Introduction to Algorithms

Jon Kleinberg Éva Tardos

Cornell University
Spring 2003

Contents

Chapter 1

Introduction

1.1 Introduction: The Stable Matching Problem

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

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

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

Gale and Shapley considered the sorts of things that could start going wrong with this process, in the absence of any mechanism to enforce the status quo. Suppose, for example, that your friend Raj has just accepted a summer job at the large telecommunications company CluNet. A few days later, the small start-up company WebExodus, which had been dragging its feet on making a few final decisions, calls up Raj and offers him a summer job as well. Now, Raj actually prefers WebExodus to CluNet — won over perhaps by the laid-back, anything-can-happen atmosphere — and so this new development may well cause him to retract his acceptance of the CluNet offer, and go to WebExodus instead. Suddenly down one summer intern, CluNet offers a job to one of its wait-listed applicants, who promptly retracts his previous acceptance of an offer from the software giant Babelsoft, and the situation begins to spiral out of control.

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

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

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

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

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

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

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

Gale and Shapley proceeded to develop a striking algorithmic solution to this problem, which we will discuss presently. Before doing this, let's comment that this is not the only origin of the Stable Matching Problem. It turns out that for a decade before the work of Gale and Shapley — unbeknownst to them — the National Resident Matching Program had been using a very similar procedure, with the same underlying motivation, to match residents to hospitals. Indeed, this system is still in use today.

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

Defining the Problem

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

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

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

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

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

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

Given a perfect matching S, what can go wrong? Guided by our initial motivation in terms of employers and applicants, we should be worried about the following situation: There are two pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. In this case, there's nothing to stop m and w' from abandoning their current partners and heading off into the sunset together; the set of marriages is not *self-enforcing.* We'll say that such a pair (m, w') is an *instability* with respect to S: (m, w') does not belong to S, but each of m and w' prefers the other to their partner in S.

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

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

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

Constructing a Stable Matching

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. At no time is anyone engaged to more than one person, and so the set of engaged pairs always forms a matching. Let us suppose that the algorithm terminates with a free man m. At termination, it must be the case that m had already proposed to every woman, for otherwise the `While` loop would not have exited. But this contradicts (1.4), which says that there cannot be a free man who has proposed to every woman. ■

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

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

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

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

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

It follows that S is a stable matching. ■

All Executions Yield the Man-Optimal Matching

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

Thus, we encounter another very natural question:

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

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

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

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

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

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

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

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

Despite all this, the proof is not so difficult.

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

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

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

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

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

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

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

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

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

Example: Multiple Stable Matchings

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

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

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

In any execution of the Gale-Shapley algorithm, m will become engaged to w, m' will become engaged to w' (perhaps in the other order), and things will stop there. Indeed, this matching is as good as possible for the men, and — as we see by inspecting the preference lists — as bad as possible for the women.

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

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

1.2 Computational Tractability

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

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

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

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

One way would be to use the following working definition:

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

Let's spend a little time considering this definition. At a certain level, it's hard to argue with; one of the goals at the bedrock of our study of algorithms is that of solving real problems quickly. And indeed, there is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Interlude: Two Definitions

Order of Growth Notation

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

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

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

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

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

Graphs

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

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

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

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