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, at least 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.

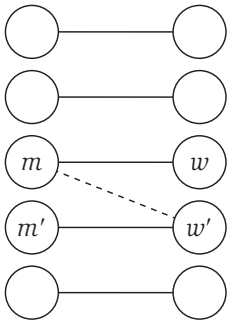

Figure 1.1 Perfect matching S with instability (m, w').

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

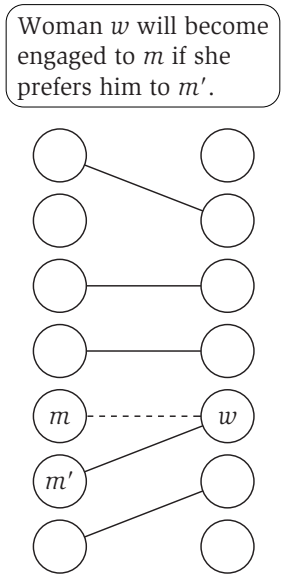

Figure 1.2 An intermediate state of the G-S algorithm when a free man m is proposing to a woman w.

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 earlier, 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 the women. If the 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 a matching at all, let alone a stable matching. After all, why couldn't it happen that two men have the same best valid partner? Second, 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 or continues 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 $(m', w) \notin S'$, it follows that (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 were a pair (m, w) in S^* such 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 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; designing an algorithm for the problem; and analyzing the algorithm by proving it 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 computational difficulty.

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 algorithm; 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