $p = 1/n$, where the maximum is achieved. Thus, we can maximize the success probability by setting $p = 1/n$. (Notice that $p = 1/n$ is a natural intuitive choice as well, if one wants exactly one process to attempt an access in any round.)

When we set $p = 1/n$, we get $\Pr[\mathcal{S}[i,t]] = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}$. It's worth getting a sense for the asymptotic value of this expression, with the help of the following extremely useful fact from basic calculus.

(12.1) *(a) The function $\left(1 - \frac{1}{n}\right)^n$ converges monotonically from $\frac{1}{4}$ up to $\frac{1}{e}$ as n increases from 2.*

(b) The function $\left(1 - \frac{1}{n}\right)^{n-1}$ converges monotonically from $\frac{1}{2}$ down to $\frac{1}{e}$ as n increases from 2.

Using (12.1), we see that $1/(en) \leq \Pr[\mathcal{S}[i,t]] \leq 1/(2n)$, and hence $\Pr[\mathcal{S}[i,t]]$ is asymptotically equal to $\Theta(1/n)$.

Waiting for a particular process to succeed. Let's consider this protocol with the optimal value $p = 1/n$ for the access probability. Suppose we are interested in how long it will take process P_i to succeed in accessing the database at least once. We see from the above calculation that the probability of its succeeding in any one round is not very good, if n is reasonably large. How about if we consider multiple rounds?

Let $\mathcal{F}[i,t]$ denote the "failure event" that process P_i does not succeed in *any* of the rounds 1 through t. This is clearly just the intersection of the complementary events $\overline{\mathcal{S}[i,r]}$ for $r = 1, 2, \ldots, t$. Moreover, since each of these events is independent, we can compute the probability of $\mathcal{F}[i,t]$ by multiplication:

$$\Pr[\mathcal{F}[i,t]] = \Pr\left[\bigcap_{r=1}^{t} \overline{\mathcal{S}[i,r]}\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] = \left[1 - \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}\right]^t.$$

This calculation does give us the value of the probability; but at this point we're in danger of ending up with some extremely complicated-looking expressions, and so it's important to start thinking asymtotically. Recall that the probability of success was $\Theta(1/n)$ after one round; specifically, it was bounded between $1/(en)$ and $1/(2n)$. Using the expression above, we have

$$\Pr\left[\mathcal{F}[i,t]\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] \leq \left(1 - \frac{1}{en}\right)^t.$$

Now, we notice that if we set $t = en$, then we have an expression that be can plugged directly into fact (12.1). Of course en will not be an integer; so we can take $t = \lceil en \rceil$ and write:

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e}.$$

This is a very compact and useful asymptotic statement: the probability that process P_i does not succeed in any of rounds 1 through $\lceil en \rceil$ is upper-bounded by the constant e^{-1}, independent of n. Now, if we increase t by some fairly small factors, the probability that P_i does not succeed in any of rounds 1 through t drops precipitously: if we set $t = \lceil en \rceil \cdot (c \ln n)$, then we have

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^t = \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}.$$

So asymptotically, we can view things as follows. After $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant; and between then and $\Theta(n \ln n)$ this probability drops to a quantity that is extremely small — bounded by an inverse polynomial in n.

Waiting for all processes to get through. Finally, we're in a position to ask the question that was implicit in the overall set-up: how many rounds must elapse before there's a high probability that all processes will have succeeded in accessing the database at least once?

To address this, we say that the protocol "fails" after t rounds if some process has not yet succeeded in accessing the database. Let $\mathcal{F}_t$ denote the event that the protocol fails after t rounds; the goal is to find a reasonably small value of t for which $\Pr\left[\mathcal{F}_t\right]$ is small.

The event $\mathcal{F}_t$ occurs if and only if one of the events $\mathcal{F}[i,t]$ occurs; so we can write

$$\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t].$$

Previously we considered intersections of independent events, which were very simple to work with; here, on the other hand, we have a union of events that are not independent. Probabilities of unions like this can be very hard to compute exactly, and in many settings it is enough to analyze them using a simple *Union Bound*, which says that the probability of a union of events is upper-bounded by the sum of their individual probabilities:

(12.2) (The Union Bound.) *Given events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, we have*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Note that this is not an equality; but the upper bound is good enough when — as here — the union on the left-hand side represents a "bad event" that we're trying to avoid, and we want a bound on its probability in terms of consituent "bad events" on the right-hand side.

For the case at hand, recall that $\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$, and so

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right].$$

The expression on the right-hand side is a sum of n terms, each with the same value; so to make the probability of $\mathcal{F}_t$ small, we need to make each of the terms on the right significantly smaller than $1/n$. From our earlier discussion, we see that choosing $t = \Theta(n)$ will not be good enough, since then each term on the right is only bounded by a constant. If we choose $t = \lceil en \rceil \cdot (c \ln n)$, then we have $\Pr\left[\mathcal{F}[i,t]\right] \leq n^{-c}$ for each i, which is what we want. Thus, in particular, taking $t = 2\lceil en \rceil \ln n$ gives us

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right] \leq n \cdot n^{-2} = n^{-1},$$

and so all processes succeed in accessing the database within t rounds with probability at least $1 - n^{-1}$.

An interesting thing to notice is that if we had chosen a value of t equal to $qn \ln n$ for a very small value of q (rather than the coefficient $2e$ that we actually used), then we would have gotten an upper bound for $\Pr\left[\mathcal{F}[i,t]\right]$ that was larger than n^{-1}, and hence a corresponding upper bound for the overall failure probability $\Pr\left[\mathcal{F}_t\right]$ that was larger than 1 — in other words, a completely worthless bound. Yet, as we saw, by choosing larger and larger values for the coefficient q, we can drive the upper bound on $\Pr\left[\mathcal{F}_t\right]$ down to n^{-c} for any constant c we want; and this is really a very tiny upper bound. So in a sense, all the "action" in the Union Bound takes place rapidly in the period when $t = \Theta(n \ln n)$; as we vary the hidden constant inside the $\Theta(\cdot)$, the Union Bound goes from providing no information to giving an extremely strong upper bound on the probability.

We can ask whether this is simply an artifact of using the Union Bound for our upper bound, or whether it's intrinsic to the process we're observing. Although we won't do the (somewhat messy) calculations here, one can show that when t is a small constant times $n \ln n$, there really is a sizable probability that some process has not yet succeeded in accessing the database. So a rapid falling-off in the value of $\Pr\left[\mathcal{F}_t\right]$ genuinely does happen over the range $t = \Theta(n \ln n)$. For this problem, as in many problems of this flavor, we're really identifying the asymptotically "correct" value of t despite our use of the seemingly weak Union Bound.

12.2 Finding the global minimum cut

Randomization naturally suggested itself in the previous example, since we were assuming a model with many processes that could not directly communicate. We now look at a problem on graphs for which a randomized approach comes as somewhat more of a surprise, since it is a problem for which perfectly reasonable deterministic algorithms exist as well.

Given an undirected graph $G = (V, E)$, we define a *cut* of G to be a partition of V into two non-empty sets A and B. Earlier, when we looked at network flows, we worked with the closely related definition of an *s-t cut*: there, given a directed graph $G = (V, E)$ with distinguished source and sink nodes s and t, an *s-t cut* was defined to be a partition of V into sets A and B such that $s \in A$ and $t \in B$. Our definition now is slightly different, since the underlying graph is now undirected, and there is no source or sink.

For a cut (A, B) in an undirected graph G, we define the *size* of (A, B) to be the number of edges with one end in A and the other in B. A *global minimum cut* (or "global min-cut" for short) is a cut of minimum size. The term "global" here is meant to connote that any cut of the graph is allowed; there is no source or sink. We first check that network flow techniques are indeed to sufficient to find a global min-cut.

(12.3) *There is a polynomial-time algorithm to find a global min-cut in an undirected graph G.*

Proof. We start from the similarity between cuts in undirected graphs and *s-t* cuts in directed graphs — and the fact that we know how to find the latter optimally.

So given an undirected graph $G = (V, E)$, we need to transform it so that there are directed edges, and so that there is a source and sink. We first replace every undirected edge $e = (u, v) \in E$ with two oppositely oriented directed edges, $e' = (u, v)$ and $e'' = (v, u)$, each of capacity 1. Let G' denote the resulting directed graph.

Now suppose we pick two arbitrary nodes $s, t \in V$, and find the minimum *s-t* cut in G'. It is easy to check that if (A, B) is this minimum cut in G', then (A, B) is also a cut of minimum size in G *among all those that separate s from t*. But we know that the global min-cut in G must separate s from *something*, since both sides A and B are non-empty, and s belongs to only one of them. So we fix any $s \in V$, and compute the minimum *s-t* cut in G' for every other node $t \in V - \{s\}$. This is $n - 1$ directed minimum cut computations, and the best among these will be a global min-cut of G. ∎

The algorithm in (12.3) gives the strong impression that finding a global min-cut in an undirected graph is some sense a *harder* problem than finding a minimum *s-t* cut in a flow network — we had to invoke a subroutine for the latter problem $n - 1$ times in our method for solving the former. But it turns out that this is just an illusion. A sequence of increasingly simple algorithms in the late 1980's and early 1990's showed that global

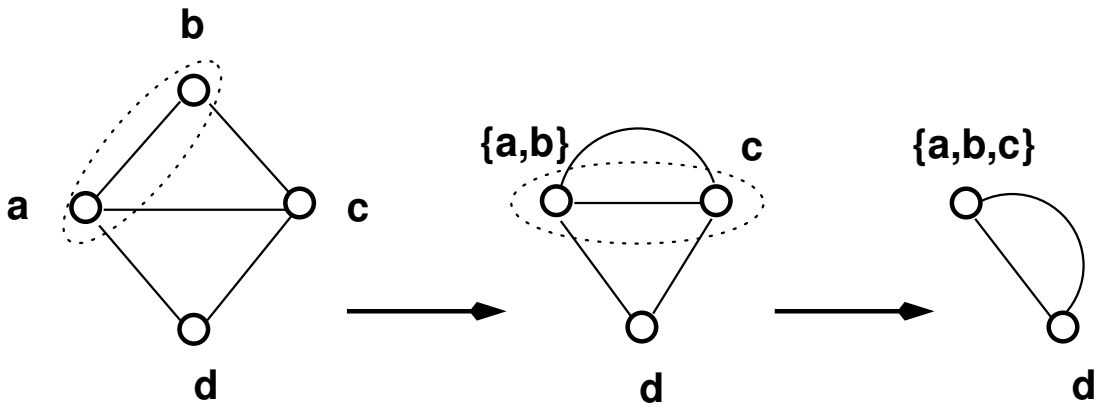


Figure 12.1: The Contraction Algorithm applied to a four-node input graph.

min-cuts in undirected graphs could actually be computed just as efficiently as s-t cuts or even more so, and by techniques that didn't require augmenting paths or even a notion of flow. The high point of this line of work came with David Karger's discovery in 1992 of the Contraction Algorithm, a randomized method that is qualitatively simpler than all previous algorithms for global min-cuts — indeed it is sufficiently simple that, on a first impression, it is very hard to believe that it actually works.

Here we describe the Contraction Algorithm in its simplest form. This version, while polynomial-time, is not among the most efficient algorithms for global min-cuts. However, subsequent optimizations to the algorithm have given it a much better running time.

The Contraction Algorithm works with a connected *multigraph* $G = (V, E)$ — this is an undirected graph that is allowed to have multiple "parallel" edges between the same pair of nodes. It begins by choosing an edge $e = (u, v)$ of G uniformly at random and *contracting* it. This means we produce a new graph G' in which u and v have been identified into a single new node w; all other nodes keep their identity. Edges that had one end equal to u and the other equal to v are deleted from G'. Each other edge e is preserved in G', but if one of its ends was equal to u or v, then this end is updated to be equal to the new node w. See Figure 12.1 for an example of this process. Note that even if G had at most one edge between any two nodes, G' may end up with parallel edges.

The Contraction Algorithm then continues recursively on G', choosing an edge uniformly at random and contracting it. As these recursive calls proceed, the constituent vertices of G' should be viewed as *super-nodes*: each super-node w corresponds to the subset $S(w) \subseteq V$ that has been "swallowed up" in the contractions that produced w. The algorithm terminates when it reaches a graph G' that has only two super-nodes nodes v_1 and v_2 (presumably with a number of parallel edges between them). Each of these super-nodes v_i has a corresponding subset $S(v_i) \subseteq V$ consisting of the nodes that have been contracted into it, and these two sets $S(v_1)$ and $S(v_2)$ form a partition of V. We output $(S(v_1), S(v_2))$ as the cut found by

the algorithm.

```
The Contraction Algorithm applied to a multigraph G = (V, E):
  For each node v, we will record
          the set S(v) of nodes that have been contracted into v.
       Initially S(v) = {v} for each v.
  If G has two nodes v1 and v2, then return the cut (S(v1), S(v2)).
  Else choose an edge e = (u, v) of G uniformly at random.
       Let G' be the graph resulting from the contraction of e,
          with new node z_uv replacing u and v.
       Define S(z_uv) = S(u) ∪ S(v).
       Apply the Contracition Algorithm recursively to G'.
  Endif
```

The algorithm is making random choices, so there is some probability that it will succeed in finding a global min-cut, and some probability that it won't. One might imagine at first that the probability of success is exponentially small — after all, there are exponentially many possible cuts of G; what's favoring the minimum cut in the process? But we'll show first that in fact the success probability is only polynomially small. It will then follow that by running the algorithm a polynomial number of times and returning the best cut found in any run, we can actually produce a global min-cut with high probability.

(12.4) *The Contraction Algorithm returns a global min-cut of G with probability at least $1/\binom{n}{2}$.*

Proof. We focus on a global min-cut (A, B) of G, and suppose it has size k; in other words, there is a set F of k edges with one end in A and the other in B. We want to give a lower bound on the probability that the Contraction Algorithm returns the cut (A, B).

Consider what could go wrong in the first step of the Contraction Algorithm — the problem would be if an edge in F were contracted. For then, a node of A and a node of B would get thrown together in the same super-node, and (A, B) could not be returned as the output of the algorithm. Conversely, if an edge not in F is contracted, then there is still a chance that (A, B) could be returned.

So what we want is an upper bound on the probability that an edge in F is contracted, and for this we need a lower bound on the size of E. Notice that if any node v had degree less than k, then the cut $(\{v\}, V-\{v\})$ would have size less than k, contradicting our assumption that (A, B) is a global min-cut. Thus, every node in G has degree at least k, and so $|E| \geq \frac{1}{2}kn$. Hence, the probability that an edge in F is contracted is at most

$$\frac{k}{\frac{1}{2}kn} = \frac{2}{n}.$$

Now consider the situation after j iterations, when there are $n-j$ super-nodes in the current graph G', and suppose that no edge in F has been contracted yet. Every cut of G' is a cut of G, and so there are at least k edges incident to every super-node of G'. Thus, G' has at least $\frac{1}{2}k(n-j)$ edges, and so the probability that an edge of F is contracted in the next iteration $j+1$ is at most

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}.$$

The cut (A, B) will actually be returned by the algorithm if no edge of F is contracted in any of iterations $1, 2, \ldots, n-2$. If we write $\mathcal{E}_j$ for the event that an edge of F is not contracted in iteration j, then we have shown $\Pr[\mathcal{E}_1] \geq 1 - 2/n$ and $\Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \geq 1 - 2/(n-j)$. We are interested in lower-bounding the quantity $\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-2}]$, and we can check by unwinding the formula for conditional probability that this is equal to

$$\begin{aligned}
&\Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \cdots \Pr[\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-3}] \\
&\quad \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{n-j}\right) \cdots \left(1 - \frac{2}{3}\right) \\
&\quad = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&\quad = \frac{2}{n(n-1)} = 1/\binom{n}{2}.
\end{aligned}$$

■

So we now know that a single run of the Contraction Algorithm fails to find a global min-cut with probability at most $(1 - 1/\binom{n}{2})$. This number is very close to 1, of course, but we can "amplify" our probability of success simply by repeatedly running the algorithm, with different random choices, and taking the best cut we find. By fact (12.1), if we run the algorithm $\binom{n}{2}$ times, then the probability we have failed to find a global min-cut in any run is at most

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2}} \leq \frac{1}{e}.$$

And it's easy to drive the failure probability below $1/e$ with further repetitions: if we run the algorithm $\binom{n}{2} \ln n$ times, then the probability we fail to find a global min-cut is at most $e^{-\ln n} = 1/n$.

The overall running time required to get a high probability of success is polynomial in n, since each run of the Contraction Algorithm takes polynomial time, and we run it a polynomial number of times. Its running time will be fairly large compared to the best network flow techniques, since we perform $\Theta(n^2)$ iterations, and each takes at least $\Omega(m)$ time. We have chosen to describe this version of the Contraction Algorithm since it is the simplest and most elegant; and it has been shown that some clever optimizations to the way in which multiple runs are performed can improve the running time considerably.

The number of global minimum cuts. The analysis of the Contraction Algorithm provides a surprisingly simple answer to the following question: given an undirected graph $G = (V, E)$ on n nodes, what is the maximum number of global min-cuts it can have (as a function of n)?

For a directed flow network, it's easy to see that the number of minimum s-t cuts can be exponential in n. For example, consider a directed graph with nodes $s, t, v_1, v_2, \ldots, v_n$, and unit-capacity edges (s, v_i) and (v_i, t) for each i. Then s together with any subset of $\{v_1, v_2, \ldots, v_n\}$ will constitute the source side of a minimum cut, and so there are 2^n minimum s-t cuts.

But for global min-cuts in an undirected graph, the situation looks quite different — if one spends some time trying out examples, one finds that the n-node cycle has $\binom{n}{2}$ global min-cuts (obtained by cutting any two edges), and it is not clear how to construct an undirected graph with more.

We know show how the analysis of the Contraction Algorithm settles this question immediately, establishing that the n-node cycle is indeed an extreme case.

(12.5) *An undirected graph $G = (V, E)$ on n nodes has at most $\binom{n}{2}$ global min-cuts.*

Proof. The key point is that the proof of (12.4) actually established more than was claimed. Suppose G is a graph, and let $C_1, \ldots, C_r$ denote all its global min-cuts. Let $\mathcal{E}_i$ denotes the event that C_i is returned by the Contraction Algorithm, and let $\mathcal{E} = \cup_{i=1}^{r} \mathcal{E}_i$ denote the event that the algorithm returns any global min-cut.

Then although (12.4) simply asserts that $\Pr[\mathcal{E}] \geq 1/\binom{n}{2}$, its proof actually shows that for each i, we have $\Pr[\mathcal{E}_i] \geq 1/\binom{n}{2}$. Now, each pair of events $\mathcal{E}_i$ and $\mathcal{E}_j$ are disjoint — since only one cut is returned by any given run of the algorithm, so by the Union Bound for disjoint events (12.26), we have

$$\Pr[\mathcal{E}] = \Pr[\cup_{i=1}^{r} \mathcal{E}_i] = \sum_{i=1}^{r} \Pr[\mathcal{E}_i] \geq r/\binom{n}{2}.$$

But clearly $\Pr[\mathcal{E}] \leq 1$, and so we must have $r \leq \binom{n}{2}$. ■

12.3 Random Variables and their Expectations

Thus far, our analysis of randomized algorithms and processes has been based on identifying certain "bad events," and bounding their probabilities. This is a qualitative type of analysis, in the sense that the algorithm either succeeds or it doesn't. A more quantitative style of analysis would consider certain parameters associated with the behavior of the algorithm — for example its running time, or the quality of the solution it produces — and seek to determine the *expected* size of these parameters over the random choices made by the

algorithm. In order to make such analysis possible, we need the fundamental notion of a *random variable*.

Given a probability space defined over a sample space Ω, A *random variable* is a function $X : \Omega \to \mathbf{N}$ such that for each natural number j, the set $X^{-1}(j) = \{\omega : X(\omega) = j\}$ is an event. Thus we can write $\Pr[X = j]$ as loose shorthand for $\Pr[X^{-1}(j)]$; it is because we can ask about X's probability of taking a given value that we think of it as a "random variable."

Given a random variable X, we are often interested in determining its *expectation* — the "average value" assumed by X. We define this as

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j],$$

declaring this to have the value ∞ if the sum diverges. Thus, for example, if X takes each of the values in $\{1, 2, \ldots, n\}$ with probability $1/n$, then $E[X] = 1(1/n)+2(1/n)+\cdots+n(1/n) = \binom{n+1}{2}/n = (n+1)/2$.

Here's a more useful example, in which we see how an appropriate random variable lets us talk about something like the "running time" of a simple random process. Suppose we have a coin that comes up `heads` with probability $p > 0$, and `tails` with probability $1-p$. Different flips of the coin have independent outcomes. If we flip the coin until we first get a `heads`, what's the expected number of flips we will perform? To answer this, we let X denote the random variable equal to the number of flips performed. We have $\Pr[X = j] = (1-p)^{j-1}p$: for the process to take exactly j steps, the first $j-1$ flips must come up `tails`, and the j^{th} must come up `heads`. Now applying the definition, we have

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \sum_{j=1}^{\infty} j(1-p)^{j-1}p = \frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^{j} = \frac{p}{1-p} \cdot \frac{(1-p)}{p^2} = \frac{1}{p}.$$

Thus we get the following intuitively sensible result:

(12.6) *If we repeatedly perform independent trials of an experiment, each of which succeeds with probability $p > 0$, then expected number of trials we need to perform before the first success is $1/p$.*

Linearity of Expectation. In previous lectures, we broke events down into unions of much simpler events, and worked with the probabilities of these simpler events. This is a powerful technique when working with random variables as well, and it is based on the principle of *linearity of expectation*.

(12.7) Linearity of Expectation. *Given two random variables X and Y defined over the same probability space, we can define $X+Y$ to be the random variable equal to $X(\omega)+Y(\omega)$ on a sample point $\omega \in \Omega$. For any X and Y, we have*

$$E[X + Y] = E[X] + E[Y].$$

We omit the proof, which is not difficult. Much of the power of (12.7) comes from the fact that it applies to the sum of *any* random variables; no restrictive assumptions are needed. As a result, if we need to compute the expectation of a complicated random variable X, we can first write it as a sum of simpler random variables $X = X_1 + X_2 + \cdots + X_n$, compute each $E[X_i]$, and then determine $E[X] = \sum E[X_i]$. We now look at some examples of this principle in action.

Guessing Cards

Memoryless Guessing. To amaze your friends, you have them shuffle a deck of 52 cards and then turn over one card at a time. Before each card is turned over, you predict its identity. Unfortunately, you don't have any particular psychic abilities — and you're not so good at remembering what's been turned over already — so your strategy is to simply guess a card uniformly at random from the full deck each time. On how many predictions do you expect to be correct?

Let's work this out for the more general setting in which the deck has n distinct cards, using X to denote the random variable equal to the number of correct predictions. A surprisingly effortless way to compute X is to define the random variable X_i, for $i = 1, 2, \ldots, n$, to be equal to 1 if the i^{th} prediction is correct, and 0 otherwise. Notice that $X = X_1 + X_2 + \cdots + X_n$, and

$$E[X_i] = 0 \cdot \Pr[X_i = 0] + 1 \cdot \Pr[X_i = 1] = \Pr[X_i = 1] = \frac{1}{n}.$$

It's worth pausing to note a useful fact that is implicitly demonstrated by the above calculation: if Z is any random variable that only takes the values 0 or 1, then $E[Z] = \Pr[Z = 1]$.

Since $E[X_i] = \frac{1}{n}$ for each i, we have

$$E[X] = \sum_{i=1}^{n} E[X_i] = n\left(\frac{1}{n}\right) = 1.$$

Thus,

(12.8) *The expected number of correct predictions under the memoryless guessing strategy is* 1*, independent of the number of cards* n*.*

Trying to compute $E[X]$ directly from the definition $\sum_{j=0}^{\infty} j \cdot \Pr[X = j]$ would be much more painful; even working out $\Pr[X = j]$ for various values of j is not immediate. A significant amount of complexity is hidden away in the seemingly innocuous statement of (12.7).

Guessing with Memory. Now let's consider a second scenario. Your psychic abilities have not developed any further since last time, but you have become very good at remembering which cards have already been turned over. Thus, when you predict the next card now, you only guess uniformly from among the cards *not yet seen*. How many correct predictions do you expect to make with this strategy?

Again, let the random variable X_i take the value 1 if the i^{th} prediction is correct, and 0 otherwise. In order for the i^{th} prediction to be correct, you need only guess the correct one out of $n - i + 1$ remaining cards; hence

$$E[X_i] = \Pr[X_i = 1] = \frac{1}{n - i + 1},$$

and so we have

$$\Pr[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n - i + 1} = \sum_{i=1}^{n} \frac{1}{i}.$$

This last expression $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is called the n^{th} *harmonic number*, and is denoted H_n. We saw it earlier in the course, when we studied approximation algorithms; there, we showed how it closely shadows the value of $\int_1^{n+1} dx/x = \ln(n+1)$. For our purposes here, we re-state the basic bound on H_n as follows.

(12.9) $\ln(n + 1) < H_n < 1 + \ln n$, *and more loosely,* $H_n = \Theta(\log n)$.

Thus, once you are able to remember the cards you've already seen, the expected number of correct predictions increases significantly above 1.

(12.10) *The expected number of correct predictions under the guessing strategy with memory is* $H_n = \Theta(\log n)$.

12.4 A Randomized Approximation Algorithm for MAX-3-SAT

When we studied NP-completeness, a core problem was *3-SAT*: Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Intuitively, we can imagine such a problem arising in a system that tries to decide the truth or falsehood of statements about the world (the variables $\{x_i\}$), given pieces of information that relate them (the clauses $\{C_j\}$). Now, the world is a fairly contradictory place, and if our system gathers enough information, it could well end up with a set of clauses that has no satisfying truth assignment. What then?

A natural approach, if we can't find a truth assignment that satisfies all clauses, is to turn the *3-SAT* instance into an optimization problem: given the set of input clauses

$C_1, \ldots, C_k$, find a truth assignment that satisfies *as many as possible*. We'll call this the *Maximum 3-Satisfiability* problem (or *MAX 3-SAT* for short). Of course, this is an NP-hard optimization problem, since it's NP-complete to decide whether the maximum number of simultaneously satisfiable clauses is equal to k. Let's see what can be said about polynomial-time approximation algorithms.

A remarkably simple randomized algorithm turns out to give a strong performance guarantee for this problem. Suppose we set each variable $x_1, \ldots, x_n$ independently to 0 or 1 with probability $\frac{1}{2}$ each. What is the expected number of clauses satisfied by such a random assignment?

Let Z denote the random variable equal to the number of satisfied clauses. As in the previous section, let's decompose Z into a sum of random variables that each take the value 0 or 1; specifically, let $Z_i = 1$ if the clause C_i is satisfied, and 0 otherwise. Thus, $Z = Z_1 + Z_2 + \cdots + Z_k$. Now, $E[Z_i]$ is equal to the probability that C_i is satisfied, and this can be computed easily as follows. In order for C_i *not* to be satisfied, each of its three variables must be assigned the value that fails to make it true; since the variables are set independently, the probability of this is $(\frac{1}{2})^3 = \frac{1}{8}$. Thus, clause C_i is satisfied with probability $1 - \frac{1}{8} = \frac{7}{8}$, and so $E[Z_i] = \frac{7}{8}$.

Using linearity of expectation, we see that the expected number of satisfied clauses is $E[Z] = E[Z_1] + E[Z_2] + \cdots + E[Z_k] = \frac{7}{8}k$. Since no assignment can satisfy more than k clauses, we have the following guarantee.

(12.11) *The expected number of clauses satisfied by a random assignment is within an approximation factor $\frac{7}{8}$ of optimal.*

But if we look at what really happened in the (admittedly simple) analysis of the random assignment, it's clear that something stronger is going on. For any random variable, there must be some point at which it assumes some value at least as large as its expectation. We've shown that for every instance of *3-SAT*, a random truth assignment satisfies a $\frac{7}{8}$ fraction of all clauses in expectation; so, in particular, there must *exist* a truth assignment that satisfies a number of clauses that is at least as large as this expectation.

(12.12) *For every instance of* 3-SAT*, there is a truth assignment that satisfies at least a $\frac{7}{8}$ fraction of all clauses.*

There is something genuinely surprisingly about the statement of (12.12) . We have arrived at a non-obvious fact about *3-SAT* — the existence of an assignment satisfying many clauses — whose statement has nothing to do with randomization; but we have done so by a randomized construction. And in fact, the randomized construction provides what is quite possibly the simplest proof of (12.12). This a fairly widespread principle in the area of combinatorics, that one can show the existence of some structure by showing that a random

construction produces it with positive probability. Constructions of this sort are said to be applications of the *probabilistic method*.

Here's an cute but minor application of (12.12): Every instance of *3-SAT* with at most seven clauses is satisfiable. Why? If the instance has $k \leq 7$ clauses, then (12.12) implies that there is an assignment satisfying at least $\frac{7}{8}k$ of them. But when $k \leq 7$, it follows that $\frac{7}{8}k > k - 1$; and since the number of clauses satisfied by this assignment must be an integer, it must be equal to k. In other words, all clauses are satisfied.

Waiting to find a good assignment. Suppose we aren't satisfied with a "one-shot" algorithm that produces a single assignment with a large number of satisfied clauses in expectation. Rather, we'd like a randomized algorithm whose expected running time is polynomial, and which is guaranteed to output a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.

A simple way to do this is to generate random truth assignments until one of them satisfies at least $\frac{7}{8}k$ clauses. We know that such an assignment exists, by (12.12); but how long will it take until we find one by random trials?

This is a natural place to apply the waiting time bound we derived in (12.6). If we can show that the probability a random assignment satisfies at least $\frac{7}{8}k$ clauses is at least p, then the expected number of trials performed by the algorithm is $1/p$. So in particular, we'd like to show that this quantity p is at least as large as an inverse polynomial in n and k.

For $j = 0, 1, 2, \ldots, k$, let p_j denote the probability that a random assignment satisfies exactly j clauses. So the expected number of clauses satisfied, by the definition of expectation, is equal to $\sum_{j=0}^{k} jp_j$; and by the analysis above, this is equal to $\frac{7}{8}k$. We are interested in the quantity $p = \sum_{j \geq 7k/8} p_j$. How can we use the lower bound on the expected value to give a lower bound on this quantity?

We start by writing

$$\frac{7}{8}k = \sum_{j=0}^{k} jp_j = \sum_{j<7k/8} jp_j + \sum_{j \geq 7k/8} jp_j.$$

Now, let k' denote the largest natural number that is strictly smaller than $\frac{7}{8}k$. The right-hand side of the above equation only increases if we replace the terms in the first sum by $k'p_j$ and the terms in the second sum by kp_j. We also observe that $p = \sum_{j<7k/8} p_j = 1 - p$, and so

$$\frac{7}{8}k \leq \sum_{j<7k/8} k'p_j + \sum_{j \geq 7k/8} kp_j = k'(1-p) + kp \leq k' + kp$$

and hence $kp \geq \frac{7}{8}k - k'$. But $\frac{7}{8}k - k' \geq \frac{1}{8}$, since k' is a natural number strictly smaller than

$\frac{7}{8}$ times another natural number, and so

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8k}.$$

This was our goal — to get a lower bound on p — and so by the waiting time bound (12.6) , we see that the expected number of trials needed to find the satisfying assignment we want is at most $8k$.

(12.13) *There is a randomized algorithm with polynomial expected running time that is guaranteed to produce a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.*

12.5 Computing the Median: Randomized Divide-and-Conquer

We've seen the divide-and-conquer paradigm for designing algorithms at various earlier points in the course. Divide-and-conquer often works well in conjunction with randomization, and we illustrate this by giving a divide-and-conquer algorithm to compute the median of n numbers. The "divide" step here is performed using randomization; consequently, we will use expectations of random variables to analyze the time spent on recursive calls.

Suppose we are given a set of n numbers $S = \{a_1, a_2, \ldots, a_n\}$. Their *median* is the number that would be in the middle position if we were to sort them. There's an annoying technical difficulty if n is even, since then there is no "middle position"; thus, we define things precisely as follows: the median of $S = \{a_1, a_2, \ldots, a_n\}$ is equal to the k^{th} largest element in S, where $k = (n+1)/2$ if n is odd, and $k = n/2$ if n is even. In what follows, we'll assume for the sake of simplicity that all the numbers are distinct. Without this assumption, the problem becomes notationally more complicated, but no new ideas are brought into play.

It is clearly easy to compute the median in time $O(n \log n)$ if we simply sort the numbers first. But if one begins thinking about the problem, it's far from clear why sorting is *necessary* for computing the median, or even why $\Omega(n \log n)$ time is necessary. In fact, we'll show how a simple randomized approach, based on divide-and-conquer, yields an expected running time of $O(n)$.

The first key step toward getting an expected linear running time is to move from median-finding to the more general problem of *selection*. Given a set of n numbers S and a number k between 1 and n, consider the function `Select`(S, k) that returns the k^{th} largest element in S. As special cases, `Select` includes the problem of finding the median of S via `Select`$(S, n/2)$ or `Select`$(S, (n+1)/2)$; it also includes the easier problems of finding the minimum (`Select`$(S, 1)$) and the maximum (`Select`(S, n)). Our goal is to design an algorithm that implements `Select` so that it runs in expected time $O(n)$.

The basic structure of the algorithm implementing `Select` is as follows. We choose an element $a_i \in S$, the "splitter," and form the sets $S^- = \{a_j : a_j < a_i\}$ and $S^+ = \{a_j : a_j > a_i\}$. We can then determine which of S^- or S^+ contains the k^{th} largest element, and iterate only on this one. Without specifying yet how we plan to choose the splitter, here's a more concrete description of how we form the two sets and iterate.

```
Select(S,k):
Choose a splitter a_i ∈ S.
For each element a_j of S
   Put a_j in S^- if a_j < a_i.
   Put a_j in S^+ if a_j > a_i.
End
If |S^-| = k − 1 then
   The splitter a_i was in fact the desired answer.
Else if |S^-| ≥ k then
   The k^th largest element lies in S^-.
   Recursively call Select(S^-, k)
Else suppose |S^-| = ℓ < k − 1.
   The k^th largest element lies in S^+.
   Recursively call Select(S^+, k − 1 − ℓ)
Endif
```

Observe that the algorithm is always called recursively on a strictly smaller set, so it must terminate. Also, observe that if $|S| = 1$, then we must have $k = 1$, and indeed the single element in S will be returned by the algorithm. Finally, from the choice of which recursive call to make, it's clear by induction that the right answer will be returned when $|S| > 1$ as well. Thus we have

(12.14) *Regardless of how the splitter is chosen, the algorithm above returns the k^{th} largest element of S.*

Now let's consider the running time of `Select`. Assuming we can select a splitter in linear time, the rest of the algorithm takes linear time plus the time for the recursive call. But how is the running time of the recursive call affected by the choice of the splitter? Essentially, it's important that the splitter significantly reduce the size of the set being considered, so that we don't keep making passes through large sets of numbers many times. So a good choice of splitter should produce sets S^- and S^+ that are approximately equal in size.

For example, if we could always choose the median as the splitter, then we could bound the running as follows. Let cn be the running time for `Select`, not counting the time for the recursive call. Then with medians as splitters, the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T(n/2) + cn$. Unwinding this recursively, we get

$$T(n) \leq cn + cn/2 + cn/4 + cn/8 + \cdots \leq 2cn,$$

since the sum in the middle is a geometric series that converges. Of course, hoping to be able to use the median as the splitter is rather circular, since the median is what we want to compute in the first place! But, in fact, we see that any "well-centered" element can serve as a good splitter: if we had a way to choose splitters a_i such that there were at least εn elements both larger and smaller than a_i, then the size of the sets in the recursive call would shrink by a factor of at least $(1-\varepsilon)$ each time. Thus, the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T((1-\varepsilon)n) + cn$. Unwinding this, we get

$$T(n) \leq cn + (1-\varepsilon)cn + (1-\varepsilon)^2 cn + \cdots = \left[1 + (1-\varepsilon) + (1-\varepsilon)^2 + \cdots\right] cn \leq \frac{1}{\varepsilon} \cdot cn,$$

since again we have a convergent geometric series.

Indeed, the only thing to really beware of is a very "off-center" splitter. For example, if we always chose the minimum element as the splitter, then we may end up with a set in the recursive call that's only one element smaller than we had before. In this case, the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T(n-1) + cn$. Unwinding this, we see that there's a problem:

$$T(n) \leq cn + c(n-1) + c(n-2) + \cdots = \frac{cn(n+1)}{2} = \Omega(n^2).$$

Choosing a "well-centered" splitter — in the sense defined above — is certainly similar in flavor to our original problem of choosing the median; but the situation is really not so bad, since *any* well-centered splitter will do. We observe that a fairly large fraction of the numbers are reasonably well-centered, and so we will be likely to end up with one simply by choosing a splitter uniformly at random. The analysis of the running time with a random splitter is intuitively based on this idea; we expect the size of the set under consideration to go down by a fixed constant fraction every iteration, so we should get a convergent series and hence a linear bound as above. We now show how to make this precise.

We'll say that the algorithm is in *phase* j when the size of the set under consideration is at most $n(\frac{3}{4})^j$ but greater than $n(\frac{3}{4})^{j+1}$. Let's try to bound the expected time spent by the algorithm in phase j. In a given iteration of the algorithm, we say that an element of the set under consideration is *central* if at least a quarter of the elements are smaller than it, and at least a quarter of the elements are larger than it.

Now, observe that if a central element is chosen as a splitter, then at least a quarter of the set will be thrown away, the set will shrink by a factor of $\frac{3}{4}$ or better, and the current phase will come to an end. Moreover, half of all the elements in the set are central, and so the probability that our random choice of splitter produces a central element is $\frac{1}{2}$. Hence, by our simple waiting-time bound (12.6), the expected number of iterations before a central element is found is two; and so the expected number of iterations spent in phase j, for any j, is at most two.

This is pretty much all we need for the analysis. Let X be a random variable equal to the number of steps taken by the algorithm. We can write it as the sum $X = X_0 + X_1 + X_2 + \cdots$, where X_j is the expected number of steps spent by the algorithm in phase j. When the algorithm is in phase j, the set has size at most $n(\frac{3}{4})^j$, and so the number of steps required for one iteration in phase j is at most $cn(\frac{3}{4})^j$ for some constant c. Above, we argued that the expected number of iterations spent in phase j is at most two, and hence we have $E[X_j] \leq 2cn(\frac{3}{4})^j$. Thus, we can bound the total expected running time using linearity of expectation:

$$E[X] = \sum_j E[X_j] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j \left(\frac{3}{4}\right)^j \leq 8cn,$$

since the sum $\sum_j (\frac{3}{4})^j$ is a geometric series that converges. Thus we have the desired result:

(12.15) *The expected running time of* `Select`(n, k) *is* $O(n)$.

12.6 Randomized Caching

In the most basic set-up of the *cache maintenance problem*, we consider a processor whose full memory has n addresses; it is also equipped with a *cache* containing k slots of memory that can be accessed very quickly. We can keep copies of k items from the full memory in the cache slots, and when a memory location is accessed, the processor will first check the cache to see if it can be quickly retrieved. We say the request is a *cache hit* if the cache contains the requested item; in this case, the access is very quick. We say that request is a *cache miss* if the requested item is not in the cache; in this case, the access takes much longer, and moreover, one of the items currently in the cache must be *evicted* to make room for the new item. (We will assume that the cache is kept full at all times.)

The goal of a cache maintenance algorithm is to minimize the number of cache misses, which are the truly expensive parts of the process. The sequence of memory references is not under the control of the algorithm — this is simply dictated by the application that is running — and so the job of the algorithms we consider is simply to decide on an *eviction policy*: which item currently in the cache should be evicted on each cache miss?

Given that the future sequence of memory requests is completely unknown, this seems like a perfect setting in which to behave randomly: without any information about which items will be needed in the future, why not just evict one from the cache uniformly at random? We will call this the *Purely Random* algorithm.

Since this is a randomized algorithm, the number of misses it incurs on a sequence σ of memory requests is a random variable R_σ. (Note that the sample space underlying this random variable encodes the random decisions made by the algorithm; thus, there is a different random variable R_σ for each request sequence σ.) To assess the performance of the algorithm, we will compare the expected number of misses it makes on a sequence σ to the

minimum number of misses it is possible to make on σ. We will use $f(\sigma)$ to denote this latter quantity.

(12.16) *For every constant $c < k$, there are arbitrarily long sequences of memory requests σ for which $E[R_\sigma] > c \cdot f(\sigma)$.*

Proof. To prove this, we need to consider any $c < k$, and produce an example of a sequence σ with the required properties. Let $S_0 = \{s_1, s_2, \ldots, s_k\}$ be the items initially in the cache, and let $S = S_0 - \{s_k\}$. For a constant r that can be as large as we like, we choose t other items from the processor's full memory, labeled $t_1, t_2, \ldots, t_r$. For $j = 1, 2, \ldots, r$, we define $S_j = S \cup \{t_j\}$.

The construction of σ is divided into r *phases*. In phase $j \leq r$, we order the elements of S_j arbitrarily, and request these elements in strict round-robin order until each element of S_j has been requested exactly b times, where b is a constant that can be as large as we like. We call one such pass through the elements of S_j a *cycle*; thus, phase j consists of b cycles. After these cycles, phase j comes to an end, and the construction of σ continues.

We first observe that it is possible to handle all the requests in σ while incurring at most one cache miss per phase. Indeed, the first time t_j is requested in phase j, the item t_{j-1} can be evicted while the item t_j is brought into the cache. After this, there will be no more misses in phase j. Thus we have $f(\sigma) \leq t$.

Now, how many misses to we expect the Purely Random algorithm to make? In other words, if X_j is the random variable equal to the number of misses in phase j, what is $E[X_j]$? In each cycle of the phase, the algorithm will incur at least one miss if the cache contents at the beginning of the cycle includes t_{j-1}. So $E[X_j]$ is lower-bounded by the expected number of cycles until t_{j-1} is evicted. Thus, for purposes of this analysis, we are essentially in the set-up of (12.6): each cycle through S_j is like an experiment in which the successful outcome for the algorithm is the eviction of t_{j-1}, an event of probability $1/k$; and so we can almost apply (12.6) to conclude that the expected number of misses is at least k. The only problem is that there's another way for the algorithm to stop incurring misses: the phase may end after b cycles, even though t_{j-1} was never evicted. So we need to re-do the calculation the proved (12.6) , taking this into account. If we let Y_j be the random variable equal to the number of cycles until the eviction of t_{j-1}, or equal to b if t_{j-1} is never evicted, then we have

$$\begin{aligned} E[X_j] &\geq E[Y_j] \\ &= \sum_{i=0}^{\infty} i \cdot \Pr[Y_j = i] = b\left(1 - \frac{1}{k}\right)^b + \sum_{i=1}^{b-1} \frac{j}{k}\left(1 - \frac{1}{k}\right)^{i-1} \\ &= b\left(1 - \frac{1}{k}\right)^b + k - (b+k)\left(1 - \frac{1}{k}\right)^b = k\left(1 - \left(1 - \frac{1}{k}\right)^b\right). \end{aligned}$$

Since $c < k$, we can choose b large enough to ensure that $E[X_j] > c$. Hence we have

$$E[R_\sigma] = \sum_{j=1}^{t} E[X_j] > ct \geq c \cdot f(\sigma),$$

as required. ■

Is the bound suggested by (12.16) the best we can hope to obtain using a randomized approach? In fact, it's possible to do much better relative to the benchmark provided by $f(\sigma)$. The point is that completely random eviction is overkill; we want an algorithm that is sensitive to the difference between the following two possibilities: (a) in the recent past, the request sequence has contained more than k distinct items; or (b) in the recent past, the request sequence has come exclusively from a set of at most k items. In the first case, we know that $f(\sigma)$ must be increasing, since no algorithm can handle more than k distinct items from cache without incurring a miss. But in the second case, it's possible that σ is passing through a long stretch in which an optimal algorithm need not incur any misses at all. It is here that our randomized algorithm must make sure that it incurs very few misses. Notice how this captures the essence of the construction used in (12.16): each phase consisted of a long interval of time over which an optimal algorithm could operate without misses; yet our Purely Random algorithm was very slow to "realize" this.

Motivated by these considerations, we now describe the *Marking* algorithm; while still randomized, it prefers evicting items that don't seem to have been used in a long time. The Marking algorithm operates in *phases*; the description of one phase is as follows.

```
Each memory item can be either marked or unmarked.
At the beginning of the phase, all items are unmarked.
On a request to item s:
   Mark s.
   If s is in the cache, then evict nothing.
   Else s is not in the cache:
      If all items currently in the cache are marked then
         Declare the phase over.
         Processing of s is deferred to start of next phase.
      Else evict an item chosen uniformly at random from
            the set of all unmarked items currently in the cache.
      Endif
   Endif
```

So the difference from the Purely Random algorithm is that the random choice for eviction is made only over the *unmarked* items, not over all items. Notice how this intuitively favors the eviction of items that don't seem to have been recently requested; in particular, this strategy works extremely well on the construction in (12.16).

In fact, the Marking algorithm obtains an exponential improvement over the Purely Random algorithm, relative to the quantity $f(\sigma)$. In the following statement, let M_σ denote the random variable equal to the number of cache misses incurred by the Marking algorithm on the request sequence σ.

(12.17) *For every request sequence σ, we have $E[M_\sigma] \leq 2H_k \cdot f(\sigma) = O(\log k) \cdot f(\sigma)$.*

Proof. In the analysis, we picture an optimal caching algorithm operating on σ alongside the Marking algorithm, incurring an overall cost of $f(\sigma)$. For simplicity, we imagine a "phase 0" that takes place before the first phase, in which all the items initially in the cache are requested once. This does not affect the cost of either the Marking algorithm or the optimal algorithm. We also imagine that the final phase r ends with an epilogue in which every item currently in the cache of the optimal algorithm is requested twice in round-robin fashion. This does not increase $f(\sigma)$; and by the end of the second pass through these items, the Marking algorithm will contain each of them in its cache, and each will be marked.

In the middle of any phase, an unmarked item s can be one of two distinct types. We call it *clean* if was not marked in the previous phase either, and we call it *stale* if was marked in the previous phase. Here is how we can picture the history of a phase, from the Marking algorithm's point of view. At the beginning of the phase, all items in the cache are stale. Over the course of the phase, there will be k requests to unmarked items: each such request increases the number of marked items by one, and the phase ends with k marked items. Requests to marked items do not result in cache misses, since every marked item is in the cache for the remainder of the phase; thus, the only possible misses come on these k requests to unmarked items. Every unmarked item is either clean or stale at the moment it is requested; so if we let c_j denote the number of requests in phase j to clean items, then there are $k - c_j$ requests in phase j to stale items.

Let X_j denote the number of misses incurred by the Marking algorithm in phase j. Each request to a clean item results in a guaranteed miss for the Marking algorithm; since the clean item was not marked in the previous phase, it cannot possibly be in the cache when it is requested in phase j. Thus, the Marking algorithm incurs at least c_j misses in phase j because of requests to clean items. Stale items, on the other hand, are a more subtle matter; on a request to a stale item s, the concern is whether the Marking algorithm evicted it earlier in the phase, and now incurs a miss as it has to bring it back in. What is the probability that the i^{th} request to a stale item, say s, results in a miss? Suppose that there have been $c \leq c_j$ requests to clean items thus far in the phase. Then the cache contains the c formerly clean items that are now marked, $i-1$ formerly stale items that are now marked, and $k-c-i+1$ items that are still stale. But there are $k - i + 1$ items overall that are still stale; and since exactly $k - c - i + 1$ of them are in the cache, the remaining c of them are not. Each of the $k - i + 1$ stale items is equally likely to be no longer in the cache, and so s is not in the

cache at this moment with probability $\frac{c}{k-i+1} \leq \frac{c_j}{k-i+1}$. This is the probability of a miss on the request to s. Summing over all requests to unmarked items, we have

$$E[X_j] \leq c_j + \sum_{i=1}^{k-c_j} \frac{c_j}{k-i+1} \leq c_j \left[1 + \sum_{\ell=c_j+1}^{k} \frac{1}{\ell}\right] = c_j(1 + H_k - H_{c_j}) \leq c_j H_k.$$

Thus, the total expected number of misses incurred by the Marking algorithm is

$$E[M_\sigma] = \sum_{j=1}^{r} E[X_j] \leq H_k \sum_{j=1}^{r} c_j.$$

Now we need a lower bound on the number of misses $f(\sigma)$ incurred by the optimal algorithm. Let $f_j(\sigma)$ denote the number of misses incurred by the optimal algorithm in phase j, so that $f(\sigma) = \sum_{j=1}^{r} f_j(\sigma)$. A key property of the way in which the Marking algorithm defines phases is that in any phase j, there are requests to k distinct items. Moreover, by our definition of *clean*, there are requests to c_{j+1} further items in phase $j+1$; so between phases j and $j+1$, there are $k + c_{j+1}$ distinct items requested. It follows that the optimal algorithm must incur at least c_j misses over the course of phases j and $j+1$, so $f_j(\sigma) + f_{j+1}(\sigma) \geq c_{j+1}$. This holds even for $j = 0$, since the optimal algorithm incurs c_1 misses in phase 1. Thus we have

$$\sum_{j=0}^{r-1} (f_j(\sigma) + f_{j+1}(\sigma)) \geq \sum_{j=0}^{r-1} c_{j+1}.$$

But the left-hand side is at most $2\sum_{j=1}^{r} f_j(\sigma) = 2f(\sigma)$, and the right-hand side is at least $E[M_\sigma]/H_k$, and so we have $E[M_\sigma] \leq 2H_k f(\sigma)$. ■

12.7 Chernoff Bounds

We defined the expectation of a random variable formally in an earlier lecture, and have worked with this definition and its consequences ever since. Intuitively, we have a sense that the value of a random variable ought to be "near" its expectation with reasonably high probability, but we have not yet explored the extent to which this is true. We now turn to some results that allow us to reach conclusions like this, and see a sampling of the applications that follow.

We say that two random variables X and Y are *independent* if for any values i and j, the events $\Pr[X = i]$ and $\Pr[Y = j]$ are independent. Now, consider a random variable X that is a sum of several independent 0-1-valued random variables: $X = X_1 + X_2 + \cdots + X_n$, where each X_i takes the value 1 with probability p_i, and the value 0 otherwise. Notice that $E[X_i] = p_i$, and so by linearity of expectation we have $\mu = E[X] = \sum_{i=1}^{n} p_i$. Intuitively, the independence of the random variables $X_1, X_2, \ldots, X_n$ suggests that their fluctuations are likely to "cancel out," and so their sum X will have a value close to its expectation μ with

high probability. This is in fact true, and we state two concrete versions of this result: one bounding the probability that X deviates above μ, the other bounding the probability that X deviates below μ. Following tradition, we call these results *Chernoff bounds*, after one of the probabilists who first established bounds of this form.

(12.18) *Let $X, X_1, X_2, \ldots, X_n$ and μ be defined as above. Then for any $\delta > 0$, we have*

$$\Pr[X > (1+\delta)\mu] < \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu.$$

(12.19) *Let $X, X_1, X_2, \ldots, X_n$ and μ be defined as above. Then for any $1 > \delta > 0$, we have*

$$\Pr[X < (1-\delta)\mu] < e^{-\frac{1}{2}\mu\delta^2}.$$

Note that the statements of the results are not symmetric, and this makes sense: for the upper bound, it is interesting to consider values of δ much larger than 1, while this would not make sense for the lower bound. The proofs of these two results are not very long, but they are fairly technical. For the applications that follow, the statements of (12.18) and (12.19) are the key things to keep in mind.

12.8 Load Balancing

In earlier lectures, we considered a distributed system in which communication among processes was difficult, and randomization to some extent replaced explicit coordination and synchronization. We now re-visit this theme through another stylized example of randomization in a distributed setting.

Suppose we have a system in which m jobs arrive in a stream and need to be processed immediately. We have a collection of n identical processors that are capable of performing the jobs; so the goal is to assign each job to a processor in a way that balances the workload evenly across the processors. If we had a central controller for the system that could receive each job and hand it off to the processors in round-robin fashion, it would be trivial to make sure that each processor received at most $\lceil m/n \rceil$ jobs — the most even balancing possible.

But suppose the system lacks the coordination or centralization to implement this. A much more light-weight approach would be to simply assign each job to one of the processors uniformly at random. Intuitively, this should also balance the jobs evenly, since each processor is equally to get each job. At the same time, since the assignment is completely random, one doesn't expect everything to end up perfectly balanced. So we ask: how well does this simple randomized approach work?

We will see that the answer depends to an extent on the relative sizes of m and n, and we start with a particularly clean case: when $m = n$. Here, it is possible for each processor to end up with exactly one job, though this is not very likely. Rather, we expect that some processors will receive no jobs, and others will receive more than one. As a way of assessing the quality of this randomized load balancing heuristic, we study how heavily loaded with jobs a processor can become.

Let X_i be the random variable equal to the number of jobs assigned to processor i, for $i = 1, 2, \ldots, n$. It is easy to determine the expected value of X_i — we let Y_{ij} is the random variable equal to 1 if job j is assigned to processor i, and 0 otherwise; then $X_i = \sum_{i=1}^{n} Y_{ij}$ and $E[Y_{ij}] = 1/n$, so $E[X_i] = \sum_{j=1}^{n} E[Y_{ij}] = 1$. But our concern is with how far X_i can deviate above its expectation; what is the probability that $X_i > c$? To give an upper bound on this, we can directly apply (12.18): X_i is a sum of independent 0-1-valued random variables $\{Y_{ij}\}$; we have $\mu = 1$ and $1 + \delta = c$. Thus,

(12.20)
$$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right).$$

In order for there to be a small probability of *any* X_i exceeding c, we will take the Union Bound over $i = 1, 2, \ldots, n$; and so we need to choose c large enough to drive $\Pr[X_i > c]$ down well below $1/n$ for each i. This requires looking at the denominator c^c in (12.20). To make this denominator large enough, we need to understand how this quantity grows with c, and we explore this by first asking the question, "What is the x such that $x^x = n$?"

Suppose we write $\gamma(n)$ to denote this number x. There is no closed-form expression for $\gamma(n)$, but we can determine its asymptotic value as follows. If $x^x = n$, then taking logarithms gives $x \log x = \log n$; and taking logarithms again gives $\log x + \log\log x = \log\log n$. Thus we have

$$2\log x > \log x + \log\log x = \log n > \log x,$$

and using this to divide through the equation $x \log x = \log n$, we get

$$\frac{1}{2}x \leq \frac{\log n}{\log\log n} \leq x = \gamma(n).$$

Thus $\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$.

Now, if we set $c = e\gamma(n)$, then by (12.20) we have

$$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right) < \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}.$$

Thus, applying the Union Bound over this upper bound for $X_1, X_2, \ldots, X_n$, we see that with probability at least $1 - 1/n$, no processor receives more than $e\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$ jobs. With

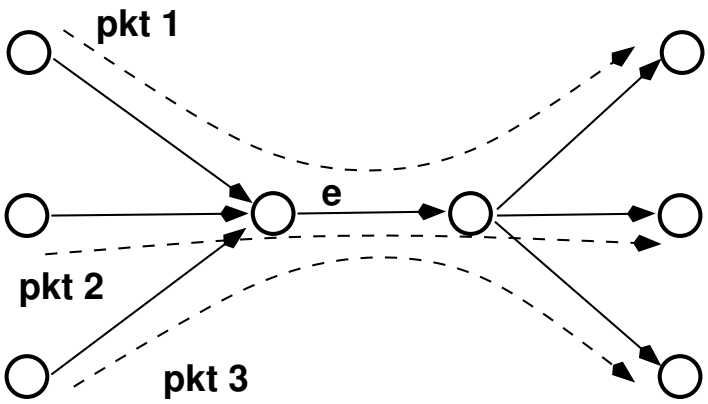


Figure 12.2: Three packets whose paths involve a shared edge e.

a more involved analysis, one can also show that this bound is asymptotically tight: with high probability, some processor actually receives $\Omega\left(\frac{\log n}{\log \log n}\right)$ jobs.

So although the load on some processors will likely exceed the expectation, this deviation is only logarithmic in the number of processors. We now use Chernoff bounds to argue that as more jobs are introduced into the system, the loads "smooth out" rapidly, so that the number of jobs on each processor quickly become the same to within constant factors.

Specifically, if we have $m = 16n \ln n$ jobs, then expected load per processor is $\mu = 16 \ln n$. Using (12.18) , we see that the probability of any processor's load exceeding $32 \ln n$ is at most

$$\Pr\left[X_i > 2\mu\right] < \left(\frac{e}{4}\right)^{16 \ln n} < \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}.$$

Also, the probability that any processor's load is below $8 \ln n$ is at most

$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}(\frac{1}{2})^2(16 \ln n)} = e^{-2 \ln n} = \frac{1}{n^2}.$$

Thus, applying the Union Bound, there is a high probability that every processor will have a load between half and twice the average, once we have $\Omega(n \log n)$ jobs.

12.9 Packet Routing

We now consider a more complex example of how randomization can alleviate contention in a distributed system — in the context of *packet routing*.

Packet routing is a mechanism to support communication among nodes of a large network, which we can model as a directed graph $G = (V, E)$. If a node s wants to send data to a node t, this data is discretized into one or more *packets*, each of which is then sent over an s-t path P in the network. At any point in time, there may be many packets in the network, associated with different sources and destinations and following different paths. However,

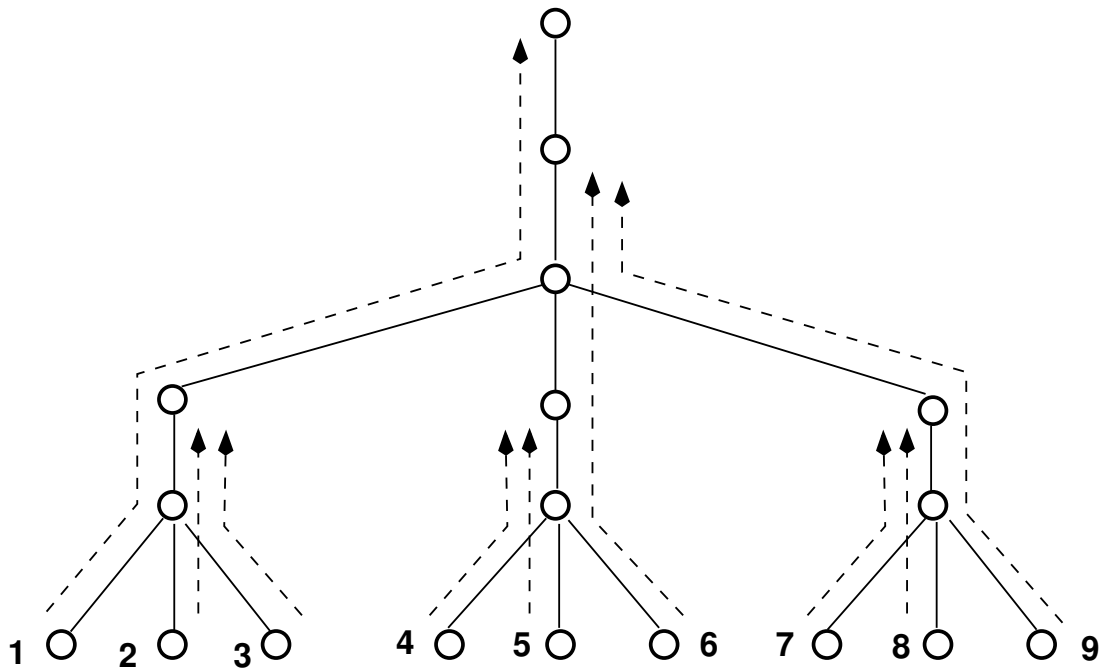


Figure 12.3: A case in which the scheduling of packets matters.

the key constraint is that a single edge e can only transmit a single packet per time step. Thus, when a packet p arrives at an edge e on its path, it may find there are several other packets already waiting to traverse e; in this case, p joins a *queue* associated with e to wait until e is ready to transmit it. In Figure 12.2, for example, three packets with different sources and destinations all want to traverse edge e; so if they all arrive at e at the same time, some of them will be forced to wait in a queue for this edge.

Suppose we are given a network G with a set of packets that need to be be sent across specified paths. We'd like to understand how many steps are needed in order for all packets to reach their destinations. Although the paths for the packets are all specified, we face the algorithmic question of timing the movements of the packets across the edges. In particular, we must decide when to release each packet from its source, as well as a *queue management policy* for each edge e — how to select the next packet for transmission from e's queue in each time step.

It's important to realize that these *packet scheduling* decisions can have a significant effect on the amount of time it takes for all the packets to reach their destinations. For example, let's consider the tree network in Figure 12.3, where there are 9 packets that want to traverse the respective dotted paths up the tree. Suppose all packets are released from their sources immediately, and each edge e manages its queue by always transmitting the packet that is closest to its destination. In this case, packet 1 will have to wait for packets 2 and 3 at the second level of the tree; and then later it will have to wait for packets 6 and 9 at the

fourth level of the tree. Thus, it will take 9 steps for this packet to reach its destination. On the other hand, suppose that each edge e manages its queue by always transmitting the packet that is farthest from its destination. Then packet 1 will never have to wait, and it will reach its destination in 5 steps; moreover, one can check that every packet will reach its destination within 6 steps.

There is a natural generalization of the tree network in Figure 12.3, in which the the tree has height h and the nodes at every other level have k children. In this case, the queue management policy that always transmits the packet nearest its destination results in some packet requiring $\Omega(hk)$ steps to reach its destination (since the packet traveling farthest is delayed by $\Omega(k)$ steps at each of $\Omega(h)$ levels), while the policy that always transmits the packet farthest from its destination results in all packets reaching their destinations within $O(h + k)$ steps. This can become quite a large difference as h and k grow large.

Schedules and their durations. Let's now move from these examples to the question of scheduling packets and managing queues in an arbitrary network G. Given packets labeled $1, 2, \ldots, N$ and associated paths $P_1, P_2, \ldots, P_N$, a *packet schedule* specifies, for each edge e and each time step t, which packet will cross edge e in step t. Of course, the schedule must satisfy some basic consistency properties: at most one packet can cross any edge e in any one step; and if packet i is scheduled to cross e at step t, then e should be on the path P_i, and the earlier portions of the schedule should cause i to have already reached e. We will say that the *duration* of the schedule is the number of steps that elapse until every packet reaches its destination; the goal is to find a schedule of minimum duration.

What are the obstacles to having a schedule of low duration? One obstacle would be a very long path that some packet must traverse; clearly, the duration will be at least the length of this path. Another obstacle would be a single edge e that many packets must cross; since each of these packets must cross e in a distinct step, this also gives a lower bound on the duration. So if we define the *dilation* d of the set of paths $\{P_1, P_2, \ldots, P_N\}$ to be the maximum length of any P_i, and the *congestion* c of the set of paths to be the maximum number that have any single edge in common, then the duration is at least $\max(c, d) = \Omega(c + d)$.

In 1988, Leighton, Maggs, and Rao proved the following striking result: congestion and dilation are the only obstacles to finding fast schedules, in the sense that there is always a schedule of duration $O(c+d)$. While the statement of this result is very simple, it turns out to be extremely difficult to prove; and it yields only a very complicated method to actually *construct* such a schedule. So instead of trying to prove this result, we'll analyze a simple algorithm (also proposed by Leighton, Maggs, and Rao) that can be easily implemented in a distributed setting, and yields a duration that is only worse by a logarithmic factor: $O(c + d \log(mN))$, where m is the number of edges and N is the number of packets.

A Randomized Schedule. If each edge simply transmits an arbitrary waiting packet in each step, it is easy to see that the resulting schedule has duration $O(cd)$ — at worst, a packet can be blocked by $c-1$ other packets on each of the d edges in its path. To reduce this bound, we need to set things up so that each packet only waits for a much smaller number of steps over the whole trip to its destination.

The reason a bound as large as $O(cd)$ can arise is that the packets are very badly timed with respect to each other — blocks of c of them all meet at an edge at the same time, and once this congestion has cleared, the same thing happens at the next edge. This sounds pathological, but one should remember that a very natural queue management policy caused it to happen in Figure 12.3. However, it is the case that such bad behavior relies on very unfortunate synchronization in the motion of the packets; so it is believable that if we introduce some randomization in the timing of the packets, then this kind of behavior is unlikely to happen. The simplest idea would be just to randomly shift the times at which the packets are released from their sources — then a block of packets all aimed at the same edge are unlikely to hit it at the same time; the contention for edges has been "smoothed out." We now show that this kind of randomization, properly implemented, in fact works quite well.

Consider first the following algorithm, which will not quite work. It involves a parameter r whose value will be determined later.

```
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r.
  i waits at its source for s time steps.
  i then moves full speed ahead, one edge per time step,
     until it reaches its destination.
```

If the set of random delays were really chosen so that no two packets ever "collided" — reaching the same edge at the same time — then this schedule would work just as advertised; its duration would be at most r (the maximum initial delay) plus d (the maximum number of edges on any path). However, unless r is chosen to be very large, it is unlikely that no collisions will occur, and so the algorithm will probably fail: two packets will show up at the same edge e in the same time step t, and both will be required to cross e in the next step.

To get around this problem, we consider the following generalization of this strategy: rather than implementing the "full speed ahead" plan at the level of individual time steps, we implement it at the level of continguous *blocks* of time steps.

```
For a parameter b, group intervals of b consecutive time steps
      into single blocks of time.
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r.
  i waits at its source for s blocks.
```

```
    i then moves foward one edge per block,
       until it reaches its destination.
```

This schedule will work provided that we avoid a more extreme type of collision: it should not be the case that more than b packets are supposed to show up at the same edge e at the start of the same block. If this happens, then at least one of them will not be able to cross e in the next block. On the other hand, if the initial delays smooth things out enough that no more than b packets arrive at any edge in the same block, then the schedule will work just as intended. In this case, the duration will be at most $b(r + d)$ — the maximum number of blocks, $r + d$, times the length of each block, b.

(12.21) *Let $\mathcal{E}$ denote the event that more than b packets are required to be at the same edge e at the start of the same block. If $\mathcal{E}$ does not occur, then the duration of the schedule is at most $b(r + d)$.*

Our goal is now to choose values of r and b so that both the probability $\Pr[\mathcal{E}]$ and the duration $b(r + d)$ are small quantities.

To give a bound on $\Pr[\mathcal{E}]$, it's useful to decompose it into a union of simpler bad events, so that we can apply the Union Bound. A natural set of bad events arises from considering each edge and each time block separately; if e is an edge, and t is a block between 1 and $r + d$, we $\mathcal{F}_{et}$ denote the event that more than b packets are required to be at e at the start of block t. Clearly $\mathcal{E} = \cup_{e,t}\mathcal{F}_{et}$. Moreover, if N_{et} is a random variable equal to the number of packets required to be at e at the start of block t, then $\mathcal{F}_{et}$ is equivalent to the event $[N_{et} > b]$.

The next step in the analysis is to decompose the random variable N_{et} into a sum of independent 0-1-valued random variables so that we can apply a Chernoff bound. This is naturally done by defining X_{eti} to be equal to 1 if packet i is required to be at edge e at the start of block t, and equal to 0 otherwise. Then $N_{et} = \sum_i X_{eti}$; and for different values of i, the random variables X_{eti} are independent, since the packets are choosing independent delays. (Note that X_{eti} and $X_{e't'i}$, where the value of i is the same, would certainly not be independent; but our analysis does not require us to add random variables of this form together.) Notice that of the r possible delays that packet i can choose, at most one will require it to be at e at block t; thus $E[X_{eti}] \leq 1/r$. Moreover, at most c packets have paths that include e; and if i is not one of these packets, then clearly $E[X_{eti}] = 0$. Thus we have

$$E[N_{et}] = \sum_i E[X_{eti}] \leq \frac{c}{r}.$$

We now have the set-up for applying the Chernoff bound (12.18), since N_{et} is a sum of the independent 0-1-valued random variables X_{eti}. Indeed, the quantities are sort of like what they were when we analyzed the problem of throwing m jobs at random onto n processors:

in that case, each constituent random variable had expectation $1/n$, the total expectation was m/n, and we needed m to be $\Omega(n \log n)$ in order for each processor load to be close to its expectation with high probability. The appropriate analogy in the case at hand is for r to play the role of n, and c to play the role of m: this makes sense both symbolically, in terms of the parameters; and also it accords with the picture that the packets are like the jobs, and the different time blocks of a single edge are like the different processors that can receive the jobs. This suggests that if we want the number of packets destined for a particular edge in a particular block to be close to its expectation, we should have $c = \Omega(r \log r)$.

This will work, except that we have to increase the logarithmic term a little to make sure that the Union Bound over all e and all t works out in the end. So let's set

$$r = \frac{c}{q \log(mN)},$$

where q is a constant that will be determined later.

Let's fix a choice of e and t, and write $\mu = E[N_{et}]$. When we go to apply the Chernoff bound, we'll see that there is one extra wrinkle; we don't know exactly when μ is, only that it is upper-bounded by $c/r = q \log(mN)$. Nevertheless, let's use (12.18) to study the probability that N_{et} does not exceed 3 times this upper bound. We choose $\delta > 0$ so that $(1+\delta)\mu = 3c/r = 3q \log(mN)$, and use this as the upper bound in the expression $\Pr[N_{et} > 3c/r] = \Pr[N_{et} > (1+\delta)\mu]$. Note that because $\mu \leq c/r$, we know that $1 + \delta \geq 3$, a fact that we'll use in the calculation below. Applying (12.18), we have

$$\begin{aligned}
\Pr[N_{et} > 3c/r] &< \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu < \left[\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right]^\mu = \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \\
&\leq \left(\frac{e}{3}\right)^{(1+\delta)\mu} = \left(\frac{e}{3}\right)^{3c/r} = \left(\frac{e}{3}\right)^{3q\log(mN)} = \frac{1}{(mN)^z},
\end{aligned}$$

where z is a constant that can be made as large as we want by choosing the constant q appropriately. The inequality at the beginning of the second line follows from the fact that $1+\delta \leq 3$, and the subsequent equalities follow from our choice of δ so that $(1+\delta)\mu = 3c/r = 3q \log(mN)$.

We can see from this calculation that it's safe to set $b = 3c/r$; for in this case, the event $\mathcal{F}_{et}$ that $N_{et} > b$ will have very small probability for each choice of e and t. There are m different choices for e, and $d + r$ different choice for t, where we observe that $d + r \leq d + c - 1 \leq N$. Thus we have

$$\Pr[\mathcal{E}] = \Pr\left[\bigcup_{e,t} \mathcal{F}_{et}\right] \leq \sum_{e,t} \Pr[\mathcal{F}_{et}] \leq mN \cdot \frac{1}{(mN)^z} = \frac{1}{(mN)^{z-1}},$$

which can be made as small as we want by choosing z large enough.

And provided this bad event does not happen, then (12.21) tells us that the duration of the schedule is bounded by

$$b(r+d) = \frac{3c}{r}(r+d) = 3c + d \cdot \frac{3c}{r} = 3c + d(3q\log(mN)) = O(c + d\log(mN)).$$

12.10 Constructing an Expander Graph

We have seen the use of randomization in a number of algorithm design applications. We now return to a somewhat different use of randomization: establishing the existence of a combinatorial object with a certain property, simply by showing that a random object has the property with positive probability. We've seen one application of this already, when we showed via a random construction that for every instance of *3-SAT*, there is a truth assignment that satisfies at least a $\frac{7}{8}$ fraction of all clauses. We now consider a much more elaborate example; we use randomization to demonstrate the existence of a highly fault-tolerant class of graphs known as *expanders*.

We call a graph an *expander* if it has a relatively sparse edge set, and has the property that every subset of nodes is highly connected to the rest of the graph. Specifically, if $G = (V, E)$ is a graph, and $S \subseteq V$, we use $N(S)$ to denote the "neighbors" of S — the set of nodes with an edge to some node in S. (Note that $N(S)$ may include some nodes in S but not others.) We say that G is an *expander with parameters* (d, c, α) if every node of G has degree at most d, and for every subset S of at most cn nodes, we have $|N(S)| \geq \alpha|S|$. In this case, we say that each set S "expands" by a factor of α. The definition will be interesting to us only when $\alpha > 1$.

Expanders are highly resilient to failures in the following natural sense: the deletion of a small number of nodes cannot separate a large set of nodes from the rest of the graph. This gives them a lot of natural "robustness" properties; here is one example.

(12.22) *Let G be an expander with parameters (d, c, α) such that $\alpha > 1$ and $\alpha c > \frac{1}{2}$. Then every pair of nodes u and v in G are connected by a path of length $O(\log n)$.*

Proof. Let u and v be two nodes of G. We perform a breadth-first search of G starting at u, stopping when the set of nodes we encounter first reaches size cn. More concretely, we use U_i to denote the set of all nodes within i steps of u, and let r be the smallest value of i such that $|U_i| > cn$. We define B_u to be the set U_{r-1}, together with as many nodes at distance exactly r from u as are necessary to produce a set of size exactly cn. We then define B'_u to be $B_u \cup N(B_u)$.

Now, for any $i \leq r$, we see that U_i is simply $U_{i-1} \cup N(U_{i-1})$. Since $i - 1 < r$, we have $|U_{i-1}| \leq cn$, and so the set U_{i-1} is covered by the expansion guarantees of G: we have $|U_i| \geq |N(U_{i-1})| \geq \alpha|U_{i-1}|$. The sets $U_1, U_2, \ldots$ grow exponentially in size up through U_{r-1};

but since G has only n nodes, this exponential growth can continue only for $O(\log n)$ steps, and so we must have $r = O(\log n)$. Hence every node in B_u, and also in B'_u, has a path to u of length $O(\log n)$.

We do this same construction for v, producing sets B_v and B'_v. Now, since $|B_u| = cn$, we have $|N(B_u)| \geq \alpha cn > n/2$, and so we have $|B'_u| \geq |N(B_u)| > n/2$. (Note that we've used the assumption that $\alpha c > \frac{1}{2}$.) Similarly, we have $|B'_v| > n/2$. Thus, the sets B'_u and B'_v must have at least one node w in their intersection. The node w has a path of length $O(\log n)$ to u and also a path of length $O(\log n)$ to v; concatenating these two paths together, we get a u-v path of length $O(\log n)$. ■

The most basic non-trivial fact about expander graphs, though, is that they exist at all: there exist fixed constant values of d, c, and $\alpha > 1$, so that for arbitrarily large values of n, there are n-node expander graphs with parameters (d, c, α). The key point is that none of the parameters d, c, or α depend on the size n of the graph. In other words, there exist arbitrarily large graphs in which each node has constant degree, and each set (of up to linear size) expands by a constant factor. To avoid explicitly discussing the underlying parameters all the time, one often speaks informally of a class of graphs having "good expansion properties" if d and c are absolute constants, and α is a constant greater than 1.

Constructing large graphs with good expansion properties — and proving these expansion properties — is much more difficult than one might imagine. Trying this oneself is the best way to drive the point home. For example, an $n \times n$ grid graph does not maintain an expansion parameter of $\alpha > 1$ as n increases: for any $c > 0$, the set S consisting of the leftmost cn columns has $|N(S)| \leq |S| + n = |S|(1 + 1/cn)$. Or consider an n-node complete binary tree: it may look like it has good expansion properties if one views it from the root downward; but if we think of the subtree S below any given node, it has $|N(S)| \leq |S| + 1$. One can show that much more sophisticated examples than these also fail to serve as good expander graphs. Ultimately, finding an explicit construction of arbitrarily large graphs that could be proved to have good expansion properties required intricate analysis and sophisticated use of some deep results from mathematics; it is only now, three decades after people began studying expanders, that somewhat simpler analyses are emerging.

We will now show that a simple random construction produces good expander graphs with constant probability. In light of our discussion, this is quite surprising: it is extremely difficult to verify that an explicitly constructed graph is a good expander, but it is easy to show that a random graph is likely to be one. The analysis of our random construction will be quite crude, and will not aim for the best possible values of all parameters; rather, its goal is to show how a completely direct use of the Union Bound is enough to verify good expansion.

The random construction. We start with a set V of n nodes, labeled $1, 2, 3, \ldots, n$, and no edges joining any of them. A *random pairing* on V is a set of edges M constructed as follows. We choose a random ordering of V, say $v_1, v_2, \ldots, v_n$, and define M to be the set of n edges $\{(i, v_i) : i = 1, 2, \ldots, n\}$. Note that each node is incident to two edges in M, unless it happens that $v_i = i$; in this case we view the edge (i, v_i) as a loop from i to itself.

Here is the full construction of G. We set $d = 600$; we compute d random pairings $M_1, M_2, \ldots, M_d$ on the set V, using orders chosen independently for each; and we define the edge set $E = M_1 \cup M_2 \cup \cdots \cup M_d$. Notice that the graph $G = (V, E)$, which may have loops and parallel edges, has maximum node degree $2d = 1200$. Notice that while G has constant node degree — independent of the number of nodes — it is quite a large constant; this is in keeping with our plan to sacrifice better parameters for the sake of the simplest analysis possible. In fact, random graphs in which each node has degree 3 can be shown to have fairly good expansion properties as well, but the proof of this becomes somewhat more involved.

(12.23) *With probability at least $2/3$, the graph $G = (V, E)$ is an expander with parameters $(2d, .35, 1.5)$. In other words, with probability at least $2/3$, every set S of at most $.35n$ nodes in G has the property that $|N(S)| > 1.5|S|$.*

Notice that $.35 \cdot 1.5 > .5$, and so the conclusion of (12.22) holds for G, provided that it satisfies these expansion properties — there will be a short path between each pair of nodes.

The proof will consist of an extended but completely direct use of the Union Bound, summing over an exponential number of possible bad events that could prevent G from being a good expander. In order to make the calculations work out, we first need some simple bounds on the growth of the factorial function and the binomial coefficients.

(12.24) *For every natural number n, we have $n! > \left(\frac{n}{e}\right)^n$.*

Proof. We prove this by induction, the cases $n = 0$ and $n = 1$ being clear. For a larger value of n, we can apply the induction hypothesis together with a close variant of fact (12.1) , asserting that $\left(1 + \frac{1}{n}\right)^n < e$ for all natural numbers n. Thus we have

$$(n+1)! = (n+1)n! > (n+1)\left(\frac{n}{e}\right)^n > (n+1)\left(\frac{n}{e}\right)^n \frac{\left(1+\frac{1}{n}\right)^n}{e} = \frac{(n+1)^{n+1}}{e^{n+1}}.$$

■

Using this bound, we now prove

(12.25) *For every pair of natural numbers n and k, where $n \geq k$, we have $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} < \left(\frac{en}{k}\right)^k$.*

Proof. By (12.24), we have

$$\binom{n}{k} < \frac{n^k}{k!} < \frac{n^k}{(k/e)^k} = \left(\frac{en}{k}\right)^k.$$

Since $\frac{n}{k} \geq \frac{n-1}{k-1}$ for any natural numbers $n \geq k$, we have

$$\binom{n}{k} \geq \frac{n^k}{k^k} = \left(\frac{n}{k}\right)^k.$$

■

Notice that $\binom{n}{k}$ is not defined when k is not a natural number. However, if k is not a natural number, we can still use (12.25) to bound $\binom{n}{\lfloor k \rfloor}$ as follows:

$$\binom{n}{\lfloor k \rfloor} < \left(\frac{en}{\lfloor k \rfloor}\right)^{\lfloor k \rfloor} < \left(\frac{en}{k}\right)^k,$$

where the first inequality is just (12.25), and the second follows from the fact that the function $(en/k)^k$ increases monotonically until $k = n$.

We are now ready for

Proof of (12.23). If G fails to have the desired property, it means that there is some set S of at most $.35n$ nodes so that $N(S) < 1.5|S|$. So for every set S of at most $.35n$ nodes, and every set T of size exactly $\lfloor 1.5|S| \rfloor$, we define the event $\mathcal{E}_{ST}$ that $N(S) \subseteq T$. We observe that if the union of all these events $\mathcal{E}_{ST}$ does not occur, then every set S expands by a sufficient amount, and G has the desired expansion properties. Thus, it is sufficient to give an upper bound on

$$\Pr\left[\bigcup_{\substack{|S| \leq .35n \\ |T| = \lfloor 1.5|S| \rfloor}} \mathcal{E}_{ST}\right].$$

To think about this, we first define a related set of events as follows. For every pair of sets S and T, we define the event $\mathcal{E}'_{ST}$ that in a single random pairing M, all nodes with an edge to S belong to T. If $k = |S| = |T|$, then $\Pr[\mathcal{E}'_{ST}] \leq 1/\binom{n}{k}$, since it is necessary that in the random ordering $v_1, v_2, \ldots, v_n$ that produces M, the set $\{v_i : i \in S\}$ is equal to T. If $k = |S| < |T|$, then

$$\Pr[\mathcal{E}'_{ST}] = \bigcup_{\substack{U \subseteq T \\ |U| = k}} \mathcal{E}'_{SU} = \sum_{\substack{U \subseteq T \\ |U| = k}} \Pr[\mathcal{E}'_{SU}] \leq \binom{|T|}{k} \Big/ \binom{n}{k} \leq 2^{|T|} \Big/ \binom{n}{k} \leq \frac{2^{|T|} k^k}{n^k}.$$

Here the last inequality follows from (12.25), and the second-to-last from the fact that T has at most $2^{|T|}$ subsets.

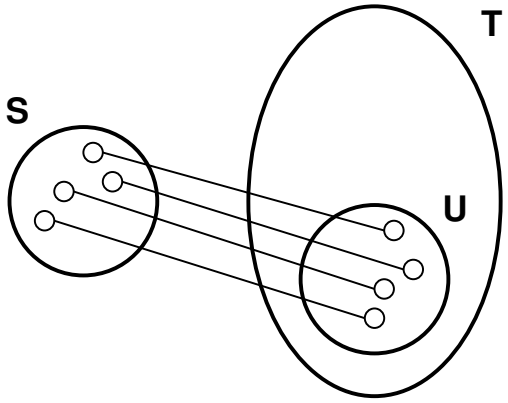


Figure 12.4: The event $\mathcal{E}_{ST}$ in the analysis of the expander construction.

Now, the graph G is built from d random pairings; so if $k = |S| \leq .35n$ and $|T| = 1.5k$, then

$$\Pr[\mathcal{E}_{ST}] = (\Pr[\mathcal{E}'_{ST}])^d \leq \frac{2^{\lfloor 1.5dk \rfloor} k^{dk}}{n^{dk}} \leq \frac{2^{1.5dk} k^{dk}}{n^{dk}} = \left(\frac{2^{1.5} k}{n}\right)^{dk}.$$

It will be crucial later that the quantity inside the parentheses in the final expression is strictly less than 1; this follows from the fact that $k \leq .35n$:

$$\frac{2^{1.5} k}{n} \leq .35 \cdot 2^{1.5} < .99.$$

As promised, we complete the proof with an enormous application of the Union Bound:

$$\Pr\left[\bigcup_{\substack{|S| \leq .35n \\ |T| = \lfloor 1.5|S| \rfloor}} \mathcal{E}_{ST}\right] \leq \sum_{\substack{|S| \leq .35n \\ |T| = \lfloor 1.5|S| \rfloor}} \Pr[\mathcal{E}_{ST}].$$

This sum involves exponentially many terms; to unravel it, we consider separately the terms for each possible size of the set S. For sets S of size k, there are $\binom{n}{k}\binom{n}{\lfloor 1.5k \rfloor}$ terms, each with probability at most $\left(\frac{2^{1.5}k}{n}\right)^{dk}$. We then upper-bound the binomial coefficients using (12.25) and begin canceling as many terms as we can:

$$\begin{aligned}
\sum_{\substack{|S| \leq .35n \\ |T| = \lfloor 1.5|S| \rfloor}} \Pr[\mathcal{E}_{ST}] &\leq \sum_{k=1}^{.35n} \binom{n}{k}\binom{n}{\lfloor 1.5k \rfloor} \left(\frac{2^{1.5}k}{n}\right)^{dk} \\
&< \sum_{k=1}^{.35n} \left(\frac{en}{k}\right)^{k} \left(\frac{en}{1.5k}\right)^{1.5k} \left(\frac{2^{1.5}k}{n}\right)^{dk} \\
&= \sum_{k=1}^{.35n} \left[\frac{e^{2.5}}{1.5^{1.5}} \cdot 2^{(1.5)(2.5)} \left(\frac{2^{1.5}k}{n}\right)^{(d-2.5)}\right]^{k}
\end{aligned}$$

$$\begin{aligned} &< \sum_{k=1}^{\infty} \left[90\,(.99)^{597.5}\right]^k \\ &< \sum_{k=1}^{\infty} \left(\frac{1}{4}\right)^k = \frac{1}{3}. \end{aligned}$$

Thus, with probability at least 2/3, the graph G has the desired expansion properties. ■

12.11 Appendix: Some Probability Definitions

For many — though certainly not all — applications of randomized algorithms, it is enough to work with probabilities defined over finite sets only; and this turns out to be much easier to think about than probabilities over arbitrary sets. So we begin by considering just this special case, We'll then end the section by re-visiting all these notions in greater generality.

Finite probability spaces

We have an intuitive understanding of sentences like, "If a fair coin is flipped, the probability of 'heads' is 1/2." Or, "If a fair die is rolled, the probability of a '6' is 1/6." What we want to do first is to describe a mathematical framework in which we can discuss such statements precisely. The framework will work well for carefully circumscribed systems such as coin flips and rolls of dice; at the same time, we will avoid the lengthy and substantial philosophical issues raised in trying to model statements like, "The probability of rain tomorrow is 20%." Fortunately, most algorithmic settings are as carefully circumscribed as those of coins and dice, if perhaps somewhat larger and more complex.

To be able to compute probabilities, we introduce the notion of a *finite probability space*. (Recall that we're dealing with just the case of finite sets for now.) A finite probability space is defined by an underlying *sample space* Ω, which consists of the possible "outcomes" of the process under consideration. Each point i in the sample space also has a non-negative *probability mass* $p(i) \geq 0$; these probability masses need only satisfy the constraint that their total sum is 1; i.e. $\sum_{i \in \Omega} p(i) = 1$. We define an *event* $\mathcal{E}$ to be any subset of Ω — an event is defined simply by the set of outcomes that constitute it — and we define the *probability* of the event to be the sum of the probability masses of all the points in $\mathcal{E}$. That is,

$$\Pr[\mathcal{E}] = \sum_{i \in \mathcal{E}} p(i).$$

In many situations that we'll consider, all points in the sample space have the same probability mass, and then the probability of an event $\mathcal{E}$ is simply its size relative to the size of Ω; that is, in this special case, $\Pr[\mathcal{E}] = |\mathcal{E}|/|\Omega|$. We use $\overline{\mathcal{E}}$ to denote the complementary event $\Omega - \mathcal{E}$; note that $\Pr\left[\overline{\mathcal{E}}\right] = 1 - \Pr[\mathcal{E}]$.

Thus, the points in the sample space and their respective probability masses form a complete description of the system under consideration; it is the events — the subsets of the sample space — whose probabilities we are interested in computing. So to represent a single flip of a "fair" coin, we can define the sample space to be $\Omega = \{\texttt{heads}, \texttt{tails}\}$ and set $p(\texttt{heads}) = p(\texttt{tails}) = 1/2$. If we want to consider a biased coin in which "heads" is twice as likely as "tails," we can define the probability masses to be $p(\texttt{heads}) = 2/3$ and $p(\texttt{tails}) = 1/3$. A key thing to notice even in this simple example is that defining the probability masses is a part of defining the underlying problem; in setting up the problem, we are specifying whether the coin is fair or biased, not "deriving" this from some more basic data.

Here's a slightly more complex example. Suppose we have n processes in a distributed system, denoted $p_1, p_2, \ldots, p_n$, and each of them chooses an identifier for itself uniformly at random from the space of all k-bit strings. Moreover, each process's choice happens concurrently with those of all the other processes, and so the outcomes of these choices are unaffected by one another. If we view each identifier as being chosen from the set $\{0, 1, 2, \ldots, 2^k - 1\}$ (by considering the numerical value of the identifier as a number in binary notation), then the sample space Ω could be represented by the set of all n-tuples of integers, with each integer between 0 and $2^k - 1$. The sample space would thus have $(2^k)^n = 2^{kn}$ points, each with probability mass 2^{-kn}.

Now, suppose we are interested in the probability that processes p_1 and p_2 each choose the same name. This is an event $\mathcal{E}$, represented by the subset consisting of all n-tuples from Ω whose first two coordinates are the same. There are $2^{k(n-1)}$ such n-tuples: we can choose any value for coordinates 3 through n, then any value for coordinate 2, and then we have no freedom of choice in coordinate 1. Thus we have

$$\Pr[\mathcal{E}] = \sum_{i \in \mathcal{E}} p(i) = 2^{k(n-1)} \cdot 2^{-kn} = 2^{-k}.$$

This, of course, corresponds to the intuitive way one might work out the probability, which is to say that we can choose any identifier we want for process p_2, after which there is only 1 choice out of 2^k for process p_1 that will cause the names to agree. It's worth checking that this intuition is really just a compact description of the calculation above.

Conditional Probability and Independence

If we view the probability of an event $\mathcal{E}$, roughly, as the likelihood that $\mathcal{E}$ is going to occur, then we may also want to ask about its probability given additional information. Thus, given another event $\mathcal{F}$ of positive probability, we define the *conditional probability of $\mathcal{E}$ given $\mathcal{F}$* as

$$\Pr[\mathcal{E} \mid \mathcal{F}] = \frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]}.$$

This is the "right" definition intuitively since it's performing the following calculation: of the portion of the sample space that consists of $\mathcal{F}$ (the event we "know" to have occurred), what fraction is occupied by $\mathcal{E}$?

Intuitively, we say that two events are *independent* if information about the outcome of one does not affect our estimate of the likelihood of the other. One way to make this concrete would be to declare events $\mathcal{E}$ and $\mathcal{F}$ to be independent if $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$, and $\Pr[\mathcal{F} \mid \mathcal{E}] = \Pr[\mathcal{F}]$. (We'll assume here that both have positive probability; otherwise, the notion of independence is not very interesting in any case.) Actually, if one of these two equalities holds, then the other must hold, for the following reason: If $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$, then

$$\frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]} = \Pr[\mathcal{E}],$$

and hence $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$, from which the other equality holds as well.

It turns out to be a little cleaner to adopt this equivalent formulation as our working definition of independence: formally, we'll say that that events $\mathcal{E}$ and $\mathcal{F}$ are *independent* if $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$.

This product formulation leads to the following natural generalization. We say that a collection of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ is *independent* if for every set of indices $I \subseteq \{1, 2, \ldots, n\}$, we have

$$\Pr\left[\bigcap_{i \in I} \mathcal{E}_i\right] = \prod_{i \in I} \Pr[\mathcal{E}_i].$$

It's important to notice the following: to check if a large set of events is independent, it's not enough to check whether every pair of them is independent. For example, suppose we flip three independent fair coins: if $\mathcal{E}_i$ denotes the event that the i^{th} coin comes up `heads`, then the events $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ are independent and each has probability $1/2$. Now, let A denote the event that coins 1 and 2 have the same value; let B denote the event that coins 2 and 3 have the same value; and let C denote the event that coins 1 and 3 have different values. It's easy to check that each of these events has probability $1/2$, and the intersection of any two has probability $1/4$. Thus every pair drawn from A, B, C is independent. But the set of all three events A, B, C is not independent, since $\Pr[A \cap B \cap C] = 0$.

The Union Bound

Suppose we are given a set of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, and we are interested in the probability that *any* of them happens; that is, we are interested in the probability $\Pr[\cup_{i=1}^{n} \mathcal{E}_i]$. If the events are all pairwise disjoint from one another, then the probability mass of their union is comprised simply of the separate contributions from each event. In other words, we have the following fact.

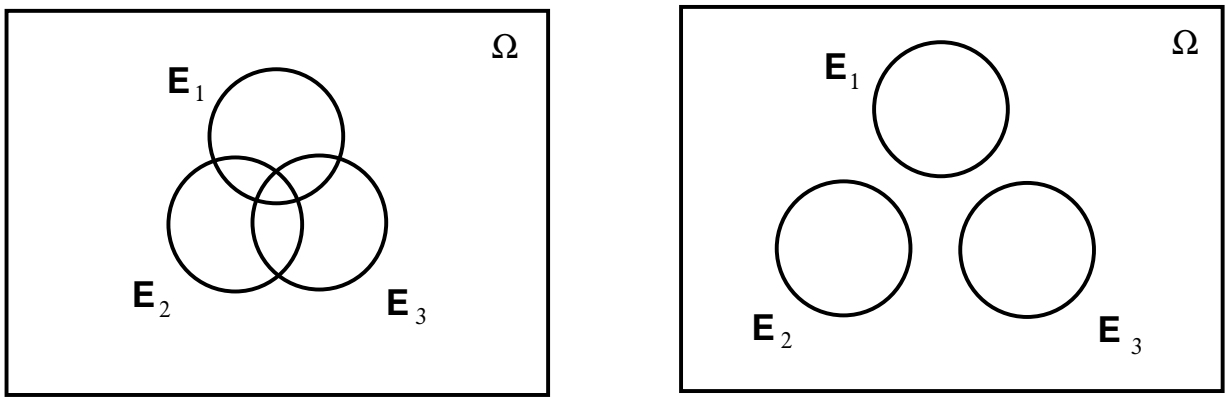


Figure 12.5: The Union Bound: the probability of a union is maximized when the events have no overlap.

(12.26) *Suppose we have events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ such that $\mathcal{E}_i \cap \mathcal{E}_j = \phi$ for each pair. Then*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] = \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

In general, a set of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ may overlap in complex ways. In this case, the equality in (12.26) no longer holds; due to the overlaps among events, the probability mass of a point that is counted once on the left-hand side will be counted one *or more* times on the right-hand side. (See Figure 12.5.) This means that for a general set of events, the equality in (12.26) is relaxed to an inequality; and this is the content of the Union Bound.

(12.27) (The Union Bound.) *Given events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, we have*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Given its innocuous appearance, the *Union Bound* is a surprisingly powerful tool in the analysis of randomized algorithms. It draws its power mainly from the following ubiquitous style of analyzing randomized algorithms. Given a randomized algorithm designed to produce a correct result with high probability, we first tabulate a set of "bad events" $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ with the following property: if none of these bad events occurs, then the algorithm will indeed produce the correct answer. In other words, if $\mathcal{F}$ denotes the event that the algorithm fails, then we have

$$\Pr\left[\mathcal{F}\right] \leq \Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right].$$

But it's hard to compute the probability of this union, so we apply the Union Bound to conclude that

$$\Pr[\mathcal{F}] \leq \Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

Now, if in fact we have an algorithm that succeeds with very high probability, and if we've chosen our bad events carefully, then each of the probabilities $\Pr[\mathcal{E}_i]$ will be so small that even their sum — and hence our overestimate of the failure probability — will be small. This is the key: decomposing a highly complicated event, the failure of the algorithm, into a horde of simple events whose probabilities can be easily computed.

Here is a simple example to make the strategy discussed above more concrete. Recall the situation we discussed earlier, in which each of a set of processes chooses a random identifier. Suppose that we have 1000 processes, each choosing a 32-bit identifier, and we are concerned that two of them will end up choosing the same identifier. Can we argue that it is unlikely this will happen? To begin with, let's denote this event by $\mathcal{F}$. While it would not be overwhelmingly difficult to compute $\Pr[\mathcal{F}]$ exactly, it is much simpler to bound it as follows. The event $\mathcal{F}$ is really a union of $\binom{1000}{2}$ "atomic" events; these are the events $\mathcal{E}_{ij}$ that processes p_i and p_j choose the same identifier. It is easy to verify that indeed, $\mathcal{F} = \cup_{i,j}\mathcal{E}_{ij}$. Now for any $i \neq j$, we have $\Pr[\mathcal{E}_{ij}] = 2^{-32}$, by the argument in one of our earlier examples. Applying the Union Bound, we have

$$\Pr[\mathcal{F}] \leq \sum_{i,j} \Pr[\mathcal{E}_{ij}] = \binom{1000}{2} \cdot 2^{-32}.$$

Now, $\binom{1000}{2}$ is at most half a million, and 2^{32} is (a little bit) more than 4 billion, so this probability is at most $\frac{.5}{4000} = .000125$.

Infinite Sample Spaces

So far we've gotten by with finite probability spaces only. In several of the lectures, however, we'll see examples in which a random process can run for arbitrarily long, and so is not well described by a sample space of finite size. As a result, we pause here to develop the notion of a probability space more generally. This will be somewhat technical, and in part we are providing it simply for the sake of completeness: although some of our subsequent applications require infinite sample spaces, none of them really exercises the full power of the formalism we describe here.

Once we move to infinite sample spaces, more care is needed in defining a probability function — we cannot simply give each point in the sample space Ω a probability mass and then compute the probability of every set by summing. Indeed, for reasons that we will not go into here, it is easy to get into trouble if one even allows every subset of Ω to be an event whose probability can be computed. Thus, a general probability space has three components:

(i) The sample space Ω.

(ii) A collection $\mathcal{S}$ of subsets of Ω; these are the only events on which we are allowed to compute probabilities.

(iii) A probability function Pr, which maps events in $\mathcal{S}$ to real numbers in $[0, 1]$.

The collection $\mathcal{S}$ of allowable events can be any family of sets that satisfies the following basic closure properties: the empty set and the full sample space Ω both belong to $\mathcal{S}$; if $\mathcal{E} \in \mathcal{S}$, then $\overline{\mathcal{E}} \in \mathcal{S}$ (closure under complement); and if $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in \mathcal{S}$, then $\cup_{i=1}^{\infty} \mathcal{E}_i \in \mathcal{S}$ (closure under countable union). The probability function Pr can be any function from $\mathcal{S}$ to $[0, 1]$ that satisfies the following basic consistency properties: $\Pr[\phi] = 0$, $\Pr[\Omega] = 1$, $\Pr[\mathcal{E}] = 1 - \Pr\left[\overline{\mathcal{E}}\right]$, and the Union Bound for disjoint events (12.26) should hold even for countable unions — if $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in \mathcal{S}$ are all pairwise disjoint, then

$$\Pr\left[\bigcup_{i=1}^{\infty} \mathcal{E}_i\right] = \sum_{i=1}^{\infty} \Pr[\mathcal{E}_i].$$

Notice how, since we are not building up Pr from the more basic notion of a probability mass any more, fact (12.26) moves from being a theorem to simply a required property of Pr.

When an infinite sample space arises in our context, it's typically for the following reason: we have an algorithm that makes a sequence of random decisions, each one from a fixed finite set of possibilities; and since it may run for arbitrarily long, it may make an arbitrarily large number of decisions. Thus, we consider the sample spaces Ω constructed as follows. We start with a finite set of symbols $X = \{1, 2, \ldots, n\}$, and assign a *weight* $w(i)$ to each symbol $i \in X$. We then define Ω to be the set of all infinite sequence of symbols from X (with repetitions allowed). So a typical element of Ω will look like $\langle x_1, x_2, x_3, \ldots \rangle$ with each entry $x_i \in X$.

The simplest type of event we will be concerned with is as follows — it is the event that a point $\omega \in \Omega$ begins with a particular finite sequence of symbols. Thus, for a finite sequence $\sigma = x_1 x_2 \ldots x_s$ of length s, we define the *prefix event associated with* σ to be the set of all sample points of Ω whose first s entries form the sequence σ. We denote this event by $\mathcal{E}_\sigma$, and we define its probability to be $\Pr[\mathcal{E}_\sigma] = w(x_1)w(x_2)\cdots w(x_s)$.

The following fact is in no sense easy to prove.

(12.28) *There is a probability space* $(\Omega, \mathcal{S}, \Pr)$*, satisfying the required closure and consistency properties, such that* Ω *is the sample space defined above,* $\mathcal{E}_\sigma \in \mathcal{S}$ *for each finite sequence* σ*, and* $\Pr[\mathcal{E}_\sigma] = w(x_1)w(x_2)\cdots w(x_s)$.

Once we have this fact, the closure of $\mathcal{S}$ under complement and countable union and the consistency of Pr with respect to these operations allows us to compute probabilities of essentially any "reasonable" subset of Ω.

In our infinite sample space Ω, with events and probabilities defined as above, we encounter a phenomenon that does not naturally arise with finite sample spaces. Suppose the set X used to generate Ω is equal to $\{0, 1\}$, and $w(0) = w(1) = 1/2$. Let $\mathcal{E}$ denote the set consisting of all sequences that contain at least one entry equal to 1. (Note that $\mathcal{E}$ omits the "all-0" sequence.) We observe that $\mathcal{E}$ is an event in $\mathcal{S}$, since we can define σ_i to be the sequence of $i-1$ 0's followed by a 1, and observe that $\mathcal{E} = \cup_{i=1}^{\infty} \mathcal{E}_{\sigma_i}$. Moreover, all the events $\mathcal{E}_{\sigma_i}$ are pairwise disjoint, and so

$$\Pr[\mathcal{E}] = \sum_{i=1}^{\infty} \Pr[\mathcal{E}_{\sigma_i}] = \sum_{i=1}^{\infty} 2^{-i} = 1.$$

Here then is the phenomenon: it's possible for an event to have probability 1 even when it's not equal to the whole sample space Ω. Similarly, $\Pr\left[\overline{\mathcal{E}}\right] = 1 - \Pr[\mathcal{E}] = 0$, and so we see that it's possible for an event to have probability 0 even when it's not the empty set. There is nothing wrong with any of these results; in a sense, it's a necessary step if we want probabilities defined over infinite sets to make sense. It's simply that in such cases, we should be careful to distinguish between the notion that an event has probability 0 and the intuitive idea that the event "can't happen."

12.12 Exercises

1. In the first lecture on randomization, we saw a simple distributed protocol to solve a particular contention-resolution problem. Here is another setting in which randomization can help with contention-resolution, through the distributed construction of an independent set.

   Suppose we have a system with n processes. Certain pairs of processes are in *conflict*, meaning that they both require access to a shared resource. In a given time interval, the goal is to schedule a large subset S of the processes to run — the rest will remain idle — so that no two conflicting processes are both in the scheduled set S. We'll call such a set S *conflict-free*.

   One can picture this process in terms of a graph $G = (V, E)$ with a node representing each process and an edge joining pairs of processes that are in conflict. It is easy to check that a set of processes S is conflict-free if and only if it forms an independent set in G. This suggests that finding a maximum-size conflict-free set S, for an arbitrary conflict G will be difficult (since the general independent set problem is reducible to this problem). Nevertheless, we can still look for heuristics that find a reasonably large conflict-free set. Moreover, we'd like a simple method for achieving this without centralized control: each process should communicate with only a small number of other processes, and then decide whether or not it should belong to the set S.

We will suppose for purposes of this question that each node has exactly d neighbors in the graph G. (That is, each process is in conflict with exactly d other processes.)

(a) Consider the following simple protocol.

> Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability $\frac{1}{2}$ and set x_i to 0 with probability $\frac{1}{2}$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.

Prove that the set S resulting from the execution of this protocol is conflict-free. Also, give a formula for the expected size of S in terms of n (the number of processes) and d (the number of conflicts per process).

(b) The choice of the probability $\frac{1}{2}$ in the protocol above was fairly arbitrary, and it's not clear that it should give the best system performance. A more general specification of the protocol would replace the probability $\frac{1}{2}$ by a parameter p between 0 and 1, as follows:

> Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability p and set x_i to 0 with probability $1 - p$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.

In terms of the parameters of the graph G, give a value of p so that the expected size of the resulting set S is as large as possible. Give a formula for the expected size of S when p is set to this optimal value.

2. In class, we designed an approximation algorithm to within a factor of 7/8 for the *MAX 3-SAT* problem, where we assumed that each clause has terms associated with 3 different variables. In this problem we will consider the analogous *MAX SAT* problem: given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, but otherwise we do not make any assumptions on the length of the clauses: there may be clauses that have a lot of variables, and others may have just a single variable.

(a) First consider the randomized approximation algorithm we used for *MAX 3-SAT*, setting each variable independently to *true* or *false* with probability 1/2 each. Show that the expected number of clauses satisfied by this random assignment is at least $k/2$, i.e., half of the clauses is satisfied in expectation. Give an example to show that

there are *MAX SAT* instances such that no assignment satisfies more than half of the clauses.

(b) If we have a clause that consists just of a single term (e.g. a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: we need to set the corresponding variable in the appropriate way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction.

Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the above randomized procedure to improve the approximation factor from $1/2$ to at least a .6 approximation, that is, change the algorithm so that the expected number of clauses satisfied by the process is at least $.6k$.

(c) Give a randomized polynomial time algorithm for the general *MAX SAT* problem, so that the expected number of clauses satisfied by the algorithm is at least a .6 fraction of the maximum possible.

(Note that by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a .6 fraction *of the maximum that can be satisfied by an optimal assignment.*)

3. Let $G = (V, E)$ be an undirected graph. We say that a set of vertices $X \subseteq V$ is a *dominating set* if every vertex of G is a member of X or is joined by an edge to a member of X.

   Give a polynomial-time algorithm that takes an arbitrary d-regular graph and finds a dominating set of size $O\left(\frac{n \log n}{d}\right)$. (A graph is *$d$-regular* if each vertex is incident to exactly d edges.) Your algorithm can be either deterministic or randomized; if it is randomized, it must always return the correct answer and have an expected running time that is polynomial in n.

4. Consider a very simple on-line auction system that works as follows. There are n *bidding agents*; agent i has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidding agents appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.)

   What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

Example: Suppose $b_1 = 20$, $b_2 = 25$, and $b_3 = 10$, and the bidders arrive in the order $1, 3, 2$. Then b^* is updated for 1 and 2, but not for 3.

5. One of the (many) hard problems that arises in genome mapping can be formulated in the following abstract way. We are given a set of n *markers* $\{\mu_1, \ldots, \mu_n\}$ — these are positions on a chromosome that we are trying to map — and our goal is to output a linear ordering of these markers. The output should be consistent with a set of k *constraints*, each specified by a triple (μ_i, μ_j, μ_k), requiring that μ_j lie *between* μ_i and μ_k in the total ordering that we produce.

   Now, it is not always possible to satisfy all constraints simultaneously, so we wish to produce an ordering that satisfies as many as possible. Unfortunately, deciding whether there is an ordering that satisfies at least k' of the k constraints is an NP-complete problem (you don't have to prove this.)

   Give a constant $\alpha > 0$ (independent of n) and an algorithm with the following property. If it is possible to satisfy k^* of the constraints, then the algorithm produces an ordering of markers satisfying at least αk^* of the constraints. You can provide either be a deterministic algorithm running in polynomial time, or a randomized algorithm which has expected polynomial running time and always produces an approximation within a factor of α.

6. Suppose that a tree network is grown according to the following randomized process. We begin in step 1, with a single isolated node v_1. In any step $k \geq 2$, we introduce a new node v_k and draw an edge from v_k to a node chosen uniformly at random from $v_1, v_2, \ldots, v_{k-1}$. We stop after step n, with a tree on n nodes.

   What is the expected number of leaves in the resulting tree?

7. Let $G = (V, E)$ be an undirected graph with n nodes and m edges. For a subset $X \subseteq V$, we use $G[X]$ to denote the subgraph *induced* on X — that is, the graph $(X, \{(u, v) \in E : u, v \in X\})$.

   We are given a natural number $k \leq n$, and are interested in finding a set of k nodes that induces a "dense" subgraph of G; we'll phrase this concretely as follows. Give a polynomial-time algorithm that produces, for a given natural number $k \leq n$, a set $X \subseteq V$ of k nodes with the property that the induced subgraph $G[X]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

   You may give either (a) a deterministic algorithm, or (b) a randomized algorithm that has an expected running time achieving the given bound, and which only outputs correct answers.

8. We are given a set of variables $x_1, x_2, \ldots, x_n$, each of which can take one of the three values $\{0, 1, 2\}$. We are also given a set of k *constraints* $C_1, C_2, \ldots, C_k$, each of which has the form $x_i \neq x_j$ for some choice of i and j. An assignment of values to the variables *satisfies* a constraint $x_i \neq x_j$ if the value assigned to x_i is different from the value assigned to x_j.

   Consider the problem of finding an assignment of values to variables that maximizes the number of satisfied constraints. This problem is NP-hard, though you don't have to prove this.

   Let c^* denote the maximum possible number of constraints that can be satisfied by an assignment of values to variables. Give a polynomial-time algorithm that produces an assignment satisfying at least $\frac{2}{3}c^*$ constraints. If you want, your algorithm can be randomized; in this case, the *expected* number of constraints it satisfies should be at least $\frac{2}{3}c^*$. In either case, you should prove that your algorithm has the desired performance guarantee.

9. Suppose you're a consultant for Price Gouging Unlimited, and they come to you with the following problem. They've contracted out their algorithmic skills to a large long-distance telephone carrier, which is trying to redesign its area-code system to improve revenue. Their area-code system works as follows: calls between two numbers within one area code are *local* and cost c_1 cents per minute; calls between two numbers in different area codes are *long distance* and cost c_2 cents per minute, where $c_2 > c_1$.

   The particular problem being tossed around in the halls of PGU at the moment is as follows. The phone company has selected a specific one of its area codes — call it A — and it plans to split it so as to maximize its revenue on the resulting set of long-distance calls. More concretely, they have a graph $G = (V, E)$ whose vertex set is the set of all phone number within area code A. Between each pair $u, v \in V$ there is an edge (u, v) whose *weight* w_{uv} is equal to the total number of minutes per month that phone numbers u and v are connected to each other. So the current revenue generated by calls within area code A is $\sum_{u,v \in V} c_1 w_{uv}$. We'll assume (unrealistically) that these weights w_{uv} remain the same from one month to the next. The phone company plans to

   - create two new area codes B and C,
   - partition the phone numbers in the set V into sets V_1 and V_2,
   - assign V_1 to area code B, and
   - assign V_2 to area code C.

   A phone call from B to C (or from C to B) now becomes a long-distance call and costs c_2 cents per minute — thus, the phone company stands to make more money per

month on phone calls among numbers in the set V after it splits the area code.

Your goal is: find the partition (V_1, V_2) of V that maximizes the phone company's monthly revenue, subject to the weights $\{w_{uv}\}$.

Here's an extremely simple randomized algorithm for this problem:

> For each phone number, assign it independently at random to one of the two area codes, with equal probability.

Show that the expected value of the revenue from the partition generated by this algorithm is at least 50% as large as the revenue generated by the optimal partition.

10. Assume you have n balls and n bins, and each ball is placed in a bin selected independently at random (with each bin equally likely). Throughout this problem use the approximation $(1 - 1/n)^n \approx 1/e$ whenever it is useful.

    (a.) Prove that the expected number of empty bins is approaches n/e for large n. Hint: remember that expectation is linear.

    (b.) Assume that you have n jobs and n machines, and each job selects a machine independently at random (with each machine equally likely). Assume that if a machine is selected by more than one job, it will do the first job, and reject the rest. What is the expected number of rejected jobs?

    (c.) Now assume in the above job-machine example each machine will do the first two jobs, and reject the rest if more than two jobs are assigned to it. What is the expected number of rejected jobs now?

11. Consider the following analog of Karger's algorithm for finding minimum $s-t$-cuts. We will contract edges analogous to Karger's algorithm. Let s and t denote the possibly contracted node that contains the original nodes s and t respectively. To make sure that s and t do not get contracted, at each iteration we delete the edges connecting s and t, and select a random edge to contract among the remaining edges. Give an example to show that the probability that this method finds a minimum $s - t$ cut can be exponentially small. Hint: How many minimum capacity s-t cuts can there be?

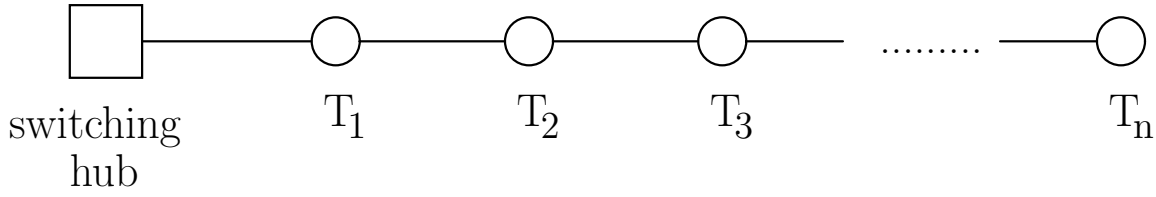
12. Consider a county in which $100,000$ people vote in an election. There are only two candidates on the ballot: a Democratic candidate (denoted D) and a Republican candidate (denoted R). As it happens, this county is heavily Democratic, so $80,000$ people go to the polls with the intention of voting for D and $20,000$ go to the polls with the intention of voting for R.

    However, the layout of the ballot is a little confusing, so each voter, independently and with probability $\frac{1}{100}$, votes for the wrong candidate; i.e. the one that he or she *didn't*

intend to vote for. (Remember that in this election, there are only two candidates on the ballot.)

Let X denote the random variable equal to the number of votes received by the Democratic candidate D, when the voting is conducted with this process of error. Determine the expected value of X, and give an explanation of your derivation of this value.

13. Out in a rural part of the state somewhere, n small towns have decided to get connected to a large Internet switching hub via a high-volume fiber-optic cable.



The towns are labeled $T_1, T_2, \ldots, T_n$, and they are all arranged on a single long highway, so that town T_i is i miles from the switching hub.

Now, this cable is quite expensive; it costs k dollars per mile, resulting in an overall cost of kn dollars for the whole cable. The towns get together and discuss how to divide up the cost of the cable.

First, one of the towns way out at the far end of the highway makes the following proposal.

> **Proposal A.** Divide the cost evenly among all towns, so each pays k dollars.

There's some sense in which Proposal A is fair, since it's like each town is paying for the mile of cable directly leading up to it.

But one of the towns very close to the switching hub objects, pointing out that the far-away towns are actually benefitting from a large section of the cable, whereas the close-in towns only benefit from a short section of it. So they make the following counter-proposal.

> **Proposal B.** Divide the cost so that the contribution of town T_i is proportional to i, its distance from the switching hub.

One of the other towns very close to the switching hub points out that there's another way to do a non-proportional division which is also natural. This is based on conceptually dividing the cable into n equal-length "edges" $e_1, \ldots, e_n$, where the first edge e_1 runs from the switching hub to T_1, and the i^{th} edge e_i $(i > 1)$ runs from T_{i-1} to T_i. Now we observe that while all the towns benefit from e_1, only the last town benefits from e_n. So they suggest

Proposal C. Divide the cost separately for each edge e_i. The cost of e_i should be shared equally by the towns $T_i, T_{i+1}, \ldots, T_n$, since these are the towns "downstream" of e_i.

0.2inSo now the towns have many different options; which is the fairest? To resolve this they turn to the work of Lloyd Shapley, one of the most famous mathematical economists of the 20th century; he proposed what is now called the *Shapley value* as a general mechanism for sharing costs or benefits among several parties. It can be viewed as determining the "marginal contribution" of each party, *assuming the parties arrive in a random order.*

Here's how it would work concretely in our setting. Consider an ordering $\mathcal{O}$ of the towns, and suppose that the towns "arrive" in this order. The *marginal cost of town T_i in order $\mathcal{O}$* is determined as follows. If T_i is first in the order $\mathcal{O}$, then T_i pays ki, the cost of running the cable all the way from the switching hub to T_i. Otherwise, look at the set of towns that come before T_i in the order $\mathcal{O}$, and let T_j be the farthest among these towns from the switching hub. When T_i arrives, we assume the cable already reaches out to T_j but no farther. So if $j > i$ (T_j is farther out than T_i), then the marginal cost of T_i is 0, since the cable already runs past T_i on its way out to T_j. On the other hand, if $j < i$, then the marginal cost of T_i is $k(i - j)$: the cost of extending the cable from T_j out to T_i.

(For example, suppose $n = 3$ and the towns arrive in the order T_1, T_3, T_2. First T_1 pays k when it arrives. Then, when T_3 arrives, it only has to pay $2k$ to extend the cable from T_1. Finally, when T_2 arrives, it doesn't have to pay anything since the cable already runs past it out to T_3.)

0.2inNow, let X_i be the random variable equal to the marginal cost of town T_i when the order $\mathcal{O}$ is selected uniformly at random from all permutations of the towns. Under the rules of the Shapley value, the amount that T_i should contribute to the overall cost of the cable is the expected value of X_i.

The question is: Which of the above three proposals, if any, gives the same division of costs as the Shapley value cost-sharing mechanism? Give a proof for your answer.

14. Suppose you're designing strategies for selling items on a popular auction Web site. Unlike other auction sites, this one uses a *one-pass auction*, in which each bid must be immediately (and irrevocably) accepted or refused. Specifically,

    - First, a seller puts up an item for sale.
    - Then buyers appear in sequence.
    - When buyer i appears, he or she makes a bid $b_i > 0$.

- The seller must decide immediately whether to accept the bid or not. If the seller accepts the bid, the item is sold and all future buyers are turned away. If the seller rejects the bid, buyer i departs and the bid is withdrawn; and only then does the seller see any future buyers.

Suppose an item is offered for sale, and there are n buyers, each with a distinct bid. Suppose further that the buyers appear in a random order, and that the seller knows the number n of buyers. We'd like to design a *strategy* whereby the seller has a reasonable chance of accepting the highest of the n bids. By a "strategy," we mean a rule by which the seller decides whether to accept each presented bid, based only on the value of n and the sequence of bids seen so far.

For example, the seller could always accept the first bid presented. This results in the seller accepting the highest of the n bids with probability only $1/n$, since it requires the highest bid to be the first one presented.

Give a strategy under which the seller accepts the highest of the n bids with probability at least $1/4$, regardless of the value of n. (For simplicity, you are allowed to assume that n is an even number.) Prove that your strategy achieves this probabilistic guarantee.

15. Suppose you are presented with a very large set S of real numbers, and you'd like to approximate the median of these numbers by sampling. You may assume all the numbers in S are distinct. Let $n = |S|$; we will say that a number x is an *ε-approximate median* of S if at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are less than x, and at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are greater than x.

    Consider an algorithm that works as follows. You select a subset $S' \subseteq S$ uniformly at random, compute the median of S', and return this as an approximate median of S. Show that there is an absolute constant c, independent of n, so that if you apply this algorithm with a sample S' of size c, then with probability at least .99, the number returned will be a (.05)-approximate median of S. (You may consider either the version of the algorithm that constructs S' by sampling with replacement, so that an element of S can be selected multiple times, or without replacement.)

16. Consider the following simple model of gambling in the presence of bad odds. At the beginning, your net profit is 0. You play for a sequence of n rounds; and in each round, your net profit increases by 1 with probability $1/3$, and decreases by 1 with probability $2/3$.

    Show that the expected number of steps in which your net profit is positive can be upper-bounded by an absolute constant, independent of the value of n.

17. Consider a balls and bins experiment with $2n$ balls but only 2 bins. As usual, each ball independently selects one of the two bins, both bins equally likely. The expected number of balls in each bin is n. In this problem we explore the question of how big their difference is likely to be. Let X_1 and X_2 denote the number of balls in the two bins respectively. (X_1 and X_2 are random variables.) Prove that for any $\varepsilon > 0$ there is a constant $c > 0$ such that the probability $\Pr[X_1 - X_2 > c\sqrt{n}] \leq \varepsilon$.

18. Consider the following (partially specified) method for transmitting a message securely between a sender and a receiver. The message will be represented as a string of bits. Let $\Sigma = \{0, 1\}$, and let Σ^* denote the set of all strings of 0 or more bits. (E.g. $0, 00, 1110001 \in \Sigma^*$. The "empty string", with no bits, will be denoted $\lambda \in \Sigma^*$.)

    The sender and receiver share a secret function $f : \Sigma^* \times \Sigma \to \Sigma$. That is, f takes a word and a bit, and returns a bit. When the receiver gets a sequence of bits $\alpha \in \Sigma^*$, he/she runs the following method to decipher it:

    ```
    Let α = α_1 α_2 ··· α_n, where n is the number of bits in α.
    The goal is to produce an n-bit deciphered message, denoted β =
    β_1 β_2 ··· β_n.
    Set β_1 := f(λ, α_1).
    For i = 2, 3, 4, ..., n
      Set β_i := f(β_1 β_2 ··· β_{i-1}, α_i).
      End for
    Output β.
    ```

    One could view this is as type of "stream cipher with feedback." One problem with this approach is that if any bit α_i gets corrupted in transmission, it will corrupt the computed value of β_j for all $j \geq i$.

    We consider the following problem. A sender S wants to transmit the same (plain-text) message β to each of k receivers $R_1, \ldots, R_k$. With each one, he shares a different secret function $f^{\langle i \rangle}$. Thus, he sends a different encrypted message $\alpha^{\langle i \rangle}$ to each receiver, so that $\alpha^{\langle i \rangle}$ decrypts to β when the above algorithm is run with the function $f^{\langle i \rangle}$.

    Unfortunately, the communication channels are very noisy, so each of the n bits in each of the k transmissions is *independently* corrupted (i.e. flipped to its complement) with probability 1/4. Thus, no single receiver on his/her own is likely to be able to decrypt the message correctly. Show, however, that if k is large enough as a function of n, then the k receivers can jointly reconstruct the plain-text message in the following way. They get together, and without revealing any of the $\alpha^{\langle i \rangle}$ or the $f^{\langle i \rangle}$, they interactively run an algorithm that will produce the correct β with probability at least 9/10. (How large do you need k to be in your algorithm?)

19. Some people designing parallel physical simulations come to you with the following problem. They have a set P of k *basic processes*, and want to assign each process to run on one of two machines, M_1 and M_2. They are then going to run a sequence of n *jobs*, $J_1, \ldots, J_n$. Each job J_i is represented by a set $P_i \subseteq P$ of exactly $2n$ basic processes which must be running (each on its assigned machine) while the job is processed. An assignment of basic processes to machines will be called *perfectly balanced* if, for each job J_i, exactly n of the basic processes associated with J_i have been assigned to each of the two machines. An assignment of basic processes to machines will be called *nearly balanced* if, for each job J_i, no more than $\frac{4}{3}n$ of the basic processes associated with J_i have been assigned to the same machine.

    (a) Show that for arbitrarily large values of n, there exist sequences of jobs $J_1, \ldots, J_n$ for which no perfectly balanced assignment exists.

    (b) Suppose that $n \geq 200$. Give an algorithm that takes an arbitrary sequence of jobs $J_1, \ldots, J_n$ and produces a nearly balanced assignment of basic processes to machines. Your algorithm may be randomized, in which case its expected running time should be polynomial, and it should always produce the correct answer.

Chapter 13

Epilogue: Algorithms that Run Forever

Every decade has its addictive puzzles; and if Rubik's Cube stands out as the pre-eminent solitaire recreation of the early eighties, then Tetris evokes a similar nostalgia for the early nineties. Rubik's Cube and Tetris have a number of things in common — they share a highly mathematical flavor, based on stylized geometric forms — but the differences between them are perhaps more interesting.

Rubik's Cube is a game whose complexity is based on an enormous search space; given a scrambled configuration of the Cube, you have to apply an intricate sequence of operations to reach the ultimate goal. On the other hand, Tetris — in its pure form — has a much fuzzier definition of "success"; rather than aiming for a particular endpoint, you're faced with a basically infinite stream of events to be dealt with, and you have to react continuously so as to keep your head above water.

These novel features of Tetris parallel an analogous set of themes that has emerged in recent thinking about algorithms. Increasingly, we face settings in which the standard view of algorithms — in which one begins with an input, runs for a finite number of steps, and produces an output — does not really apply. Rather, if we think about Internet routers that move packets while avoiding congestion, or decentralized file-sharing mechanisms that replicate and distribute content to meet user demand, or machine learning routines that form predictive models of concepts that change over time, then we are dealing with algorithms that effectively are designed *to run forever*. Instead of producing an eventual output, they succeed if they can keep up with an environment that is in constant flux and continuously throwing new tasks at them. For such applications, we have shifted from the world of Rubik's Cube to the world of Tetris.

There are many settings in which we could explore this theme, and for this final lecture we consider one of the most compelling — the design of algorithms for high-speed switching on the Internet.

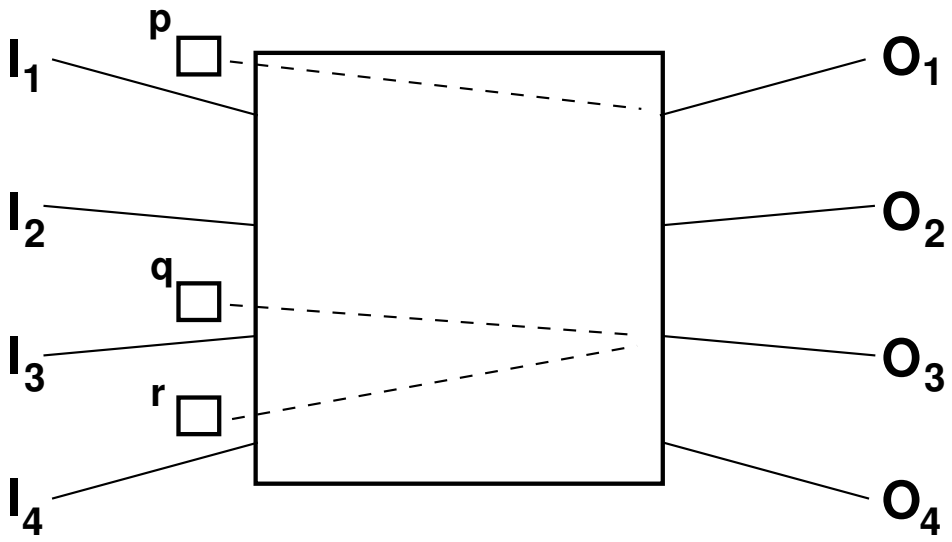


Figure 13.1: A switch with $n = 4$ inputs and outputs. In one time step, packets p, q, and r have arrived.

Algorithms for Packet Switching

A packet traveling from a source to a destination on the Internet can be thought of as traversing a path in a large graph whose nodes are switches, and whose edges are the cables that link switches together. Each packet p has a header from which a switch can determine, when p arrives on an input link, the output link on which p needs to depart. The goal of a switch is thus to take streams of packets arriving on its input links and move each packet, as quickly as possible, to the particular output link on which it needs to depart. How quickly? In high-volume settings, it is possible for a packet to arrive on each input link once every few tens of nanoseconds; if they aren't off-loaded to their respective output links at a comparable rate, then traffic will back up and packets will be dropped.

In order to think about the algorithms operating inside a switch, we model the switch itself as follows. It has n *input links* $I_1, \ldots, I_n$ and n *output links* $O_1, \ldots, O_n$. Packets arrive on the input links; a given packet p has an associated input/output *type* $(I[p], O[p])$ indicating that it has arrived at input link $I[p]$ and needs to depart on output link $O[p]$. Time moves in discrete *steps*; in each step, at most one new packet arrives on each input link, and at most one packet can depart on each output link.

Consider the example in Figure 13.1. In a single time step, the three packets p, q, and r have arrived at an empty switch on input links I_1, I_3, and I_4 respectively. Packet p is destined for O_1, packet q is destined for O_3, and packet r is also destined for O_3. Now, there's no problem sending packet p out on link O_1; but only one packet can depart on link O_3, and so the switch has to resolve the contention between q and r. How can it do this?

The simplest model of switch behavior is known as *pure output queueing*, and it's essentially an idealized picture of how we wished a switch behaved. In this model, all nodes that arrive in a given time step are placed in an *output buffer* associated with their output link, and one of the packets in each output buffer actually gets to depart. More concretely, here's the model of a single time step.

```
One step under pure output queueing:
  Packets arrive on input links.
  Each packet p of type (I[p], O[p]) is moved to output buffer O[p].
  At most one packet departs from each output buffer.
```

So in Figure 13.1, the given time step could end with packets p and q having departed on their output links, and with packet r sitting in the output buffer O_3. (In discussing this example here and below, we'll assume that q is favored over r when decisions are made.) Under this model, the switch is basically a "frictionless" object through which packets pass unimpeded to their output buffer.

In reality, however, a packet that arrives on an input link must be copied over to its appropriate output link, and this operation requires some processing that ties up both the input and output links for a few nanoseconds. So really, constraints within the switch do pose some obstacles to the movement of packets from inputs to outputs.

The most restrictive model of these constraints, *input/output queueing*, works as follows. We now have an *input buffer* for each input link I, as well as an output buffer for each output link O. When each packet arrives, it immediately lands in its associated input buffer. In a single time step, a switch can read at most one packet from each input buffer and write at most one packet to each output buffer. So under input/output queueing, the example of Figure 13.1 would work as follows. Each of p, q, and r would arrive in different input buffers; the switch could then move p and q to their output buffers, but it could not move all three — since moving all three would involve writing two packets into the output buffer O_3. Thus, the first step would end with p and q have departed on their output links, and r sitting in the input buffer I_4 (rather than in the output buffer O_3).

More generally, the restriction of limited reading and writing amounts to the following: if packets $p_1, \ldots, p_\ell$ are moved in a single time steps from input buffers to output buffers, then all their input buffers and all their output buffers must be distinct. In other words, their types $\{(I[p_i], O[p_i]) : i = 1, 2, \ldots, \ell\}$ must form a biparite *matching*. Thus, we can model a single time step as follows.

```
One step under input/output queueing:
  Packets arrive on input links and are placed in input buffers.
  A set of packets whose types form a matching are moved to their
    associated output buffers.
  At most one packet departs from each output buffer.
```

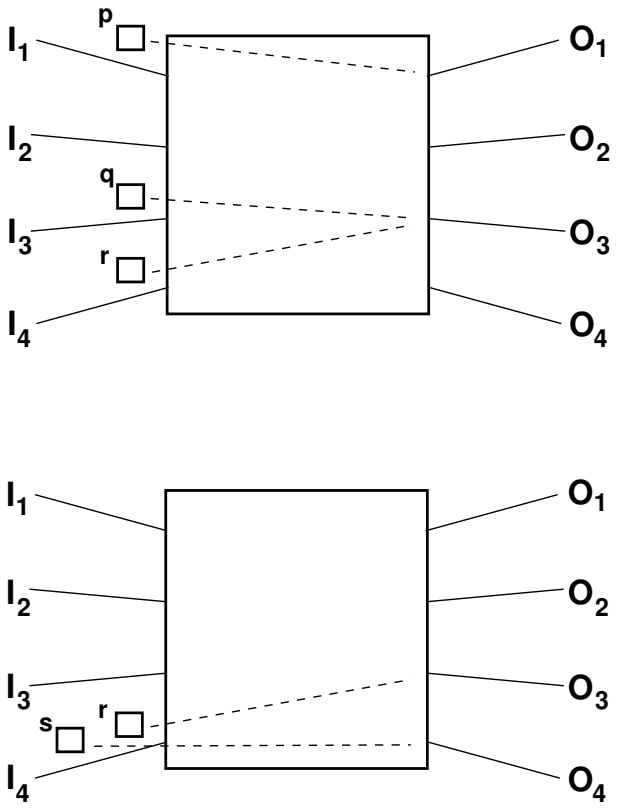


Figure 13.2: A two-step example in which head-of-line blocking occurs.

The choice of which matching to move is left unspecified for now; this is a point that will become crucial below.

So under input/output queueing, the switch introduces some "friction" on the movement of packets, and this is an observable phenomenon: if we view the switch as a black box, and simply watch the sequence of departures on the output links, then we can see the difference between pure output queueing and input/output queueing. Consider an example whose first step is just like Figure 13.1, and in whose second step a single packet s of type (I_4, O_4) arrives. Under pure output queueing, p and q would depart in the first step, and r and s would depart in the second step. Under input/output queueing, however, the sequence of events depicted in Figure 13.2 occurs: at the end of the first step, r is still sitting in the input buffer I_4, and so at the end of the second step one of r or s is still in the input buffer I_4 and has not yet departed. This conflict between r and s is called "head-of-line" blocking, and it causes a switch with input/output queueing to exhibit inferior delay characteristics compared with pure output queueing.

Simulating a Switch with Pure Output Queueing. While pure output queueing would be nice to have, the arguments above indicate why it's not feasible to design a switch

with this behavior: in a single time step (lasting only tens of nanoseconds) it would not generally be possible to move packets from each of n input links to a common output buffer.

But what if we were to take a switch that used input/output queueing and ran it somewhat faster, moving several matchings in a single time step instead of just one? Would it be possible to simulate a switch that used pure output queueing? By this we mean that the sequence of departures on the output links (viewing the switch as a black box) should be the same under the behavior of pure output queueing and the behavior of our sped-up input/output queueing algorithm.

It is not hard to see that a speed-up of n would suffice: if we could move n matchings in each time step, then even if every arriving packet needed to reach the same output buffer, we could move them all in the course of one step. But a speed-up of n is completely infeasible; and if we think about this worst-case example, we begin to worry that we might need a speed-up of n to make this work — after all, what if all the arriving packets really did need to go to the same output buffer?

The crux of this lecture is to show that a much more modest (and arguably feasible) speed-up is sufficient: we'll describe a striking result of Chuang, Goel, McKeown, and Prabhakar, showing that a switch using input/output queueing with a speed-up of 2 can simulate a switch that uses pure output queueing. Intuitively, the result exploits the fact that the behavior of the switch at an internal level need not resemble the behavior under pure output queueing, provided that the sequence of output link departures is the same. (Hence, to continue the example in the previous paragraph, it's okay that we don't move all n arriving packets to a common output buffer in one time step; we can afford more time for this, since their departures on this common output link will be spread out over a long period of time anyway.)

Just to be precise, here's our model for a speed-up of 2.

```
One step under sped-up input/output queueing:
  Packets arrive on input links and are placed in input buffers.
  A set of packets whose types form a matching are moved to their
    associated output buffers.
  At most one packet departs from each output buffer.
  A set of packets whose types form a matching are moved to their
    associated output buffers.
```

In order to prove that this model can simulate pure output queueing, we need to resolve the crucial under-specified point in the above model: *which matchings should be moved in each step*? The answer to this question will form the core of the result, and we build up to it through a sequence of intermediate steps. To begin with, we make one simple observation right away: if a packet of type (I, O) is part of a matching selected by the switch, then the switch will move the packet of this type that has the *earliest* time to leave.

Maintaining Input and Output Buffers. To decide which two matchings the switch should move in a given time step, we define some quantities that track the current state of the switch relative to pure output queueing. To begin with, for a packet p, we define its *time to leave*, $TL(p)$, to be the time step in which it would depart on its output link from a switch that was running pure output queueing. The goal is to make sure that each packet p departs from our switch (running sped-up input/output queueing) in precisely the time step $TL(p)$.

Conceptually, each input buffer is maintained as an ordered list; however, we retain the freedom to insert an arriving packet into the middle of this order, and to move a packet to its output buffer even when it is not yet at the front of the line. Despite this, the linear ordering of the buffer will form a useful progress measure. Each output buffer, on the other hand, does not need to be ordered; when a packet's time to leave comes up, we simply let it depart. We can think of the whole set-up as resembling a busy airport terminal, with the input buffers corresponding to check-in counters, the output buffers to the departure lounge, and the internals of the switch to a congested security checkpoint. The input buffers are stressful places: if you don't make it to the head of the line by the time your departure is announced, you could miss your time to leave; to mitigate this, there are airport personnel who are allowed to helpfully extract you from the middle of the line and hustle you through security. The output buffers, by way of contrast, are relaxing places: you sit around until your time to leave is announced, and then you just go. The goal is to get everyone through the congestion in the middle so that they depart on time.

One consequence of these observations is that we don't need to worry about packets that are already in output buffers; they'll just depart at the right time. Hence we refer to a packet p as *unprocessed* if it is still in its input buffer, and we define some further useful quantities for such packets. The *input cushion* $IC(p)$ is the number of packets ordered in front of p in its input buffer. The *output cushion* $OC(p)$ is the number of packets already in p's output buffer that have an earlier time to leave. Things are going well for an unprocessed packet p if $OC(p)$ is significantly greater than $IC(p)$; in this case, p is near the front of the line in its input buffer, and there are still a lot of packets before it in the output buffer. To capture this relationship we define $Slack(p) = OC(p) - IC(p)$, observing that large values of $Slack(p)$ are good.

Here is our plan: we will move matchings through the switch so as to maintain the following two properties at all times:

(i) $Slack(p) \geq 0$ for all unprocessed packets p.

(ii) In any step that begins with $IC(p) = OC(p) = 0$, packet p will be moved to its output buffer in the first matching.

We first claim that it is sufficient to maintain these two properties.

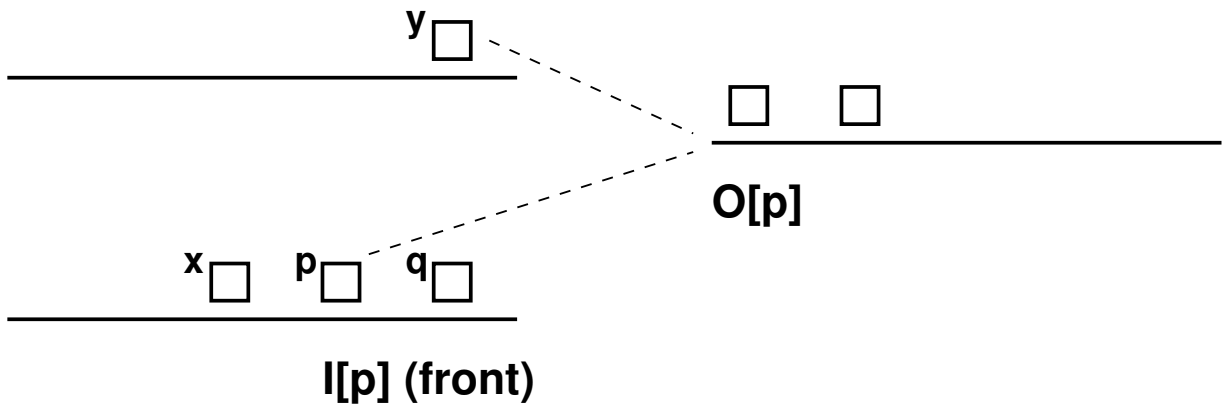


Figure 13.3: Choosing a matching to move.

(13.1) *If properties (i) and (ii) are maintained for all unprocessed packets at all times, then every packet p will depart at its time to leave $TL(p)$.*

Proof. If p is in its output buffer at the start of step $TL(p)$, then it can clearly depart. Otherwise, it must be in its input buffer. In this case, we have $OC(p) = 0$ at the start of the step. By property (i), we have $Slack(p) = OC(p) - IC(p) \geq 0$, and hence $IC(p) = 0$. It then follows from property (ii) that p will be moved to the output buffer in the first matching of this step, and hence will depart in this step as well. ■

It turns out that property (ii) is easy to guarantee — and it will arise naturally from the solution below — so we focus on the tricky task of choosing matchings so as to maintain property (i).

Moving a Matching Through a Switch. When a packet p first arrives on an input link, we insert it as far back in the input buffer as possible (potentially somewhere in the middle) consistent with the requirement $Slack(p) \geq 0$. This makes sure property (i) is satisfied initially for p.

Now, if we want to maintain non-negative slacks over time, then we need to worry about counter-balancing events that cause $Slack(p)$ to decrease. Let's return to the description of a single time step and think about how such decreases can occur.

```
One step under double input/output queueing:
  Packets arrive on input links and are placed in input buffers.
  The switch moves a matching.
  At most one packet departs from each output buffer.
  The switch moves a matching.
```

Consider a given packet p that is unprocessed at the beginning of a time step. In the arrival phase of the step, $IC(p)$ could increase by 1 if the arriving packet is placed in the

input buffer ahead of p. This would cause $Slack(p)$ to decrease by 1. In the departure phase of the step, $OC(p)$ could decrease by 1, since a packet with an earlier time to leave will no longer be in the output buffer. This too would cause $Slack(p)$ to decrease by 1. So in summary, $Slack(p)$ can potentially decrease by 1 in each of the arrival and departure phases. Consequently, we will be able to maintain property (i) if we can guarantee that $Slack(p)$ increases by at least 1 each time the switch moves a matching. How can we do this?

If the matching to be moved includes a packet in $I[p]$ that is *ahead* of p, then $IC(p)$ will decrease and hence $Slack(p)$ will increase. If the matching includes a packet destined for $O[p]$ with an earlier time to leave than p, then $OC(p)$ and $Slack(p)$ will increase. So the only problem is if neither of these things happens. Figure 13.3 gives a schematic picture of such a situation: suppose that packet x is moved out of $I[p]$ even though it is farther back in order, and packet y is moved to $O[p]$ even though it has a later time to leave. In this situation, it seems that buffers $I[p]$ and $O[p]$ have both been treated "unfairly" — it would have been better for $I[p]$ to send a packet like p that was farther forward, and it would have been better for $O[p]$ to receive a packet like p that had an earlier time to leave. Taken together, the two buffers form something reminiscent of an *instability* from the stable matching problem ...

In fact, we can make this precise, and it provides the key to finishing the algorithm. Suppose we say that output buffer O *prefers* input buffer I to I' if the earliest time to leave among packets of type (I, O) is smaller than the earliest time to leave among packets of type (I', O). (In other words, buffer I is more in need of sending something to buffer O.) Further, we say that input buffer I *prefers* output buffer O to output buffer O' if the forward-most packet of type (I, O) comes ahead of the forward-most packet of type (I, O') in the ordering of I. We construct a preference list for each buffer from these rules; and if there are no packets at all of type (I, O), then I and O are placed at the end of each other's preference lists, with ties broken arbitrarily.

The following fact now answers the question of how to choose a matching.

(13.2) *Suppose the switch always moves a stable matching M with respect to the preference lists defined above. (And for each type (I, O) contained in M, we select the packet of this type with the earliest time to leave). Then for all unprocessed packets p, the value $Slack(p)$ decreases by at least 1 when the matching M is moved.*

Proof. Consider any unprocessed packet p. Following the discussion above, suppose that no packet ahead of p in $I[p]$ is moved as part of the matching M, and no packet destined for $O[p]$ with an earlier time to leave is moved as part of M. So in particular, the pair $(I[p], O[p])$ is not in M; suppose that pairs $(I', O[p])$ and $(I[p], O')$ belong to M.

Now, p has an earlier time to leave than any packet of type $(I', O[p])$, and it comes ahead of every packet of type $(I[p], O')$ in the ordering of $I[p]$. It follows that $I[p]$ prefers $O[p]$ to O', and $O[p]$ prefers $I[p]$ to I' — hence, the pair $(I[p], O[p])$ forms an instability, which contradicts our assumption that M is stable. ■

Thus, by moving a stable matching in every step, the switch maintains the property $Slack(p) \geq 0$ for all packets p; hence, by (13.1), the switch is able to simulate the behavior of pure output queueing. Overall, it makes for a surprising last-minute appearance by the topic with which we began the course — and rather than matching men with women or applicants with employers, we find ourselves matching input links to output links in a high-speed Internet router.

This has been one glimpse into the issue of algorithms that run forever, keeping up with an infinite stream of new events; it is an intriguing topic, full of open directions and unresolved issues. But that is for another time, and another course; and as for us — we are done.