Chapter 9

PSPACE: A Class of Problems beyond NP

Throughout the book, one of the main issues has been the notion of *time* as a computational resource. It was this notion that formed the basis for adopting *polynomial time* as our working definition of efficiency; and, implicitly, it underlies the distinction between $\mathcal{P}$ and $\mathcal{NP}$. To some extent, we have also been concerned with the *space* (i.e., memory) requirements of algorithms. In this chapter, we investigate a class of problems defined by treating space as the fundamental computational resource. In the process, we develop a natural class of problems that appear to be even harder than $\mathcal{NP}$ and co-$\mathcal{NP}$.

9.1 PSPACE

The basic class we study is PSPACE, the set of all problems that can be solved by an algorithm with polynomial space complexity—that is, an algorithm that uses an amount of *space* that is polynomial in the size of the input.

We begin by considering the relationship of PSPACE to classes of problems we have considered earlier. First of all, in polynomial time, an algorithm can consume only a polynomial amount of space; so we can say

(9.1) $\mathcal{P} \subseteq PSPACE$.

But PSPACE is much broader than this. Consider, for example, an algorithm that just counts from 0 to $2^n - 1$ in base-2 notation. It simply needs to implement an n-bit counter, which it maintains in exactly the same way one increments an odometer in a car. Thus this algorithm runs for an exponential amount of time, and then halts; in the process, it has used only a polynomial amount of space. Although this algorithm is not doing anything particularly

interesting, it illustrates an important principle: Space can be reused during a computation in ways that time, by definition, cannot.

Here is a more striking application of this principle.

(9.2) *There is an algorithm that solves 3-SAT using only a polynomial amount of space.*

Proof. We simply use a brute-force algorithm that tries all possible truth assignments; each assignment is plugged into the set of clauses to see if it satisfies them. The key is to implement this all in polynomial space.

To do this, we increment an n-bit counter from 0 to $2^n - 1$ just as described above. The values in the counter correspond to truth assignments in the following way: When the counter holds a value q, we interpret it as a truth assignment v that sets x_i to be the value of the i^{th} bit of q.

Thus we devote a polynomial amount of space to enumerating all possible truth assignments v. For each truth assignment, we need only polynomial space to plug it into the set of clauses and see if it satisfies them. If it does satisfy the clauses, we can stop the algorithm immediately. If it doesn't, we delete the intermediate work involved in this "plugging in" operation and *reuse* this space for the next truth assignment. Thus we spend only polynomial space cumulatively in checking all truth assignments; this completes the bound on the algorithm's space requirements. ■

Since 3-SAT is an NP-complete problem, (9.2) has a significant consequence.

(9.3) $\mathcal{NP} \subseteq$ *PSPACE.*

Proof. Consider an arbitrary problem Y in $\mathcal{NP}$. Since $Y \leq_P$ 3-SAT, there is an algorithm that solves Y using a polynomial number of steps plus a polynomial number of calls to a black box for 3-SAT. Using the algorithm in (9.2) to implement this black box, we obtain an algorithm for Y that uses only polynomial space. ■

Just as with the class $\mathcal{P}$, a problem X is in PSPACE if and only if its complementary problem $\overline{X}$ is in PSPACE as well. Thus we can conclude that co-$\mathcal{NP} \subseteq$ PSPACE. We draw what is known about the relationships among these classes of problems in Figure 9.1.

Given that PSPACE is an enormously large class of problems, containing both $\mathcal{NP}$ and co-$\mathcal{NP}$, it is very likely that it contains problems that cannot be solved in polynomial time. But despite this widespread belief, amazingly

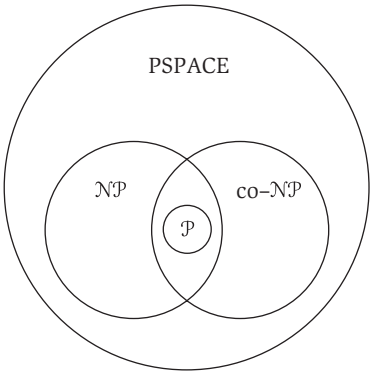

Figure 9.1 The subset relationships among various classes of problems. Note that we don't know how to prove the conjecture that all of these classes are different from one another.

it has not been proven that $\mathcal{P} \neq$ PSPACE. Nevertheless, the nearly universal conjecture is that PSPACE contains problems that are not even in $\mathcal{NP}$ or co-$\mathcal{NP}$.

9.2 Some Hard Problems in PSPACE

We now survey some natural examples of problems in PSPACE that are not known—and not believed—to belong to $\mathcal{NP}$ or co-$\mathcal{NP}$.

As was the case with $\mathcal{NP}$, we can try understanding the structure of PSPACE by looking for *complete problems*—the hardest problems in the class. We will say that a problem X is PSPACE-*complete* if (i) it belongs to PSPACE; and (ii) for all problems Y in PSPACE, we have $Y \leq_P X$.

It turns out, analogously to the case of $\mathcal{NP}$, that a wide range of natural problems are PSPACE-complete. Indeed, a number of the basic problems in artificial intelligence are PSPACE-complete, and we describe three genres of these here.

Planning

Planning problems seek to capture, in a clean way, the task of interacting with a complex environment to achieve a desired set of goals. Canonical applications include large logistical operations that require the movement of people, equipment, and materials. For example, as part of coordinating a disaster-relief effort, we might decide that twenty ambulances are needed at a particular high-altitude location. Before this can be accomplished, we need to get ten snowplows to clear the road; this in turn requires emergency fuel and snowplow crews; but if we use the fuel for the snowplows, then we may not have enough for the ambulances; and . . . you get the idea. Military operations

also require such reasoning on an enormous scale, and automated planning techniques from artificial intelligence have been used to great effect in this domain as well.

One can see very similar issues at work in complex solitaire games such as Rubik's Cube or the *fifteen-puzzle*—a 4×4 grid with fifteen movable tiles labeled $1, 2, \ldots, 15$, and a single *hole*, with the goal of moving the tiles around so that the numbers end up in ascending order. (Rather than ambulances and snowplows, we now are worried about things like getting the tile labeled 6 one position to the left, which involves getting the 11 out of the way; but that involves moving the 9, which was actually in a good position; and so on.) These toy problems can be quite tricky and are often used in artificial intelligence as a test-bed for planning algorithms.

Having said all this, how should we define the problem of planning in a way that's general enough to include each of these examples? Both solitaire puzzles and disaster-relief efforts have a number of abstract features in common: There are a number of *conditions* we are trying to achieve and a set of allowable *operators* that we can apply to achieve these conditions. Thus we model the environment by a set $\mathcal{C} = \{C_1, \ldots, C_n\}$ of *conditions*: A given state of the world is specified by the subset of the conditions that currently hold. We interact with the environment through a set $\{\mathcal{O}_1, \ldots, \mathcal{O}_k\}$ of *operators*. Each operator $\mathcal{O}_i$ is specified by a *prerequisite list*, containing a set of conditions that must hold for $\mathcal{O}_i$ to be invoked; an *add list*, containing a set of conditions that will become true after $\mathcal{O}_i$ is invoked; and a *delete list*, containing a set of conditions that will cease to hold after $\mathcal{O}_i$ is invoked. For example, we could model the fifteen-puzzle by having a condition for each possible location of each tile, and an operator to move each tile between each pair of adjacent locations; the prerequisite for an operator is that its two locations contain the designated tile and the hole.

The problem we face is the following: Given a set $\mathcal{C}_0$ of *initial conditions*, and a set $\mathcal{C}^*$ of *goal conditions*, is it possible to apply a sequence of operators beginning with $\mathcal{C}_0$ so that we reach a situation in which precisely the conditions in $\mathcal{C}^*$ (and no others) hold? We will call this an instance of the *Planning Problem*.

Quantification

We have seen, in the 3-SAT problem, some of the difficulty in determining whether a set of disjunctive clauses can be simultaneously satisfied. When we add quantifiers, the problem appears to become even more difficult.

Let $\Phi(x_1, \ldots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each C_i is a disjunction of three terms (in other words, it is an instance of 3-SAT). Assume for simplicity that n is an odd number, and suppose we ask

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

That is, we wish to know whether there is a choice for x_1, so that for both choices of x_2, there is a choice for x_3, and so on, so that Φ is satisfied. We will refer to this decision problem as *Quantified 3-SAT* (or, briefly, QSAT).

The original 3-SAT problem, by way of comparison, simply asked

$$\exists x_1 \exists x_2 \cdots \exists x_{n-2} \exists x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

In other words, in 3-SAT it was sufficient to look for a single setting of the Boolean variables.

Here's an example to illustrate the kind of reasoning that underlies an instance of QSAT. Suppose that we have the formula

$$\Phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

and we ask

$$\exists x_1 \forall x_2 \exists x_3 \Phi(x_1, x_2, x_3)?$$

The answer to this question is yes: We can set x_1 so that for both choices of x_2, there is a way to set x_3 so that Φ is satisfied. Specifically, we can set $x_1 = 1$; then if x_2 is set to 1, we can set x_3 to 0 (satisfying all clauses); and if x_2 is set to 0, we can set x_3 to 1 (again satisfying all clauses).

Problems of this type, with a sequence of quantifiers, arise naturally as a form of *contingency planning*—we wish to know whether there is a decision we can make (the choice of x_1) so that for all possible responses (the choice of x_2) there is a decision we can make (the choice of x_3), and so forth.

Games

In 1996 and 1997, world chess champion Garry Kasparov was billed by the media as the defender of the human race, as he faced IBM's program Deep Blue in two chess matches. We needn't look further than this picture to convince ourselves that computational game-playing is one of the most visible successes of contemporary artificial intelligence.

A large number of two-player games fit naturally into the following framework. Players alternate moves, and the first one to achieve a specific goal wins. (For example, depending on the game, the goal could be capturing the king, removing all the opponent's checkers, placing four pieces in a row, and so on.) Moreover, there is often a natural, polynomial, upper bound on the maximum possible length of a game.

The Competitive Facility Location Problem that we introduced in Chapter 1 fits naturally within this framework. (It also illustrates the way in which games can arise not just as pastimes, but through competitive situations in everyday life.) Recall that in Competitive Facility Location, we are given a graph G, with a nonnegative *value* b_i attached to each node i. Two players alternately select nodes of G, so that the set of selected nodes at all times forms an independent set. Player 2 wins if she ultimately selects a set of nodes of total value at least B, for a given bound B; Player 1 wins if he prevents this from happening. The question is: Given the graph G and the bound B, is there a strategy by which Player 2 can force a win?

9.3 Solving Quantified Problems and Games in Polynomial Space

We now discuss how to solve all of these problems in polynomial space. As we will see, this will be trickier—in one case, a lot trickier—than the (simple) task we faced in showing that problems like 3-SAT and Independent Set belong to $\mathcal{NP}$.

We begin here with QSAT and Competitive Facility Location, and then consider Planning in the next section.

Designing an Algorithm for QSAT

First let's show that QSAT can be solved in polynomial space. As was the case with 3-SAT, the idea will be to run a brute-force algorithm that reuses space carefully as the computation proceeds.

Here is the basic brute-force approach. To deal with the first quantifier $\exists x_1$, we consider both possible values for x_1 in sequence. We first set $x_1 = 0$ and see, recursively, whether the remaining portion of the formula evaluates to 1. We then set $x_1 = 1$ and see, recursively, whether the remaining portion of the formula evaluates to 1. The full formula evaluates to 1 if and only if *either* of these recursive calls yields a 1—that's simply the definition of the $\exists$ quantifier.

This is essentially a divide-and-conquer algorithm, which, given an input with n variables, spawns two recursive calls on inputs with $n - 1$ variables each. If we were to save all the work done in both these recursive calls, our space usage $S(n)$ would satisfy the recurrence

$$S(n) \leq 2S(n-1) + p(n),$$

where $p(n)$ is a polynomial function. This would result in an exponential bound, which is too large.

Fortunately, we can perform a simple optimization that greatly reduces the space usage. When we're done with the case $x_1 = 0$, all we really need to save is the single bit that represents the outcome of the recursive call; we can throw away all the other intermediate work. This is another example of "reuse"—we're reusing the space from the computation for $x_1 = 0$ in order to compute the case $x_1 = 1$.

Here is a compact description of the algorithm.

```
If the first quantifier is ∃ x_i then
  Set x_i = 0 and recursively evaluate the quantified expression
                    over the remaining variables
  Save the result (0 or 1) and delete all other intermediate work
  Set x_i = 1 and recursively evaluate the quantified expression
                    over the remaining variables
  If either outcome yielded an evaluation of 1, then
     return 1
  Else return 0
  Endif
If the first quantifier is ∀ x_i then
  Set x_i = 0 and recursively evaluate the quantified expression
                    over the remaining variables
  Save the result (0 or 1) and delete all other intermediate work
  Set x_i = 1 and recursively evaluate the quantified expression
                    over the remaining variables
  If both outcomes yielded an evaluation of 1, then
     return 1
  Else return 0
  Endif
Endif
```

Analyzing the Algorithm

Since the recursive calls for the cases $x_1 = 0$ and $x_1 = 1$ overwrite the same space, our space requirement $S(n)$ for an n-variable problem is simply a polynomial in n plus the space requirement for one recursive call on an $(n-1)$-variable problem:

$$S(n) \leq S(n-1) + p(n),$$

where again $p(n)$ is a polynomial function. Unrolling this recurrence, we get

$$S(n) \leq p(n) + p(n-1) + p(n-2) + \cdots + p(1) \leq n \cdot p(n).$$

Since $p(n)$ is a polynomial, so is $n \cdot p(n)$, and hence our space usage is polynomial in n, as desired.

In summary, we have shown the following.

(9.4) *QSAT can be solved in polynomial space.*

Extensions: An Algorithm for Competitive Facility Location

We can determine which player has a forced win in a game such as Competitive Facility Location by a very similar type of algorithm.

Suppose Player 1 moves first. We consider all of his possible moves in sequence. For each of these moves, we see who has a forced win in the resulting game, with Player 2 moving first. If Player 1 has a forced win in any of them, then Player 1 has a forced win from the initial position. The crucial point, as in the QSAT algorithm, is that we can reuse the space from one candidate move to the next; we need only store the single bit representing the outcome. In this way, we only consume a polynomial amount of space plus the space requirement for one recursive call on a graph with fewer nodes. As in the case of QSAT, we get the recurrence

$$S(n) \leq S(n-1) + p(n)$$

for a polynomial $p(n)$.

In summary, we have shown the following.

(9.5) *Competitive Facility Location can be solved in polynomial space.*

9.4 Solving the Planning Problem in Polynomial Space

Now we consider how to solve the basic Planning Problem in polynomial space. The issues here will look quite different, and it will turn out to be a much more difficult task.

The Problem

Recall that we have a set of *conditions* $\mathcal{C} = \{C_1, \ldots, C_n\}$ and a set of *operators* $\{\mathcal{O}_1, \ldots, \mathcal{O}_k\}$. Each operator $\mathcal{O}_i$ has a *prerequisite list* P_i, an *add list* A_i, and a *delete list* D_i. Note that $\mathcal{O}_i$ can still be applied even if conditions other than those in P_i are present; and it does not affect conditions that are not in A_i or D_i.

We define a *configuration* to be a subset $\mathcal{C}' \subseteq \mathcal{C}$; the state of the Planning Problem at any given time can be identified with a unique configuration $\mathcal{C}'$

consisting precisely of the conditions that hold at that time. For an initial configuration $\mathcal{C}_0$ and a goal configuration $\mathcal{C}^*$, we wish to determine whether there is a sequence of operators that will take us from $\mathcal{C}_0$ to $\mathcal{C}^*$.

We can view our Planning instance in terms of a giant, implicitly defined, directed graph $\mathcal{G}$. There is a node of $\mathcal{G}$ for each of the 2^n possible configurations (i.e., each possible subset of $\mathcal{C}$); and there is an edge of $\mathcal{G}$ from configuration $\mathcal{C}'$ to configuration $\mathcal{C}''$ if, in one step, one of the operators can convert $\mathcal{C}'$ to $\mathcal{C}''$. In terms of this graph, the Planning Problem has a very natural formulation: Is there a path in $\mathcal{G}$ from $\mathcal{C}_0$ to $\mathcal{C}^*$? Such a path corresponds precisely to a sequence of operators leading from $\mathcal{C}_0$ to $\mathcal{C}^*$.

It's possible for a Planning instance to have a short solution (as in the example of the fifteen-puzzle), but this need not hold in general. That is, there need not always be a short path in $\mathcal{G}$ from $\mathcal{C}_0$ to $\mathcal{C}^*$. This should not be so surprising, since $\mathcal{G}$ has an exponential number of nodes. But we must be careful in applying this intuition, since $\mathcal{G}$ has a special structure: It is defined very compactly in terms of the n conditions and k operators.

(9.6) *There are instances of the Planning Problem with n conditions and k operators for which there exists a solution, but the shortest solution has length $2^n - 1$.*

Proof. We give a simple example of such an instance; it essentially encodes the task of incrementing an n-bit counter from the all-zeros state to the all-ones state.

- We have conditions $C_1, C_2, \ldots, C_n$.
- We have operators $\mathcal{O}_i$ for $i = 1, 2, \ldots, n$.
- $\mathcal{O}_1$ has no prerequisites or delete list; it simply adds C_1.
- For $i > 1$, $\mathcal{O}_i$ requires C_j for all $j < i$ as prerequisites. When invoked, it adds C_i and deletes C_j for all $j < i$.

Now we ask: Is there a sequence of operators that will take us from $\mathcal{C}_0 = \phi$ to $\mathcal{C}^* = \{C_1, C_2, \ldots, C_n\}$?

We claim the following, by induction on i:

From any configuration that does not contain C_j for any $j \leq i$, there exists a sequence of operators that reaches a configuration containing C_j for all $j \leq i$; but any such sequence has at least $2^i - 1$ steps.

This is clearly true for $i = 1$. For larger i, here's one solution.

- By induction, achieve conditions $\{C_{i-1}, \ldots, C_1\}$ using operators $\mathcal{O}_1, \ldots, \mathcal{O}_{i-1}$.
- Now invoke operator $\mathcal{O}_i$, adding C_i but deleting everything else.

- Again, by induction, achieve conditions $\{C_{i-1}, \ldots, C_1\}$ using operators $\mathcal{O}_1, \ldots, \mathcal{O}_{i-1}$. Note that condition C_i is preserved throughout this process.

Now we take care of the other part of the inductive step—that *any* such sequence requires at least $2^i - 1$ steps. So consider the first moment when C_i is added. At this step, $C_{i-1}, \ldots, C_1$ must have been present, and by induction, this must have taken at least $2^{i-1} - 1$ steps. C_i can only be added by $\mathcal{O}_i$, which deletes all C_j for $j < i$. Now we have to achieve conditions $\{C_{i-1}, \ldots, C_1\}$ again; this will take another $2^{i-1} - 1$ steps, by induction, for a total of at least $2(2^{i-1} - 1) + 1 = 2^i - 1$ steps.

The overall bound now follows by applying this claim with $i = n$. ■

Of course, if every "yes" instance of Planning had a polynomial-length solution, then Planning would be in $\mathcal{NP}$—we could just exhibit the solution. But (9.6) shows that the shortest solution is not necessarily a good certificate for a Planning instance, since it can have a length that is exponential in the input size.

However, (9.6) describes essentially the worst case, for we have the following matching upper bound. The graph $\mathcal{G}$ has 2^n nodes, and if there is a path from $\mathcal{C}_0$ to $\mathcal{C}^*$, then the shortest such path does not visit any node more than once. As a result, the shortest path can take at most $2^n - 1$ steps after leaving $\mathcal{C}_0$.

(9.7) *If a Planning instance with n conditions has a solution, then it has one using at most $2^n - 1$ steps.*

Designing the Algorithm

We've seen that the shortest solution to the Planning Problem may have length exponential in n, which is bad news: After all, this means that in polynomial space, we can't even store an explicit representation of the solution. But this fact doesn't necessarily close out our hopes of solving an arbitrary instance of Planning using only polynomial space. It's possible that there could be an algorithm that decides the answer to an instance of Planning without ever being able to survey the entire solution at once.

In fact, we now show that this is the case: we design an algorithm to solve Planning in polynomial space.

Some Exponential Approaches To get some intuition about this problem, we first consider the following brute-force algorithm to solve the Planning instance. We build the graph $\mathcal{G}$ and use any graph connectivity algorithm—depth-first search or breadth-first search—to decide whether there is a path from $\mathcal{C}_0$ to $\mathcal{C}^*$.

Of course, this algorithm is too brute-force for our purposes; it takes exponential space just to construct the graph $\mathcal{G}$. We could try an approach in which we never actually build $\mathcal{G}$, and just simulate the behavior of depth-first search or breadth-first search on it. But this likewise is not feasible. Depth-first search crucially requires us to maintain a list of all the nodes in the current path we are exploring, and this can grow to exponential size. Breadth-first requires a list of all nodes in the current "frontier" of the search, and this too can grow to exponential size.

We seem stuck. Our problem is transparently equivalent to finding a path in $\mathcal{G}$, and all the standard path-finding algorithms we know are too lavish in their use of space. Could there really be a fundamentally different path-finding algorithm out there?

A More Space-Efficient Way to Construct Paths In fact, there is a fundamentally different kind of path-finding algorithm, and it has just the properties we need. The basic idea, proposed by Savitch in 1970, is a clever use of the divide-and-conquer principle. It subsequently inspired the trick for reducing the space requirements in the Sequence Alignment Problem; so the overall approach may remind you of what we discussed there, in Section 6.7. Our plan, as before, is to find a clever way to reuse space, admittedly at the expense of increasing the time spent. Neither depth-first search nor breadth-first search is nearly aggressive enough in its reuse of space; both need to maintain a large history at all times. We need a way to solve half the problem, throw away almost all the intermediate work, and then solve the other half of the problem.

The key is a procedure that we will call $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$. It determines whether there is a sequence of operators, *consisting of at most L steps*, that leads from configuration $\mathcal{C}_1$ to configuration $\mathcal{C}_2$. So our initial problem is to determine the result (yes or no) of $\texttt{Path}(\mathcal{C}_0, \mathcal{C}^*, 2^n)$. Breadth-first search can be viewed as the following dynamic programming implementation of this procedure: To determine $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$, we first determine all $\mathcal{C}'$ for which $\texttt{Path}(\mathcal{C}_1, \mathcal{C}', L-1)$ holds; we then see, for each such $\mathcal{C}'$, whether any operator leads directly from $\mathcal{C}'$ to $\mathcal{C}_2$.

This indicates some of the wastefulness, in terms of space, that breadth-first search entails. We are generating a huge number of intermediate configurations just to reduce the parameter L by one. More effective would be to try determining whether there is any configuration $\mathcal{C}'$ that could serve as the *midpoint* of a path from $\mathcal{C}_1$ to $\mathcal{C}_2$. We could first generate all possible midpoints $\mathcal{C}'$. For each $\mathcal{C}'$, we then check recursively whether we can get from $\mathcal{C}_1$ to $\mathcal{C}'$ in at most $L/2$ steps; and also whether we can get from $\mathcal{C}'$ to $\mathcal{C}_2$ in at most $L/2$ steps. This involves two recursive calls, but we care only about the yes/no outcome of each; other than this, we can reuse space from one to the next.

Does this really reduce the space usage to a polynomial amount? We first write down the procedure carefully, and then analyze it. We will think of L as a power of 2, which it is for our purposes.

```
Path(C1, C2, L)
  If L = 1 then
    If there is an operator O converting C1 to C2 then
      return ''yes''
    Else
      return ''no''
    Endif
  Else (L > 1)
    Enumerate all configurations C' using an n-bit counter
    For each C' do the following:
       Compute x = Path(C1, C', ⌈L/2⌉)
       Delete all intermediate work, saving only the return value x
       Compute y = Path(C', C2, ⌈L/2⌉)
       Delete all intermediate work, saving only the return value y
       If both x and y are equal to ''yes'', then return ''yes''
    Endfor
    If ''yes'' was not returned for any C' then
       Return ''no''
    Endif
  Endif
```

Again, note that this procedure solves a generalization of our original question, which simply asked for $\texttt{Path}(\mathcal{C}_0, \mathcal{C}^*, 2^n)$. This does mean, however, that we should remember to view L as an exponentially large parameter: $\log L = n$.

Analyzing the Algorithm

The following claim therefore implies that Planning can be solved in polynomial space.

(9.8) $\texttt{Path}(\mathcal{C}_1, \mathcal{C}_2, L)$ *returns "yes" if and only if there is a sequence of operators of length at most L leading from* $\mathcal{C}_1$ *to* $\mathcal{C}_2$. *Its space usage is polynomial in n, k, and* $\log L$.

Proof. The correctness follows by induction on L. It clearly holds when $L = 1$, since all operators are considered explicitly. Now consider a larger value of L. If there is a sequence of operators from $\mathcal{C}_1$ to $\mathcal{C}_2$, of length $L' \leq L$, then there is a configuration $\mathcal{C}'$ that occurs at position $\lceil L'/2 \rceil$ in this sequence. By

induction, `Path`$(\mathcal{C}_1, \mathcal{C}', \lceil L/2 \rceil)$ and `Path`$(\mathcal{C}', \mathcal{C}_2, \lceil L/2 \rceil)$ will both return "yes," and so `Path`$(\mathcal{C}_1, \mathcal{C}_2, L)$ will return "yes." Conversely, if there is a configuration $\mathcal{C}'$ so that `Path`$(\mathcal{C}_1, \mathcal{C}', \lceil L/2 \rceil)$ and `Path`$(\mathcal{C}', \mathcal{C}_2, \lceil L/2 \rceil)$ both return "yes," then the induction hypothesis implies that there exist corresponding sequences of operators; concatenating these two sequences, we obtain a sequence of operators from $\mathcal{C}_1$ to $\mathcal{C}_2$ of length at most L.

Now we consider the space requirements. Aside from the space spent inside recursive calls, each invocation of `Path` involves an amount of space polynomial in n, k, and $\log L$. But at any given point in time, only a single recursive call is active, and the intermediate work from all other recursive calls has been deleted. Thus, for a polynomial function p, the space requirement $S(n, k, L)$ satisfies the recurrence

$$S(n, k, L) \leq p(n, k, \log L) + S(n, k, \lceil L/2 \rceil).$$
$$S(n, k, 1) \leq p(n, k, 1).$$

Unwinding the recurrence for $O(\log L)$ levels, we obtain the bound $S(n, k, L) = O(\log L \cdot p(n, k, \log L))$, which is a polynomial in n, k, and $\log L$. ■

If dynamic programming has an opposite, this is it. Back when we were solving problems by dynamic programming, the fundamental principle was to save all the intermediate work, so you don't have to recompute it. Now that conserving space is our goal, we have just the opposite priorities: throw away all the intermediate work, since it's just taking up space and it can always be recomputed.

As we saw when we designed the space-efficient Sequence Alignment Algorithm, the best strategy often lies somewhere in between, motivated by these two approaches: throw away some of the intermediate work, but not so much that you blow up the running time.

9.5 Proving Problems PSPACE-Complete

When we studied $\mathcal{NP}$, we had to prove a *first* problem NP-complete directly from the definition of $\mathcal{NP}$. After Cook and Levin did this for Satisfiability, many other NP-complete problems could follow by reduction.

A similar sequence of events followed for PSPACE, shortly after the results for $\mathcal{NP}$. Recall that we defined PSPACE-completeness, by direct analogy with NP-completeness, in Section 9.1. The natural analogue of Circuit Satisfiability and 3-SAT for PSPACE is played by QSAT, and Stockmeyer and Meyer (1973) proved

(9.9) *QSAT is PSPACE-complete.*

This basic PSPACE-complete problem can then serve as a good "root" from which to discover other PSPACE-complete problems. By strict analogy with the case of $\mathcal{NP}$, it's easy to see from the definition that if a problem Y is PSPACE-complete, and a problem X in PSPACE has the property that $Y \leq_P X$, then X is PSPACE-complete as well.

Our goal in this section is to show an example of such a PSPACE-completeness proof, for the case of the Competitive Facility Location Problem; we will do this by reducing QSAT to Competitive Facility Location. In addition to establishing the hardness of Competitive Facility Location, the reduction also gives a sense for how one goes about showing PSPACE-completeness results for games in general, based on their close relationship to quantifiers.

We note that Planning can also be shown to be PSPACE-complete by a reduction from QSAT, but we will not go through that proof here.

Relating Quantifiers and Games

It is actually not surprising at all that there should be a close relation between quantifiers and games. Indeed, we could have equivalently defined QSAT as the problem of deciding whether the first player has a forced win in the following *Competitive 3-SAT* game. Suppose we fix a formula $\Phi(x_1, \ldots, x_n)$ consisting, as in QSAT, of a conjunction of length-3 clauses. Two players alternate turns picking values for variables: the first player picks the value of x_1, then the second player picks the value of x_2, then the first player picks the value of x_3, and so on. We will say that the first player wins if $\Phi(x_1, \ldots, x_n)$ ends up evaluating to 1, and the second player wins if it ends up evaluating to 0.

When does the first player have a forced win in this game (i.e., when does our instance of Competitive 3-SAT have a yes answer)? Precisely when there is a choice for x_1 so that for all choices of x_2 there is a choice for x_3 so that . . . and so on, resulting in $\Phi(x_1, \ldots, x_n)$ evaluating to 1. That is, the first player has a forced win if and only if (assuming n is an odd number)

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n).$$

In other words, our Competitive 3-SAT game is directly equivalent to the instance of QSAT defined by the same Boolean formula Φ, and so we have proved the following.

(9.10) *QSAT $\leq_P$ Competitive 3-SAT and Competitive 3-SAT $\leq_P$ QSAT.*

Proving Competitive Facility Location is PSPACE-Complete

Statement (9.10) moves us into the world of games. We use this connection to establish the PSPACE-completeness of Competitive Facility Location.

(9.11) *Competitive Facility Location is PSPACE-complete.*

Proof. We have already shown that Competitive Facility Location is in PSPACE. To prove it is PSPACE-complete, we now show that Competitive 3-SAT $\leq_P$ Competitive Facility Location. Combined with the fact that QSAT $\leq_P$ Competitive 3-SAT, this will show that QSAT $\leq_P$ Competitive Facility Location and hence will establish the PSPACE-completeness result.

We are given an instance of Competitive 3-SAT, defined by a formula Φ. Φ is the conjunction of clauses

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k;$$

each C_j has length 3 and can be written $C_j = t_{j1} \vee t_{j2} \vee t_{j3}$. As before, we will assume that there is an odd number n of variables. We will also assume, quite naturally, that no clause contains both a term and its negation; after all, such a clause would be automatically satisfied by any truth assignment. We must show how to encode this Boolean structure in the graph that underlies Competitive Facility Location.

We can picture the instance of Competitive 3-SAT as follows. The players alternately select values in a truth assignment, beginning and ending with Player 1; at the end, Player 2 has won if she can select a clause C_j in which none of the terms has been set to 1. Player 1 has won if Player 2 cannot do this.

It is this notion that we would like to encode in an instance of Competitive Facility Location: that the players alternately make a fixed number of moves, in a highly constrained fashion, and then there's a final chance by Player 2 to win the whole thing. But in its general form, Competitive Facility Location looks much more wide-open than this. Whereas the players in Competitive 3-SAT must set one variable at a time, in order, the players in Competitive Facility Location can jump all over the graph, choosing nodes wherever they want.

Our fundamental trick, then, will be to use the values b_i on the nodes to tightly constrain where the players can move, under any "reasonable" strategy. In other words, we will set things up so that if the either of the players deviates from a particular narrow course, he or she will lose instantly.

As with our more complicated NP-completeness reductions in Chapter 8, the construction will have gadgets to represent assignments to the variables, and further gadgets to represent the clauses. Here is how we encode the variables. For each variable x_i, we define two nodes v_i, v_i' in the graph G, and include an edge (v_i, v_i'), as in Figure 9.2. Selecting v_i will represent setting $x_i = 1$; selecting v_i' will represent $x_i = 0$. The constraint that the chosen variables

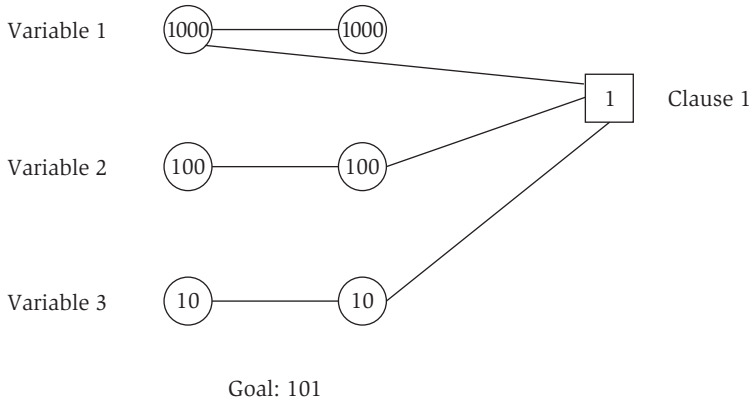

Figure 9.2 The reduction from Competitive 3-SAT to Competitive Facility Location.

must form an independent set naturally prevents both v_i and v'_i from being chosen. At this point, we do not define any other edges.

How do we get the players to set the variables in order—first x_1, then x_2, and so forth? We place values on v_1 and v'_1 so high that Player 1 will lose instantly if he does not choose them. We place somewhat lower values on v_2 and v'_2, and continue in this way. Specifically, for a value $c \geq k+2$, we define the node values b_{v_i} and $b_{v'_i}$ to be c^{1+n-i}. We define the bound that Player 2 is trying to achieve to be

$$B = c^{n-1} + c^{n-3} + \cdots + c^2 + 1.$$

Let's pause, before worrying about the clauses, to consider the game played on this graph. In the opening move of the game, Player 1 must select one of v_1 or v'_1 (thereby obliterating the other one); for if not, then Player 2 will immediately select one of them on her next move, winning instantly. Similarly, in the second move of the game, Player 2 must select one of v_2 or v'_2. For otherwise, Player 1 will select one on his next move; and then, even if Player 2 acquired all the remaining nodes in the graph, she would not be able to meet the bound B. Continuing by induction in this way, we see that to avoid an immediate loss, the player making the i^{th} move must select one of v_i or v'_i. Note that our choice of node values has achieved precisely what we wanted: The players must set the variables in order. And what is the outcome on this graph? Player 2 ends up with a total of value of $c^{n-1} + c^{n-3} + \cdots + c^2 = B - 1$: she has lost by one unit!

We now complete the analogy with Competitive 3-SAT by giving Player 2 one final move on which she can try to win. For each clause C_j, we define a node c_j with value $b_{c_j} = 1$ and an edge associated with each of its terms as

follows. If $t = x_i$, we add an edge (c_j, v_i); if $t = \overline{x_i}$, we add an edge (c_j, v'_i). In other words, we join c_j to the node that represents the term t.

This now defines the full graph G. We can verify that, because their values are so small, the addition of the clause nodes did not change the property that the players will begin by selecting the variable nodes $\{v_i, v'_i\}$ in the correct order. However, after this is done, Player 2 will win if and only if she can select a clause node c_j that is not adjacent to any selected variable node—in other words, if and only the truth assignment defined alternately by the players failed to satisfy some clause.

Thus Player 2 can win the Competitive Facility Location instance we have defined if and only if she can win the original Competitive 3-SAT instance. The reduction is complete. ■

Solved Exercises

Solved Exercise 1

Self-avoiding walks are a basic object of study in the area of statistical physics; they can be defined as follows. Let $\mathcal{L}$ denote the set of all points in $\mathbf{R}^2$ with integer coordinates. (We can think of these as the "grid points" of the plane.) A *self-avoiding walk* W of length n is a sequence of points $(p_1, p_2, \ldots, p_n)$ drawn from $\mathcal{L}$ so that

(i) $p_1 = (0, 0)$. *(The walk starts at the origin.)*

(ii) No two of the points are equal. *(The walk "avoids" itself.)*

(iii) For each $i = 1, 2, \ldots, n-1$, the points p_i and p_{i+1} are at distance 1 from each other. *(The walk moves between neighboring points in $\mathcal{L}$.)*

Self-avoiding walks (in both two and three dimensions) are used in physical chemistry as a simple geometric model for the possible conformations of long-chain polymer molecules. Such molecules can be viewed as a flexible chain of beads that flops around, adopting different geometric layouts; self-avoiding walks are a simple combinatorial abstraction for these layouts.

A famous unsolved problem in this area is the following. For a natural number $n \geq 1$, let $A(n)$ denote the number of distinct self-avoiding walks of length n. Note that we view walks as *sequences* of points rather than sets; so two walks can be distinct even if they pass through the same set of points, provided that they do so in different orders. (Formally, the walks $(p_1, p_2, \ldots, p_n)$ and $(q_1, q_2, \ldots, q_n)$ are distinct if there is some i $(1 \leq i \leq n)$ for which $p_i \neq q_i$.) See Figure 9.3 for an example. In polymer models based on self-avoiding walks, $A(n)$ is directly related to the *entropy* of a chain molecule,

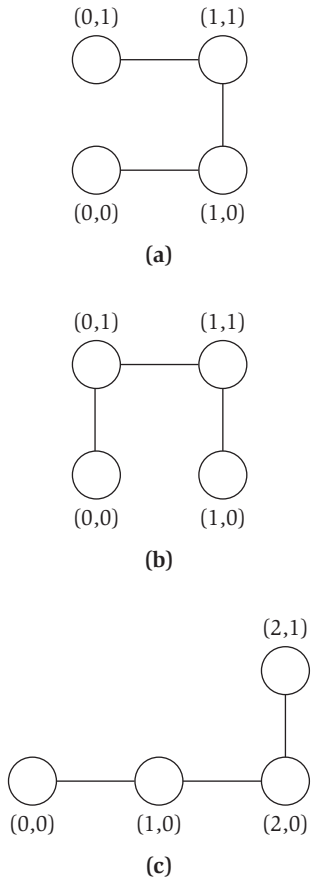

Figure 9.3 Three distinct self-avoiding walks of length 4. Note that although walks (a) and (b) involve the same set of points, they are considered different walks because they pass through them in a different order.

and so it appears in theories concerning the rates of certain metabolic and organic synthesis reactions.

Despite its importance, no simple formula is known for the value $A(n)$. Indeed, no algorithm is known for computing $A(n)$ that runs in time polynomial in n.

(a) Show that $A(n) \geq 2^{n-1}$ for all natural numbers $n \geq 1$.

(b) Give an algorithm that takes a number n as input, and outputs $A(n)$ as a number in binary notation, using space (i.e., memory) that is polynomial in n.

(Thus the running time of your algorithm can be exponential, as long as its space usage is polynomial. Note also that *polynomial* here means "polynomial in n," not "polynomial in $\log n$." Indeed, by part (a), we see that it will take at least $n-1$ bits to write the value of $A(n)$, so clearly $n-1$ is a lower bound on the amount of space you need for producing a correct answer.)

Solution We consider part (b) first. One's first thought is that enumerating all self-avoiding walks sounds like a complicated prospect; it's natural to imagine the search as growing a chain starting from a single bead, exploring possible conformations, and backtracking when there's no way to continue growing and remain self-avoiding. You can picture attention-grabbing screen-savers that do things like this, but it seems a bit messy to write down exactly what the algorithm would be.

So we back up; polynomial space is a very generous bound, and we can afford to take an even more brute-force approach. Suppose that instead of trying just to enumerate all self-avoiding walks of length n, we simply enumerate *all* walks of length n, and then check which ones turn out to be self-avoiding. The advantage of this is that the space of all walks is much easier to describe than the space of self-avoiding walks.

Indeed, any walk $(p_1, p_2, \ldots, p_n)$ on the set $\mathcal{L}$ of grid points in the plane can be described by the sequence of directions it takes. Each step from p_i to p_{i+1} in the walk can be viewed as moving in one of four directions: north, south, east, or west. Thus any walk of length n can be mapped to a distinct string of length $n-1$ over the alphabet $\{N, S, E, W\}$. (The three walks in Figure 9.3 would be ENW, NES, and EEN.) Each such string corresponds to a walk of length n, but not all such strings correspond to walks that are self-avoiding: for example, the walk NESW revisits the point $(0, 0)$.

We can use this encoding of walks for part (b) of the question as follows. Using a counter in base 4, we enumerate all strings of length $n-1$ over the alphabet $\{N, S, E, W\}$, by viewing this alphabet equivalently as $\{0, 1, 2, 3\}$. For each such string, we construct the corresponding walk and test, in polynomial space, whether it is self-avoiding. Finally, we increment a second counter A (initialized to 0) if the current walk is self-avoiding. At the end of this algorithm, A will hold the value of $A(n)$.

Now we can bound the space used by this algorithm as follows. The first counter, which enumerates strings, has $n-1$ positions, each of which requires two bits (since it can take four possible values). Similarly, the second counter holding A can be incremented at most 4^{n-1} times, and so it too needs at most $2n$ bits. Finally, we use polynomial space to check whether each generated walk is self-avoiding, but we can reuse the same space for each walk, and so the space needed for this is polynomial as well.

The encoding scheme also provides a way to answer part (a). We observe that all walks that can be encoded using only the letters $\{N, E\}$ are self-avoiding, since they only move up and to the right in the plane. As there are 2^{n-1} strings of length $n-1$ over these two letters, there are at least 2^{n-1} self-avoiding walks; in other words, $A(n) \geq 2^{n-1}$.

(Note that we argued earlier that our encoding technique also provides an upper bound, showing immediately that $A(n) \leq 4^{n-1}$.)

Exercises

1. Let's consider a special case of Quantified 3-SAT in which the underlying Boolean formula has no negated variables. Specifically, let $\Phi(x_1, \ldots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each C_i is a disjunction of three terms. We say Φ is *monotone* if each term in each clause consists of a nonnegated variable—that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$.

We define Monotone QSAT to be the decision problem

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

where the formula Φ is monotone.

Do one of the following two things: (a) prove that Monotone QSAT is PSPACE-complete; or (b) give an algorithm to solve arbitrary instances of Monotone QSAT that runs in time polynomial in n. (Note that in (b), the goal is polynomial *time*, not just polynomial space.)

2. Consider the following word game, which we'll call *Geography*. You have a set of names of places, like the capital cities of all the countries in the world. The first player begins the game by naming the capital city c of the country the players are in; the second player must then choose a city c' that starts with the letter on which c ends; and the game continues in this way, with each player alternately choosing a city that starts with the letter on which the previous one ended. The player who loses is the first one who cannot choose a city that hasn't been named earlier in the game.

For example, a game played in Hungary would start with "Budapest," and then it could continue (for example), "Tokyo, Ottawa, Ankara, Amsterdam, Moscow, Washington, Nairobi."

This game is a good test of geographical knowledge, of course, but even with a list of the world's capitals sitting in front of you, it's also a major strategic challenge. Which word should you pick next, to try forcing

your opponent into a situation where they'll be the one who's ultimately stuck without a move?

To highlight the strategic aspect, we define the following abstract version of the game, which we call *Geography on a Graph*. Here, we have a directed graph $G = (V, E)$, and a designated *start node* $s \in V$. Players alternate turns starting from s; each player must, if possible, follow an edge out of the current node to a node that hasn't been visited before. The player who loses is the first one who cannot move to a node that hasn't been visited earlier in the game. (There is a direct analogy to *Geography*, with nodes corresponding to words.) In other words, a player loses if the game is currently at node v, and for edges of the form (v, w), the node w has already been visited.

Prove that it is PSPACE-complete to decide whether the first player can force a win in *Geography on a Graph*.

3. Give a polynomial-time algorithm to decide whether a player has a forced win in *Geography on a Graph*, in the special case when the underlying graph G has no directed cycles (in other words, when G is a DAG).

Notes and Further Reading

PSPACE is just one example of a class of intractable problems beyond NP; charting the landscape of computational hardness is the goal of the field of *complexity theory*. There are a number of books that focus on complexity theory; see, for example, Papadimitriou (1995) and Savage (1998).

The PSPACE-completeness of QSAT is due to Stockmeyer and Meyer (1973).

Some basic PSPACE-completeness results for two-player games can be found in Schaefer (1978) and in Stockmeyer and Chandra (1979). The Competitive Facility Location Problem that we consider here is a stylized example of a class of problems studied within the broader area of *facility location*; see, for example, the book edited by Drezner (1995) for surveys of this topic.

Two-player games have provided a steady source of difficult questions for researchers in both mathematics and artificial intelligence. Berlekamp, Conway, and Guy (1982) and Nowakowski (1998) discuss some of the mathematical questions in this area. The design of a world-champion-level chess program was for fifty years the foremost applied challenge problem in the field of computer game-playing. Alan Turing is known to have worked on devising algorithms to play chess, as did many leading figures in artificial intelligence over the years. Newborn (1996) gives a readable account of the history of work

on this problem, covering the state of the art up to a year before IBM's Deep Blue finally achieved the goal of defeating the human world champion in a match.

Planning is a fundamental problem in artificial intelligence; it features prominently in the text by Russell and Norvig (2002) and is the subject of a book by Ghallab, Nau, and Traverso (2004). The argument that Planning can be solved in polynomial space is due to Savitch (1970), who was concerned with issues in complexity theory rather than the Planning Problem per se.

Notes on the Exercises Exercise 1 is based on a problem we learned from Maverick Woo and Ryan Williams; Exercise 2 is based on a result of Thomas Schaefer.