We have defined the class of **NP**-complete problems, which have the property that if there is a polynomial time algorithm for any one of these problems, there is a polynomial time algorithm for all of them. Unfortunately, nobody has found an algorithm for any **NP**-complete problem, and it is widely believed that it is impossible to do so.

This might seem like a big hint that we should just give up, and go back to solving problems like MAX-FLOW, where we can find a polynomial time solution. Unfortunately, **NP**-complete show up all the time in the real world, and people want solutions to these problems. What can we do?

What Can We Do?

Actually, there is a great deal we can do. Here are just a few possibilities:

- Restrict the inputs.

  NP-completeness refers to the worst case inputs for a problem. Often, inputs are not as bad as those that arise in **NP**-completeness proofs. For example, although the general SAT problem is hard, we have seen that the cases of 2SAT and Horn formulae have simple polynomial times algorithms.

- Provide approximations.

  NP-completeness results often arise because we want an exact answer. If we relax the problem so that we only have to return a good answer, then we might be able to develop a polynomial time algorithm. For example, we have seen that a greedy algorithm provides an approximate answer for the SET COVER problem.

- Develop heuristics.

  Sometimes we might not be able to make absolute guarantees, but we can develop algorithms that seem to work well in practice, and have arguments suggesting why they should work well. For example, the simplex algorithm for linear programming is exponential in the worst case, but in practice it's generally the right tool for solving linear programming problems.

- Use randomness.

  So far, all our algorithms have been *deterministic*; they always run the same way on the same input. Perhaps if we let our algorithm do some things randomly, we can avoid the **NP**-completeness problem?

Actually, the question of whether one can use randomness to solve an **NP**-complete problem is still open, though it appears unlikely. (As is, of course, the problem of whether one can solve an **NP**-complete problem in polynomial time!) However, randomness proves a useful tool when we try to come up with approximation algorithms and heuristics. Also, if one can assume the input comes from a suitable "random distribution", then often one can develop an algorithm that works well on average. (Notice the distinction between *using randomness in an algorithm* and *assuming the input is random*.)

We now consider approximation algorithms and heuristics. The difference between approximation algorithms and heuristic algorithms is really pretty small; we generally call something an approximation algorithm when we can prove something about its performance, and we generally call something a heuristic when its performance is good but we can't prove why.

To begin, we will look at heuristic methods. The amount we can prove about these methods is (as yet) very limited. However, these techniques have had some success in practice, and there are arguments in favor of why they are reasonable thing to try for some problems.

Approximation Algorithm Basics

Approximation algorithms give guarantees. It is worth keeping in mind that sometimes approximation algorithms do not always perform as well as heuristic-based algorithms. Other times they provide insight into the problem, so they can help determine good heuristics.

Often when we talk about an approximation algorithm, we give an *approximation ratio*. The approximation ratio gives the ratio between our solution and the actual solution. The goal is to obtain an approximation ratio as close to 1 as possible. If the problem involves a minimization, the approximation ratio will be greater than 1; if it involves a maximization, the approximation ratio will be less than 1.

Set Cover Approximation

The inputs to the set cover problem are a finite set $X = \{x_1, \ldots, x_n\}$, and a collection of subsets $\mathcal{S}$ of X such that $\bigcup_{S \in \mathcal{S}} S = X$. The problem is to find the subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that the sets of $\mathcal{T}$ cover X, that is

$$\bigcup_{T \in \mathcal{T}} T = X.$$

Notice that such a cover exists, since $\mathcal{S}$ is itself a cover.

A greedy heuristic would be to build a cover by repeatedly including the set in $\mathcal{S}$ that will cover the maximum

number of as yet uncovered elements. In this case, the greedy heuristic does not yield an optimal solution. Interestingly, however, we can prove that the greedy solution is a good solution, in the sense that it is not too far from the optimal. That is, it is an example of an *approximation algorithm*.

Claim 20.1 *Let k be the size of the smallest set cover for the instance $(X,\mathcal{S})$. Then the greedy heuristic finds a set cover of size at most $k\ln n$.*

Proof: Let $Y_i \subseteq X$ be the set of elements that are still not covered after i sets have been chosen with the greedy heuristic. Clearly $Y_0 = X$. We claim that there must be a set $A \in \mathcal{S}$ such that $|A \cap Y_i| \geq |Y_i|/k$. To see this, consider the sets in the optimal set cover of X. These sets cover Y_i, and there are only k of them, so one of these sets must cover at least a $1/k$ fraction of Y_i. Hence

$$|Y_{i+1}| \leq |Y_i| - |Y_i|/k = (1-1/k)|Y_i|,$$

and by induction,

$$|Y_i| \leq (1-1/k)^i |Y_0| = n(1-1/k)^i < n\mathrm{e}^{-i/k},$$

where the last inequality uses the fact that $1+x \leq \mathrm{e}^x$ with equality iff $x = 0$. Hence when $i \geq k\ln n$ we have $|Y_i| < 1$, meaning there are no uncovered elements, and hence the greedy algorithm finds a set cover of size at most $k\ln n$. ■

Exercise: Show that this bound is tight, up to constant factors. That is, give a family of examples where the set cover has size k and the greedy algorithm finds a cover of size $\Omega(k\ln n)$.

Vertex Cover Approximations

In the Vertex Cover problem, we wish to find a set of vertices of minimal size such that every edge is adjacent to some vertex in the cover. That is, given an undirected graph $G = (V,E)$, we wish to find $U \subseteq V$ such that every edge $e \in E$ has an endpoint in U. We have seen that Vertex Cover is **NP**-complete.

A natural greedy algorithm for Vertex Cover is to repeatedly choose a vertex with the highest degree, and put it into the cover. When we put the vertex in the cover, we remove the vertex and all its adjacent edges from the graph, and continue. Unfortunately, in this case the greedy algorithm gives us a rather poor aprroximation, as can be seen with the following example:

In the example, all edges are connected to the base level; there are $m/2$ vertices at the next level, $m/3$ vertices at the next level, and so on. Each vertex at the base level is connected to one vertex at each other level, and the connections are spread as evenly as possible at each level. A greedy algorithm could always choose a rightmost

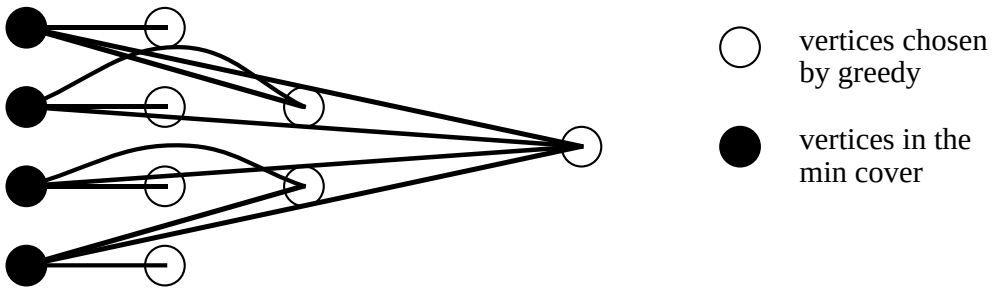

Figure 20.1: A bad greedy example.

vertex, whereas the optimal cover consists of the leftmost vertices. This example shows that, in general, the greedy approach could be off by a factor of $\Omega(\log n)$, where n is the number of vertices.

A better algorithm for vertex cover is the following: repeatedly choose an edge, and throw *both* of its endpoints into the cover. Throw the vertices and its adjacent edges out of the graph, and continue.

It is easy to show that this second algorithm uses at most twice as many vertices as the optimal vertex cover. This is because each edge that gets chosen during the course of the algorithm must have one of its endpoints in the cover; hence we have merely always thrown two vertices in where we might have gotten away with throwing in 1.

Somewhat surprisingly, this simple algorithm is still the best known approximation algorithm for the vertex cover problem. That is, no algorithm has been proven to do better than within a factor of 2.

Maximum Cut Approximation

We will provide both a randomized and a deterministic approximation algorithm for the MAX CUT problem. The MAX CUT problem is to divide the vertices in a graph into two disjoint sets so that the numbers of edges between vertices in different sets is maximized. This problem is **NP**-hard. Notice that the MIN CUT problem can be solved in polynomial time by repeated using the min cut-max flow algorithm. (Exercise: Prove this!)

The randomized version of the algorithm is as follows: we divide the vertices into two sets, HEADS and TAILS. We decide where each vertex goes by flipping a (fair) coin.

What is the probability an edge crosses between the sets of the cut? This will happen only if its two endpoints lie on different sides, which happens 1/2 of the time. (There are 4 possibilities for the two endpoints – HH,HT,TT,TH – and two of these put the vertices on different sides.) So, on average, we expect 1/2 the edges in the graph to cross

the cut. Since the most we could have is for all the edges to cross the cut, this random assignment will, on average, be within a factor of 2 of optimal.

We now examine a deterministic algorithm with the same "approximation ratio". (In fact, the two algorithms are intrinsically related– but this is not so easy to see!) The algorithm implements the hill climbing approximation heuristic. We will split the vertices into sets S_1 and S_2. Start with all vertices on one side of the cut. Now, if you can switch a vertex to a different side so that it increases the number of edges across the cut, do so. Repeat this action until the cut can no longer be improved by this simple switch.

We switch vertices at most $|E|$ times (since each time, the number of edges across the cut increases). Moreover, when the process finishes we are within a factor of 2 of the optimal, as we shall now show. In fact, when the process finishes, at least $|E|/2$ edges lie in the cut.

We can count the edges in the cut in the following way: consider any vertex $v \in S_1$. For every vertex w in S_2 that it is connected to by an edge, we add $1/2$ to a running sum. We do the same for each vertex in S_2. Note that each edge crossing the cut contributes 1 to the sum– 1/2 for each vertex of the edge.

Hence the cut C satisfies

$$C = \frac{1}{2}\left(\sum_{v\in S_1} |\{w : (v,w) \in E, w \in S_2\}| + \sum_{v\in S_2} |\{w : (v,w) \in E, w \in S_1\}|\right).$$

Since we are using the local search algorithm, at least half the edges from any vertex v must lie in the set opposite from v; otherwise, we could switch what side vertex v is on, and improve the cut! Hence, if vertex v has degree $\delta(v)$, then

$$\begin{aligned} C &= \frac{1}{2}\left(\sum_{v\in S_1} |\{w : (v,w) \in E, w \in S_2\}| + \sum_{v\in S_2} |\{w : (v,w) \in E, w \in S_1\}|\right) \\ &\geq \frac{1}{2}\left(\sum_{v\in S_1} \frac{\delta(v)}{2} + \sum_{v\in S_2} \frac{\delta(v)}{2}\right) \\ &= \frac{1}{4}\sum_{v\in V} \delta(v) \\ &= \frac{1}{2}|E|, \end{aligned}$$

where the last equality follows from the fact that if we sum the degree of all vertices, we obtain twice the number of edges, since we have counted each edge twice.

In practice, we might expect that hill climbing algorithm (see hill climbing discussion below) would do better than just getting a cut within a factor of 2.

Euclidean Travelling Salesperson Problem Approximation

In the Euclidean Travelling Salesman Problem, we are given n points (cities) in the $x-y$ plane, and we seek the tour (cycle) of minimum length that travels through all the cities. This problem is NP-complete (showing this is somewhat difficult).

Our approximation algorithm involves the following steps:

1. Find a minimum spanning tree T for the points.
2. Create a *pseudo tour* by walking around the tree. The pseudo tour may visit some vertices twice.
3. Remove repeats from the tour by *short-cutting* through the repeated vertices. (See Figure 20.2.)

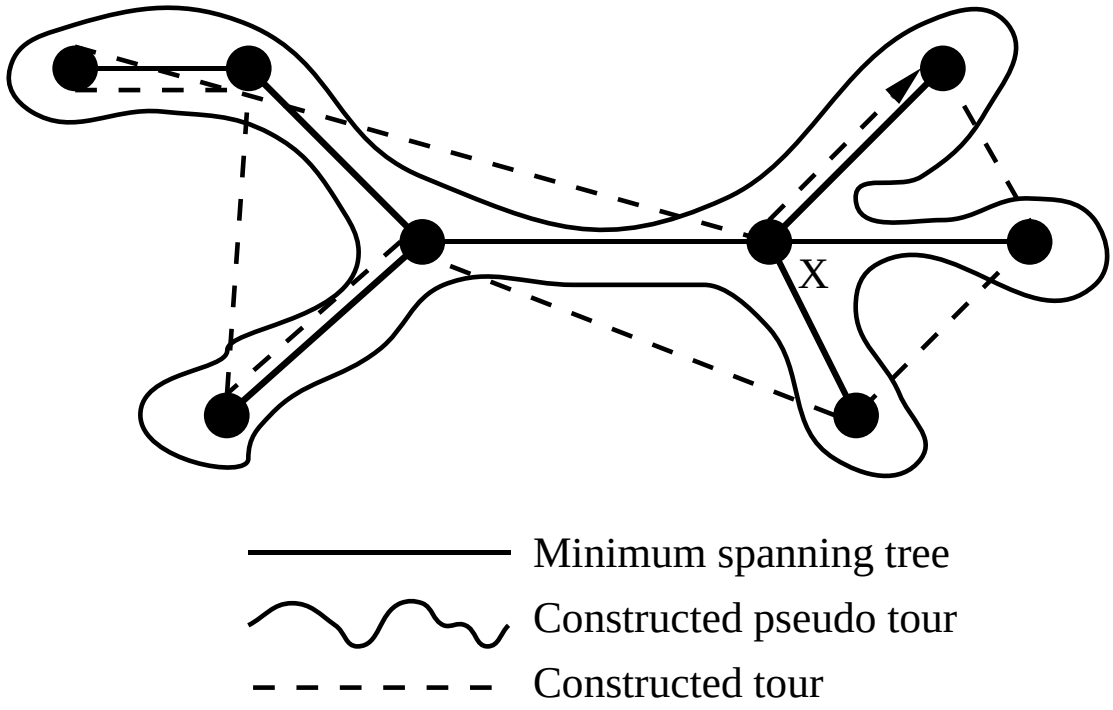

Figure 20.2: Building an approximate tour. Start at X, move in the direction shown, short-cutting repeated vertices.

We now show the following inequalities:

$$\begin{aligned} \text{length of tour} &\leq \text{length of pseudo tour} \\ &\leq 2(\text{size of T}) \\ &\leq 2(\text{length of optimal tour}) \end{aligned}$$

Short-cutting edges can only decrease the length of the tour, so the tour given by the algorithm is at most the length of the pseudo tour. The length of our pseudo tour is at most twice the size of the spanning tree, since this pseudo tour consists of walking through each edge of the tree at most twice. Finally, the length of the optimal tour is at least the size of the minimum spanning tree, since any tour contains a spanning tree (plus an edge!).

Using a similar idea, one can come up with an approximation algorithm that returns a tour that is within a factor of $3/2$ of the optimal. Also, note that this algorithm will work in any setting where short-cutting is effective. More specifically, it will work for any instance of the travelling salesperson problem that satisfies the *triangle inequality* for distances: that is, if $d(x,y)$ represents the distance between vertices x and y, and $d(x,z) \leq d(x,y) + d(y,z)$ for all x, y and z.

Linear Programming Relaxation

The next approach we describe, linear programming relaxation, can often be used as a good heuristic, and in some cases it leads to approximation algorithms with provable guarantees. Again, we will use the MAX-SAT problem as an example of how to use this technique.

The idea is simple. Most **NP**-complete problems can be easily described by a natural Integer Programming problem. (Of course, all **NP**-complete problems can be transformed into some Integer Programming problem, since Integer Programming is **NP**-complete; but what we mean here is in many cases the transformation is quite natural.) Even though we cannot solve the related Integer Program, *if we pretend it is a linear program*, then we can solve it, using (for example) the simplex method. This idea is known as *relaxation*, since we are relaxing the constraints on the solution; we are no longer requiring that we get a solution where the variables take on integer values.

If we are extremely lucky, we might find a solution of the linear program where all the variables are integers, in which case we will have solved our original problem. Usually, we will not. In this case we will have to try to somehow take the linear programming solution, and modify it into a solution where all the variables take on integer values. *Randomized Rouding* is one technique for doing this.

MAX-SAT

We may formulate MAX-SAT as an integer programming problem in a straightforward way (in fact, we have seen a similar reduction before, back when we examined reducitons; it is repeated here). Suppose the formula contains variables $x_1, x_2, \ldots, x_n$ which must be set to TRUE or FALSE, and clauses $C_1, C_2, \ldots, C_m$. For each variable

x_i we associate a variable y_i which should be 1 if the variable is TRUE, and 0 if it is FALSE. For each clause C_j we have a variable z_j which should be 1 if the clause is satisfied and 0 otherwise.

We wish to maximize the number of satisfied clauses s, or

$$\sum_{j=1}^{m} z_j.$$

The constraints include that that $0 \leq y_i, z_j \leq 1$; since this is an integer program, this forces all these variables to be either 0 or 1. Finally, we need a constraint for each clause saying that its associated variable z_j can be 1 if and only if the clause is actually satisfied. If the clause C_j is $(x_2 \vee \overline{x_4} \vee x_6 \vee \overline{x_8})$, for example, then we need the restriction:

$$y_2 + y_6 + (1 - y_4) + (1 - y_8) \geq z_j.$$

This forces z_j to be 0 unless the clause can be satisfied. In general, we replace x_i by y_i, $\overline{x_i}$ by $1 - y_i$, $\vee$ by $+$, and set the whole thing $\geq z_j$ to get the appropriate constraint.

When we solve the linear program, we will get a solution that might have $y_1 = 0.7$ and $z_1 = 0.6$, for instance. This initially appears to make no sense, since a variable cannot be 0.7 TRUE. But we can still use these values in a reasonable way. If $y_1 = 0.7$, it suggests that we would prefer to set the variable x_1 to TRUE (1). In fact, we could try just rounding each variable up or down to 0 or 1, and use that as a solution! This would be one way to turn the non-integer solution into an integer solution. Unfortunately, there are problems with this method. For example, suppose we have the clause $C1 = (x_1 \vee x_2 \vee x_3)$, and $y_1 = y_2 = y_3 = 0.4$. Then by simple rounding, this clause will not be TRUE, even though it "seems satisfied" to our linear program (that is, $z_1 = 1$). If we have a lot of these clauses, regular rounding might perform very poorly.

It turns out that there an interpretation for 0.7 that suggests a better way than simple rounding. We think of the 0.7 as *a probability*. That is, we interpret $y_1 = 0.7$ as meaning that x_1 would like to be true with probability 0.7. So we take each variable x_i, and independently we set it to 1 with the probability given by y_i (and with probability $1 - y_i$ we set x_i to 0). This process is known as *randomized rounding*. One reason randomized rounding is useful is it allows us *to prove* that the expected number of clauses we satisfy using this rounding is a within a constant factor of the true optimum.

First, note that whatever the maximum number of clauses s we can satisfy is, the value found by the linear program, or $\sum_{j=1}^{m} z_j$, is at least as big as s. This is because the linear program could achieve a value of at least s simply by using as the values for y_i the truth assignment that make satisfying s clauses possible.

Now consider a clause with k variables; for convenience, suppose the clause is just $C_1 = (x_1 \vee x_2 \ldots \vee x_k)$. Suppose that when we solve the linear program, we find $z_1 = \beta$. Then we claim that the probability that this clause

is satisfied after the rounding is at least $(1-1/e)\beta$. This can be checked (using a bit of sophisticated math), but it follows by noting (with experiments) that the worst possibility is that $y_1 = y_2 \ldots = y_k = \beta/k$. In this case, each x_1 is FALSE with probability $(1-\beta/k)$, and so C_1 ends up being unsatisfied with probability $(1-\beta/k)^k$. Hence the probability it is satisfied is at least (again using some math) $1-(1-\beta/k)^k \geq (1-1/e)\beta$.

Hence the ith clause is satisfied with probability at least $(1-1/e)z_i$, so the expected number of satisfied clauses after randomized rounding is at least $(1-1/e)\sum_{j=1}^{m} z_j$. This is within a factor of $(1-1/e)$ of our upper bound on the maximum number of satisfiable clauses, $\sum_{j=1}^{m} z_j$. Hence we expected to get within a constant factor of the maximum.

Surprisingly, by combining the simple coin flipping algorithm with the randomized rounding algorithm, we can get an even better algorithm. The idea is that the coin flipping algorithm does best on long clauses, since each literal in the clause makes it more likely the clause gets set to TRUE. On the other hand, randomized rounding does best on short clauses; the probability the clause is satisfied $(1-(1-\beta/k)^k)$ decreases with k. It turns out that if we try both algorithms, and take the better result, on average we will satisfy $3/4$ of the clauses.

We also point out that there are even more sophisticated approximation algorithms for MAX-SAT, with better approximation ratios.

Local Search

We have seen one general heuristic method: relaxation of an integer linear program. We now turn to other heuristics, beginning with local search.

"Local search" is meant to represent a large class of similar techniques that can be used to find a good solution for a problem. The idea is to think of the solution space as being represented by an undirected graph. That is, each possible solution is a node in the graph. An edge in the graph represents a possible move we can make between solutions.

For example, consider the Number Partition problem for the homework assignment. Each possible solution, or division of the set of numbers into two groups, would be a vertex in the graph of all possible solutions. For our possible moves, we could move between solutions by changing the sign associated with a number. So in this case, our graph of all possible solutions, we have an edge between any two possible solutions that differ in only one sign. Of course this graph of all possible solutions is huge; there are 2^n possible solutions when there are n numbers in the original problem! We could never hope to even write this graph down. The idea of local search is that we never actually try to write the whole graph down; we just move from one possible solution to a "nearby" possible solution, either for as long as we like, or until we happen to find an optimal solution.

To set up a local search algorithm, we need to have the following:

1. A set of possible solutions, which will be the vertices in our local search graph.
2. A notion of what the *neighbors* of each vertex in the graph are. For each vertex x, we will call the set of adjacent vertices $N(x)$. The neighbors must satisfy several properties: $N(x)$ must be easy to compute from x (since if we try to move from x we will need to compute the neighbors), if $y \in N(x)$ then $x \in N(y)$ (so it makes sense to represent neighbors as undirected edges), and $N(x)$ cannot be too big, or more than polynomial in the input size (so that the neighbors of a node are easy to search through).
3. A cost function, from possible solutions to the real numbers.

The most basic local search algorithm (say to minimize the cost function) is easily described:

1. Pick a starting point x.
2. While there is a neighbor y of x with $f(y) < f(x)$, move to it; that is, set x to y and continue.
3. Return the final solution.

The Reasoning Behind Local Search

The idea behind local search is clear; if keep getting better and better solutions, we should end up with a good one. Pictorially, if we "project" the state space down to a two dimensional curve, we are hoping that the picture has a sink, or *global optimum*, and that we will quickly move toward it. See Figure 20.3.

There are two possible problems with this line of thinking. First, even if the space does look this way, we might not move quickly enough toward the right solution. For example, for the number partition problem from the homework, it might be that each move improves our solution, but only by improving the residue by 1 each time. If we start with a bad solution, it will take a lot of moves to reach the minimum. Generally, however, this is not much of a problem, as long as the cost function is reasonably simple.

The more important problem is that the solution space might not look this way at all. For example, our cost function might not change smoothly when we move from a state to it neighbor. Also, it may be that there are several *local optima*, in which cas our local search algorithm will hone in a local optimum and get stuck. See Figure 20.4.

This second problem, that the solution space might not "look nice", is crucial, and it underscores *the importance of setting up the problem*. When we choose the possible moves between solutions – that is, when we construct the

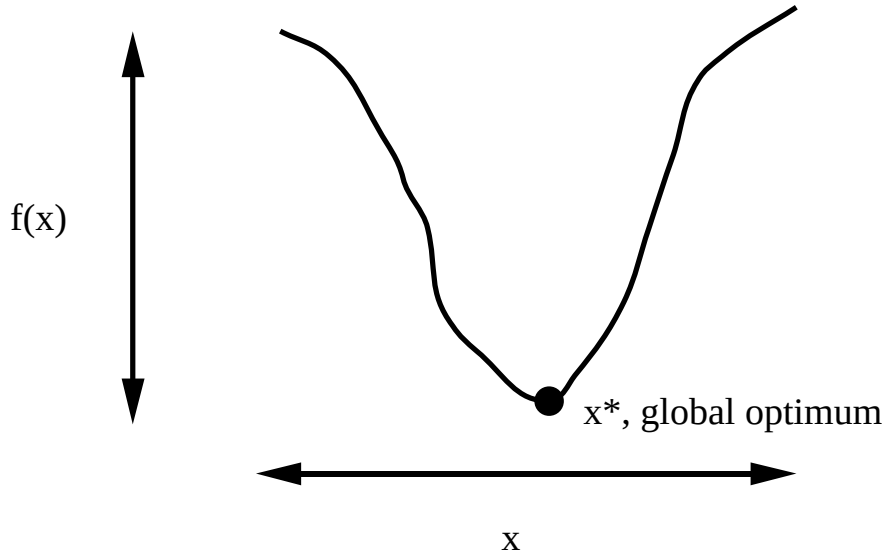

Figure 20.3: A very nice state space.

mapping that gives us the neighborhood of each node– we are setting up how local search will behave, including how the cost function will change between neighbors, and how many local optima there are. How well local search will work depends tremendously on how smart one is in setting up the right neighborhoods, so that the solution space really does look the way we would like it to.

Examples of Neighborhoods

We have already seen an example of a neighborhood for the homework problem. Here are possible neighborhoods for other problems:

- MAX3SAT: A possible neighborhood structure is two truth assignments are neighbors if they differ in only one variables. A more extensive neighborhood could make two truth assignments neighbors if they differ in at most two variables; this trades increased flexibility for increase size in the neighborhood.
- Traveling Salesperson: The k-opt neighborhood of x is given by all tours that differ in at most k edges from x. In practice, using the 3-opt neighborhood seems to perform better than the 2-opt neighborhood, and using 4-opt or larger increases the neighborhood size to a point where it is inefficient.

Lots of Room to Experiment

There are several aspects of local search algorithms that we can vary, and all can have an impact on performance.

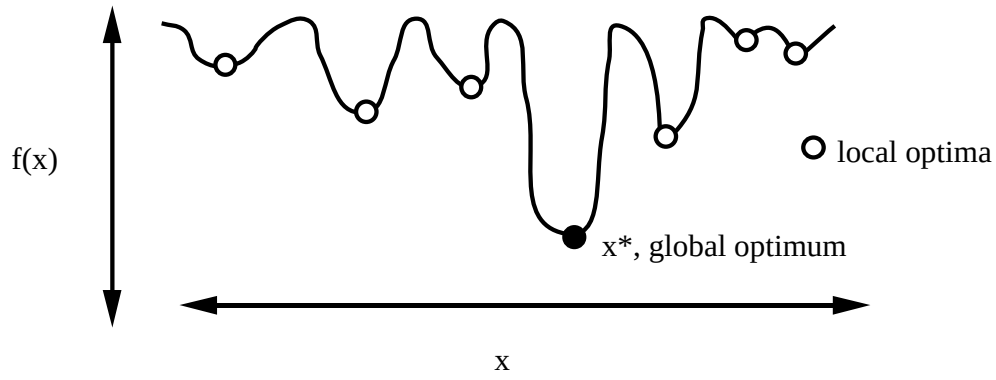

Figure 20.4: A state space with many local optima; it will be hard to find the best solution.

For example:

1. What are the nieghborhoods $N(x)$?
2. How do we choose an inital starting point?
3. How do we choose a neighbor y to move to? (Do we take the first one we find, a random neighbor that improves f, the neighbor that improves f the most, or do we use other criteria?)
4. What if there are ties?

There are other practical considerations to keep in mind. Can we re-run the algorithm several times? Can we try several of the algorithms on different machines? Issues like these can have a big impact on actual performance. However, perhaps the most important issue is to think of the right neighborhood structure to begin with; if this is right, then other issues are generally secondary, and if this is wrong, you are likely to fail no matter what you do.

Local Search Variations

There are many variations on the local search technique (below, assume the goal is to minimize the cost function):

- Hill-climbing – this is the name for the basic variation, where one moves to a vertex of lower (or possibly equal) cost.
- Metropolis rule – pick a random neighbor, and if the cost is lower, move there. If the cost is higher, move there

with some probability (that is usually set to depend on the cost differential). The idea is that possibly moving to a worse state helps avoid getting trapped at local minima.

- Simulated annealing – this method is similar to the Metropolis rule, except that the probability of going to a higher cost neighbor varies with time. This is analogous to a physical system (such as a chemical polymer) being cooled down over time.
- Tabu search – this adds some memory to hill climbing. Like with the Metropolis rule and simulated annealing, you can go to worse solutions. A penalty function is added to the cost function to try to prevent cycling and promote searching new areas of the search space.
- Parallel search ("go with the winners")– do multiple searches in parallel, occasionally killing off searches that appear less successful and replacing them with copies of searches that appear to be doing better.
- Genetic algorithms – this trendy area is actually quite related to local search. An important difference is that instead instead of keeping one solution at a time, a group of them (called a population) is kept, and the population changes at each step.

It is still quite unclear what exactly each of these techniques adds to the pot. For example, some people swear that genetic algorithms lead to better solutions more quickly than other methods, while others claim that by choosing the right neighborhood function one can do as well with hill climbing. In the years to come, hopefully more will become understood about all of these methods.

Hardness of Approximation

As we've seen, many optimization problems that are NP-hard to solve can still be well-approximated in polynomial time. However, it is natural to ask whether the approximation ratios achieved are the best possible (such as $\ln n$ for SET COVER and 2 for VERTEX COVER) or one can have arbitrarily good approximations in polynomial time (like we have for EUCLIDEAN TSP). For the first 20 years of NP-completeness, there was almost no understanding of the hardness of approximation. The breakthrough came from a very unexpected direction — the study of *probabilistic proof systems*, where we incorporate randomness (and sometimes interaction) into the classical notion of an NP proof:

Theorem 20.2 (PCP Theorem) *Every language in NP has a randomized polynomial-time verifier V, such that:*

1. *"Extreme" Efficiency: on input $\langle x,y\rangle$, V reads x, tosses $O(\log n)$ coins, reads $O(1)$ bits of y, and then decides whether to accept or reject.*
2. *Completeness: if $x \in L$, then there exists $y \in \{0,1\}^{\text{poly}(|x|)}$ such that $\Pr[V(\langle x,y\rangle) = 1] = 1$.*
3. *Soundness: if $x \notin L$, then for all $y \in \{0,1\}^{\text{poly}(|x|)}$, we have $\Pr[V(\langle x,y\rangle) = 1] \leq 1/2$.*

What is remarkable is that arbitrary NP statements can be verified by probabilistically checking only a *constant* number of bits of a witness y! Since SHORT PROOF is NP-complete, this means that any mathematical proof (in a standard logical system like ZFC Set Theory) can be transformed into one that has this probabilistic verification property. Like with RP, the error probability of 1/2 can be made an arbitrarily small constant by repeating the verifier V several times.

The study of PCPs and related concepts ("interactive proofs" and "zero-knowledge proofs") emerged from work on cryptography, where the goal was to enable distrustful parties to convince each other of various facts (e.g. I have an encrypted and digitally signed bank statement showing that I have at least $10,000 in my account), without revealing their secrets (e.g. a decryption key). But they turned out to have great significance for computational complexity, including unlocking the mysteries of approximation. Indeed, the following turns out to be *equivalent* to the PCP Theorem.

Theorem 20.3 (PCP Theorem, restated as a Gap-Producing Reduction) *There is a constant $\varepsilon > 0$ and a polynomial-time reduction $\varphi \mapsto \varphi'$ from 3-SAT to 3-SAT such that:*

1. *If φ is satisfiable, then so is φ'.*
2. *If φ is unsatisfiable, then no assignment satisfies even a $(1-\varepsilon)$ fraction of the clauses of φ'.*

Note that such a gap-producing reduction implies that there is no $(1-\varepsilon)$-approximation algorithm for 3-SAT. The equivalence of the two versions of the PCP Theorem is not too hard to establish (can you see why Gap-Producing Reductions imply the first statement of the PCP Theorem?), but proofs of the PCP Theorem itself are quite involved and use many deep and beautiful ideas (encoding computations using algebra/polynomials and/or gap-amplification using random walks on expander graphs). If you're interested in this area, see a graduate-level textbook on computational complexity (like the one by Goldreich or the one by Arora and Barak), or take a graduate course on complexity (like CS 221). While now many tight inapproximability results are known (like $\ln n$ for SET COVER and $7/8$ for MAX 3-SAT), there are still major open problems like the Unique Games Conjecture (discussed in Boaz Barak's CS Colloquium on Dec. 1).