Chapter 12

Local Search

In the previous two chapters, we have considered techniques for dealing with computationally intractable problems: in Chapter 10, by identifying structured special cases of NP-hard problems, and in Chapter 11, by designing polynomial-time approximation algorithms. We now develop a third and final topic related to this theme: the design of *local search algorithms*.

Local search is a very general technique; it describes any algorithm that "explores" the space of possible solutions in a sequential fashion, moving in one step from a current solution to a "nearby" one. The generality and flexibility of this notion has the advantage that it is not difficult to design a local search approach to almost any computationally hard problem; the counterbalancing disadvantage is that it is often very difficult to say anything precise or provable about the quality of the solutions that a local search algorithm finds, and consequently very hard to tell whether one is using a good local search algorithm or a poor one.

Our discussion of local search in this chapter will reflect these trade-offs. Local search algorithms are generally heuristics designed to find good, but not necessarily optimal, solutions to computational problems, and we begin by talking about what the search for such solutions looks like at a global level. A useful intuitive basis for this perspective comes from connections with energy minimization principles in physics, and we explore this issue first. Our discussion for this part of the chapter will have a somewhat different flavor from what we've generally seen in the book thus far; here, we'll introduce some algorithms, discuss them qualitatively, but admit quite frankly that we can't prove very much about them.

There are cases, however, in which it is possible to prove properties of local search algorithms, and to bound their performance relative to an

optimal solution. This will be the focus of the latter part of the chapter: We begin by considering a case—the dynamics of Hopfield neural networks—in which local search provides the natural way to think about the underlying behavior of a complex process; we then focus on some NP-hard problems for which local search can be used to design efficient algorithms with provable approximation guarantees. We conclude the chapter by discussing a different type of local search: the game-theoretic notions of best-response dynamics and Nash equilibria, which arise naturally in the study of systems that contain many interacting agents.

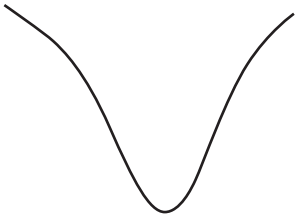

Figure 12.1 When the potential energy landscape has the structure of a simple funnel, it is easy to find the lowest point.

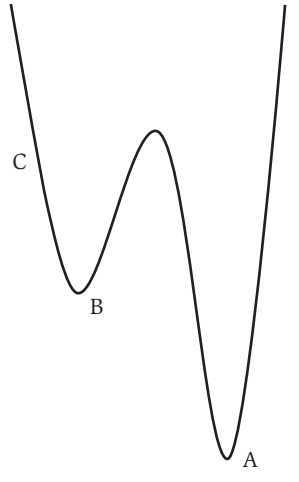

Figure 12.2 Most landscapes are more complicated than simple funnels; for example, in this "double funnel," there's a deep global minimum and a shallower local minimum.

12.1 The Landscape of an Optimization Problem

Much of the core of local search was developed by people thinking in terms of analogies with physics. Looking at the wide range of hard computational problems that require the minimization of some quantity, they reasoned as follows. Physical systems are performing minimization all the time, when they seek to minimize their potential energy. What can we learn from the ways in which nature performs minimization? Does it suggest new kinds of algorithms?

Potential Energy

If the world really looked the way a freshman mechanics class suggests, it seems that it would consist entirely of hockey pucks sliding on ice and balls rolling down inclined surfaces. Hockey pucks usually slide because you push them; but why do balls roll downhill? One perspective that we learn from Newtonian mechanics is that the ball is trying to minimize its *potential energy*. In particular, if the ball has mass m and falls a distance of h, it loses an amount of potential energy proportional to mh. So, if we release a ball from the top of the funnel-shaped landscape in Figure 12.1, its potential energy will be minimized at the lowest point.

If we make the landscape a little more complicated, some extra issues creep in. Consider the "double funnel" in Figure 12.2. Point A is lower than point B, and so is a more desirable place for the ball to come to rest. But if we start the ball rolling from point C, it will not be able to get over the barrier between the two funnels, and it will end up at B. We say that the ball has become trapped in a *local minimum*: It is at the lowest point if one looks in the neighborhood of its current location; but stepping back and looking at the whole landscape, we see that it has missed the *global minimum*.

Of course, enormously large physical systems must also try to minimize their energy. Consider, for example, taking a few grams of some homogeneous substance, heating it up, and studying its behavior over time. To capture the potential energy exactly, we would in principle need to represent the

behavior of each atom in the substance, as it interacts with nearby atoms. But it is also useful to speak of the properties of the system as a whole—as an aggregate—and this is the domain of statistical mechanics. We will come back to statistical mechanics in a little while, but for now we simply observe that our notion of an "energy landscape" provides useful visual intuition for the process by which even a large physical system minimizes its energy. Thus, while it would in reality take a huge number of dimensions to draw the true "landscape" that constrains the system, we can use one-dimensional "cartoon" representations to discuss the distinction between local and global energy minima, the "funnels" around them, and the "height" of the energy barriers between them.

Taking a molten material and trying to cool it to a perfect crystalline solid is really the process of trying to guide the underlying collection of atoms to its global potential energy minimum. This can be very difficult, and the large number of local minima in a typical energy landscape represent the pitfalls that can lead the system astray in its search for the global minimum. Thus, rather than the simple example of Figure 12.2, which simply contains a single wrong choice, we should be more worried about landscapes with the schematic cartoon representation depicted in Figure 12.3. This can be viewed as a "jagged funnel," in which there are local minima waiting to trap the system all the way along its journey to the bottom.

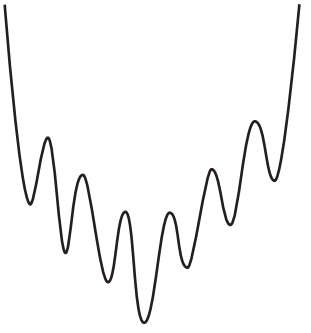

Figure 12.3 In a general energy landscape, there may be a very large number of local minima that make it hard to find the global minimum, as in the "jagged funnel" drawn here.

The Connection to Optimization

This perspective on energy minimization has really been based on the following core ingredients: The physical system can be in one of a large number of possible states; its energy is a function of its current state; and from a given state, a small perturbation leads to a "neighboring" state. The way in which these neighboring states are linked together, along with the structure of the energy function on them, defines the underlying energy landscape.

It's from this perspective that we again start to think about computational minimization problems. In a typical such problem, we have a large (typically exponential-size) set $\mathcal{C}$ of possible solutions. We also have a *cost function* $c(\cdot)$ that measures the quality of each solution; for a solution $S \in \mathcal{C}$, we write its cost as $c(S)$. The goal is to find a solution $S^* \in \mathcal{C}$ for which $c(S^*)$ is as small as possible.

So far this is just the way we've thought about such problems all along. We now add to this the notion of a *neighbor relation* on solutions, to capture the idea that one solution S' can be obtained by a small modification of another solution S. We write $S \sim S'$ to denote that S' is a neighboring solution of S, and we use $N(S)$ to denote the *neighborhood* of S, the set $\{S' : S \sim S'\}$. We will primarily be considering symmetric neighbor relations here, though the

basic points we discuss will apply to asymmetric neighbor relations as well. A crucial point is that, while the set $\mathcal{C}$ of possible solutions and the cost function $c(\cdot)$ are provided by the specification of the problem, we have the freedom to make up any neighbor relation that we want.

A *local search algorithm* takes this setup, including a neighbor relation, and works according to the following high-level scheme. At all times, it maintains a current solution $S \in \mathcal{C}$. In a given step, it chooses a neighbor S' of S, declares S' to be the new current solution, and iterates. Throughout the execution of the algorithm, it remembers the minimum-cost solution that it has seen thus far; so, as it runs, it gradually finds better and better solutions. The crux of a local search algorithm is in the choice of the neighbor relation, and in the design of the rule for choosing a neighboring solution at each step.

Thus one can think of a neighbor relation as defining a (generally undirected) graph on the set of all possible solutions, with edges joining neighboring pairs of solutions. A local search algorithm can then be viewed as performing a walk on this graph, trying to move toward a good solution.

An Application to the Vertex Cover Problem

This is still all somewhat vague without a concrete problem to think about; so we'll use the Vertex Cover Problem as a running example here. It's important to keep in mind that, while Vertex Cover makes for a good example, there are many other optimization problems that would work just as well for this illustration.

Thus we are given a graph $G = (V, E)$; the set $\mathcal{C}$ of possible solutions consists of all subsets S of V that form vertex covers. Hence, for example, we always have $V \in \mathcal{C}$. The *cost* $c(S)$ of a vertex cover S will simply be its size; in this way, minimizing the cost of a vertex cover is the same as finding one of minimum size. Finally, we will focus our examples on local search algorithms that use a particularly simple neighbor relation: we say that $S \sim S'$ if S' can be obtained from S by adding or deleting a single node. Thus our local search algorithms will be walking through the space of possible vertex covers, adding or deleting a node to their current solution in each step, and trying to find as small a vertex cover as possible.

One useful fact about this neighbor relation is the following.

(12.1) *Each vertex cover S has at most n neighboring solutions.*

The reason is simply that each neighboring solution of S is obtained by adding or deleting a distinct node. A consequence of (12.1) is that we can efficiently examine all possible neighboring solutions of S in the process of choosing which to select.

Let's think first about a very simple local search algorithm, which we'll term *gradient descent*. Gradient descent starts with the full vertex set V and uses the following rule for choosing a neighboring solution.

> *Let S denote the current solution. If there is a neighbor S' of S with strictly lower cost, then choose the neighbor whose cost is as small as possible. Otherwise terminate the algorithm.*

So gradient descent moves strictly "downhill" as long as it can; once this is no longer possible, it stops.

We can see that gradient descent terminates precisely at solutions that are *local minima*: solutions S such that, for all neighboring S', we have $c(S) \leq c(S')$. This definition corresponds very naturally to our notion of local minima in energy landscapes: They are points from which no one-step perturbation will improve the cost function.

How can we visualize the behavior of a local search algorithm in terms of the kinds of energy landscapes we illustrated earlier? Let's think first about gradient descent. The easiest instance of Vertex Cover is surely an n-node graph with no edges. The empty set is the optimal solution (since there are no edges to cover), and gradient descent does exceptionally well at finding this solution: It starts with the full vertex set V, and keeps deleting nodes until there are none left. Indeed, the set of vertex covers for this edge-less graph corresponds naturally to the funnel we drew in Figure 12.1: The unique local minimum is the global minimum, and there is a downhill path to it from any point.

When can gradient descent go astray? Consider a "star graph" G, consisting of nodes $x_1, y_1, y_2, \ldots, y_{n-1}$, with an edge from x_1 to each y_i. The minimum vertex cover for G is the singleton set $\{x_1\}$, and gradient descent can reach this solution by successively deleting $y_1, \ldots, y_{n-1}$ in any order. But, if gradient descent deletes the node x_1 first, then it is immediately stuck: No node y_i can be deleted without destroying the vertex cover property, so the only neighboring solution is the full node set V, which has higher cost. Thus the algorithm has become trapped in the local minimum $\{y_1, y_2, \ldots, y_{n-1}\}$, which has very high cost relative to the global minimum.

Pictorially, we see that we're in a situation corresponding to the double funnel of Figure 12.2. The deeper funnel corresponds to the optimal solution $\{x_1\}$, while the shallower funnel corresponds to the inferior local minimum $\{y_1, y_2, \ldots, y_{n-1}\}$. Sliding down the wrong portion of the slope at the very beginning can send one into the wrong minimum. We can easily generalize this situation to one in which the two minima have any relative depths we want. Consider, for example, a bipartite graph G with nodes $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_\ell$, where $k < \ell$, and there is an edge from every node of the form x_i

to every node of the form y_j. Then there are two local minima, corresponding to the vertex covers $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_\ell\}$. Which one is discovered by a run of gradient descent is entirely determined by whether it first deletes an element of the form x_i or y_j.

With more complicated graphs, it's often a useful exercise to think about the kind of landscape they induce; and conversely, one sometimes may look at a landscape and consider whether there's a graph that gives rise to something like it.

For example, what kind of graph might yield a Vertex Cover instance with a landscape like the jagged funnel in Figure 12.3? One such graph is simply an n-node path, where n is an odd number, with nodes labeled $v_1, v_2, \ldots, v_n$ in order. The unique minimum vertex cover S^* consists of all nodes v_i where i is even. But there are many local optima. For example, consider the vertex cover $\{v_2, v_3, v_5, v_6, v_8, v_9, \ldots\}$ in which every third node is omitted. This is a vertex cover that is significantly larger than S^*; but there's no way to delete any node from it while still covering all edges. Indeed, it's very hard for gradient descent to find the minimum vertex cover S^* starting from the full vertex set V: Once it's deleted just a single node v_i with an even value of i, it's lost the chance to find the global optimum S^*. Thus the even/odd parity distinction in the nodes captures a plethora of different wrong turns in the local search, and hence gives the overall funnel its jagged character. Of course, there is not a direct correspondence between the ridges in the drawing and the local optima; as we warned above, Figure 12.3 is ultimately just a cartoon rendition of what's going on.

But we see that even for graphs that are structurally very simple, gradient descent is much too straightforward a local search algorithm. We now look at some more refined local search algorithms that use the same type of neighbor relation, but include a method for "escaping" from local minima.

12.2 The Metropolis Algorithm and Simulated Annealing

The first idea for an improved local search algorithm comes from the work of Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953). They considered the problem of simulating the behavior of a physical system according to principles of statistical mechanics. A basic model from this field asserts that the probability of finding a physical system in a state with energy E is proportional to the *Gibbs-Boltzmann function* $e^{-E/(kT)}$, where $T > 0$ is the temperature and $k > 0$ is a constant. Let's look at this function. For any temperature T, the function is monotone decreasing in the energy E, so this states that a physical

system is more likely to be in a lower energy state than in a high energy state. Now let's consider the effect of the temperature T. When T is small, the probability for a low-energy state is significantly larger than the probability for a high-energy state. However, if the temperature is large, then the difference between these two probabilities is very small, and the system is almost equally likely to be in any state.

The Metropolis Algorithm

Metropolis et al. proposed the following method for performing step-by-step simulation of a system at a fixed temperature T. At all times, the simulation maintains a current state of the system and tries to produce a new state by applying a perturbation to this state. We'll assume that we're only interested in states of the system that are "reachable" from some fixed initial state by a sequence of small perturbations, and we'll assume that there is only a finite set $\mathcal{C}$ of such states. In a single step, we first generate a small random perturbation to the current state S of the system, resulting in a new state S'. Let $E(S)$ and $E(S')$ denote the energies of S and S', respectively. If $E(S') \leq E(S)$, then we update the current state to be S'. Otherwise let $\Delta E = E(S') - E(S) > 0$. We update the current state to be S' with probability $e^{-\Delta E/(kT)}$, and otherwise leave the current state at S.

Metropolis et al. proved that their simulation algorithm has the following property. To prevent too long a digression, we omit the proof; it is actually a direct consequence of some basic facts about random walks.

(12.2) *Let*

$$Z = \sum_{S \in \mathcal{C}} e^{-E(S)/(kT)}.$$

For a state S, let $f_S(t)$ denote the fraction of the first t steps in which the state of the simulation is in S. Then the limit of $f_S(t)$ as t approaches ∞ is, with probability approaching 1, equal to $\frac{1}{Z} \cdot e^{-E(S)/(kT)}$.

This is exactly the sort of fact one wants, since it says that the simulation spends roughly the correct amount of time in each state, according to the Gibbs-Boltzmann equation.

If we want to use this overall scheme to design a local search algorithm for minimization problems, we can use the analogies of Section 12.1 in which states of the system are candidate solutions, with energy corresponding to cost. We then see that the operation of the Metropolis Algorithm has a very desirable pair of features in a local search algorithm: It is biased toward "downhill"

moves but will also accept "uphill" moves with smaller probability. In this way, it is able to make progress even when situated in a local minimum. Moreover, as expressed in (12.2), it is globally biased toward lower-cost solutions.

Here is a concrete formulation of the Metropolis Algorithm for a minimization problem.

```
Start with an initial solution S_0, and constants k and T
In one step:
   Let S be the current solution
   Let S' be chosen uniformly at random from the neighbors of S
   If c(S') ≤ c(S) then
      Update S ← S'
   Else
      With probability e^{-(c(S')-c(S))/(kT)}
         Update S ← S'
      Otherwise
         Leave S unchanged
   EndIf
```

Thus, on the Vertex Cover instance consisting of the star graph in Section 12.1, in which x_1 is joined to each of $y_1, \ldots, y_{n-1}$, we see that the Metropolis Algorithm will quickly bounce out of the local minimum that arises when x_1 is deleted: The neighboring solution in which x_1 is put back in will be generated and will be accepted with positive probability. On more complex graphs as well, the Metropolis Algorithm is able, to some extent, to correct the wrong choices it makes as it proceeds.

At the same time, the Metropolis Algorithm does not always behave the way one would want, even in some very simple situations. Let's go back to the very first graph we considered, a graph G with no edges. Gradient descent solves this instance with no trouble, deleting nodes in sequence until none are left. But, while the Metropolis Algorithm will start out this way, it begins to go astray as it nears the global optimum. Consider the situation in which the current solution contains only c nodes, where c is much smaller than the total number of nodes, n. With very high probability, the neighboring solution generated by the Metropolis Algorithm will have size $c + 1$, rather than $c - 1$, and with reasonable probability this uphill move will be accepted. Thus it gets harder and harder to shrink the size of the vertex cover as the algorithm proceeds; it is exhibiting a sort of "flinching" reaction near the bottom of the funnel.

This behavior shows up in more complex examples as well, and in more complex ways; but it is certainly striking for it to show up here so simply. In order to figure out how we might fix this behavior, we return to the physical analogy that motivated the Metropolis Algorithm, and ask: What's the meaning of the temperature parameter T in the context of optimization?

We can think of T as a one-dimensional knob that we're able to turn, and it controls the extent to which the algorithm is willing to accept uphill moves. As we make T very large, the probability of accepting an uphill move approaches 1, and the Metropolis Algorithm behaves like a random walk that is basically indifferent to the cost function. As we make T very close to 0, on the other hand, uphill moves are almost never accepted, and the Metropolis Algorithm behaves almost identically to gradient descent.

Simulated Annealing

Neither of these temperature extremes—very low or very high—is an effective way to solve minimization problems in general, and we can see this in physical settings as well. If we take a solid and heat it to a very high temperature, we do not expect it to maintain a nice crystal structure, even if this is energetically favorable; and this can be explained by the large value of kT in the expression $e^{-E(S)/(kT)}$, which makes the enormous number of less favorable states too probable. This is a way in which we can view the "flinching" behavior of the Metropolis Algorithm on an easy Vertex Cover instance: It's trying to find the lowest energy state at too high a temperature, when all the competing states have too high a probability. On the other hand, if we take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either; rather, we get a deformed crystal structure with many imperfections. This is because, with T very small, we've come too close to the realm of gradient descent, and the system has become trapped in one of the numerous ridges of its jagged energy landscape. It is interesting to note that when T is very small, then statement (12.2) shows that in the limit, the random walk spends most of its time in the lowest energy state. The problem is that the random walk will take an enormous amount of time before getting anywhere near this limit.

In the early 1980s, as people were considering the connection between energy minimization and combinatorial optimization, Kirkpatrick, Gelatt, and Vecchi (1983) thought about the issues we've been discussing, and they asked the following question: How do we solve this problem for physical systems, and what sort of algorithm does this suggest? In physical systems, one guides a material to a crystalline state by a process known as *annealing*: The material is cooled very gradually from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures. In this

way, it is able to escape from the energy minima that it encounters all the way through the cooling process, eventually arriving at the global optimum.

We can thus try to mimic this process computationally, arriving at an algorithmic technique known as *simulated annealing*. Simulated annealing works by running the Metropolis Algorithm while gradually decreasing the value of T over the course of the execution. The exact way in which T is updated is called, for natural reasons, a *cooling schedule*, and a number of considerations go into the design of the cooling schedule. Formally, a cooling schedule is a function τ from $\{1, 2, 3, \ldots\}$ to the positive real numbers; in iteration i of the Metropolis Algorithm, we use the temperature $T = \tau(i)$ in our definition of the probability.

Qualitatively, we can see that simulated annealing allows for large changes in the solution in the early stages of its execution, when the temperature is high. Then, as the search proceeds, the temperature is lowered so that we are less likely to undo progress that has already been made. We can also view simulated annealing as trying to optimize a trade-off that is implicit in (12.2). According to (12.2), values of T arbitrarily close to 0 put the highest probability on minimum-cost solutions; *however*, (12.2) by itself says nothing about the rate of convergence of the functions $f_S(t)$ that it uses. It turns out that these functions converge, in general, much more rapidly for large values of T; and so to find minimum-cost solutions quickly, it is useful to speed up convergence by starting the process with T large, and then gradually reducing it so as to raise the probability on the optimal solutions. While we believe that physical systems reach a minimum energy state via annealing, the simulated annealing method has no guarantee of finding an optimal solution. To see why, consider the double funnel of Figure 12.2. If the two funnels take equal area, then at high temperatures the system is essentially equally likely to be in either funnel. Once we cool the temperature, it will become harder and harder to switch between the two funnels. There appears to be no guarantee that at the end of annealing, we will be at the bottom of the lower funnel.

There are many open problems associated with simulated annealing, both in proving properties of its behavior and in determining the range of settings for which it works well in practice. Some of the general questions that come up here involve probabilistic issues that are beyond the scope of this book.

Having spent some time considering local search at a very general level, we now turn, in the next few sections, to some applications in which it is possible to prove fairly strong statements about the behavior of local search algorithms and about the local optima that they find.

12.3 An Application of Local Search to Hopfield Neural Networks

Thus far we have been discussing local search as a method for trying to find the global optimum in a computational problem. There are some cases, however, in which, by examining the specification of the problem carefully, we discover that it is really just an arbitrary *local* optimum that is required. We now consider a problem that illustrates this phenomenon.

The Problem

The problem we consider here is that of finding *stable configurations* in *Hopfield neural networks*. Hopfield networks have been proposed as a simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states. Concretely, a Hopfield network can be viewed as an undirected graph $G = (V, E)$, with an integer-valued weight w_e on each edge e; each weight may be positive or negative. A *configuration* S of the network is an assignment of the value -1 or $+1$ to each node u; we will refer to this value as the *state* s_u of the node u. The meaning of a configuration is that each node u, representing a unit of the neural network, is trying to choose between one of two possible states ("on" or "off"; "yes" or "no"); and its choice is influenced by those of its neighbors as follows. Each edge of the network imposes a *requirement* on its endpoints: If u is joined to v by an edge of negative weight, then u and v want to have the same state, while if u is joined to v by an edge of positive weight, then u and v want to have opposite states. The absolute value $|w_e|$ will indicate the *strength* of this requirement, and we will refer to $|w_e|$ as the *absolute weight* of edge e.

Unfortunately, there may be no configuration that respects the requirements imposed by all the edges. For example, consider three nodes a, b, c all mutually connected to one another by edges of weight 1. Then, no matter what configuration we choose, two of these nodes will have the same state and thus will be violating the requirement that they have opposite states.

In view of this, we ask for something weaker. With respect to a given configuration, we say that an edge $e = (u, v)$ is *good* if the requirement it imposes is satisfied by the states of its two endpoints: either $w_e < 0$ and $s_u = s_v$, or $w_e > 0$ and $s_u \neq s_v$. Otherwise we say e is *bad*. Note that we can express the condition that e is good very compactly, as follows: $w_e s_u s_v < 0$. Next we say that a node u is *satisfied* in a given configuration if the total absolute weight

of all good edges incident to u is at least as large as the total absolute weight of all bad edges incident to u. We can write this as

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0.$$

Finally, we call a configuration *stable* if all nodes are satisfied.

Why do we use the term *stable* for such configurations? This is based on viewing the network from the perspective of an individual node u. On its own, the only choice u has is whether to take the state -1 or $+1$; and like all nodes, it wants to respect as many edge requirements as possible (as measured in absolute weight). Suppose u asks: Should I flip my current state? We see that if u does flip its state (while all other nodes keep their states the same), then all the good edges incident to u become bad, and all the bad edges incident to u become good. So, to maximize the amount of good edge weight under its direct control, u should flip its state if and only if it is not satisfied. In other words, a stable configuration is one in which no individual node has an incentive to flip its current state.

A basic question now arises: Does a Hopfield network always have a stable configuration, and if so, how can we find one?

Designing the Algorithm

We will now design an algorithm that establishes the following result.

(12.3) *Every Hopfield network has a stable configuration, and such a configuration can be found in time polynomial in n and $W = \sum_e |w_e|$.*

We will see that stable configurations in fact arise very naturally as the local optima of a certain local search procedure on the Hopfield network.

To see that the statement of (12.3) is not entirely trivial, we note that it fails to remain true if one changes the model in certain natural ways. For example, suppose we were to define a *directed Hopfield network* exactly as above, except that each edge is directed, and each node determines whether or not it is satisfied by looking only at edges for which it is the tail. Then, in fact, such a network need not have a stable configuration. Consider, for example, a directed version of the three-node network we discussed earlier: There are nodes a, b, c, with directed edges $(a, b), (b, c), (c, a)$, all of weight 1. Then, if all nodes have the same state, they will all be unsatisfied; and if one node has a different state from the other two, then the node directly in front of it will be unsatisfied. Thus there is no configuration of this directed network in which all nodes are satisfied.

It is clear that a proof of (12.3) will need to rely somewhere on the undirected nature of the network.

To prove (12.3), we will analyze the following simple iterative procedure, which we call the State-Flipping Algorithm, to search for a stable configuration.

```
While the current configuration is not stable
  There must be an unsatisfied node
  Choose an unsatisfied node u
  Flip the state of u
Endwhile
```

An example of the execution of this algorithm is depicted in Figure 12.4, ending in a stable configuration.

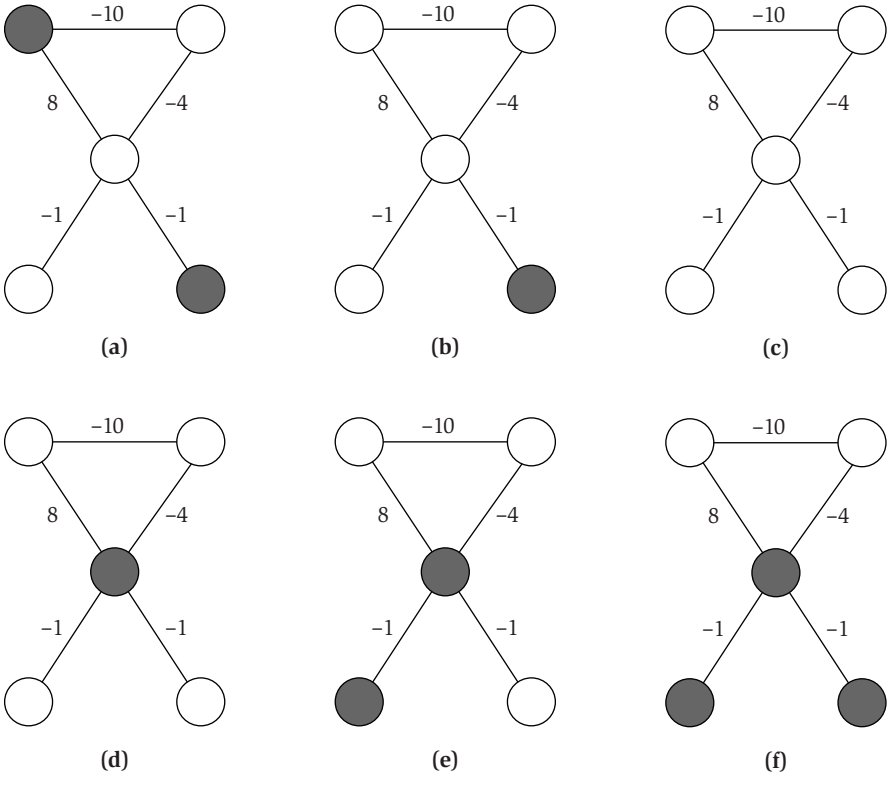

Figure 12.4 Parts (a)–(f) depict the steps in an execution of the State-Flipping Algorithm for a five-node Hopfield network, ending in a stable configuration. (Nodes are colored black or white to indicate their state.)

Analyzing the Algorithm

Clearly, if the *State-Flipping Algorithm* we have just defined terminates, we will have a stable configuration. What is not obvious is whether it must in fact terminate. Indeed, in the earlier directed example, this process will simply cycle through the three nodes, flipping their states sequentially forever.

We now prove that the State-Flipping Algorithm always terminates, and we give a bound on the number of iterations it takes until termination. This will provide a proof of (12.3). The key to proving that this process terminates is an idea we've used in several previous situations: to look for a measure of *progress*—namely, a quantity that strictly increases with every flip and has an absolute upper bound. This can be used to bound the number of iterations.

Probably the most natural progress measure would be the number of satisfied nodes: If this increased every time we flipped an unsatisfied node, the process would run for at most n iterations before terminating with a stable configuration. Unfortunately, this does not turn out to work. When we flip an unsatisfied node v, it's true that it has now become satisfied, but several of its previously satisfied neighbors could now become unsatisfied, resulting in a net decrease in the number of satisfied nodes. This actually happens in one of the iterations depicted in Figure 12.4: when the middle node changes state, it renders both of its (formerly satisfied) lower neighbors unsatisfied.

We also can't try to prove termination by arguing that every node changes state at most once during the execution of the algorithm: Again, looking at the example in Figure 12.4, we see that the node in the lower right changes state twice. (And there are more complex examples in which we can get a single node to change state many times.)

However, there is a more subtle progress measure that *does* increase with each flip of an unsatisfied node. Specifically, for a given configuration S, we define $\Phi(S)$ to be the total absolute weight of all good edges in the network. That is,

$$\Phi(S) = \sum_{\text{good } e} |w_e|.$$

Clearly, for any configuration S, we have $\Phi(S) \geq 0$ (since $\Phi(S)$ is a sum of positive integers), and $\Phi(S) \leq W = \sum_e |w_e|$ (since, at most, every edge is good).

Now suppose that, in a nonstable configuration S, we choose a node u that is unsatisfied and flip its state, resulting in a configuration S'. What can we say about the relationship of $\Phi(S')$ to $\Phi(S)$? Recall that when u flips its state, all good edges incident to u become bad, all bad edges incident to u become good, and all edges that don't have u as an endpoint remain the same. So,

if we let g_u and b_u denote the total absolute weight on good and bad edges incident to u, respectively, then we have

$$\Phi(S') = \Phi(S) - g_u + b_u.$$

But, since u was unsatisfied in S, we also know that $b_u > g_u$; and since b_u and g_u are both integers, we in fact have $b_u \geq g_u + 1$. Thus

$$\Phi(S') \geq \Phi(S) + 1.$$

Hence the value of Φ begins at some nonnegative integer, increases by at least 1 on every flip, and cannot exceed W. Thus our process runs for at most W iterations, and when it terminates, we must have a stable configuration. Moreover, in each iteration we can identify an unsatisfied node using a number of arithmetic operations that is polynomial in n; thus a running-time bound that is polynomial in n and W follows as well.

So we see that, in the end, the existence proof for stable configurations was really about local search. We first set up an objective function Φ that we sought to maximize. Configurations were the possible solutions to this maximization problem, and we defined what it meant for two configurations S and S' to be neighbors: S' should be obtainable from S by flipping a single state. We then studied the behavior of a simple iterative improvement algorithm for local search (the upside-down form of gradient descent, since we have a maximization problem); and we discovered the following.

(12.4) *Any local maximum in the State-Flipping Algorithm to maximize Φ is a stable configuration.*

It's worth noting that while our algorithm proves the existence of a stable configuration, the running time leaves something to be desired when the absolute weights are large. Specifically, and analogously to what we saw in the Subset Sum Problem and in our first algorithm for maximum flow, the algorithm we obtain here is polynomial only in the actual magnitude of the weights, not in the size of their binary representation. For very large weights, this can lead to running times that are quite infeasible.

However, no simple way around this situation is currently known. It turns out to be an open question to find an algorithm that constructs stable states in time polynomial in n and $\log W$ (rather than n and W), or in a number of primitive arithmetic operations that is polynomial in n alone, independent of the value of W.

12.4 Maximum-Cut Approximation via Local Search

We now discuss a case where a local search algorithm can be used to provide a provable approximation guarantee for an optimization problem. We will do this by analyzing the structure of the local optima, and bounding the quality of these locally optimal solutions relative to the global optimum. The problem we consider is the Maximum-Cut Problem, which is closely related to the problem of finding stable configurations for Hopfield networks that we saw in the previous section.

The Problem

In the *Maximum-Cut Problem*, we are given an undirected graph $G = (V, E)$, with a positive integer weight w_e on each edge e. For a partition (A, B) of the vertex set, we use $w(A, B)$ to denote the total weight of edges with one end in A and the other in B:

$$w(A, B) = \sum_{\substack{e=(u,v) \\ u \in A,\, v \in B}} w_e.$$

The goal is to find a partition (A, B) of the vertex set so that $w(A, B)$ is maximized. Maximum Cut is NP-hard, in the sense that, given a weighted graph G and a bound β, it is NP-complete to decide whether there is a partition (A, B) of the vertices of G with $w(A, B) \geq \beta$. At the same time, of course, Maximum Cut resembles the polynomially solvable Minimum s-t Cut Problem for flow networks; the crux of its intractability comes from the fact that we are seeking to maximize the edge weight across the cut, rather than minimize it.

Although the problem of finding a stable configuration of a Hopfield network was not an optimization problem per se, we can see that Maximum Cut is closely related to it. In the language of Hopfield networks, Maximum Cut is an instance in which all edge weights are positive (rather than negative), and configurations of nodes states S correspond naturally to partitions (A, B): Nodes have state -1 if and only if they are in the set A, and state $+1$ if and only if they are in the set B. The goal is to assign states so that as much weight as possible is on *good edges*—those whose endpoints have opposite states. Phrased this way, Maximum Cut seeks to maximize precisely the quantity $\Phi(S)$ that we used in the proof of (12.3), in the case when all edge weights are positive.

Designing the Algorithm

The State-Flipping Algorithm used for Hopfield networks provides a local search algorithm to approximate the Maximum Cut objective function $\Phi(S) =$

$w(A, B)$. In terms of partitions, it says the following: If there exists a node u such that the total weight of edges from u to nodes in its own side of the partition exceeds the total weight of edges from u to nodes on the other side of the partition, then u itself should be moved to the other side of the partition.

We'll call this the "single-flip" neighborhood on partitions: Partitions (A, B) and (A', B') are neighboring solutions if (A', B') can be obtained from (A, B) by moving a single node from one side of the partition to the other. Let's ask two basic questions.

- Can we say anything concrete about the quality of the local optima under the single-flip neighborhood?
- Since the single-flip neighborhood is about as simple as one could imagine, what other neighborhoods might yield stronger local search algorithms for Maximum Cut?

We address the first of these questions here, and we take up the second one in the next section.

Analyzing the Algorithm

The following result addresses the first question, showing that local optima under the single-flip neighborhood provide solutions achieving a guaranteed approximation bound.

(12.5) *Let (A, B) be a partition that is a local optimum for Maximum Cut under the single-flip neighborhood. Let (A^*, B^*) be a globally optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.*

Proof. Let $W = \sum_e w_e$. We also extend our notation a little: for two nodes u and v, we use w_{uv} to denote w_e if there is an edge e joining u and v, and 0 otherwise.

For any node $u \in A$, we must have

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv},$$

since otherwise u should be moved to the other side of the partition, and (A, B) would not be locally optimal. Suppose we add up these inequalities for all $u \in A$; any edge that has both ends in A will appear on the left-hand side of exactly two of these inequalities, while any edge that has one end in A and one end in B will appear on the right-hand side of exactly one of these inequalities. Thus, we have

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B). \tag{12.1}$$

We can apply the same reasoning to the set B, obtaining

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B). \tag{12.2}$$

If we add together inequalities (12.1) and (12.2), and divide by 2, we get

$$\sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B). \tag{12.3}$$

The left-hand side of inequality (12.3) accounts for all edge weight that does not cross from A to B; so if we add $w(A, B)$ to both sides of (12.3), the left-hand side becomes equal to W. The right-hand side becomes $2w(A, B)$, so we have $W \leq 2w(A, B)$, or $w(A, B) \geq \frac{1}{2}W$.

Since the globally optimal partition (A^*, B^*) clearly satisfies $w(A^*, B^*) \leq W$, we have $w(A, B) \geq \frac{1}{2}w(A^*, B^*)$. ■

Notice that we never really thought much about the optimal partition (A^*, B^*) in the proof of (12.5); we really showed the stronger statement that, in any locally optimal solution under the single-flip neighborhood, at least half the total edge weight in the graph crosses the partition.

Statement (12.5) proves that a local optimum is a 2-approximation to the maximum cut. This suggests that the local optimization may be a good algorithm for approximately maximizing the cut value. However, there is one more issue that we need to consider: the running time. As we saw at the end of Section 12.3, the Single-Flip Algorithm is only pseudo-polynomial, and it is an open problem whether a local optimum can be found in polynomial time. However, in this case we can do almost as well, simply by stopping the algorithm when there are no "big enough" improvements.

Let (A, B) be a partition with weight $w(A, B)$. For a fixed $\epsilon > 0$, let us say that a single node flip is a *big-improvement-flip* if it improves the cut value by at least $\frac{2\epsilon}{n} w(A, B)$ where $n = |V|$. Now consider a version of the Single-Flip Algorithm when we only accept big-improvement-flips and terminate once no such flip exists, even if the current partition is not a local optimum. We claim that this will lead to almost as good an approximation and will run in polynomial time. First we can extend the previous proof to show that the resulting cut is almost as good. We simply have to add the term $\frac{2\epsilon}{n} w(A, B)$ to each inequality, as all we know is that there are no big-improvement-flips.

(12.6) *Let (A, B) be a partition such that no big-improvement-flip is possible. Let (A^*, B^*) be a globally optimal partition. Then $(2 + \epsilon)w(A, B) \geq w(A^*, B^*)$.*

Next we consider the running time.

(12.7) *The version of the Single-Flip Algorithm that only accepts big-improvement-flips terminates after at most $O(\epsilon^{-1} n \log W)$ flips, assuming the weights are integral, and $W = \sum_e w_e$.*

Proof. Each flip improves the objective function by at least a factor of $(1 + \epsilon/n)$. Since $(1 + 1/x)^x \geq 2$ for any $x \geq 1$, we see that $(1 + \epsilon/n)^{n/\epsilon} \geq 2$, and so the objective function increases by a factor of at least 2 every n/ϵ flips. The weight cannot exceed W, and hence it can only be doubled at most $\log W$ times. ■

12.5 Choosing a Neighbor Relation

We began the chapter by saying that a local search algorithm is really based on two fundamental ingredients: the choice of the neighbor relation, and the rule for choosing a neighboring solution at each step. In Section 12.2 we spent time thinking about the second of these: both the Metropolis Algorithm and simulated annealing took the neighbor relation as given and modified the way in which a neighboring solution should be chosen.

What are some of the issues that should go into our choice of the neighbor relation? This can turn out to be quite subtle, though at a high level the trade-off is a basic one.

(i) The neighborhood of a solution should be rich enough that we do not tend to get stuck in bad local optima; but

(ii) the neighborhood of a solution should not be too large, since we want to be able to efficiently search the set of neighbors for possible local moves.

If the first of these points were the only concern, then it would seem that we should simply make all solutions neighbors of one another—after all, then there would be no local optima, and the global optimum would always be just one step away! The second point exposes the (obvious) problem with doing this: If the neighborhood of the current solution consists of every possible solution, then the local search paradigm gives us no leverage whatsoever; it reduces simply to brute-force search of this neighborhood.

Actually, we've already encountered one case in which choosing the right neighbor relation had a profound effect on the tractability of a problem, though we did not explicitly take note of this at the time: This was in the Bipartite Matching Problem. Probably the simplest neighbor relation on matchings would be the following: M' is a neighbor of M if M' can be obtained by the insertion or deletion of a single edge in M. Under this definition, we get "landscapes" that are quite jagged, quite like the Vertex Cover examples we

saw earlier; and we can get locally optimal matchings under this definition that have only half the size of the maximum matching.

But suppose we try defining a more complicated (indeed, asymmetric) neighbor relation: We say that M' is a neighbor of M if, when we set up the corresponding flow network, M' can be obtained from M by a single augmenting path. What can we say about a matching M if it is a local maximum under this neighbor relation? In this case, there is no augmenting path, and so M must in fact be a (globally) maximum matching. In other words, with this neighbor relation, the only local maxima are global maxima, and so direct gradient ascent will produce a maximum matching. If we reflect on what the Ford-Fulkerson algorithm is doing in our reduction from Bipartite Matching to Maximum Flow, this makes sense: the size of the matching strictly increases in each step, and we never need to "back out" of a local maximum. Thus, by choosing the neighbor relation very carefully, we've turned a jagged optimization landscape into a simple, tractable funnel.

Of course, we do not expect that things will always work out this well. For example, since Vertex Cover is NP-complete, it would be surprising if it allowed for a neighbor relation that simultaneously produced "well-behaved" landscapes and neighborhoods that could be searched efficiently. We now look at several possible neighbor relations in the context of the Maximum Cut Problem, which we considered in the previous section. The contrasts among these neighbor relations will be characteristic of issues that arise in the general topic of local search algorithms for computationally hard graph-partitioning problems.

Local Search Algorithms for Graph Partitioning

In Section 12.4, we considered a state-flipping algorithm for the Maximum-Cut Problem, and we showed that the locally optimal solutions provide a 2-approximation. We now consider neighbor relations that produce larger neighborhoods than the single-flip rule, and consequently attempt to reduce the prevalence of local optima. Perhaps the most natural generalization is the *k-flip neighborhood*, for $k \geq 1$: we say that partitions (A, B) and (A', B') are neighbors under the k-flip rule if (A', B') can be obtained from (A, B) by moving at most k nodes from one side of the partition to the other.

Now, clearly if (A, B) and (A', B') are neighbors under the k-flip rule, then they are also neighbors under the k'-flip rule for every $k' > k$. Thus, if (A, B) is a local optimum under the k'-flip rule, it is also a local optimum under the k-flip rule for every $k < k'$. But reducing the set of local optima by raising the value of k comes at a steep computational price: to examine the set of neighbors of (A, B) under the k-flip rule, we must consider all $\Theta(n^k)$ ways of moving up to

k nodes to the opposite side of the partition. This becomes prohibitive even for small values of k.

Kernighan and Lin (1970) proposed an alternate method for generating neighboring solutions; it is computationally much more efficient, but still allows large-scale transformations of solutions in a single step. Their method, which we'll call the K-L heuristic, defines the neighbors of a partition (A, B) according the following n-phase procedure.

- In phase 1, we choose a single node to flip, in such a way that the value of the resulting solution is as large as possible. We perform this flip even if the value of the solution decreases relative to $w(A, B)$. We *mark* the node that has been flipped and let (A_1, B_1) denote the resulting solution.
- At the start of phase k, for $k > 1$, we have a partition (A_{k-1}, B_{k-1}); and $k - 1$ of the nodes are marked. We choose a single unmarked node to flip, in such a way that the value of the resulting solution is as large as possible. (Again, we do this even if the value of the solution decreases as a result.) We mark the node we flip and let (A_k, B_k) denote the resulting solution.
- After n phases, each node is marked, indicating that it has been flipped precisely once. Consequently, the final partition (A_n, B_n) is actually the mirror image of the original partition (A, B): We have $A_n = B$ and $B_n = A$.
- Finally, the K-L heuristic defines the $n - 1$ partitions $(A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$ to be the neighbors of (A, B). Thus (A, B) is a local optimum under the K-L heuristic if and only if $w(A, B) \geq w(A_i, B_i)$ for $1 \leq i \leq n - 1$.

So we see that the K-L heuristic tries a very long sequence of flips, even while it appears to be making things worse, in the hope that some partition (A_i, B_i) generated along the way will turn out better than (A, B). But even though it generates neighbors very different from (A, B), it only performs n flips in total, and each takes only $O(n)$ time to perform. Thus it is computationally much more reasonable than the k-flip rule for larger values of k. Moreover, the K-L heuristic has turned out to be very powerful in practice, despite the fact that rigorous analysis of its properties has remained largely an open problem.

* 12.6 Classification via Local Search

We now consider a more complex application of local search to the design of approximation algorithms, related to the Image Segmentation Problem that we considered as an application of network flow in Section 7.10. The more complex version of Image Segmentation that we focus on here will serve as an example where, in order to obtain good performance from a local search algorithm, one needs to use a rather complex neighborhood structure on the

set of solutions. We will find that the natural "state-flipping" neighborhood that we saw in earlier sections can result in very bad local optima. To obtain good performance, we will instead use an exponentially large neighborhood. One problem with such a large neighborhood is that we can no longer afford to search though all neighbors of the current solution one by one for an improving solution. Rather, we will need a more sophisticated algorithm to find an improving neighbor whenever one exists.

The Problem

Recall the basic Image Segmentation Problem that we considered as an application of network flow in Section 7.10. There we formulated the problem of segmenting an image as a *labeling* problem; the goal was to label (i.e., classify) each pixel as belonging to the foreground or the background of the image. At the time, it was clear that this was a very simple formulation of the problem, and it would be nice to handle more complex labeling tasks—for example, to segment the regions of an image based on their distance from the camera. Thus we now consider a labeling problem with more than two labels. In the process, we will end up with a framework for classification that applies more broadly than just to the case of pixels in an image.

In setting up the two-label foreground/background segmentation problem, we ultimately arrived at the following formulation. We were given a graph $G = (V, E)$ where V corresponded to the pixels of the image, and the goal was to classify each node in V as belonging to one of two possible classes: foreground or background. Edges represented pairs of nodes likely to belong to the same class (e.g., because they were next to each other), and for each edge (i, j) we were given a separation penalty $p_{ij} \geq 0$ for placing i and j in different classes. In addition, we had information about the likelihood of whether a node or pixel was more likely to belong to the foreground or the background. These likelihoods translated into penalties for assigning a node to the class where it was less likely to belong. Then the problem was to find a labeling of the nodes that minimized the total separation and assignment penalties. We showed that this minimization problem could be solved via a minimum-cut computation. For the rest of this section, we will refer to the problem we defined there as *Two-Label Image Segmentation*.

Here we will formulate the analogous classification/labeling problem with more than two classes or labels. This problem will turn out to be NP-hard, and we will develop a local search algorithm where the local optima are 2-approximations for the best labeling. The general labeling problem, which we will consider in this section, is formulated as follows. We are given a graph $G = (V, E)$ and a set L of k labels. The goal is to label each node in V with one of the labels in L so as to minimize a certain penalty. There are two competing

forces that will guide the choice of the best labeling. For each edge $(i, j) \in E$, we have a *separation penalty* $p_{ij} \geq 0$ for labeling the two nodes i and j with different labels. In addition, nodes are more likely to have certain labels than others. This is expressed through an *assignment penalty*. For each node $i \in V$ and each label $a \in L$, we have a nonnegative penalty $c_i(a) \geq 0$ for assigning label a to node i. (These penalties play the role of the likelihoods from the Two-Label Image Segmentation Problem, except that here we view them as costs to be minimized.) The *Labeling Problem* is to find a labeling $f : V \to L$ that minimizes the total penalty:

$$\Phi(f) = \sum_{i \in V} c_i(f(i)) + \sum_{(i,j) \in E: f(i) \neq f(j)} p_{ij}.$$

Observe that the Labeling Problem with only two labels is precisely the Image Segmentation Problem from Section 7.10. For three labels, the Labeling Problem is already NP-hard, though we will not prove this here.

Our goal is to develop a local search algorithm for this problem, in which local optima are good approximations to the optimal solution. This will also serve as an illustration of the importance of choosing good neighborhoods for defining the local search algorithm. There are many possible choices for neighbor relations, and we'll see that some work a lot better than others. In particular, a fairly complex definition of the neighborhoods will be used to obtain the approximation guarantee.

Designing the Algorithm

A First Attempt: The Single-Flip Rule The simplest and perhaps most natural choice for neighbor relation is the single-flip rule from the State-Flipping Algorithm for the Maximum-Cut Problem: Two labelings are neighbors if we can obtain one from the other by relabeling a single node. Unfortunately, this neighborhood can lead to quite poor local optima for our problem even when there are only two labels.

This may be initially surprising, since the rule worked quite well for the Maximum-Cut Problem. However, our problem is related to the Minimum-Cut Problem. In fact, Minimum *s*-*t* Cut corresponds to a special case when there are only two labels, and s and t are the only nodes with assignment penalties. It is not hard to see that this State-Flipping Algorithm is not a good approximation algorithm for the Minimum-Cut Problem. See Figure 12.5, which indicates how the edges incident to s may form the global optimum, while the edges incident to t can form a local optimum that is much worse.

A Closer Attempt: Considering Two Labels at a Time Here we will develop a local search algorithm in which the neighborhoods are much more elaborate. One interesting feature of our algorithm is that it allows each solution to have

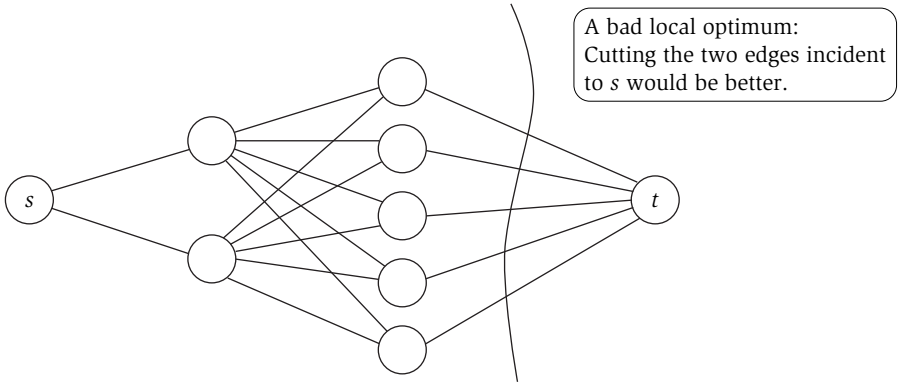

Figure 12.5 An instance of the Minimum *s-t* Cut Problem, where all edges have capacity 1.

exponentially many neighbors. This appears to be contrary to the general rule that "the neighborhood of a solution should not be too large," as stated in Section 12.5. However, we will be working with neighborhoods in a more subtle way here. Keeping the size of the neighborhood small is good if the plan is to search for an improving local step by brute force; here, however, we will use a polynomial-time minimum-cut computation to determine whether any of a solution's exponentially many neighbors represent an improvement.

The idea of the local search is to use our polynomial-time algorithm for Two-Label Image Segmentation to find improving local steps. First let's consider a basic implementation of this idea that does not always give a good approximation guarantee. For a labeling f, we pick two labels $a, b \in L$ and restrict attention to the nodes that have labels a or b in labeling f. In a single local step, we will allow any subset of these nodes to flip labels from a to b, or from b to a. More formally, two labelings f and f' are neighbors if there are two labels $a, b \in L$ such that for all other labels $c \notin \{a, b\}$ and all nodes $i \in V$, we have $f(i) = c$ if and only if $f'(i) = c$. Note that a state f can have exponentially many neighbors, as an arbitrary subset of the nodes labeled a and b can flip their label. However, we have the following.

(12.8) *If a labeling f is not locally optimal for the neighborhood above, then a neighbor with smaller penalty can be found via k^2 minimum-cut computations.*

Proof. There are fewer than k^2 pairs of distinct labels, so we can try each pair separately. Given a pair of labels $a, b \in L$, consider the problem of finding an improved labeling via swapping labels of nodes between labels a and b. This is exactly the Segmentation Problem for two labels on the subgraph of nodes that f labels a or b. We use the algorithm developed for Two-Label Image Segmentation to find the best such relabeling. ■

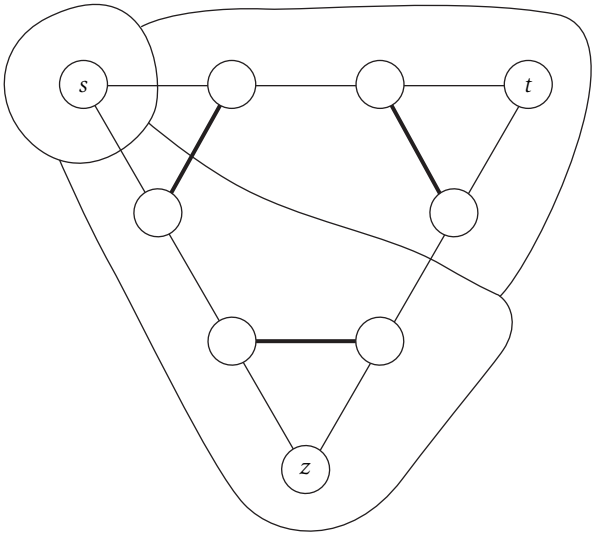

Figure 12.6 A bad local optimum for the local search algorithm that considers only two labels at a time.

This neighborhood is much better than the single-flip neighborhood we considered first. For example, it solves the case of two labels optimally. However, even with this improved neighborhood, local optima can still be bad, as shown in Figure 12.6. In this example, there are three nodes s, t, and z that are each required to keep their initial labels. Each other node lies on one of the sides of the triangle; it has to get one of the two labels associated with the nodes at the ends of this side. These requirements can be expressed simply by giving each node a very large assignment penalty for the labels that we are not allowing. We define the edge separation penalties as follows: The light edges in the figure have penalty 1, while the heavy edges have a large separation penalty of M. Now observe that the labeling in the figure has penalty $M + 3$ but is locally optimal. The (globally) optimal penalty is only 3 and is obtained from the labeling in the figure by relabeling both nodes next to s.

A Local Search Neighborhood That Works Next we define a different neighborhood that leads to a good approximation algorithm. The local optimum in Figure 12.6 may be suggestive of what would be a good neighborhood: We need to be able to relabel nodes of different labels in a single step. The key is to find a neighbor relation rich enough to have this property, yet one that still allows us to find an improving local step in polynomial time.

Consider a labeling f. As part of a local step in our new algorithm, we will want to do the following. We pick one label $a \in L$ and restrict attention to the

nodes that do *not* have label a in labeling f. As a single local step, we will allow any subset of these nodes to change their labels to a. More formally, for two labelings f and f', we say that f' is a neighbor of f if there is a label $a \in L$ such that, for all nodes $i \in V$, either $f'(i) = f(i)$ or $f'(i) = a$. Note that this neighbor relation is not symmetric; that is, we cannot get f back from f' via a single step. We will now show that for any labeling f we can find its best neighbor via k minimum-cut computations, and further, a local optimum for this neighborhood is a 2-approximation for the minimum penalty labeling.

Finding a Good Neighbor To find the best neighbor, we will try each label a separately. Consider a label a. We claim that the best relabeling in which nodes may change their labels to a can be found via a minimum-cut computation. The construction of the minimum-cut graph $G' = (V', E')$ is analogous to the minimum-cut computation developed for Two-Label Image Segmentation. There we introduced a source s and a sink t to represent the two labels. Here we will also introduce a source and a sink, where the source s will represent label a, while the sink t will effectively represent the alternate option nodes have—namely, to keep their original labels. The idea will be to find the minimum cut in G' and relabel all nodes on the s-side of the cut to label a, while letting all nodes on the t-side keep their original labels.

For each node of G, we will have a corresponding node in the new set V' and will add edges (i, t) and (s, i) to E', as was done in Figure 7.18 from Chapter 7 for the case of two labels. The edge (i, t) will have capacity $c_i(a)$, as cutting the edge (i, t) places node i on the source side and hence corresponds to labeling node i with label a. The edge (i, s) will have capacity $c_i(f(i))$, if $f(i) \neq a$, and a very large number M (or $+\infty$) if $f(i) = a$. Cutting edge (i, t) places node i on the sink side and hence corresponds to node i retaining its original label $f(i) \neq a$. The large capacity of M prevents nodes i with $f(i) = a$ from being placed on the sink side.

Node e can always be placed so that at most one incident edge is cut.

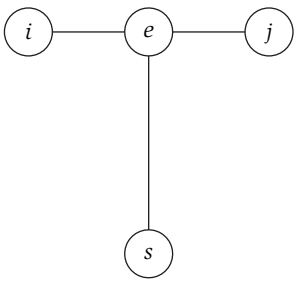

Figure 12.7 The construction for edge $e = (i, j)$ with $a \neq f(i) \neq f(j) \neq a$.

In the construction for the two-label problem, we added edges between the nodes of V and used the separation penalties as capacities. This works well for nodes that are separated by the cut, or nodes on the source side that are both labeled a. However, if both i and j are on the sink side of the cut, then the edge connecting them is not cut, yet i and j are separated if $f(i) \neq f(j)$. We deal with this difficulty by enhancing the construction of G' as follows. For an edge (i, j), if $f(i) = f(j)$ or one of i or j is labeled a, then we add an edge (i, j) to E' with capacity p_{ij}. For the edges $e = (i, j)$ where $f(i) \neq f(j)$ and neither has label a, we'll have to do something different to correctly encode via the graph G' that i and j remain separated even if they are both on the sink side. For each such edge e, we add an extra node e to V' corresponding to edge e, and add the edges (i, e), (e, j), and (e, s) all with capacity p_{ij}. See Figure 12.7 for these edges.

(12.9) *Given a labeling f and a label a, the minimum cut in the graph $G' = (V', E')$ corresponds to the minimum-penalty neighbor of labeling f obtained by relabeling a subset of nodes to label a. As a result, the minimum-penalty neighbor of f can be found via k minimum-cut computations, one for each label in L.*

Proof. Let (A, B) be an s-t cut in G'. The large value of M ensures that a minimum-capacity cut will not cut any of these high-capacity edges. Now consider a node e in G' corresponding to an edge $e = (i, j) \in E$. The node $e \in V'$ has three adjacent edges, each with capacity p_{ij}. Given any partition of the other nodes, we can place e so that at most one of these three edges is cut. We'll call a cut *good* if no edge of capacity M is cut and, for all the nodes corresponding to edges in E, at most one of the adjacent edges is cut. So far we have argued that all minimum-capacity cuts are good.

Good s-t cuts in G' are in one-to-one correspondence with relabelings of f obtained by changing the label of a subset of nodes to a. Consider the capacity of a good cut. The edges (s, i) and (i, t) contribute exactly the assignment penalty to the capacity of the cut. The edges (i, j) directly connecting nodes in V contribute exactly the separation penalty of the nodes in the corresponding labeling: p_{ij} if they are separated, and 0 otherwise. Finally, consider an edge $e = (i, j)$ with a corresponding node $e \in V'$. If i and j are both on the source side, none of the three edges adjacent to e are cut, and in all other cases exactly one of these edges is cut. So again, the three edges adjacent to e contribute to the cut exactly the separation penalty between i and j in the corresponding labeling. As a result, the capacity of a good cut is exactly the same as the penalty of the corresponding labeling, and so the minimum-capacity cut corresponds to the best relabeling of f. ■

Analyzing the Algorithm

Finally, we need to consider the quality of the local optima under this definition of the neighbor relation. Recall that in our previous two attempts at defining neighborhoods, we found that they can both lead to bad local optima. Now, by contrast, we'll show that any local optimum under our new neighbor relation is a 2-approximation to the minimum possible penalty.

To begin the analysis, consider an optimal labeling f^*, and for a label $a \in L$ let $V_a^* = \{i : f^*(i) = a\}$ be the set of nodes labeled by a in f^*. Consider a locally optimal labeling f. We obtain a neighbor f_a of labeling f by starting with f and relabeling all nodes in V_a^* to a. The labeling f is locally optimal, and hence this neighbor f_a has no smaller penalty: $\Phi(f_a) \geq \Phi(f)$. Now consider the difference $\Phi(f_a) - \Phi(f)$, which we know is nonnegative. What quantities contribute to

this difference? The only possible change in the assignment penalties could come from nodes in V_a^*: for each $i \in V_a^*$, the change is $c_i(f^*(i)) - c_i(f(i))$. The separation penalties differ between the two labelings only in edges (i, j) that have at least one end in V_a^*. The following inequality accounts for these differences.

(12.10) *For a labeling f and its neighbor f_a, we have*

$$\Phi(f_a) - \Phi(f) \leq \sum_{i \in V_a^*} \left[c_i(f^*(i)) - c_i(f(i))\right] + \sum_{(i,j) \text{ leaving } V_a^*} p_{ij} - \sum_{\substack{(i,j) \text{ in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij}.$$

Proof. The change in the assignment penalties is exactly $\sum_{i \in V_a^*} c_i(f^*(i)) - c_i(f(i))$. The separation penalty for an edge (i, j) can differ between the two labelings only if edge (i, j) has at least one end in V_a^*. The total separation penalty of labeling f for such edges is exactly

$$\sum_{\substack{(i,j) \text{ in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij},$$

while the labeling f_a has a separation penalty of at most

$$\sum_{(i,j) \text{ leaving } V_a^*} p_{ij}$$

for these edges. (Note that this latter expression is only an upper bound, since an edge (i, j) leaving V_a^* that has its other end in a does not contribute to the separation penalty of f_a.) ■

Now we are ready to prove our main claim.

(12.11) *For any locally optimal labeling f, and any other labeling f^*, we have $\Phi(f) \leq 2\Phi(f^*)$.*

Proof. Let f_a be the neighbor of f defined previously by relabeling nodes to label a. The labeling f is locally optimal, so we have $\Phi(f_a) - \Phi(f) \geq 0$ for all $a \in L$. We use (12.10) to bound $\Phi(f_a) - \Phi(f)$ and then add the resulting inequalities for all labels to obtain the following:

$$\begin{aligned} 0 &\leq \sum_{a \in L} (\Phi(f_a) - \Phi(f)) \\ &\leq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i(f^*(i)) - c_i(f(i)) + \sum_{(i,j) \text{ leaving } V_a^*} p_{ij} - \sum_{\substack{(i,j) \text{ in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij} \right]. \end{aligned}$$

We will rearrange the inequality by grouping the positive terms on the left-hand side and the negative terms on the right-hand side. On the left-hand side, we get $c_i(f^*(i))$ for all nodes i, which is exactly the assignment penalty of f^*. In addition, we get the term p_{ij} twice for each of the edges separated by f^* (once for each of the two labels $f^*(i)$ and $f^*(j)$).

On the right-hand side, we get $c_i(f(i))$ for each node i, which is exactly the assignment penalty of f. In addition, we get the terms p_{ij} for edges separated by f. We get each such separation penalty at least once, and possibly twice if it is also separated by f^*.

In summary, we get the following.

$$2\Phi(f^*) \geq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i(f^*(i)) + \sum_{(i,j) \text{ leaving } V_a^*} p_{ij} \right]$$

$$\geq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i(f(i)) + \sum_{\substack{(i,j) \text{ in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij} \right] \geq \Phi(f),$$

proving the claimed bound. ■

We proved that all local optima are good approximations to the labeling with minimum penalty. There is one more issue to consider: How fast does the algorithm find a local optimum? Recall that in the case of the Maximum-Cut Problem, we had to resort to a variant of the algorithm that accepts only big improvements, as repeated local improvements may not run in polynomial time. The same is also true here. Let $\epsilon > 0$ be a constant. For a given labeling f, we will consider a neighboring labeling f' a *significant improvement* if $\Phi(f') \leq (1 - \epsilon/3k)\Phi(f)$. To make sure the algorithm runs in polynomial time, we should only accept significant improvements, and terminate when no significant improvements are possible. After at most $\epsilon^{-1}k$ significant improvements, the penalty decreases by a constant factor; hence the algorithm will terminate in polynomial time. It is not hard to adapt the proof of (12.11) to establish the following.

(12.12) *For any fixed $\epsilon > 0$, the version of the local search algorithm that only accepts significant improvements terminates in polynomial time and results in a labeling f such that $\Phi(f) \leq (2 + \epsilon)\Phi(f^*)$ for any other labeling f^*.*

12.7 Best-Response Dynamics and Nash Equilibria

Thus far we have been considering local search as a technique for solving optimization problems with a single objective—in other words, applying local operations to a candidate solution so as to minimize its total cost. There are many settings, however, where a potentially large number of agents, each with its own goals and objectives, collectively interact so as to produce a solution to some problem. A solution that is produced under these circumstances often reflects the "tug-of-war" that led to it, with each agent trying to pull the solution in a direction that is favorable to it. We will see that these interactions can be viewed as a kind of local search procedure; analogues of local minima have a natural meaning as well, but having multiple agents and multiple objectives introduces new challenges.

The field of game theory provides a natural framework in which to talk about what happens in such situations, when a collection of agents interacts strategically—in other words, with each trying to optimize an individual objective function. To illustrate these issues, we consider a concrete application, motivated by the problem of routing in networks; along the way, we will introduce some notions that occupy central positions in the area of game theory more generally.

The Problem

In a network like the Internet, one frequently encounters situations in which a number of nodes all want to establish a connection to a single *source node* s. For example, the source s may be generating some kind of data stream that all the given nodes want to receive, as in a style of one-to-many network communication known as *multicast*. We will model this situation by representing the underlying network as a directed graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge. There is a designated source node $s \in V$ and a collection of k agents located at distinct *terminal nodes* $t_1, t_2, \ldots, t_k \in V$. For simplicity, we will not make a distinction between the agents and the nodes at which they reside; in other words, we will think of the agents as being $t_1, t_2, \ldots, t_k$. Each agent t_j wants to construct a path P_j from s to t_j using as little total cost as possible.

Now, if there were no interaction among the agents, this would consist of k separate shortest-path problems: Each agent t_j would find an s-t_j path for which the total cost of all edges is minimized, and use this as its path P_j. What makes this problem interesting is the prospect of agents being able to *share* the costs of edges. Suppose that after all the agents have chosen their paths, agent t_j only needs to pay its "fair share" of the cost of each edge e on its path; that is, rather than paying c_e for each e on P_i, it pays c_e divided by the number of

agents whose paths contain e. In this way, there is an incentive for the agents to choose paths that overlap, since they can then benefit by splitting the costs of edges. (This sharing model is appropriate for settings in which the presence of multiple agents on an edge does not significantly degrade the quality of transmission due to congestion or increased latency. If latency effects do come into play, then there is a countervailing penalty for sharing; this too leads to interesting algorithmic questions, but we will stick to our current focus for now, in which sharing comes with benefits only.)

Best-Response Dynamics and Nash Equilibria: Definitions and Examples

To see how the option of sharing affects the behavior of the agents, let's begin by considering the pair of very simple examples in Figure 12.8. In example (a), each of the two agents has two options for constructing a path: the middle route through v, and the outer route using a single edge. Suppose that each agent starts out with an initial path but is continually evaluating the current situation to decide whether it's possible to switch to a better path.

In example (a), suppose the two agents start out using their outer paths. Then t_1 sees no advantage in switching paths (since $4 < 5 + 1$), but t_2 does (since $8 > 5 + 1$), and so t_2 updates its path by moving to the middle. Once this happens, things have changed from the perspective of t_1: There is suddenly an advantage for t_1 in switching as well, since it now gets to share the cost of the middle path, and hence its cost to use the middle path becomes $2.5 + 1 < 4$. Thus it will switch to the middle path. Once we are in a situation where both

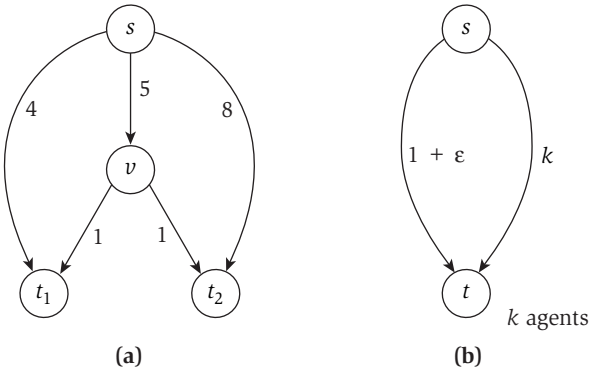

Figure 12.8 (a) It is in the two agents' interest to share the middle path. (b) It would be better for all the agents to share the edge on the left. But if all k agents start on the right-hand edge, then no one of them will want to unilaterally move from right to left; in other words, the solution in which all agents share the edge on the right is a bad Nash equilibrium.

sides are using the middle path, neither has an incentive to switch, and so this is a stable solution.

Let's discuss two definitions from the area of game theory that capture what's going on in this simple example. While we will continue to focus on our particular multicast routing problem, these definitions are relevant to any setting in which multiple agents, each with an individual objective, interact to produce a collective solution. As such, we will phrase the definitions in these general terms.

- First of all, in the example, each agent was continually prepared to improve its solution in response to changes made by the other agent(s). We will refer to this process as *best-response dynamics*. In other words, we are interested in the dynamic behavior of a process in which each agent updates based on its best response to the current situation.
- Second, we are particularly interested in stable solutions, where the best response of each agent is to stay put. We will refer to such a solution, from which no agent has an incentive to deviate, as a *Nash equilibrium*. (This is named after the mathematician John Nash, who won the Nobel Prize in economics for his pioneering work on this concept.) Hence, in example (a), the solution in which both agents use the middle path is a Nash equilibrium. Note that the Nash equilibria are precisely the solutions at which best-response dynamics terminate.

The example in Figure 12.8(b) illustrates the possibility of multiple Nash equilibria. In this example, there are k agents that all reside at a common node t (that is, $t_1 = t_2 = \cdots = t_k = t$), and there are two parallel edges from s to t with different costs. The solution in which all agents use the left-hand edge is a Nash equilibrium in which all agents pay $(1+\varepsilon)/k$. The solution in which all agents use the right-hand edge is also a Nash equilibrium, though here the agents each pay $k/k = 1$. The fact that this latter solution is a Nash equilibrium exposes an important point about best-response dynamics. If the agents could somehow synchronously agree to move from the right-hand edge to the left-hand one, they'd all be better off. But under best-response dynamics, each agent is only evaluating the consequences of a unilateral move by itself. In effect, an agent isn't able to make any assumptions about future actions of other agents—in an Internet setting, it may not even know anything about these other agents or their current solutions—and so it is only willing to perform updates that lead to an immediate improvement for itself.

To quantify the sense in which one of the Nash equilibria in Figure 12.8(b) is better than the other, it is useful to introduce one further definition. We say that a solution is a *social optimum* if it minimizes the total cost to all agents. We can think of such a solution as the one that would be imposed by

a benevolent central authority that viewed all agents as equally important and hence evaluated the quality of a solution by summing the costs they incurred. Note that in both (a) and (b), there is a social optimum that is also a Nash equilibrium, although in (b) there is also a second Nash equilibrium whose cost is much greater.

The Relationship to Local Search

Around here, the connections to local search start to come into focus. A set of agents following best-response dynamics are engaged in some kind of gradient descent process, exploring the "landscape" of possible solutions as they try to minimize their individual costs. The Nash equilibria are the natural analogues of local minima in this process: solutions from which no improving move is possible. And the "local" nature of the search is clear as well, since agents are only updating their solutions when it leads to an immediate improvement.

Having said all this, it's important to think a bit further and notice the crucial ways in which this differs from standard local search. In the beginning of this chapter, it was easy to argue that the gradient descent algorithm for a combinatorial problem must terminate at a local minimum: each update decreased the cost of the solution, and since there were only finitely many possible solutions, the sequence of updates could not go on forever. In other words, the cost function itself provided the progress measure we needed to establish termination.

In best-response dynamics, on the other hand, each agent has its own personal objective function to minimize, and so it's not clear what overall "progress" is being made when, for example, agent t_i decides to update its path from s. There's progress for t_i, of course, since its cost goes down, but this may be offset by an even larger increase in the cost to some other agent. Consider, for example, the network in Figure 12.9. If both agents start on the middle path, then t_1 will in fact have an incentive to move to the outer path; its cost drops from 3.5 to 3, but in the process the cost of t_2 increases from 3.5 to 6. (Once this happens, t_2 will also move to its outer path, and this solution—with both nodes on the outer paths—is the unique Nash equilibrium.)

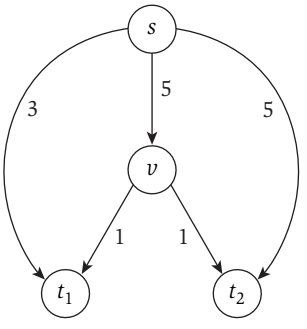

Figure 12.9 A network in which the unique Nash equilibrium differs from the social optimum.

There are examples, in fact, where the cost-increasing effects of best-response dynamics can be much worse than this. Consider the situation in Figure 12.10, where we have k agents that each have the option to take a common outer path of cost $1+\varepsilon$ (for some small number $\varepsilon > 0$), or to take their own alternate path. The alternate path for t_j has cost $1/j$. Now suppose we start with a solution in which all agents are sharing the outer path. Each agent pays $(1+\varepsilon)/k$, and this is the solution that minimizes the total cost to all agents. But running best-response dynamics starting from this solution causes things to unwind rapidly. First t_k switches to its alternate path, since $1/k < (1+\varepsilon)/k$.

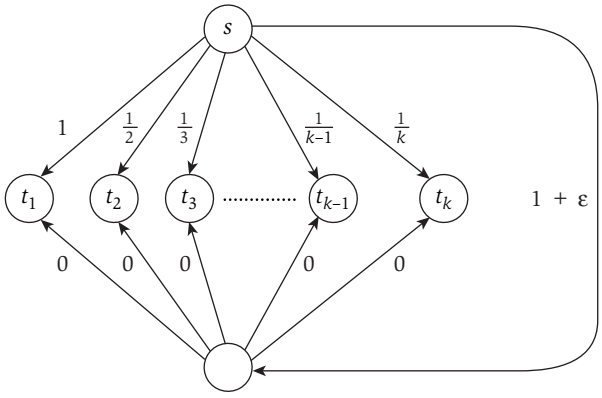

Figure 12.10 A network in which the unique Nash equilibrium costs $H(k) = \Theta(\log k)$ times more than the social optimum.

As a result of this, there are now only $k-1$ agents sharing the outer path, and so t_{k-1} switches to its alternate path, since $1/(k-1) < (1+\varepsilon)/(k-1)$. After this, t_{k-2} switches, then t_{k-3}, and so forth, until all k agents are using the alternate paths directly from s. Things come to a halt here, due to the following fact.

(12.13) *The solution in Figure 12.10, in which each agent uses its direct path from s, is a Nash equilibrium, and moreover it is the unique Nash equilibrium for this instance.*

Proof. To verify that the given solution is a Nash equilibrium, we simply need to check that no agent has an incentive to switch from its current path. But this is clear, since all agents are paying at most 1, and the only other option—the (currently vacant) outer path—has cost $1+\varepsilon$.

Now suppose there were some other Nash equilibrium. In order to be different from the solution we have just been considering, it would have to involve at least one of the agents using the outer path. Let $t_{j_1}, t_{j_2}, \ldots, t_{j_\ell}$ be the agents using the outer path, where $j_1 < j_2 < \cdots < j_\ell$. Then all these agents are paying $(1+\varepsilon)/\ell$. But notice that $j_\ell \geq \ell$, and so agent t_{j_ℓ} has the option to pay only $1/j_\ell \leq 1/\ell$ by using its alternate path directly from s. Hence t_{j_ℓ} has an incentive to deviate from the current solution, and hence this solution cannot be a Nash equilibrium. ■

Figure 12.8(b) already illustrated that there can exist a Nash equilibrium whose total cost is much worse than that of the social optimum, but the examples in Figures 12.9 and 12.10 drive home a further point: The total cost to all agents under even the *most favorable* Nash equilibrium solution can be

worse than the total cost under the social optimum. How much worse? The total cost of the social optimum in this example is $1 + \varepsilon$, while the cost of the unique Nash equilibrium is $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} = \sum_{i=1}^{k} \frac{1}{i}$. We encountered this expression in Chapter 11, where we defined it to be the *harmonic number* $H(k)$ and showed that its asymptotic value is $H(k) = \Theta(\log k)$.

These examples suggest that one can't really view the social optimum as the analogue of the global minimum in a traditional local search procedure. In standard local search, the global minimum is always a stable solution, since no improvement is possible. Here the social optimum can be an unstable solution, since it just requires one agent to have an interest in deviating.

Two Basic Questions

Best-response dynamics can exhibit a variety of different behaviors, and we've just seen a range of examples that illustrate different phenomena. It's useful at this point to step back, assess our current understanding, and ask some basic questions. We group these questions around the following two issues.

- **The existence of a Nash equilibrium.** At this point, we actually don't have a proof that there even *exists* a Nash equilibrium solution in every instance of our multicast routing problem. The most natural candidate for a progress measure, the total cost to all agents, does not necessarily decrease when a single agent updates its path.

  Given this, it's not immediately clear how to argue that the best-response dynamics must terminate. Why couldn't we get into a cycle where agent t_1 improves its solution at the expense of t_2, then t_2 improves its solution at the expense of t_1, and we continue this way forever? Indeed, it's not hard to define other problems in which exactly this can happen and in which Nash equilibria don't exist. So if we want to argue that best-response dynamics leads to a Nash equilibrium in the present case, we need to figure out what's special about our routing problem that causes this to happen.

- **The price of stability.** So far we've mainly considered Nash equilibria in the role of "observers": essentially, we turn the agents loose on the graph from an arbitrary starting point and watch what they do. But if we were viewing this as protocol designers, trying to define a procedure by which agents could construct paths from s, we might want to pursue the following approach. Given a set of agents, located at nodes $t_1, t_2, \ldots, t_k$, we could propose a collection of paths, one for each agent, with two properties.

  (i) The set of paths forms a Nash equilibrium solution; and

  (ii) Subject to (i), the total cost to all agents is as small as possible.

Of course, ideally we'd like just to have the smallest total cost, as this is the social optimum. But if we propose the social optimum and it's not a Nash equilibrium, then it won't be stable: Agents will begin deviating and constructing new paths. Thus properties (i) and (ii) together represent our protocol's attempt to optimize in the face of stability, finding the best solution from which no agent will want to deviate.

We therefore define the *price of stability*, for a given instance of the problem, to be the ratio of the cost of the best Nash equilibrium solution to the cost of the social optimum. This quantity reflects the blow-up in cost that we incur due to the requirement that our solution must be stable in the face of the agents' self-interest.

Note that this pair of questions can be asked for essentially any problem in which self-interested agents produce a collective solution. For our multicast routing problem, we now resolve both these questions. Essentially, we will find that the example in Figure 12.10 captures some of the crucial aspects of the problem in general. We will show that for any instance, best-response dynamics starting from the social optimum leads to a Nash equilibrium whose cost is greater by at most a factor of $H(k) = \Theta(\log k)$.

Finding a Good Nash Equilibrium

We focus first on showing that best-response dynamics in our problem always terminates with a Nash equilibrium. It will turn out that our approach to this question also provides the necessary technique for bounding the price of stability.

The key idea is that we don't need to use the total cost to all agents as the progress measure against which to bound the number of steps of best-response dynamics. Rather, any quantity that strictly decreases on a path update by any agent, and which can only decrease a finite number of times, will work perfectly well. With this in mind, we try to formulate a measure that has this property. The measure will not necessarily have as strong an intuitive meaning as the total cost, but this is fine as long as it does what we need.

We first consider in more detail why just using the total agent cost doesn't work. Suppose, to take a simple example, that agent t_j is currently sharing, with x other agents, a path consisting of the single edge e. (In general, of course, the agents' paths will be longer than this, but single-edge paths are useful to think about for this example.) Now suppose that t_j decides it is in fact cheaper to switch to a path consisting of the single edge f, which no agent is currently using. In order for this to be the case, it must be that $c_f < c_e/(x+1)$. Now, as a result of this switch, the total cost to all agents goes up by c_f: Previously,

$x+1$ agents contributed to the cost c_e, and no one was incurring the cost c_f; but, after the switch, x agents still collectively have to pay the full cost c_e, and t_j is now paying an additional c_f.

In order to view this as progress, we need to redefine what "progress" means. In particular, it would be useful to have a measure that could offset the added cost c_f via some notion that the overall "potential energy" in the system has dropped by $c_e/(x+1)$. This would allow us to view the move by t_j as causing a net decrease, since we have $c_f < c_e/(x+1)$. In order to do this, we could maintain a "potential" on each edge e, with the property that this potential drops by $c_e/(x+1)$ when the number of agents using e decreases from $x+1$ to x. (Correspondingly, it would need to increase by this much when the number of agents using e increased from x to $x+1$.)

Thus, our intuition suggests that we should define the potential so that, if there are x agents on an edge e, then the potential should decrease by c_e/x when the first one stops using e, by $c_e/(x-1)$ when the next one stops using e, by $c_e/(x-2)$ for the next one, and so forth. Setting the potential to be $c_e(1/x + 1/(x-1) + \cdots + 1/2 + 1) = c_e \cdot H(x)$ is a simple way to accomplish this. More concretely, we define the *potential* of a set of paths $P_1, P_2, \ldots, P_k$, denoted $\Phi(P_1, P_2, \ldots, P_k)$, as follows. For each edge e, let x_e denote the number of agents whose paths use the edge e. Then

$$\Phi(P_1, P_2, \ldots, P_k) = \sum_{e \in E} c_e \cdot H(x_e).$$

(We'll define the harmonic number $H(0)$ to be 0, so that the contribution of edges containing no paths is 0.)

The following claim establishes that Φ really works as a progress measure.

(12.14) *Suppose that the current set of paths is $P_1, P_2, \ldots, P_k$, and agent t_j updates its path from P_j to P'_j. Then the new potential $\Phi(P_1, \ldots, P_{j-1}, P'_j, P_{j+1}, \ldots, P_k)$ is strictly less than the old potential $\Phi(P_1, \ldots, P_{j-1}, P_j, P_{j+1}, \ldots, P_k)$.*

Proof. Before t_j switched its path from P_j to P'_j, it was paying $\sum_{e \in P_j} c_e/x_e$, since it was sharing the cost of each edge e with $x_e - 1$ other agents. After the switch, it continues to pay this cost on the edges in the intersection $P_j \cap P'_j$, and it also pays $c_f/(x_f+1)$ on each edge $f \in P'_j - P_j$. Thus the fact that t_j viewed this switch as an improvement means that

$$\sum_{f \in P'_j - P_j} \frac{c_f}{x_f + 1} < \sum_{e \in P_j - P'_j} \frac{c_e}{x_e}.$$

Now let's ask what happens to the potential function Φ. The only edges on which it changes are those in $P_j' - P_j$ and those in $P_j - P_j'$. On the former set, it increases by

$$\sum_{f \in P_j' - P_j} c_f[H(x_f + 1) - H(x_f)] = \sum_{f \in P_j' - P_j} \frac{c_f}{x_f + 1},$$

and on the latter set, it decreases by

$$\sum_{e \in P_j - P_j'} c_e[H(x_e) - H(x_e - 1)] = \sum_{e \in P_j - P_j'} \frac{c_e}{x_e}.$$

So the criterion that t_j used for switching paths is precisely the statement that the total increase is strictly less than the total decrease, and hence the potential Φ decreases as a result of t_j's switch. ■

Now there are only finitely many ways to choose a path for each agent t_j, and (12.14) says that best-response dynamics can never revisit a set of paths $P_1, \ldots, P_k$ once it leaves it due to an improving move by some agent. Thus we have shown the following.

(12.15) *Best-response dynamics always leads to a set of paths that forms a Nash equilibrium solution.*

Bounding the Price of Stability Our potential function Φ also turns out to be very useful in providing a bound on the price of stability. The point is that, although Φ is not equal to the total cost incurred by all agents, it tracks it reasonably closely.

To see this, let $C(P_1, \ldots, P_k)$ denote the total cost to all agents when the selected paths are $P_1, \ldots, P_k$. This quantity is simply the sum of c_e over all edges that appear in the union of these paths, since the cost of each such edge is completely covered by the agents whose paths contain it.

Now the relationship between the cost function C and the potential function Φ is as follows.

(12.16) *For any set of paths $P_1, \ldots, P_k$, we have*

$$C(P_1, \ldots, P_k) \leq \Phi(P_1, \ldots, P_k) \leq H(k) \cdot C(P_1, \ldots, P_k).$$

Proof. Recall our notation in which x_e denotes the number of paths containing edge e. For the purposes of comparing C and Φ, we also define E^+ to be the set of all edges that belong to at least one of the paths $P_1, \ldots, P_k$. Then, by the definition of C, we have $C(P_1, \ldots, P_k) = \sum_{e \in E^+} c_e$.

A simple fact to notice is that $x_e \leq k$ for all e. Now we simply write

$$C(P_1, \ldots, P_k) = \sum_{e \in E^+} c_e \leq \sum_{e \in E^+} c_e H(x_e) = \Phi(P_1, \ldots, P_k)$$

and

$$\Phi(P_1, \ldots, P_k) = \sum_{e \in E^+} c_e H(x_e) \leq \sum_{e \in E^+} c_e H(k) = H(k) \cdot C(P_1, \ldots, P_k). \quad \blacksquare$$

Using this, we can give a bound on the price of stability.

(12.17) *In every instance, there is a Nash equilibrium solution for which the total cost to all agents exceeds that of the social optimum by at most a factor of $H(k)$.*

Proof. To produce the desired Nash equilibrium, we start from a social optimum consisting of paths $P_1^*, \ldots, P_k^*$ and run best-response dynamics. By (12.15), this must terminate at a Nash equilibrium $P_1, \ldots, P_k$.

During this run of best-response dynamics, the total cost to all agents may have been going up, but by (12.14) the potential function was decreasing. Thus we have $\Phi(P_1, \ldots, P_k) \leq \Phi(P_1^*, \ldots, P_k^*)$.

This is basically all we need since, for any set of paths, the quantities C and Φ differ by at most a factor of $H(k)$. Specifically,

$$C(P_1, \ldots, P_k) \leq \Phi(P_1, \ldots, P_k) \leq \Phi(P_1^*, \ldots, P_k^*) \leq H(k) \cdot C(P_1^*, \ldots, P_k^*). \quad \blacksquare$$

Thus we have shown that a Nash equilibrium always exists, and there is always a Nash equilibrium whose total cost is within an $H(k)$ factor of the social optimum. The example in Figure 12.10 shows that it isn't possible to improve on the bound of $H(k)$ in the worst case.

Although this wraps up certain aspects of the problem very neatly, there are a number of questions here for which the answer isn't known. One particularly intriguing question is whether it's possible to construct a Nash equilibrium for this problem in polynomial time. Note that our proof of the existence of a Nash equilibrium argued simply that as best-response dynamics iterated through sets of paths, it could never revisit the same set twice, and hence it could not run forever. But there are exponentially many possible sets of paths, and so this does not give a polynomial-time algorithm. Beyond the question of finding any Nash equilibrium efficiently, there is also the open question of efficiently finding a Nash equilibrium that achieves a bound of $H(k)$ relative to the social optimum, as guaranteed by (12.17).

It's also important to reiterate something that we mentioned earlier: It's not hard to find problems for which best-response dynamics may cycle forever

and for which Nash equilibria do not necessarily exist. We were fortunate here that best-response dynamics could be viewed as iteratively improving a *potential function* that guaranteed our progress toward a Nash equilibrium, but the point is that potential functions like this do not exist for all problems in which agents interact.

Finally, it's interesting to compare what we've been doing here to a problem that we considered earlier in this chapter: finding a stable configuration in a Hopfield network. If you recall the discussion of that earlier problem, we analyzed a process in which each node "flips" between two possible states, seeking to increase the total weight of "good" edges incident to it. This can in fact be viewed as an instance of best-response dynamics for a problem in which each node has an objective function that seeks to maximize this measure of good edge weight. However, showing the convergence of best-response dynamics for the Hopfield network problem was much easier than the challenge we faced here: There it turned out that the state-flipping process was in fact a "disguised" form of local search with an objective function obtained simply by adding together the objective functions of all nodes—in effect, the analogue of the total cost to all agents served as a progress measure. In the present case, it was precisely because this total cost function did not work as a progress measure that we were forced to embark on the more complex analysis described here.

Solved Exercises

Solved Exercise 1

The Center Selection Problem from Chapter 11 is another case in which one can study the performance of local search algorithms.

Here is a simple local search approach to Center Selection (indeed, it's a common strategy for a variety of problems that involve locating facilities). In this problem, we are given a set of sites $S = \{s_1, s_2, \ldots, s_n\}$ in the plane, and we want to choose a set of k centers $C = \{c_1, c_2, \ldots, c_k\}$ whose *covering radius*—the farthest that people in any one site must travel to their nearest center—is as small as possible.

We start by arbitrarily choosing k points in the plane to be the centers $c_1, c_2, \ldots, c_k$. We now alternate the following two steps.

(i) Given the set of k centers $c_1, c_2, \ldots, c_k$, we divide S into k sets: For $i = 1, 2, \ldots, k$, we define S_i to be the set of all the sites for which c_i is the closest center.

(ii) Given this division of S into k sets, construct new centers that will be as "central" as possible relative to them. For each set S_i, we find the smallest

circle in the plane that contains all points in S_i, and define center c_i to be the center of this circle.

If steps (i) and (ii) cause the covering radius to strictly decrease, then we perform another iteration; otherwise the algorithm stops.

The alternation of steps (i) and (ii) is based on the following natural interplay between sites and centers. In step (i) we partition the sites as well as possible given the centers; and then in step (ii) we place the centers as well as possible given the partition of the sites. In addition to its role as a heuristic for placing facilities, this type of two-step interplay is also the basis for local search algorithms in statistics, where (for reasons we won't go into here) it is called the *Expectation Maximization* approach.

(a) Prove that this local search algorithm eventually terminates.

(b) Consider the following statement.

There is an absolute constant $b > 1$ (independent of the particular input instance), so when the local search algorithm terminates, the covering radius of its solution is at most b times the optimal covering radius.

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

Solution To prove part (a), one's first thought is the following: The set of covering radii decreases in each iteration; it can't drop below the optimal covering radius; and so the iterations must terminate. But we have to be a bit careful, since we're dealing with real numbers. What if the covering radii decreased in every iteration, but by less and less, so that the algorithm was able to run arbitrarily long as its covering radii converged to some value from above?

It's not hard to take care of this concern, however. Note that the covering radius at the end of step (ii) in each iteration is completely determined by the current partition of the sites into $S_1, S_2, \ldots, S_k$. There are a finite number of ways to partition the sites into k sets, and if the local search algorithm ran for more than this number of iterations, it would have to produce the same partition in two of these iterations. But then it would have the same covering radius at the end of each of these iterations, and this contradicts the assumption that the covering radius strictly decreases from each iteration to the next.

This proves that the algorithm always terminates. (Note that it only gives an exponential bound on the number of iterations, however, since there are exponentially many ways to partition the sites into k sets.)

To disprove part (b), it would be enough to find a run of the algorithm in which the iterations gets "stuck" in a configuration with a very large covering radius. This is not very hard to do. For any constant $b > 1$, consider a set S

of four points in the plane that form the corners of a tall, narrow rectangle of width w and height $h = 2bw$. For example, we could have the four points be $(0, 0), (0, h), (w, h), (w, 0)$.

Now suppose $k = 2$, and we start the two centers anywhere to the left and right of the rectangle, respectively (say, at $(-1, h/2)$ and $(w + 1, h/2)$). The first iteration proceeds as follows.

- Step (i) will divide S into the two points S_1 on the left side of the rectangle (with x-coordinate 0) and the two points S_2 on the right side of the rectangle (with x-coordinate w).
- Step (ii) will place centers at the midpoints of S_1 and S_2 (i.e., at $(0, h/2)$ and $(w, h/2)$).

We can check that in the next iteration, the partition of S will not change, and so the locations of the centers will not change; the algorithm terminates here at a local minimum.

The covering radius of this solution is $h/2$. But the optimal solution would place centers at the midpoints of the top and bottom sides of the rectangle, for a covering radius of $w/2$. Thus the covering radius of our solution is $h/w = 2b > b$ times that of the optimum.

Exercises

1. Consider the problem of finding a stable state in a Hopfield neural network, in the special case when all edge weights are positive. This corresponds to the Maximum-Cut Problem that we discussed earlier in the chapter: For every edge e in the graph G, the endpoints of G would prefer to have opposite states.

   Now suppose the underlying graph G is connected and bipartite; the nodes can be partitioned into sets X and Y so that each edge has one end in X and the other in Y. Then there is a natural "best" configuration for the Hopfield net, in which all nodes in X have the state $+1$ and all nodes in Y have the state -1. This way, all edges are *good*, in that their ends have opposite states.

   The question is: In this special case, when the best configuration is so clear, will the State-Flipping Algorithm described in the text (as long as there is an unsatisfied node, choose one and flip its state) always find this configuration? Give a proof that it will, or an example of an input instance, a starting configuration, and an execution of the State-Flipping Algorithm that terminates at a configuration in which not all edges are good.

2. Recall that for a problem in which the goal is to maximize some underlying quantity, gradient descent has a natural "upside-down" analogue, in which one repeatedly moves from the current solution to a solution of strictly greater value. Naturally, we could call this a *gradient ascent algorithm*. (Often in the literature you'll also see such methods referred to as *hill-climbing* algorithms.)

   By straight symmetry, the observations we've made in this chapter about gradient descent carry over to gradient ascent: For many problems you can easily end up with a local optimum that is not very good. But sometimes one encounters problems—as we saw, for example, with the Maximum-Cut and Labeling Problems—for which a local search algorithm comes with a very strong guarantee: Every local optimum is close in value to the global optimum. We now consider the Bipartite Matching Problem and find that the same phenomenon happens here as well.

   Thus, consider the following Gradient Ascent Algorithm for finding a matching in a bipartite graph.

   > *As long as there is an edge whose endpoints are unmatched, add it to the current matching. When there is no longer such an edge, terminate with a locally optimal matching.*

   (a) Give an example of a bipartite graph G for which this gradient ascent algorithm does not return the maximum matching.

   (b) Let M and M' be matchings in a bipartite graph G. Suppose that $|M'| > 2|M|$. Show that there is an edge $e' \in M'$ such that $M \cup \{e'\}$ is a matching in G.

   (c) Use (b) to conclude that any locally optimal matching returned by the gradient ascent algorithm in a bipartite graph G is at least *half* as large as a maximum matching in G.

3. Suppose you're consulting for a biotech company that runs experiments on two expensive high-throughput assay machines, each identical, which we'll label M_1 and M_2. Each day they have a number of jobs that they need to do, and each job has to be assigned to one of the two machines. The problem they need help on is how to assign the jobs to machines to keep the loads balanced each day. The problem is stated as follows. There are n jobs, and each job j has a required processing time t_j. They need to partition the jobs into two groups A and B, where set A is assigned to M_1 and set B to M_2. The time needed to process all of the jobs on the two machines is $T_1 = \sum_{j \in A} t_j$ and $T_2 = \sum_{j \in B} t_j$. The problem is to have the two machines work roughly for the same amounts of time—that is, to minimize $|T_1 - T_2|$.

A previous consultant showed that the problem is NP-hard (by a reduction from Subset Sum). Now they are looking for a good local search algorithm. They propose the following. Start by assigning jobs to the two machines arbitrarily (say jobs $1, \ldots, n/2$ to M_1, the rest to M_2). The local moves are to move a single job from one machine to the other, and we only move jobs if the move decreases the absolute difference in the processing times. You are hired to answer some basic questions about the performance of this algorithm.

(a) The first question is: How good is the solution obtained? Assume that there is no single job that dominates all the processing time—that is, that $t_j \leq \frac{1}{2}\sum_{i=1}^{n} t_i$ for all jobs j. Prove that for every locally optimal solution, the times the two machines operate are roughly balanced: $\frac{1}{2}T_1 \leq T_2 \leq 2T_1$.

(b) Next you worry about the running time of the algorithm: How often will jobs be moved back and forth between the two machines? You propose the following small modification in the algorithm. If, in a local move, many different jobs can move from one machine to the other, then the algorithm should always move the job j with maximum t_j. Prove that, under this variant, each job will move at most once. (Hence the local search terminates in at most n moves.)

(c) Finally, they wonder if they should work on better algorithms. Give an example in which the local search algorithm above will not lead to an optimal solution.

4. Consider the Load Balancing Problem from Section 11.1. Some friends of yours are running a collection of Web servers, and they've designed a local search heuristic for this problem, different from the algorithms described in Chapter 11.

Recall that we have m machines $M_1, \ldots, M_m$, and we must assign each job to a machine. The load of the i^{th} job is denoted t_i. The makespan of an assignment is the *maximum load* on any machine:

$$\max_{\text{machines } M_i} \sum_{\text{jobs } j \text{ assigned to } M_i} t_j.$$

Your friends' local search heuristic works as follows. They start with an arbitrary assignment of jobs to machines, and they then repeatedly try to apply the following type of "swap move."

Let $A(i)$ and $A(j)$ be the jobs assigned to machines M_i and M_j, respectively. To perform a swap move on M_i and M_j, choose subsets of jobs $B(i) \subseteq A(j)$ and $B(j) \subseteq A(j)$, and "swap" these jobs between the two machines. That is, update $A(i)$ to be $A(i) \cup B(j) - B(i)$,

and update $A(j)$ to be $A(j) \cup B(i) - B(j)$. (One is allowed to have $B(i) = A(i)$, or to have $B(i)$ be the empty set; and analogously for $B(j)$.)

Consider a swap move applied to machines M_i and M_j. Suppose the loads on M_i and M_j before the swap are T_i and T_j, respectively, and the loads after the swap are T_i' and T_j'. We say that the swap move is *improving* if $\max(T_i', T_j') < \max(T_i, T_j)$—in other words, the larger of the two loads involved has strictly decreased. We say that an assignment of jobs to machines is *stable* if there does not exist an improving swap move, beginning with the current assignment.

Thus the local search heuristic simply keeps executing improving swap moves until a stable assignment is reached; at this point, the resulting stable assignment is returned as the solution.

Example. Suppose there are two machines: In the current assignment, the machine M_1 has jobs of sizes $1, 3, 5, 8$, and machine M_2 has jobs of sizes $2, 4$. Then one possible improving swap move would be to define $B(1)$ to consist of the job of size 8, and define $B(2)$ to consist of the job of size 2. After these two sets are swapped, the resulting assignment has jobs of size $1, 2, 3, 5$ on M_1, and jobs of size $4, 8$ on M_2. This assignment is stable. (It also has an optimal makespan of 12.)

(a) As specified, there is no explicit guarantee that this local search heuristic will always terminate. What if it keeps cycling forever through assignments that are not stable?

Prove that, in fact, the local search heuristic terminates in a finite number of steps, with a stable assignment, on any instance.

(b) Show that any stable assignment has a makespan that is within a factor of 2 of the minimum possible makespan.

Notes and Further Reading

Kirkpatrick, Gelatt, and Vecchi (1983) introduced simulated annealing, building on an algorithm of Metropolis et al. (1953) for simulating physical systems. In the process, they highlighted the analogy between energy landscapes and the solution spaces of computational problems.

The book of surveys edited by Aarts and Lenstra (1997) covers a wide range of applications of local search techniques for algorithmic problems. Hopfield neural networks were introduced by Hopfield (1982) and are discussed in more detail in the book by Haykin (1999). The heuristic for graph partitioning discussed in Section 12.5 is due to Kernighan and Lin (1970).

The local search algorithm for classification based on the Labeling Problem is due to Boykov, Veksler, and Zabih (1999). Further results and computational experiments are discussed in the thesis by Veksler (1999).

The multi-agent routing problem considered in Section 12.7 raises issues at the intersection of algorithms and game theory, an area concerned with the general issue of strategic interaction among agents. The book by Osborne (2003) provides an introduction to game theory; the algorithmic aspects of the subject are discussed in surveys by Papadimitriou (2001) and Tardos (2004) and the thesis and subsequent book by Roughgarden (2002, 2004). The use of potential functions to prove the existence of Nash equilibria has a long history in game theory (Beckmann, McGuire, and Winsten, 1956; Rosenthal 1973), and potential functions were used to analyze best-response dynamics by Monderer and Shapley (1996). The bound on the price of stability for the routing problem in Section 12.7 is due to Anshelevich et al. (2004).