7.3.6. Show that the following problem is $\mathcal{NP}$-complete. DOMINATING SET: Given a directed graph G and an integer B, is there a set S of B nodes of G such that for every node $u \notin S$ of G, there is a node $v \in S$ such that (v, u) is an edge of G.

7.3.7. Call a nondeterministic finite automaton $M = (K, \Sigma, \Delta, s, F)$ *acyclic* if there is no state q and string $w \neq e$ such that $(q, w) \vdash_M^* (q, e)$. Show that the problem of telling whether two acyclic nondeterministic finite automata are inequivalent is $\mathcal{NP}$-complete.

7.4 COPING WITH NP-COMPLETENESS

Problems do not go away when they are proved $\mathcal{NP}$-complete. But once we know that the problem we are interested in is an $\mathcal{NP}$-complete problem, we are more willing to lower our sights, to settle for solutions that are less than perfect, for algorithms that are not always polynomial, or do not work on all possible instances. In this section we review some of the most useful maneuvers of this sort.

Special Cases

Once our problem has been shown $\mathcal{NP}$-complete, the first question to ask is this: Do we *really* need to solve this problem in the full generality in which it was formulated —and proved $\mathcal{NP}$-complete? $\mathcal{NP}$-completeness reductions often produce instances of the problem that are unnaturally complex. Perhaps what we really need to solve is a more tractable *special case* of the problem.

For example, we have already seen that there is an important special case of SATISFIABILITY that can be easily solved efficiently: 2-SATISFIABILITY (recall Section 6.3). If all instances of SATISFIABILITY that we must solve have clauses of this kind, then the fact that the general problem is $\mathcal{NP}$-complete is rather irrelevant. But often a special case of interest turns out to be *itself* $\mathcal{NP}$-complete —for example, 3-SATISFIABILITY is such a case, recall Theorem 7.2.3. We next see another example.

Example 7.4.1: Most problems involving undirected graphs become easy when the graph is a **tree** —that is to say, it has no cycles, see Figure 7-12. Looking back at our collection of $\mathcal{NP}$-complete graph problems, HAMILTON CYCLE is of course trivial in trees (no tree has a cycle, Hamilton or otherwise), but so is HAMILTON PATH —a tree has a Hamilton path only if *it is* a Hamilton path. The CLIQUE problem also becomes trivial —no tree can have a clique with more than two nodes.

The INDEPENDENT SET problem is also easy when the graph is a tree. The method used for its solution takes advantage of the "hierarchical structure" of

trees. It is often useful in a tree to pick an arbitrary node and designate it as the **root** (see Figure 7-12); once this has been done, each node u in the tree becomes itself the root of a *subtree* $T(u)$ —the set of all nodes v such that the (unique) path from v to the root goes through u; see Figure 7-12. Then problems can be solved bottom up, by going from the leaves (subtrees with one node) to larger and larger subtrees, until the whole tree (the subtree of the root) has been dealt with. For each node u we can define the set of its *children* $C(u)$ —the nodes in its subtree that are adjacent to it, excluding u itself— and its set of *grandchildren* $G(u)$ —the children of its children. Naturally, these sets could be empty. For example, in Figure 7-12, the root, denoted r, has two children and five grandchildren. Nodes with no children are called *leaves*.

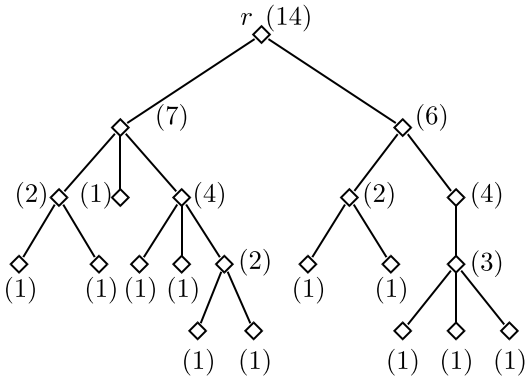

Figure 7-12

The size of the largest independent set of the tree can now be found by computing, for each node u, the number $I(u)$, defined to be the size of the largest independent set of $T(u)$. It is easy to see that the following equation holds:

$$I(u) = \max\{ \sum_{v \in C(u)} I(v), \quad 1 + \sum_{v \in G(u)} I(u)\} \tag{2}$$

What this equation says is that, in designing the largest independent set of $T(u)$, we have two choices: Either (this is the first term in the max) we do not put u into the independent set, in which case we can put together all maximum independent sets in the subtrees of its children, or (and this is the second term) we put u in the independent set, in which case we must omit all its children, and assemble the maximum independent sets of the subtrees of all its grandchildren.

It is now easy to see that a *dynamic programming* algorithm can solve the INDEPENDENT SET problem in the special case of trees in polynomial time. The algorithm starts at the leaves (where $I(u)$ is trivially one) and computes $I(u)$ for

larger and larger subtrees. The value of I at the root is the size of the maximum independent set of the tree. The algorithm is polynomial, because for each node u, all we have to do is compute the expression in (2), which only takes linear time. For example, in the tree of Figure 7-12, the values of $I(u)$ are shown in parentheses. The largest independent set of the tree has size 14.

Needless to say, the closely related NODER COVER problem can also be solved the same way (recall the reductions between NODE COVER and INDEPENDENT SET). So, if the graphs we are interested in happen to be trees, the fact that NODE COVER and INDEPENDENT SET are $\mathcal{NP}$-complete is irrelevant. Many other $\mathcal{NP}$-complete problems on graphs are solved by similar algorithms when specialized to trees, see for example Problem 7.4.1.$\diamondsuit$

Approximation Algorithms

When facing an $\mathcal{NP}$-complete optimization problem, we may want to consider algorithms that do not produce optimum solutions, but solutions *guaranteed to be close to the optimum.* Suppose that we wish to obtain such solutions for an optimization problem, maximization or minimization. For each instance x of this problem, there is an optimum solution with value $\text{opt}(x)$; let us assume that $\text{opt}(x)$ is always a positive integer (this is the case with all optimization problems we study here; we can easily spot and solve instances in which opt is zero).

Suppose now that we have a polynomial algorithm A which, when presented with instance x of the optimization problem, returns some solution with value $A(x)$. Since the problem is $\mathcal{NP}$-complete and A is polynomial, we cannot realistically hope that $A(x)$ is always the optimum value. But suppose that we know that the following inequality always holds:

$$\frac{|\text{opt}(x) - A(x)|}{\text{opt}(x)} \leq \epsilon,$$

where ϵ is some positive real number, hopefully very small, that bounds from above the worst-case relative error of algorithm A. (The absolute value in this inequality allows us to treat both minimization and maximization problems within the same framework.) If algorithm A satisfies this inequality for all instances x of the problem, then it is called an **ϵ-approximation algorithm.**

Once an optimization problem has been shown to be $\mathcal{NP}$-complete, the following question becomes most important: Are there ϵ-approximation algorithms for this problem? And if so, how small can ϵ be? Let us observe at the outset that such questions are meaningful only if we assume that $\mathcal{P} \neq \mathcal{NP}$, because, if $\mathcal{P} = \mathcal{NP}$, then the problem can be solved exactly, with $\epsilon = 0$.

All $\mathcal{NP}$-complete optimization problems can therefore be subdivided into three large categories:

(a) Problems that are **fully approximable**, in that there is an ϵ-approximate polynomial-time algorithm for them *for all* $\epsilon > 0$, however small. Of the $\mathcal{NP}$-complete optimization problems we have seen, only TWO-MACHINE SCHEDULING (in which we wish to minimize the finishing time D) falls into this most fortunate category.

(b) Problems that are **partly approximable**, in that there are ϵ-approximate polynomial-time algorithms for them for some range of ϵ's, but —unless of course $\mathcal{P} = \mathcal{NP}$— this range does not reach all the way down to zero, as with the fully approximable problems. Of the $\mathcal{NP}$-complete optimization problems we have seen, NODE COVER and MAX SAT fall into this intermediate class.

(c) Problems that are **inapproximable**, that is, there is no ϵ-approximation algorithm for them, with however large ϵ —unless of course $\mathcal{P} = \mathcal{NP}$. Of the $\mathcal{NP}$-complete optimization problems we have seen in this chapter, unfortunately many fall into this category: the TRAVELING SALESMAN PROBLEM, CLIQUE, INDEPENDENT SET, as well as the problem of minimizing the number of states of a deterministic automaton equivalent to a given regular expression in output polynomial time (recall the corollary to Theorem 7.3.8).

Example 7.4.2: Let us describe a 1-approximation algorithm for NODE COVER —that is to say, an algorithm which, for any graph, returns a node cover that is at most *twice* the optimum size. The algorithm is very simple:

$C := \emptyset$
while there is an edge $[u, v]$ left in G **do**
  add u and v to C, and delete them from G

For example, in the graph in Figure 7-13, the algorithm might start by choosing edge $[a, b]$ and inserting both endpoints in C; both nodes (and their adjacent edges, of course) are then deleted from G. Next $[e, f]$ might be chosen, and finally $[g, h]$. The resulting set C is a node cover, because each edge in G must touch one of its nodes (either because it was chosen by the algorithm, or because it was deleted by it). In the present example, $C = \{a, b, e, f, g, h\}$, has six nodes, which is at most twice the optimum value —in this case, four.

To prove the "at most twice" guarantee, consider the cover C returned by the algorithm, and let $\hat{C}$ be the optimum node cover. $|C|$ is exactly twice the number of edges chosen by the algorithm. However, these edges by the very way they were chosen, have no vertices in common, and for each of them at least one of its endpoints must be in $\hat{C}$ —because $\hat{C}$ is a node cover. It follows that the number of edges chosen by the algorithm is no larger than the optimum set cover, and hence $|C| \leq 2 \cdot |\hat{C}|$, and this is indeed a 1-approximation algorithm.

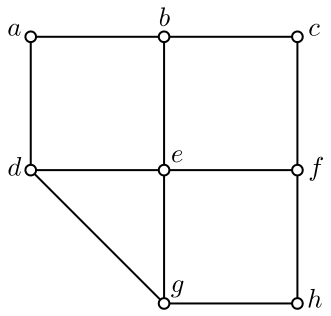

Figure 7-13

Can we do better? Depressingly, this simple approximation algorithm is the best one known for the NODE COVER problem. And only very recently have we been able to prove that, unless $\mathcal{P} = \mathcal{NP}$, there is no ϵ-approximation algorithm for NODE COVER for any $\epsilon < \frac{1}{6}$.$\diamondsuit$

Example 7.4.3: However, for TWO-MACHINE SCHEDULING, there is no limit to how close to the optimum we can get: For any $\epsilon > 0$ there is an ϵ-approximation algorithm for this problem.

This family of algorithms is based on an idea that we have already seen: Recall that the PARTITION problem can be solved in time $\mathcal{O}(nS)$ (where n is the number of integers, and S is their sum; see Section 6.2). It is very easy to see that this algorithm can be rather trivially adapted to solve the TWO-MACHINE SCHEDULING (finding the smallest D): The $B(i)$ sets are extended to include sums up to S (not just up to $H = \frac{1}{2}S$). The smallest sum in $B(n)$ that is $\geq \frac{1}{2}S$ is the desired minimum D.

One more idea is needed to arrive at our approximation algorithm: Consider an instance of TWO-MACHINE SCHEDULING with these task lengths

$$45362, 134537, 85879, 56390, 145627, 197342, 83625, 126789, 38562, 75402,$$

with $n = 10$, and $S \approx 10^6$. Solving it by our exact $\mathcal{O}(nS)$ algorithm would cost us an unappetizing 10^7 steps. But suppose instead that we *round up* the task lengths to the next hundred. We obtain the numbers

$$45400, 134600, 85900, 56400, 145700, 197400, 83700, 126800, 38600, 75500,$$

which is really the same as

$$454, 1346, 859, 564, 1457, 1974, 837, 1268, 386, 755,$$

(normalizing by 100); thus we can now solve this instance in about 10^5 steps. By sacrificing a little in accuracy (the optimum of the new problem is clearly not very far from the original one), we have decreased the time requirements a hundredfold!

It is easy to prove that, if we round up to the next kth power of ten, the difference between the two optimal values is no more than $n10^k$. To calculate the relative error, this quantity must be divided by the optimum, which, obviously, can be no less than $\frac{S}{2}$. We have thus a $\frac{2n10^k}{S}$-approximation algorithm, whose running time is $\mathcal{O}(\frac{nS}{10^k})$. By setting $\frac{2n10^k}{S}$ equal to any desirable $\epsilon > 0$, we arrive at an algorithm whose running time is $\mathcal{O}(\frac{n^2}{\epsilon})$ —certainly a polynomial.$\diamondsuit$

Example 7.4.4: How does one prove that a problem is inapproximable (or not fully approximable)? For most optimization problems of interest, this question had been one of the most stubborn open problems, and required the development of novel ideas and mathematical techniques (see the references at the end of this chapter). But let us look at a case in which such a proof is relatively easy, that of the TRAVELING SALESMAN PROBLEM.

Suppose that we are given some large number ϵ, and we must prove that, unless $\mathcal{P} = \mathcal{NP}$, there is no ϵ-approximation algorithm for the TRAVELING SALESMAN PROBLEM. We know that the HAMILTON CYCLE problem is $\mathcal{NP}$-complete; we shall show that, if there is an ϵ-approximation algorithm for the TRAVELING SALESMAN PROBLEM, then there is a polynomial-time algorithm for the HAMILTON CYCLE problem. Let us start with any instance G of the HAMILTON CYCLE problem, with n nodes. We apply to it the simple reduction from HAMILTON CYCLE to TRAVELING SALESMAN PROBLEM (recall the proof of Theorem 7.3.4), but with a twist: The distances d_{ij} are now the following (compare with the proof of Theorem 7.3.4):

$$d_{ij} = \begin{cases} 0 & \text{if } i = j; \\ 1 & \text{if } (v_i, v_j) \in G; \\ 2 + n\epsilon & \text{otherwise.} \end{cases}$$

The instance constructed has the following interesting property: If G has a Hamilton cycle, then the optimum cost of a tour is n; if, however, there is no Hamilton cycle, then the optimum cost is *greater than* $n(1 + \epsilon)$ —because at least one distance $2 + n\epsilon$ must be traversed, in addition to at least $n - 1$ others of cost at least 1.

Suppose that we had a polynomial-time ϵ-approximation algorithm A for the TRAVELING SALESMAN PROBLEM. Then we would be able to tell whether G has a Hamilton cycle as follows: Run algorithm A on the given instance of the TRAVELING SALESMAN PROBLEM. Then we have these two cases:

(a) If the solution returned has cost $\geq n(1+\epsilon)+1$, then we know that the optimum cannot be n, because in that case the relative error of A would have been at least

$$\frac{|n(1+\epsilon)+1-n|}{n} > \epsilon,$$

which contradicts our hypothesis that A is an ϵ-approximation algorithm. Since the optimum solution is larger than n, we conclude that G has no Hamilton cycle.

(b) If, however, the solution returned by A has cost $\leq n(1+\epsilon)$, then we know that the optimum solution must be n. This is because our instance was designed so that it cannot have a tour of cost between $n+1$ and $n(1+\epsilon)$. Hence, in this case G has a Hamilton cycle.

It follows that, by applying the polynomial algorithm A on the instance of the TRAVELING SALESMAN PROBLEM that we constructed from G in polynomial time, we can tell whether G has a Hamilton cycle —which implies that $\mathcal{P} = \mathcal{NP}$. Since this argument can be carried out for any $\epsilon > 0$, however large, we must conclude that the TRAVELING SALESMAN PROBLEM is inapproximable.$\diamondsuit$

Ways of coping with $\mathcal{NP}$-completeness often mix well: Once we realize that the TRAVELING SALESMAN PROBLEM is inapproximable, we may want to *approximate special cases* of the problem. Indeed, let us consider the special case in which the distances d_{ij} satisfy the *triangle inequality*

$$d_{ij} \leq d_{ik} + d_{kj} \text{ for each } i, j, k,$$

a fairly natural assumption on distance matrices, which holds in most instances of the TRAVELING SALESMAN PROBLEM arising in practice. As it turns out, this special case is partly approximable, and the best known error bound is $\frac{1}{2}$. What is more, when the cities are restricted to be points on the plane with the usual Euclidean distances —another special case of obvious appeal and relevance— then the problem becomes fully approximable! Both special cases are known to be $\mathcal{NP}$-complete (see Problem 7.4.3 for the proof for the triangle inequality case).

Backtracking and Branch-and-Bound

All $\mathcal{NP}$-complete problems are, by definition, solvable by polynomially bounded nondeterministic Turing machines; unfortunately we only know of exponential methods to simulate such machines. We examine next a class of algorithms that tries to improve on this exponential behavior with clever, problem-dependent stratagems. This approach typically produces algorithms that are exponential in the worst case, but often do much better.

A typical $\mathcal{NP}$-complete problem asks whether any member of a large set S_0 of "candidate certificates", or "candidate witnesses" (truth assignments, sets of vertices, permutations of nodes, and so onrecall Section 6.4) satisfies certain constraints specified by the instance (satisfies all clauses, is a clique of size K, is a Hamilton path). We call these candidate certificates or witnesses *solutions*. For all interesting problems, the size of the set S_0 of all possible solutions is typically exponentially large, and only depends on the given instance x (its size depends exponentially on the number of variables in the formula, on the number of nodes in the graph, and so on).

Now, a nondeterministic Turing machine "solving" an instance of this $\mathcal{NP}$-complete problem produces a tree of configurations (recall Figure 6-3). Each of these configurations corresponds to a subset of the set of potential solutions S_0, call it S, and the "task" facing this configuration is to determine whether there is a solution in S satisfying the constraints of x. Hence, S_0 is the set corresponding to the initial configuration. Telling whether S contains a solution is often a problem not very different from the original one. Thus, we can see each of the configurations in the tree as a *subproblem* of the same kind as the original (this useful "self-similarity" property of $\mathcal{NP}$-complete problems is called *self-reducibility*). Making a nondeterministic choice out of a configuration, say leading to r possible next configurations, corresponds to replacing S with r sets, $S_1, \ldots, S_r$, whose union must be S, so that no candidate solution ever falls between the cracks.

This suggests the following genre of algorithms for solving $\mathcal{NP}$-complete problems: We always maintain a set of *active subproblems,* call it $\mathcal{A}$; initially, $\mathcal{A}$ contains only the original problem S_0; that is, $\mathcal{A} = \{S_0\}$. At each point we choose a subproblem from $\mathcal{A}$ (presumably the one that seems most "promising" to us), we remove it from $\mathcal{A}$, and replace it with several smaller subproblems (whose union of candidate solutions must cover the one just removed). This is called *branching.*

Next, each newly generated subproblem is submitted to a quick *heuristic test.* This test looks at a subproblem, and comes up with one of three answers:

(a) It may come up with the answer "empty," meaning that the subproblem under consideration has no solutions satisfying the constraint of the instance, and hence it can be omitted. This event is called *backtracking.*

(b) It may come up with an actual solution of the original problem contained in the current subproblem (a satisfying truth assignment of the original formula, a Hamilton cycle of the original graph, etc.), in which case the algorithm terminates successfully.

(c) Since the problem is $\mathcal{NP}$-complete, we cannot hope to have a quick heuristic test that always comes up with one of the above answers (otherwise, we would submit the original subproblem S_0 to it). Hence, the test will often reply "?", meaning that it cannot prove that the subproblem is empty, but it

cannot find a quick solution in it either; in this case, we add the subproblem in hand to the set $\mathcal{A}$ of active subproblems. The hope is that the test will come up with one of the two other answers often enough, and thus will substantially reduce the number of subproblems we will have to examine —and ultimately the running time of the algorithm.

We can now show the full **backtracking algorithm**:

$\mathcal{A} := \{S_0\}$
while $\mathcal{A}$ is not empty do
  choose a subproblem S and delete it from $\mathcal{A}$
  choose a way of branching out of S, say to subproblems $S_1, \ldots, S_r$
  for each subproblem S_i in this list do
    if test(S_i) returns "solution found" then halt
    else if test(S_i) returns "?" then add S_i to $\mathcal{A}$
return "no solution"

The backtracking algorithm terminates because, in the end, the subproblems will become so small and specialized that they will contain just one candidate solution (these are the leaves of the tree of the nondeterministic computation); in this case the test will be able to decide quickly whether or not this solution satisfies the constraints of the instance.

The effectiveness of a backtracking algorithm depends on three important "design decisions:"

(1) How does one choose the next subproblem out of which to branch?
(2) How is the chosen subproblem further split into smaller subproblems?
(3) Which test is used?

Example 7.4.5: In order to design a backtracking algorithm for SATISFIABILITY, we must make the design decisions (1) through (3) above.

In SATISFIABILITY the most natural way to split a subproblem is to choose a variable x and create two subproblems: one in which $x = \top$, and one in which $x = \bot$. As promised, each subproblem is of the same sort as the original problem: a set of clauses, but with fewer variables (plus a fixed truth assignment for each of the original variables not appearing in the current subproblem). In the $x = \top$ subproblem, the clauses in which x appears are omitted, and $\overline{x}$ is omitted from the clauses in which it appears; exactly the opposite happens in the $x = \bot$ subproblem.

The question regarding design decision (2) is, how to choose the variable x on which to branch. Let us use the following rule: *Choose a variable that appears in the smallest clause* (if there are ties, break them arbitrarily). This is a sensible strategy, because smaller clauses are "tighter" constraints, and may lead sooner to backtracking. In particular, an empty clause is the unmistakable sign of unsatisfiability.

Now for design decision (1) —how to choose the next subproblem. In line with our strategy for (2), let us choose the subproblem that contains the smallest clause (again, we break ties arbitrarily).

Finally, the test (design decision (3)) is very simple:

if there is an empty clause, return "subproblem is empty;"
if there are no clauses, return "solution found;"
otherwise return "?"

See Figure 7-14 for an application of the backtracking algorithm described above to the instance

$$(x \vee y \vee z), (\overline{x} \vee y), (\overline{y} \vee z), (\overline{z} \vee x), (\overline{x} \vee \overline{y} \vee \overline{z}),$$

which we know is unsatisfiable (recall Example 6.3.3). As it turns out, this algorithm is a variant of a well-known algorithm for SATISFIABILITY, known as the **Davis-Putnam procedure**. Significantly, when the instance has at most two literals per clause, the backtracking algorithm *becomes exactly the polynomial purge algorithm of Section 6.3.*$\diamondsuit$

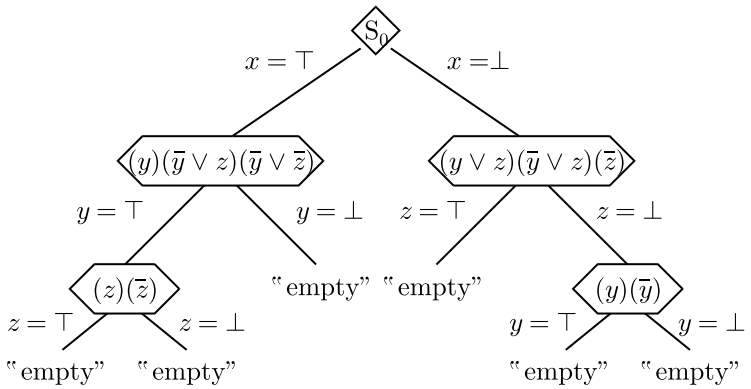

Figure 7-14

Example 7.4.6: Let us now design a backtracking algorithm for HAMILTON CYCLE. In each subproblem we already have a path with endpoints a and b, say, and going through a set of nodes $T \subseteq V - \{a, b\}$. We are looking for a Hamilton path from a to b through the remaining nodes in V, to close the Hamilton cycle. Initially $a = b$ is an arbitrary node, and $T = \emptyset$.

Branching is easy —we just choose how to extend the path by a new edge, say $[a, c]$, leading from a to a node $c \notin T$. This node c becomes the new value

of a in the subproblem (node b is always fixed throughout the algorithm). We leave the choice of the subproblem from which to branch unspecified (we pick any subproblem from $\mathcal{A}$). Finally, the test is the following (remember that in a subproblem we are looking for a path from a to b in a graph $G-T$, the original graph with the nodes in T deleted).

if $G-T-\{a,b\}$ is disconnected, or if $G-T$ has a degree-one node
other than a or b, return "subproblem is empty;"
if $G-T$ is a path from a to b, return "solution found;"
otherwise return "?"

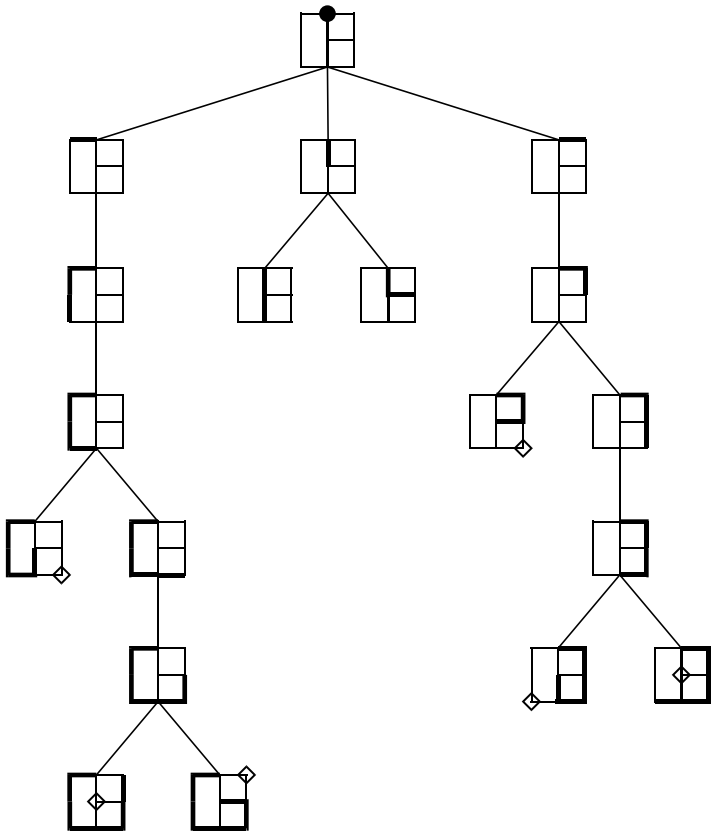

Figure 7-15: Execution of backtracking algorithm for HAMILTON CYCLE on the graph shown in the root. Initially both a and b coincide with the dotted node. In the leaf (backtracking) nodes the degree-one nodes are circled (in the middle leaves there are many choices). A total of nineteen subproblems is considered.

The application of this algorithm to a simple graph is shown in Figure

7-15. Although the number of partial solutions constructed may seem large (nineteen), it is minuscule compared to the number of solutions examined by the full-blown nondeterministic "algorithm" for the same instance (this number would be $(n-1)! = 5,040$). Needless to say, it is possible to devise more sophisticated and effective branching rules and tests than the one used here.$\diamondsuit$

Determining the best design decisions (1) through (3) depends a lot not only on the problem, but also on the kinds of instances of interest, and usually requires *extensive experimentation.*

Backtracking algorithms are of interest when solving a "yes-no" problem. For optimization problems one often uses an interesting variant of backtracking called **branch-and-bound.** In an optimization problem we can also think that we have an exponentially large set of candidate solutions; however, this time each solution has a **cost**† associated with it, and we wish to find the candidate solution in S_0 with the smallest cost. The branch-and-bound algorithm is in general the one shown below (the algorithm shown only returns the optimal cost, but it can be easily modified to return the optimal solution).

$\mathcal{A} := \{S_0\}$, bestsofar:= ∞
while $\mathcal{A}$ is not empty do
  choose a subproblem S and delete it from $\mathcal{A}$
  choose a way of branching out of S, say to subproblems $S_1, \ldots, S_r$
  for each subproblem S_i in this list do
    if $|S_i| = 1$ (that is, S_i is a complete solution) then update bestsofar
    else if lowerbound(S_i) <bestsofar then add S_i to $\mathcal{A}$
return bestsofar

The algorithm always remembers the smallest cost of any solution seen so far, initially ∞ (performance often improves a lot if bestsofar is initialized to the cost of a solution obtained by another heuristic). Every time a full solution to the original problem is found, bestsofar is updated. The key ingredient of a branch-and-bound algorithm (besides the design decisions (1) and (2) it shares with backtracking) is a method for obtaining a *lower bound* on the cost of any solution in a subproblem S. That is, the function lowerbound(S) returns a number that is guaranteed to be less than or equal to the lowest cost of any solution in S. The branch-and-bound algorithm above will always terminate with the optimal solution. This is because the only subproblems left unconsidered are those for which lowerbound(S_i) $\geq$bestsofar —that is, those subproblems of which the optimal solution is provably no better than the best solution we have seen so far.

† We shall assume that the optimization problem in question is a minimization problem; maximization problems can be treated in a very similar way.

Naturally, there are many ways of obtaining lower bounds (lowerbound$(S) = 0$ would usually do...). The point is that, if lowerbound(S) is a sophisticated algorithm returning a value that is usually very close to the optimum solution in S, then the branch-and-bound algorithm is likely to perform very well, that is, to terminate reasonably fast.

Example 7.4.7: Let us adapt the backtracking algorithm we developed for HAMILTON CYCLE to obtain a branch-and-bound algorithm for the TRAVELING SALESMAN PROBLEM. As before, a subproblem S is characterized by a path from a to b through a set T of cities. What is a reasonable lower bound? Here is one idea: For each city outside $T \cup \{a, b\}$, calculate the sum of its *two shortest distances* to another city outside T. For a and b, calculate their shortest distance to another city outside T. It is not hard to prove (see Problem 7.4.4) that the *half* sum of these numbers, plus the cost of the already fixed path from a to b through T, is a valid lower bound on the cost of any tour in the subproblem S. The branch-and-bound algorithm is now completely specified.

There are far more sophisticated lower bounds for the TRAVELING SALESMAN PROBLEM.$\diamondsuit$

Local Improvement

Our final family of algorithms is inspired by evolution: What if we allow a solution of an optimization problem to change a little, and adopt the new solution if it has improved cost? Concretely, let S_0 be the set of candidate solutions in an instance of an optimization problem (again, we shall assume that it is a minimization problem). Define a **neighborhood relation** N on the set of solutions $N \subseteq S_0 \times S_0$ —it captures the intuitive notion of "changing a little." For $s \in S_0$, the set $\{s' : (s, s') \in N\}$ is called the **neighborhood** of s.

The algorithm is simply this (see Figure 7-16 for a suggestive depiction of the operation of local improvement algorithms):

s :=initialsolution
while there is a solution s' such that
  $N(s, s')$ and cost(s') <cost(s) do: $s := s'$
return s

That is, the algorithm keeps improving s by replacing with a neighbor s' with a better cost, until there is no s' in the neighborhood of s with better cost; in the latter case we say that s *is a local optimum.* Obviously, a local optimum is not guaranteed to be an optimal solution —unless of course $N = S_0 \times S_0$. The quality of local optima obtained and the running time of the algorithm both depend critically on N: the larger the neighborhoods, the better the local optimum; on the other hand, large neighborhoods imply that the iteration of the algorithm (an execution of the while loop, and the ensuing search through

the neighborhood of the current solution s) will be slower. Local improvement algorithms seek a favorable compromise in this trade-off. As usual, there are no general principles to guide us in designing a good neighborhood; the choice seems very problem-dependent, even instance-dependent, and is best made through experimentation.

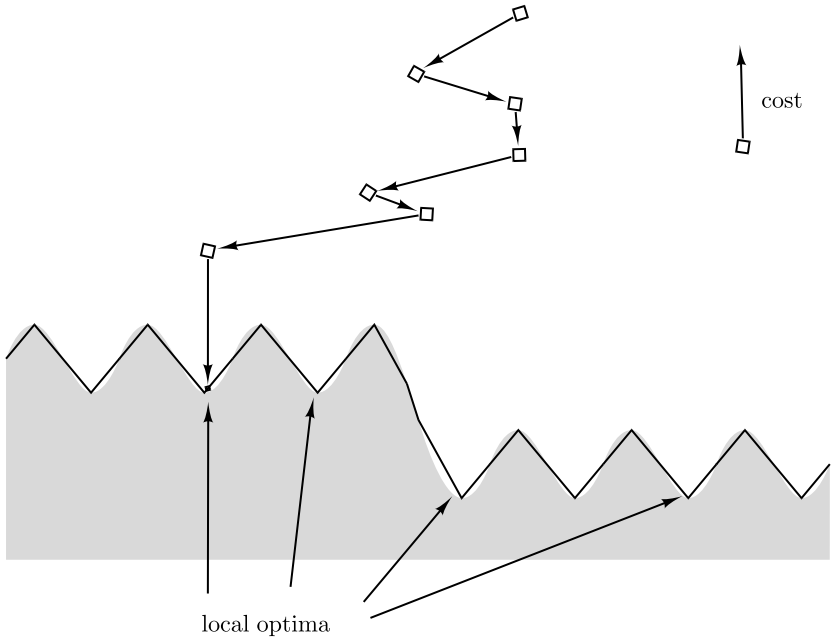

Figure 7-16: Once the neighborhood relation has been fixed, the solutions of an optimization problem can be pictured as an *energy landscape*, in which local optima are depicted as valleys. Local improvement heuristics jump from solution to solution, until a local optimum is found.

Another issue that affects the performance of a local improvement algorithm is the method used in finding s'. Do we adopt the first better solution we find in the neighborhood of s, or do we wait to find the best? Is the longer iteration justified by the speed of descent —and do we want speedy descent anyway? Finally, the performance of a local improvement algorithm also depends on the procedure initialsolution. It is not clear at all that better initial solutions will result in better performance —often a mediocre starting point is preferable, because it gives the algorithm more freedom to explore the solution space (see Figure 7-16). Incidentally, the procedure initialsolution should best be *randomized* —that is, able to generate different initial solutions when called many times. This allows us to *restart* many times the local improvement algorithm above, and obtain

many local optima.

Example 7.4.8: Let us take again the TRAVELING SALESMAN PROBLEM. When should we consider two tours as neighbors? Since a tour can be considered as a set of n undirected inter-city "links," one plausible answer is, *when they share all but very few links.* Two is the minimum possible number of links in which two tours may differ, and this suggests a well-known neighborhood relation for the TRAVELING SALESMAN PROBLEM that we call **2-change** (see Figure 7-17). That is, two tours are related by N if and only if they differ in just two links. The local improvement algorithm using the 2-change neighborhood performs reasonably well in practice. However, much better results are achieved by adopting the **3-change** neighborhood; furthermore, it is reported in the literature that **4-change** does not return sufficiently better tours to justify the increase in iteration time.

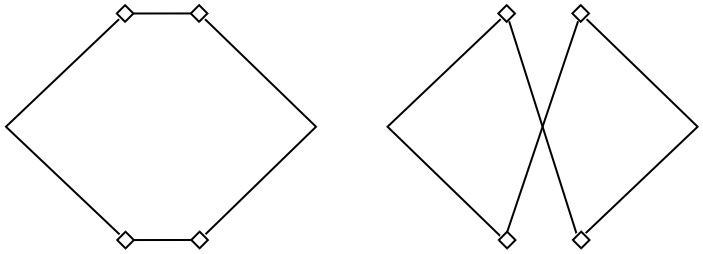

Figure 7-17

Perhaps the best heuristic algorithm currently known for the TRAVELING SALESMAN PROBLEM, the **Lin-Kernighan algorithm**, relies on **λ-change**, a neighborhood so sophisticated and complex that it does not even fit in our framework (whether two solutions are neighbors depends on the distances). As its name suggests, λ-change allows arbitrary many link changes in one step (but of course, not all possible such changes are explored, this would make the iteration exponentially slow).$\diamondsuit$

Example 7.4.9: In order to develop a local improvement algorithm for MAX SAT (the version of SATISFIABILITY in which we wish to satisfy as many clauses as possible; recall Theorem 7.2.4), we might choose to consider two truth assignments to be related by N if they only differ in the value of a single variable. This immediately defines an interesting, and empirically successful, local improvement algorithm for MAX SAT. It is apparently advantageous in this case to adopt as s' the best neighbor of s, instead of the first one found that is better than s. Also, it has been reported that it pays to make "lateral moves" (adopt a solution even if the inequality in the third line of the algorithm is not strict).

This heuristic is considered a very effective way of obtaining good solutions to MAX SAT, and is often used to solve SATISFIABILITY (in this use, it is hoped that in the end the algorithm will return a truth assignment that satisfies *all* clauses).$\diamondsuit$

An interesting twist on local improvement algorithms is a method called **simulated annealing.** As the name suggests, the inspiration comes from the physics of cooling solids. Simulated annealing allows the algorithm to "escape" from bad local optima (see Figure 7-18, and compare with 7-17) by performing occasional cost-increasing changes.

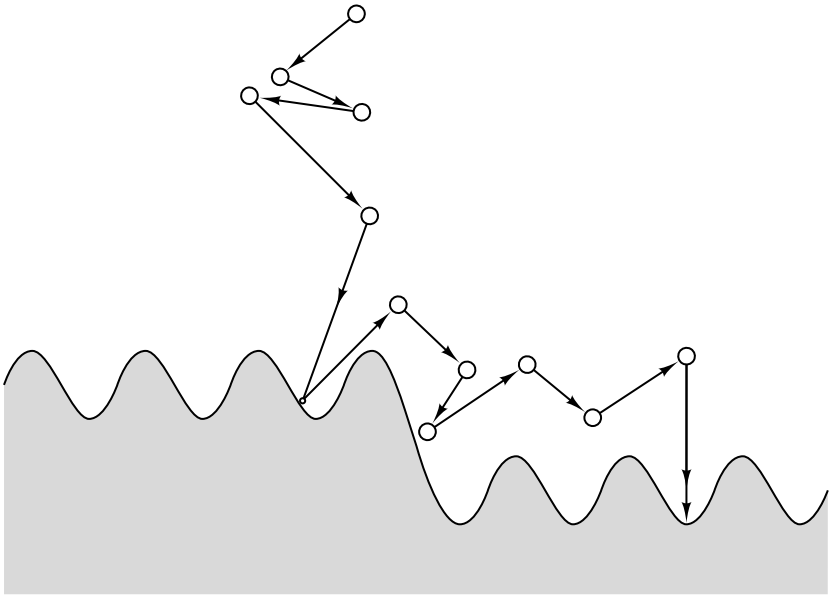

Figure 7-18: Simulated annealing has an advantage over the basic local improvement algorithm because its occasional cost-increasing moves help it avoid early convergence in a bad local optimum. This often comes at a great loss of efficiency.

s :=initialsolution, $T := T_0$
repeat
  generate a random solution s' such that $N(s, s')$,
    and let Δ :=cost(s')−cost(s)
  if $\Delta \leq 0$ then $s := s'$, else
    $s := s'$ with probability $e^{-\frac{\Delta}{T}}$
  update(T)
until $T = 0$

return the best solution seen

Intuitively, the probability that a cost-increasing change will be adopted is determined by the amount of the cost increase Δ, as well as by an important parameter T, the **temperature**. The higher the temperature, the more aggressively more expensive solutions are pursued. The way in which T is updated in the penultimate line of the algorithm —the *annealing schedule* of the algorithm, as it is called— is perhaps the most crucial design decision in these algorithms —besides, of course, the choice of neighborhood.

There are several other related genres of local improvement methods, many of them based, like the ones we described here, on some loose analogy with physical or biological systems (*genetic algorithms*, *neural networks,* etc.; see the references).

From the point of view of the formal criteria that we have developed in this book, the local improvement algorithms and their many variants are totally unattractive: They they do not in general return the optimum solution, they tend to have exponential worst-case complexity, and they are not even guaranteed to return solutions that are in any well-defined sense "close" to the optimum. Still, for many $\mathcal{NP}$-complete problems, in practice they often turn out to be the ones that perform best! Explaining and predicting the impressive empirical success of some of these algorithms is one of the most challenging frontiers of the theory of computation today.

Problems for Section 7.4

7.4.1. Give a polynomial algorithm for the DOMINATING SET problem (recall Problem 7.3.6) in the special case of trees (considered as symmetric directed graphs).

7.4.2. Suppose that all clauses in an instance of satisfiability contain *at most one positive literal;* such clauses are called **Horn clauses**. Show that, if all clauses of a Boolean formula are Horn clauses, then the satisfiability question for this formula can be settled in polynomial time. (*Hint:* When does a variable in a Horn formula *have* to be assigned $\top$?)

7.4.3. Show that the TRAVELING SALESMAN PROBLEM remains $\mathcal{NP}$-complete even if the distances are required to obey the triangle inequality. (*Hint:* Look back at our original proof that the TRAVELING SALESMAN PROBLEM is $\mathcal{NP}$-complete.)

7.4.4. Suppose that, in an instance of the traveling salesman problem with cities $1, 2, \ldots, n$ and distance matrix d_{ij}, we only consider tours that start from a, traverse by some path of length L the cities in a set $T \subseteq \{1, 2, \ldots, n\}$, end up in another city b, and then visit the remaining cities and return to a. Let us call this set of tours S.

(a) For each city $i \in \{1,2,\ldots,n\}-T-\{a,b\}$, let m_i be the sum of the smallest and next-to-smallest distances from i to another city in $\{1,2,\ldots,n\}-T-\{a,b\}$. Let s be the shortest distances from a to any city in $\{1,2,\ldots,n\}-T-\{a,b\}$, plus the corresponding shortest distance from b. Show that any tour in S has cost at least

$$L + \frac{1}{2}[\sum_{i \in \{1,2,\ldots,n\}-T-\{a,b\}} m_i + s].$$

That is, the formula above is a valid lower bound for S.
(b) The **minimum spanning tree** of the n cities is the smallest tree that has the cities as set of nodes; it can be computed very efficiently. Derive a better lower bound for S from this information.

7.4.5. How many 2-change neighbors does a tour of n cities have? How many 3-change neighbors? 4-change neighbors?

7.4.6. (a) Suppose that in the simulated annealing algorithm the temperature is kept at zero. Show that this is the basic local improvement algorithm.
(b) What is the simulated annealing algorithm with the temperature kept at infinity?
(c) Suppose now that the temperature is zero for a few iterations, then infinity for a few, then zero again, etc. How is the resulting algorithm related to the basic version of local improvement?

REFERENCES

Stephen A. Cook was the first to exhibit an $\mathcal{NP}$-complete language in his paper

- S. A. Cook "The Complexity of Theorem-Proving Procedures," *Proceedings of the Third Annual ACM Symposium on the Theory of Computing* pp. 151–158). New York: Association for Computing Machinery, 1971.

Richard M. Karp established the scope and importance of $\mathcal{NP}$-completeness in his paper

- R. M. Karp "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, (pp. 85–104), ed. R. E. Miller and J. W. Thatcher. New York: Plenum Press, 1972,

where, among a host of other results, Theorems 7.3.1–7.3.7, and the results in problems 7.3.4 and 7.3.6, are proved. $\mathcal{NP}$-completeness was independently discovered by Leonid Levin in

- L. A. Levin "Universal Sorting Problems," *Problemi Peredachi Informatsii,* 9, 3, pp. 265–266 (in Russian), 1973.

The following book contains a useful catalog of over 300 $\mathcal{NP}$-complete problems from many and diverse areas; many more problems have been proved $\mathcal{NP}$-complete since its appearence.

- M. R. Garey and D. S. Johnson *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness,* New York: Freeman, 1979.

This book is also an early source of information on complexity as it applies to concrete problems, as well as on approximation algorithms. For much more recent and extensive treatment of this latter subject see

- D. Hochbaum (ed.) *Approximation Algorithms for $\mathcal{NP}$-hard Problems,* Boston, Mass: PWS Publishers, 1996,

and for more information about other ways of coping with $\mathcal{NP}$-completeness see, for example,

- C. R. Reeves, (ed.) *Modern Heuristic Techniques for Combinatorial Problems,* New York: John Wiley, 1993, and
- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity* Englewood Cliffs, N.J.: Prentice-Hall, 1982; second edition, New York: Dover, 1997.