Chapter 1

Introduction: Some Representative Problems

1.1 A First Problem: Stable Matching

As an opening topic, we look at an algorithmic problem that nicely illustrates many of the themes we will be emphasizing. It is 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.

The Problem

The Stable Matching Problem originated, in part, 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 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 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 that Raj's friend Chelsea, destined to go to 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 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 the process as well as 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 redirected. Consider another student, 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 over 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 note 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, with relatively little change, is still in use today.

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

Formulating the Problem To get at the essence of 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.

It is useful, at least initially, to 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 issues inherent in the problem; in particular, our solution to this simplified version will extend directly to the more general case as well.

Following Gale and Shapley, we observe that this special case can be viewed as the problem of devising a system by which each of n men and n women can end up getting married: our problem naturally has the analogue of two "genders"—the applicants and the companies—and in the case we are considering, everyone is seeking to be paired with exactly one individual of the opposite gender.[1]

[1] Gale and Shapley considered the same-sex stable matching problem as well, where there is only a single gender. This is motivated by related applications, but it turns out to be fairly different at a technical level. Given the applicant-employer application we're considering here, we'll be focusing on the version with two genders.

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 throughout the book; 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 the men with the women, 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.

An instability: m and w' each prefer the other to their current partners.

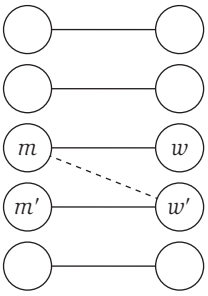

Figure 1.1 Perfect matching S with instability (m, w').

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 (as depicted in Figure 1.1) 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 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?

Some Examples To illustrate these definitions, consider the following two very simple instances of the Stable Matching Problem.

First, 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'.

If we think about this set of preference lists intuitively, it represents complete agreement: the men agree on the order of the women, and the women agree on the order of the men. There is a unique stable matching here, consisting of the pairs (m, w) and (m', w'). The other perfect matching, consisting of the pairs (m', w) and (m, w'), would not be a stable matching, because the pair (m, w) would form an instability with respect to this matching. (Both m and w would want to leave their respective partners and pair up.)

Next, here's an example where things are a bit more intricate. Suppose the preferences are

m prefers w to w'.

m' prefers w' to w.

w prefers m' to m.

w' prefers m to m'.

What's going on in this case? The two men's preferences mesh perfectly with each other (they rank different women first), and the two women's preferences likewise mesh perfectly with each other. But the men's preferences clash completely with the women's preferences.

In this second example, there are two different stable matchings. The matching consisting of the pairs (m, w) and (m', w') is stable, because both men are as happy as possible, so neither would leave their matched partner. But the matching consisting of the pairs (m', w) and (m, w') is also stable, for the complementary reason that both women are as happy as possible. This is an important point to remember as we go forward—it's possible for an instance to have more than one stable matching.

Designing the Algorithm

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 also answer the second question that we asked above: 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

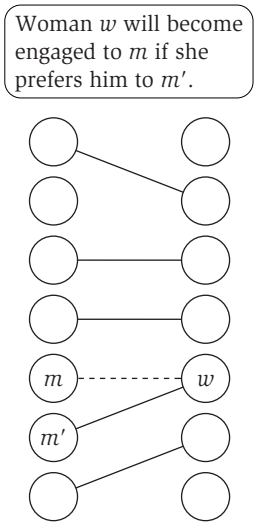

Figure 1.2 An intermediate state of the G-S algorithm when a free man m is proposing to a woman w.

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*, with Figure 1.2 depicting a state of the 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 whom 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.

Analyzing the Algorithm

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. 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. ■

Extensions

We began by defining the notion of a stable matching; we have just proven that the G-S algorithm actually constructs one. We now consider some further questions about the behavior of the G-S algorithm and its relation to the properties of different stable matchings.

To begin with, recall that we saw an example earlier in which there could be multiple stable matchings; to recap, the preference lists in this example were as follows:

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

Now, 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. Thus, the *other* stable matching, consisting of the pairs (m', w) and (m, w'), is not attainable from an execution of the G-S algorithm in which the men propose. On the other hand, it would be reached if we ran a version of the algorithm in which the women propose. 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.

This example shows a certain "unfairness" in the G-S algorithm, favoring men. If the men's preferences mesh perfectly (they all list different women as their first choice), then in all runs of the G-S algorithm all men end up matched with their first choice, independent of the preferences of women. If women's preferences clash completely with the men's preferences (as was the case in this example), then the resulting stable matching is as bad as possible for the women. So this simple set of preference lists compactly summarizes a world in which *someone* is destined to end up unhappy: women are unhappy if men propose, and men are unhappy if women propose.

Let's now analyze the G-S algorithm in more detail and try to understand how general this "unfairness" phenomenon is.

To begin with, our example reinforces the point that the G-S algorithm is actually underspecified: 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 interleaved in complex ways, and we want to know how much variability this asynchrony causes in the final outcome. To consider a very different kind of example, the independent components may not be men and women but electronic components activating parts of an airplane wing; the effect of asynchrony in 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 of the G-S algorithm yield the same matching. We proceed to prove this now.

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? We'll show that each man ends up with the "best possible partner" in a concrete sense. (Recall that this is true if all men prefer different women.) 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 $best(m)$ to denote the best valid partner of m.

Now, let S^* denote the set of pairs $\{(m, best(m)) : m \in M\}$. 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. 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 $best(m)$.

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 woman 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 was 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. ■

Thus, we find that our simple example above, in which the men's preferences clashed with the women's, hinted at a very general phenomenon: for any input, the side that does the proposing in the G-S algorithm ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

1.2 Five Representative Problems

The Stable Matching Problem provides us with a rich example of the process of algorithm design. For many problems, this design process involves a few significant steps: formulating the problem with enough mathematical precision that we can ask a concrete question and start thinking about algorithms to solve it; developing an algorithm for the problem; proving the algorithm is correct; and giving a bound on the running time so as to establish the algorithm's efficiency.

This high-level strategy is carried out in practice with the help of a few fundamental design techniques, which are very useful in assessing the inherent complexity of a problem and in formulating an algorithm to solve it. As in any area, becoming familiar with these design techniques is a gradual process; but with experience one can start recognizing problems as belonging to identifiable genres and appreciating how subtle changes in the statement of a problem can have an enormous effect on its complexity.

To get this discussion started, then, it helps to pick out a few representative milestones that we'll be encountering in our study of algorithms: cleanly formulated problems, all resembling one another at a general level, but differing greatly in their difficulty and in the kinds of approaches that one brings to bear on them. The first three will be solvable efficiently by a sequence of increasingly subtle algorithmic techniques; the fourth marks a major turning point in our discussion, serving as an example of a problem believed to be unsolvable by any efficient algorithms; and the fifth hints at a class of problems believed to be harder still.

The problems are self-contained and are all motivated by computing applications. To talk about some of them, though, it will help to use the terminology of *graphs*. While graphs are a common topic in earlier computer science courses, we'll be introducing them in a fair amount of depth in

Chapter 3; due to their enormous expressive power, we'll also be using them extensively throughout the book. For the discussion here, it's enough to think of a graph G as 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 *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 1.3, with each node as a small circle and each edge as a line segment joining its two ends.

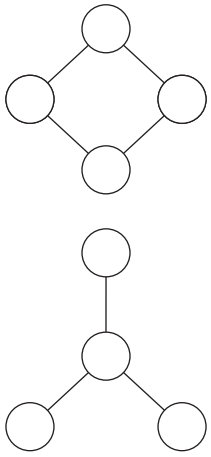

Figure 1.3 Two graphs, each on four nodes.

Let's now turn to a discussion of the five representative problems.

Interval Scheduling

Consider the following very simple scheduling problem. You have a resource—it may be a lecture room, a supercomputer, or an electron microscope—and many people request to use the resource for periods of time. A *request* takes the form: Can 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 1.4. Note that there is a single compatible set of size four, and this is the largest compatible set.

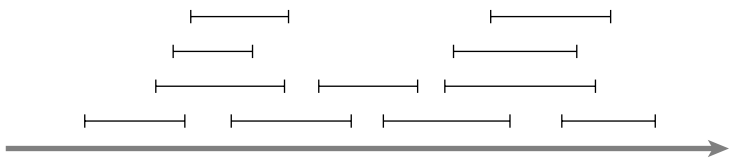

Figure 1.4 An instance of the Interval Scheduling Problem.

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*, $v_i > 0$; 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 value.

The case in which $v_i = 1$ for each i is simply the basic Interval Scheduling Problem; but the appearance of arbitrary values changes the nature of the maximization problem quite a bit. Consider, for example, that if v_1 exceeds the sum of all other v_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, and yet degenerate to a method for solving (unweighted) interval scheduling when all the values 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 values. Instead, we employ a technique, *dynamic programming*, that builds up the optimal value over all possible solutions in a compact, tabular way that leads to a very efficient algorithm.

Bipartite Matching

When we considered the Stable Matching Problem, 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 graphs, and in order to do this it is useful to define the notion of a *bipartite graph*. 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 1.5; 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 1.3 are also bipartite.

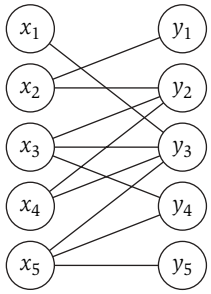

Figure 1.5 A bipartite graph.

Now, in the problem of finding a stable matching, matchings were built from pairs of men and women. 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 matchings 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. In other words, it is as though only certain pairs of men and women are willing to be paired off, and we want to figure out how to pair off many people in a way that is consistent with this. Consider, for example, the bipartite graph G in Figure 1.5: 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, computer science departments across the country are often seen pondering a bipartite graph in which X is the set of professors in the department, Y is the set of offered courses, and an edge $(x_i.y_j)$ indicates that professor x_i is capable of teaching course y_j. 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 earlier do not seem adequate

for providing an efficient algorithm for this problem. There is, however, a very elegant and efficient 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*.

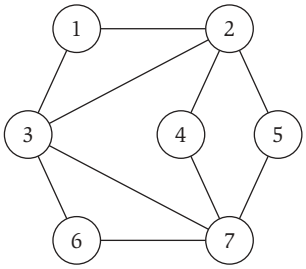

Figure 1.6 A graph whose largest independent set has size four.

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. For example, the maximum size of an independent set in the graph in Figure 1.6 is four, achieved by the four-node independent set $\{1, 4, 5, 6\}$.

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 interpersonal 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.[2]

[2] 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 1.3 cannot arise as the "conflict graph" in an instance

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 efficient 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 book that Independent Set is one of a large class of problems that are termed *NP-complete*. No efficient algorithm is known for any of them; but they are all *equivalent* in the sense that a solution to any one of them would imply, in a precise sense, a solution to 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 1,000 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 be 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 two large companies that operate café franchises across the country—let's call them JavaPlanet and Queequeg's Coffee—and they are currently competing for market share in a geographic area. First JavaPlanet opens a franchise; then Queequeg's Coffee opens a franchise; then JavaPlanet; then Queequeg's; and so on. . . . Suppose they must deal with zoning regulations that require no two franchises be located too close 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

of Interval Scheduling, and the second graph in Figure 1.3 cannot arise as the "conflict graph" in an instance of Bipartite Matching.

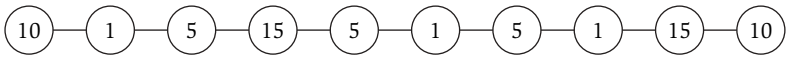

Figure 1.7 An instance of the Competitive Facility Location Problem.

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 containing a franchise, 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 in 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 with a total value of at least B? We will call this an instance of the *Competitive Facility Location Problem*.

Consider, for example, the instance pictured in Figure 1.7, 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.

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.

Solved Exercises

Solved Exercise 1

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.

Show that in every stable matching, every good man is married to a good woman.

Solution A natural way to get started thinking about this problem is to assume the claim is false and try to work toward obtaining a contradiction. What would it mean for the claim to be false? There would exist some stable matching M in which a good man m was married to a bad woman w.

Now, let's consider what the other pairs in M look like. There are k good men and k good women. Could it be the case that every good woman is married to a good man in this matching M? No: one of the good men (namely m) is already married to a bad woman, and that leaves only $k-1$ other good men. So even if all of them were married to good women, that would still leave some good woman who is married to a bad man.

Let w' be such a good woman, and let m' be the bad man she is married to. It is now easy to identify an instability in M: consider the pair (m, w'). Each is good, but is married to a bad partner. Thus, each of m and w' prefers the other to their current partner, and hence (m, w') is an instability. This contradicts our assumption that M is stable, and hence concludes the proof.

Solved Exercise 2

We can think about a generalization of the Stable Matching Problem, in which certain man-woman pairs are explicitly *forbidden*. In the case of employers and applicants, we could imagine that certain applicants simply lack the necessary qualifications or certifications; and so they cannot be employed at certain companies, however desirable they may seem. Using the analogy to marriage between men and women, 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, w') not in F, 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.)*

(iii) 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.)*

(iv) There is a man m and a woman w, neither of whom 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) give an algorithm that, for any set of preference lists and forbidden pairs, produces a stable matching; or (b) give an example of a set of preference lists and forbidden pairs for which there is no stable matching.

Solution The Gale-Shapley algorithm is remarkably robust to variations on the Stable Matching problem. So, if you're faced with a new variation of the problem and can't find a counterexample to stability, it's often a good idea to check whether a direct adaptation of the G-S algorithm will in fact produce stable matchings.

That turns out to be the case here. We will show that there is always a stable matching, even in this more general model with forbidden pairs, and we will do this by adapting the G-S algorithm. To do this, let's consider why the original G-S algorithm can't be used directly. The difficulty, of course, is that the G-S algorithm doesn't know anything about forbidden pairs, and so the condition in the `While` loop,

> `While there is a man` m `who is free and hasn't proposed to every woman`,

won't work: we don't want m to propose to a woman w for which the pair (m, w) is forbidden.

Thus, let's consider a variation of the G-S algorithm in which we make only one change: we modify the `While` loop to say,

> `While there is a man` m `who is free and hasn't proposed to every woman` w `for which` $(m, w) \notin F$.

Here is the algorithm in full:

```
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 w for which (m, w) ∉ F
   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
```

We now prove that this yields a stable matching, under our new definition of stability.

To begin with, facts (1.1), (1.2), and (1.3) from the text remain true (in particular, the algorithm will terminate in at most n^2 iterations). Also, we don't have to worry about establishing that the resulting matching S is perfect (indeed, it may not be). We also notice an additional pairs of facts. If m is a man who is not part of a pair in S, then m must have proposed to every non-forbidden woman; and if w is a woman who is not part of a pair in S, then it must be that no man ever proposed to w.

Finally, we need only show

(1.9) *There is no instability with respect to the returned matching* S.

Proof. Our general definition of instability has four parts: This means that we have to make sure that none of the four bad things happens.

First, suppose there is an instability of type (i), consisting of pairs (m, w) and (m', w') in S with the property that (m, w) not in F, and m prefers w' to w, and w' prefers m to m'. It follows that m must have proposed to w'; so w' rejected m, and thus she prefers her final partner to m—a contradiction.

Next, suppose there is an instability of type (ii), consisting of 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. Then m' must have proposed to w and been rejected; again, it follows that w prefers her final partner to m'—a contradiction.

Third, suppose there is an instability of type (ii′), consisting of 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. Then no man proposed to w' at all; in particular, m never proposed to w', and so he must prefer w to w'—a contradiction.

Finally, suppose there is an instability of type (iii), consisting of a man m and a woman w, neither of which is part of any pair in the matching, so that $(m, w) \notin F$. But for m to be single, he must have proposed to every non-forbidden woman; in particular, he must have proposed to w, which means she would no longer be single—a contradiction. ■

Exercises

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair (m, w) such that m is ranked first on the preference list of w and w is ranked first on the preference list of m.

2. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m. Then in every stable matching S for this instance, the pair (m, w) belongs to S.

3. 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, whom we'll call $\mathcal{A}$ and $\mathcal{D}$. 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 *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 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) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or (b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

4. 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 the section, 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.

5. The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be *indifferent* between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with $n = 4$), a woman could say that m_1 is ranked in first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in last place. We will say that w *prefers* m to m' if m is ranked higher than m' on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A *strong instability* in a perfect matching S consists of a man m and a woman w, such that m and w prefer each other to their partners in S. Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give

an efficient algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) A *weak instability* in a perfect matching S is a man m and a woman w, such that their partners in S are w' and m', respectively, and one of the following holds:

- m prefers w to w', and w either prefers m to m' or is indifferent between these two choices; or
- w prefers m to m', and m either prefers w to w' or is indifferent between these two choices.

In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an efficient algorithm that is guaranteed to find a perfect matching with no weak instability.

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 the 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 problem. 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. (Figure 1.8 gives an example with its solution.)

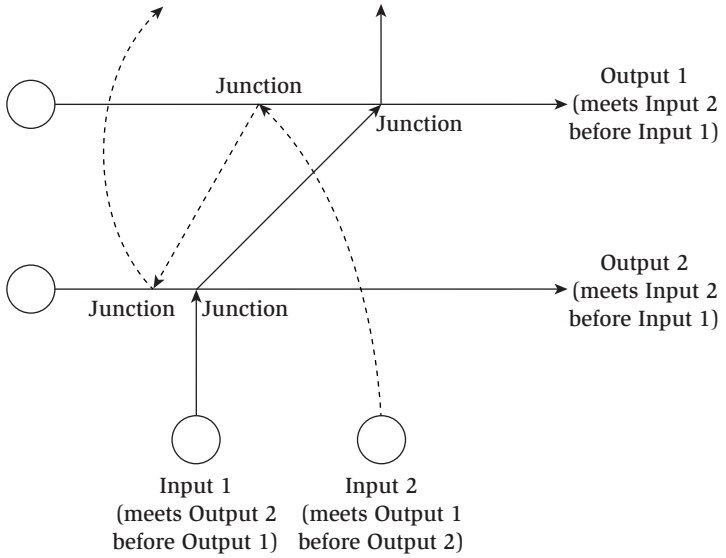

Figure 1.8 An example with two input wires and two output wires. Input 1 has its junction with Output 2 upstream from its junction with Output 1; Input 2 has its junction with Output 1 upstream from its junction with Output 2. A valid solution is to switch the data stream of Input 1 onto Output 2, and 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.

8. 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) Give 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) Give 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.

Notes and Further Reading

The Stable Matching Problem was first defined and analyzed by Gale and Shapley (1962); according to David Gale, their motivation for the problem came from a story they had recently read in the *New Yorker* about the intricacies of the college admissions process (Gale, 2001). Stable matching has grown into an area of study in its own right, covered in books by Gusfield and Irving (1989) and Knuth (1997). Gusfield and Irving also provide a nice survey of the "parallel" history of the Stable Matching Problem as a technique invented for matching applicants with employers in medicine and other professions.

As discussed in the chapter, our five representative problems will be central to the book's discussions, respectively, of greedy algorithms, dynamic programming, network flows, NP-completeness, and PSPACE-completeness. We will discuss the problems in these contexts later in the book.