Part V

Intractable Problems
Chapter 16

$NP$-Completeness

Up to now, we have focused on developing efficient algorithms for solving problems. The word “efficient” is somewhat subjective, and the degree of efficiency has varied depending on the problem. Still, in each case, we have shown a running time that was no worse than a low-order polynomial in some natural description of the problem size.

It is possible to prove, however, that some problems cannot be solved by any algorithm with polynomial running time. In fact, it is possible to prove that some problems cannot be solved by any algorithm at all. We will not be examining any of these problems, but in this chapter and the next, we will take a look at a very interesting class of problems for which no efficient algorithms are known. Part of the reason that this class of problems is interesting is that if a polynomial-time algorithm were to be found for any one of these problems, then we could derive polynomial-time algorithms for all of the problems in this class. Furthermore, no one to date has given a convincing proof that there are no such algorithms. At the heart of these issues is the most famous open question in computational complexity theory.

16.1 Boolean Satisfiability

Suppose we are given an expression $F$ containing boolean variables and the following operators:

- $\neg$: logical negation;
- $\lor$: logical or; and
- $\land$: logical and.
There are two questions we might ask regarding $\mathcal{F}$:

- Is $\mathcal{F}$ valid? That is, does $\mathcal{F}$ evaluate to true for every possible assignment of true or false to the variables in $\mathcal{F}$?

- Is $\mathcal{F}$ satisfiable? That is, does there exist some assignment of true or false to the variables in $\mathcal{F}$ so that $\mathcal{F}$ evaluates to true?

For example, let $\mathcal{F} = \neg x \lor (y \land x)$. This expression is not valid, because setting $x$ to true and $y$ to false yields

$$-\text{true} \lor (\text{false} \land \text{true}) = \text{false} \lor \text{false}$$

$$= \text{false}.$$  

However, $\mathcal{F}$ is satisfiable — in fact, any other assignment of values to $x$ and $y$ makes $\mathcal{F}$ true.

Note that it follows immediately from the definitions of validity and satisfiability that for any expression $\mathcal{F}$, $\mathcal{F}$ is valid iff $\neg \mathcal{F}$ is unsatisfiable. Because of this duality, we will focus on only one of these problems, the satisfiability problem. We would like to find a satisfiability algorithm whose worst-case running time is bounded by some polynomial in the size of the given expression, where the size is defined to be the total number of occurrences of variables and operators. However, as of this writing, no such algorithm has been found. Indeed, as we will see shortly, there is good reason to believe that no such algorithm is possible. On the other hand, there currently exists no proof that such an algorithm is impossible.

Before we look at the satisfiability problem in more detail, let us first consider a simpler problem, that of evaluating a boolean expression $\mathcal{F}$, given boolean values for its variables. We first observe that $\mathcal{F}$ must be of one of the following forms:

- a single variable;
- the negation of an expression, i.e., $\neg \mathcal{F}_1$;
- the or of two expressions, i.e., $\mathcal{F}_1 \lor \mathcal{F}_2$; or
- the and of two expressions, i.e., $\mathcal{F}_1 \land \mathcal{F}_2$.

It is therefore convenient to represent the formula using a binary tree in which the leaves represent variables and the internal nodes represent operators. For the $\neg$ operator, the right-hand child will always be empty. To make this representation more concrete, we can use special constants not,
and, or to represent the three operators, and we can represent the variables using positive integers. In order to avoid unnecessary complications, we will assume that if \( j \) represents a variable in the expression, then for \( 1 \leq i \leq j \), \( i \) represents some variable in the expression (note, however, that we cannot apply this assumption to arbitrary subtrees). For example, Figure 16.1 shows the tree representing the formula \( \neg x \lor (y \land x) \), first in a more abstract form, then in a more concrete form using 1 to represent \( x \) and 2 to represent \( y \). Finally, we can represent an assignment of truth values to the variables using an array \( A[1..n] \) of boolean values, where \( A[i] \) gives the value assigned to the variable represented by \( i \).

Given an expression tree for \( \mathcal{F} \) and an array representing an assignment of truth values, we can then evaluate \( \mathcal{F} \) using \textsc{BoolEval}, shown in Figure 16.2. It is not hard to see that this algorithm runs in \( \Theta(m) \) time, where \( m \) is the number of operators in \( \mathcal{F} \) (note that the number of leaves in the expression tree can be no more than \( m + 1 \)).

Returning to the satisfiability problem, we now point out some characteristics that this problem has in common with the other problems we will study in this chapter. First, it is a decision problem — the output is either “yes” or “no”. Second, when the answer is “yes”, there is a relatively short proof of this fact — an assignment of truth values that satisfies the expression. Third, given a proposed proof of satisfiability (i.e., some assignment of truth values to the variables), we can efficiently verify whether it does, in fact, prove the satisfiability of the expression. However, finding such a proof, or proving that none exists, appears to be an expensive task in the worst case (note that there are \( 2^n \) possible assignments of truth values to \( n \) variables).
Figure 16.2 Algorithm for evaluating a boolean expression.

**Precondition:** \( \mathcal{F} \) is a `BinaryTreeNode` referring to the root of a nonempty boolean expression tree, and \( A[1..n] \) is an array of `Bools`.

**Postcondition:** Returns the value of \( \mathcal{F} \) assuming that the variable represented by \( i \) has value \( A[i] \) if \( i \leq n \), or `false` if \( i > n \).

\[
\begin{align*}
\text{BoolEval}(\mathcal{F}, A[1..n]) \\
&\quad \text{if } \mathcal{F}.\text{LeftChild}() = \text{nil} \\
&\quad \quad \text{if } \mathcal{F}.\text{Root}() \leq n \\
&\quad \quad \quad \text{return } A[\mathcal{F}.\text{Root}()] \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{return } \text{false} \\
&\quad \text{else if } \mathcal{F}.\text{Data}() = \text{not} \\
&\quad \quad \text{return not } \text{BoolEval}(\mathcal{F}.\text{LeftChild}(), A) \\
&\quad \text{else} \\
&\quad \quad l \leftarrow \text{BoolEval}(\mathcal{F}.\text{LeftChild}(), A) \\
&\quad \quad r \leftarrow \text{BoolEval}(\mathcal{F}.\text{RightChild}(), A) \\
&\quad \quad \text{if } \mathcal{F}.\text{Root}() = \text{and} \\
&\quad \quad \quad \text{return } l \text{ and } r \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{return } l \text{ or } r
\end{align*}
\]

16.2 The Set \( \mathcal{P} \)

We have suggested that the boolean satisfiability problem may not be efficiently solvable. However, we have not yet formalized what this means. In this section we will define a set of decision problems that we will consider to be those that are efficiently decidable.

We will begin by adopting a couple of conventions. First, we will assume that each problem instance is a bit string. Certainly, we can encode other types of input data as bit strings, provided the set of all instances is countable. We do need to ensure, however, that this encoding is done in such a way that the length of the encoding is not unnecessarily long. With this convention, we can now express the running time of an algorithm in terms of the length of its input string. This gives us a uniform way of expressing the running times of all algorithms for all problems. Second, we will view a decision problem as a subset of its instances. Specifically, let \( I \) denote
the set of all bit strings that encode boolean expressions. We can then let $\text{Sat}$ denote the set of all expressions in $I$ that are satisfiable. It will also be convenient to use $\text{Sat}$ to denote the problem itself. In general, given a set of instances $I$, we will refer to any subset $X \subseteq I$ as a decision problem over $I$.

We must now address the question of how efficient is “efficient”. We will somewhat arbitrarily delineate the “efficient” algorithms as those that operate within a running time bounded by some polynomial in the length of the input. This delineation, however, is not entirely satisfactory. On the one hand, one could make a persuasive argument that an algorithm with a running time in $\Theta(n^{1000})$ is not “efficient”. On the other hand, suppose some algorithm has a running time in $\Theta(n^{[\alpha(n)/4]})$, where $\alpha$ is as defined in Section 8.4. Because $n^{[\alpha(n)/4]} \leq n$ for every positive $n$ that can be coded in binary within our universe, such an algorithm might reasonably be considered to be “efficient”. However, $n^{[\alpha(n)/4]}$ is not bounded above by any polynomial.

The main reason we equate polynomial running time with efficiency is that polynomials have several useful closure properties. Specifically, if $p_1(n)$ and $p_2(n)$ are polynomials, then so are $p_1(n) + p_2(n)$, $p_1(n)p_2(n)$, and $p_1(p_2(n))$. As we will see, these closure properties make the theory much cleaner. Furthermore, if we can say that a particular decision problem cannot be solved by any polynomial-time algorithm, then we can be fairly safe in concluding that there is no algorithm that will terminate in a reasonable amount of time on large inputs in the worst case.

Before we formalize this idea, we need to be careful about one aspect of our running-time measures. Specifically, we have assumed in this text that arithmetic operations can be performed in a single step. This assumption is valid if we can reasonably expect the numbers to fit in a single machine word. For larger values, we should use the BigNum type in order to get an appropriate measure of the running time. Also note that all of the algorithms in this text except those using real or complex numbers can be expressed using only booleans and natural numbers as primitive types (i.e., those not defined in terms of other variables). Furthermore, only the algorithms of Section 15.1 require real or complex numbers — all other algorithms can be restricted to rational numbers, which can be expressed as a pair of natural numbers and a sign bit. Thus, it will make sense to stipulate that an “efficient” algorithm contains only boolean or natural number variables, along with other types built from these, and that each natural number variable will contain only a polynomial number of bits.

We can now formalize our notion of “efficient”. We say that an algorithm $A$ is a polynomial-time algorithm if there is a polynomial $p(n)$ such that
• A always completes within \( p(n) \) steps, where \( n \) is the number of bits in the input string; and

• all primitive variables used by \( A \) are either booleans or natural numbers whose values remain strictly less than \( 2^{p(n)} \).

We then define \( \mathcal{P} \) to be the set of all decision problems \( X \) such that there exists a deterministic (i.e., not randomized) polynomial-time algorithm \( A \) deciding \( X \). It is this set \( \mathcal{P} \) that we will consider to be the set of efficiently solvable decision problems.

The decision to define \( \mathcal{P} \) using only deterministic algorithms is rather arbitrary. Indeed, there is a branch of computational complexity theory that focuses on efficient randomized algorithms. However, the study of deterministic algorithms is more fundamental, and therefore is a more reasonable starting point for us.

### 16.3 The Set \( \mathcal{NP} \)

\textsc{Sat} is clearly a decision problem, but as we have suggested, it is currently not known whether or not it belongs to \( \mathcal{P} \). In this section, we will define a related set called \( \mathcal{NP} \) that includes all of \( \mathcal{P} \), but also includes \textsc{Sat}, as well as many other problems not known to be in \( \mathcal{P} \). Furthermore, \( \mathcal{NP} \) will have the property that if \textsc{Sat} is, in fact, in \( \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \).

In order to extend the definition of \( \mathcal{P} \) to include problems like \textsc{Sat}, we need to formalize the idea that each element of a decision problem \( X \) has a short proof which can be efficiently checked. For the sake of concreteness, let us assume that all proofs will be encoded as bit strings. We then denote the set of all bit strings by \( \mathcal{B} \).

We now define \( \mathcal{NP} \) to be the set of all decision problems \( X \) for which there exist:

• a polynomial \( p(n) \); and

• a decision problem \( Y \subseteq I \times \mathcal{B} \), where \( I \) is the set of instances for \( X \); such that

• \( Y \in \mathcal{P} \);

• for each \( x \in I \), \( x \in X \) iff there is a proof \( \phi \in \mathcal{B} \) such that \( (x, \phi) \in Y \); and
• for each \( x \in X \), there is a proof \( \phi \in B \) such that \( (x, \phi) \in Y \) and \( |\phi| \leq p(|x|). \)

From our earlier discussion, it follows that \( \text{SAT} \in \mathcal{NP} \). We can clearly consider any array \( A[1..n] \) of boolean values to be a bit string, and vice versa. We then let the decision problem \( Y \) be the problem solved by \( \text{BOOLVAL} \). Hence, \( Y \in \mathcal{P} \). We can then let \( p(n) = n \). Then a given expression \( F \) is satisfiable iff there is a proof \( \phi \) such that \( (x, \phi) \in Y \). Because we have assumed that whenever an integer \( j \) represents a variable in \( F \), all positive integers less than \( j \) also represent variables in \( F \), it follows that if a proof \( \phi \) exists, there will always be one with length no more than \( |F| \).

We therefore have an example of a problem in \( \mathcal{NP} \) that may or may not be in \( \mathcal{P} \). The following theorem gives the relationship between \( \mathcal{P} \) and \( \mathcal{NP} \).

**Theorem 16.1** \( \mathcal{P} \subseteq \mathcal{NP} \).

**Proof:** Let \( X \in \mathcal{P} \), and let \( I \) be the set of instances of \( X \). We then define \( Y = X \times B \). Thus, \( Y \) is comprised of all pairs \( (x, \phi) \) such that \( x \in X \) and \( \phi \in B \). We can therefore decide whether \( (x, \phi) \in I \times B \) belongs to \( Y \) by simply deciding whether \( x \in X \). Because \( X \in \mathcal{P} \), it follows that \( Y \in \mathcal{P} \). Let \( p(n) = 1 \). Then \( x \in X \) iff \( (x, 0) \in Y \), and the length of \( 0 \) is \( 1 \). Therefore \( X \in \mathcal{NP} \). \( \square \)

It is currently unknown whether the above containment is proper, or whether \( \mathcal{P} = \mathcal{NP} \). In fact the “\( \mathcal{P} = \mathcal{NP} \)” question is the most famous open question in computational complexity theory. Most complexity theorists believe that these sets are not equal. Though many of the reasons for this belief are beyond the scope of this book, we will soon see one compelling reason. For now, let us simply state that we take as a working hypothesis that \( \mathcal{P} \neq \mathcal{NP} \). Thus, if we can show that some particular statement implies that \( \mathcal{P} = \mathcal{NP} \), we take this as strong evidence that the statement is false.

In order to focus on the relationship between \( \mathcal{P} \) and \( \mathcal{NP} \), it is useful to identify problems that seem more likely to be in \( \mathcal{NP} \setminus \mathcal{P} \). In some sense, we want to identify the hardest problems in \( \mathcal{NP} \). We can do this using a refinement of the notion of problem reduction. Specifically, let \( X \) and \( Y \) be two decision problems whose instances comprise the sets \( I \) and \( J \), respectively. We say that \( X \) is *polynomially many-one reducible* to \( Y \), written \( X \leq_m^p Y \), if there is a function \( f : I \to J \) such that

• for all \( x \in I \), \( x \in X \) iff \( f(x) \in Y \); and
• there is a deterministic polynomial-time algorithm for computing \( f \).

Note that polynomial many-one reductions are transformations — given an instance \( x \) of problem \( X \), we can decide whether \( x \in X \) simply by computing \( f(x) \) and deciding whether \( f(x) \in Y \). The notation may seem confusing at first, because when we use the word “reduce”, we usually think of decreasing the size. As a result, denoting a reduction from \( X \) to \( Y \) by \( X \leq_p m Y \) seems backwards. The proper way to understand the notation is to realize that when there is a polynomial many-one reduction from \( X \) to \( Y \), then in some sense, \( X \) is no harder than \( Y \). This idea is formalized by the following theorem.

**Theorem 16.2** If \( X \leq_p m Y \) and \( Y \in \mathcal{P} \), then \( X \in \mathcal{P} \).

**Proof:** Let \( I \) and \( J \) be the sets of instances of \( X \) and \( Y \), respectively, and let \( f : I \rightarrow J \) be the function computing the polynomial many-one reduction from \( X \) to \( Y \). Let \( p_1(n) \) be a polynomial bounding the running time and the values of variables for some algorithm to compute \( f \), and let \( p_2(n) \) be a polynomial bounding the running time and the values of variables for some algorithm to decide \( Y \). We can then decide whether a given \( x \in I \) belongs to \( X \) by first computing \( f(x) \), then deciding whether \( f(x) \in Y \). Let \( A \) denote this algorithm.

The time required to compute \( f(x) \) is no more than \( p_1(|x|) \). The time required to decide whether \( f(x) \in Y \) is no more than \( p_2(|f(x)|) \). Because \( f(x) \) is computed using at most \( p_1(|x|) \) steps, \( |f(x)| \leq c p_1(|x|) \), where \( c \) is some positive integer constant bounding the number of bits that can be written in a single step. The total time required by \( A \) is therefore no more than
\[
p(|x|) = p_1(|x|) + p_2(c p_1(|x|)),
\]
which is a polynomial in \( |x| \). Furthermore, the values of the variables in \( A \) do not exceed \( \max(p_1(|x|), p_2(|x|)) \). Because \( p(n) \), \( p_1(n) \), and \( p_2(n) \) must be nonnegative for all \( n \in \mathbb{N} \), all are bounded above by the polynomial \( p(n) + p_1(n) + p_2(n) \). Therefore, \( X \in \mathcal{P} \). \( \Box \)

Note that Theorem 16.2 does not say that if \( Y \) can be decided in \( O(f(n)) \) time, then \( X \) can be decided in \( O(f(n)) \) time. Indeed, in the proof of the theorem, the bound on the time to decide \( X \) can be much larger than the time to decide \( Y \). Thus, if we interpret \( X \leq_p m Y \) as indicating that \( X \) is no harder than \( Y \), we must understand “no harder than” in a very loose sense — simply that if \( Y \in \mathcal{P} \), then \( X \in \mathcal{P} \).
We will often utilize Theorem 16.2 in the following equivalent form.

**Corollary 16.3** If \( X \leq^p_m Y \) and \( X \notin \mathcal{P} \), then \( Y \notin \mathcal{P} \).

Suppose we have some problem \( Y \in \mathcal{NP} \) such that for every \( X \in \mathcal{NP} \), \( X \leq^p_m Y \). If \( \mathcal{P} \neq \mathcal{NP} \), then there is some \( X \in \mathcal{NP} \setminus \mathcal{P} \). Because \( X \leq^p_m Y \) and \( X \notin \mathcal{P} \), Corollary 16.3 tells us that \( Y \notin \mathcal{P} \). This fact motivates the following definitions.

**Definition 16.4** If \( Y \) is a decision problem such that for every \( X \in \mathcal{NP} \), \( X \leq^p_m Y \), then we say that \( Y \) is \( \mathcal{NP} \)-hard.

**Definition 16.5** If \( Y \in \mathcal{NP} \) is \( \mathcal{NP} \)-hard, then we say that \( Y \) is \( \mathcal{NP} \)-complete.

Suppose we have some \( \mathcal{NP} \)-hard problem \( Y \), and also suppose that \( \mathcal{P} \neq \mathcal{NP} \). Then there is some \( X \in \mathcal{NP} \setminus \mathcal{P} \). Because \( X \in \mathcal{NP} \) and \( Y \) is \( \mathcal{NP} \)-hard, \( X \leq^p_m Y \). Hence, from Corollary 16.3, \( Y \notin \mathcal{P} \). We therefore have the following theorem and its corollary.

**Theorem 16.6** If \( Y \) is \( \mathcal{NP} \)-hard and \( \mathcal{P} \neq \mathcal{NP} \), then \( Y \notin \mathcal{P} \).

**Corollary 16.7** If \( Y \) is \( \mathcal{NP} \)-complete and \( \mathcal{P} \neq \mathcal{NP} \), then \( Y \in \mathcal{NP} \setminus \mathcal{P} \).

It turns out that thousands of \( \mathcal{NP} \)-complete problems in a wide variety of problem domains have been identified. If we could find a polynomial-time algorithm for any one of these problems, Corollary 16.7 would then imply that \( \mathcal{P} = \mathcal{NP} \). The fact that this has not been accomplished is one reason to suspect that \( \mathcal{P} \neq \mathcal{NP} \). Let us now identify an \( \mathcal{NP} \)-complete problem.

**Theorem 16.8 (Cook’s Theorem)** \( \text{Sat} \) is \( \mathcal{NP} \)-complete.

The idea of the proof of Cook’s Theorem is to give a method for constructing, from an arbitrary \( X \in \mathcal{NP} \), a polynomial-time algorithm that takes as input an instance \( x \) of \( X \) and produces as output a boolean expression \( \mathcal{F} \) such that \( \mathcal{F} \) is satisfiable iff \( x \in X \). In constructing this algorithm, we can use the polynomial \( p(n) \) bounding the size of a proof \( \phi \) and the algorithm for deciding whether \( \phi \) proves that \( x \in X \). In order to complete the construction, we must carefully define the computational model so that the boolean formula can encode the algorithm. Due to the large amount of work involved, we will delay the proof of Cook’s Theorem until Section 16.8.
Fortunately, once we have one \( \mathcal{NP} \)-complete problem, the task of showing other problems to be \( \mathcal{NP} \)-complete becomes much easier. The reason for this is that polynomial many-one reducibility is transitive, as we show in the following theorem. Its proof is similar to the proof of Theorem 16.2, and is therefore left as an exercise. Its corollary then gives us a proof technique for showing \( \mathcal{NP} \)-hardness, provided we already have at least one \( \mathcal{NP} \)-hard problem.

**Theorem 16.9** If \( X \leq_{m}^{p} Y \) and \( Y \leq_{m}^{p} Z \), then \( X \leq_{m}^{p} Z \).

**Corollary 16.10** If \( X \) is \( \mathcal{NP} \)-hard and \( X \leq_{m}^{p} Y \), then \( Y \) is \( \mathcal{NP} \)-hard.

Thus, to show a decision problem \( Y \) to be \( \mathcal{NP} \)-hard, we need only to show that \( X \leq_{m}^{p} Y \) for some \( \mathcal{NP} \)-hard problem \( X \). This is the technique that we will use for all subsequent \( \mathcal{NP} \)-hardness proofs. Note also that the more problems we have shown to be \( \mathcal{NP} \)-complete, the more tools we have for showing additional problems to be \( \mathcal{NP} \)-complete. For this reason, we will devote the next few sections to identifying a variety of \( \mathcal{NP} \)-complete problems.

### 16.4 Restricted Satisfiability Problems

In order to illustrate the technique of proving \( \mathcal{NP} \)-hardness using a polynomial many-one reduction from a known \( \mathcal{NP} \)-hard problem, we consider in this section two special cases of boolean satisfiability. In the first special case, the expression is in **conjunctive normal form (CNF)**; i.e., the expression is of the form

\[
\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_{i}} \alpha_{ij},
\]

where each \( \alpha_{ij} \) is a **literal** — either a variable or the negation of a variable. Let us refer to this problem as \( \text{CSat} \). We will now show that \( \text{CSat} \) is \( \mathcal{NP} \)-complete. The proof of \( \mathcal{NP} \)-hardness will consist of showing that \( \text{Sat} \leq_{m}^{p} \text{CSat} \).

It makes sense to represent the input for \( \text{CSat} \) using a data structure that follows the form of a CNF formula more closely than does an expression tree. Specifically, we represent a CNF formula as a **ConsList** of **clauses**, each of which is a disjunction of literals. We then represent a clause as a **ConsList** of literals. Finally, we represent each literal as an integer as follows. For a non-negated variable, we simply use a positive integer, as in an
expression tree. For a negated variable $\neg x$, we use $-i$, where $i$ is the integer representing the variable $x$. We will again assume that for an input formula $\mathcal{F}$, if $j$ represents a variable in $\mathcal{F}$, then for $1 \leq i \leq j$, $i$ represents a variable in $\mathcal{F}$. Again, this assumption will not apply to arbitrary sub-formulas.

One obvious way of reducing SAT to CSAT is to convert a given boolean expression to an equivalent expression in CNF. However, there are boolean expressions for which the shortest equivalent CNF expression has size exponential in the size of the original expression. As a result, any such conversion algorithm must require at least exponential time in the worst case.

Fortunately, our reduction doesn’t need to construct an equivalent expression, but only one that is satisfiable iff the given expression is satisfiable. In fact, the constructed expression isn’t even required to contain the same variables. We will use this flexibility in designing our reduction.

For the first step of our reduction, we will construct an equivalent formula in which negations are applied only to variables. Because of this restriction, we can simplify our representation for this kind of expression by allowing leaves to contain either positive or negative integers, as in our representation of CNF formulas. Using this representation, we no longer need nodes representing the $\neg$ operation. We will refer to this representation as a normalized expression tree.

Fortunately, there is a polynomial-time algorithm for normalizing a boolean expression tree. The algorithm uses DeMorgan’s laws:

- $\neg(x \land y) = \neg x \land \neg y$; and
- $\neg(x \lor y) = \neg x \lor \neg y$.

The algorithm is shown in Figure 16.3. This algorithm solves a slightly more general problem for which the input includes a boolean neg, which indicates whether the normalized expression should be equivalent to $\mathcal{F}$ or $\neg \mathcal{F}$. It is easily seen that its running time is proportional to the number of nodes in the tree, which is in $O(m)$, where $m$ is the number of operators in $\mathcal{F}$.

As the second step in our reduction, we need to find the largest integer used to represent a variable in a normalized expression tree. We need this value in order to be able to introduce new variables. Such an algorithm is shown in Figure 16.4. Clearly, its running time is in $O(|\mathcal{F}|)$.

As the third step in our reduction, we will construct from a normalized expression tree $\mathcal{F}$ and a value larger than any integer representing a variable in $\mathcal{F}$, a CNF expression $\mathcal{F}'$ having the following properties:

- $P_1$: $\mathcal{F}'$ contains all of the variables in $\mathcal{F}$;
Figure 16.3 Algorithm to normalize a boolean expression tree

**Precondition:** \( \mathcal{F} \) is a \texttt{BinaryTreeNode} referring to the root of a boolean expression tree, and \( \text{neg} \) is a \texttt{Bool}.

**Postcondition:** Returns a normalized expression tree \( \mathcal{F}' \) such that if \( \text{neg} \) is false, \( \mathcal{F}' \equiv \mathcal{F} \), and if \( \text{neg} \) is true, \( \mathcal{F}' \equiv \neg \mathcal{F} \).

\[
\text{Normalize}(\mathcal{F}, \text{neg}) \rightarrow \text{new} \ \texttt{BinaryTreeNode}(); \ \text{op} \rightarrow \mathcal{F}.\text{Root}()
\]

if \( \mathcal{F}.\text{LeftChild}() = \text{nil} \)

if \( \text{neg} \)

\( \mathcal{F}'.\text{SetRoot}(\neg \text{op}) \)

else

\( \mathcal{F}'.\text{SetRoot}(\text{op}) \)

else if \( \text{op} = \neg \)

\( \mathcal{F}' \leftarrow \text{Normalize}(\mathcal{F}.\text{LeftChild}(), \neg \text{neg}) \)

else

\( \mathcal{F}'.\text{SetLeftChild}((\text{Normalize}(\mathcal{F}.\text{LeftChild}(), \text{neg}))) \)

\( \mathcal{F}'.\text{SetRightChild}((\text{Normalize}(\mathcal{F}.\text{RightChild}(), \text{neg}))) \)

if \( (\text{op} = \text{and} \ \text{and} \ \text{neg}) \ \text{or} \ (\text{op} = \text{or} \ \text{and} \ \neg \text{neg}) \)

\( \mathcal{F}'.\text{SetRoot}(\text{or}) \)

else

\( \mathcal{F}'.\text{SetRoot}(\text{and}) \)

\text{return } \mathcal{F}'

---

\( P_2 \): for any satisfying assignment \( A \) for \( \mathcal{F} \), there is a satisfying assignment \( A' \) for \( \mathcal{F}' \) in which all the variables in \( \mathcal{F} \) have the same values as in \( A \); and

\( P_3 \): for any satisfying assignment \( A' \) for \( \mathcal{F}' \), the assignment \( A \) for \( \mathcal{F} \) in which each variable in \( \mathcal{F} \) is assigned its value from \( A' \) satisfies \( \mathcal{F} \).

Thus, \( \mathcal{F}' \) will be satisfiable iff \( \mathcal{F} \) is satisfiable. We consider three cases.

**Case 1:** \( \mathcal{F} \) is a literal. Then because \( \mathcal{F} \) is in CNF, we let \( \mathcal{F}' = \mathcal{F} \).

**Case 2:** \( \mathcal{F} = \mathcal{F}_1 \land \mathcal{F}_2 \). We then construct CNF formulas \( \mathcal{F}'_1 \) and \( \mathcal{F}'_2 \) from \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \), respectively, such that properties \( P_1-P_3 \) are satisfied. Then
Figure 16.4 Algorithm to find the largest integer representing a variable in a normalized expression tree

**Precondition:** \( \mathcal{F} \) is a \texttt{BinaryTreeNode} representing a normalized expression tree.

**Postcondition:** Returns the largest absolute value of any integer in \( \mathcal{F} \).

\[
\text{MaxVar}(\mathcal{F}) \quad \begin{cases} 
    l &\leftarrow \text{MaxVar}(\mathcal{F}\.\text{LeftChild}()) \\
    r &\leftarrow \text{MaxVar}(\mathcal{F}\.\text{RightChild}()) \\
    \text{return} & \text{Max}(l, r) \\
    \text{else} & \text{return} \mid \mathcal{F}\.\text{Root}() \mid 
\end{cases}
\]

\( \mathcal{F}' = \mathcal{F}'_1 \land \mathcal{F}'_2 \) is in CNF and clearly satisfies properties \( P_1-P_3 \) with respect to \( \mathcal{F} \).

**Case 3:** \( \mathcal{F} = \mathcal{F}_1 \lor \mathcal{F}_2 \). We then construct CNF formulas \( \mathcal{F}'_1 \) and \( \mathcal{F}'_2 \) from \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \), respectively, such that properties \( P_1-P_3 \) are satisfied. Let \( u \) be a variable that is contained in neither \( \mathcal{F}'_1 \) nor \( \mathcal{F}'_2 \). We construct \( \mathcal{F}_2'' \) by including \( u \) in each clause of \( \mathcal{F}'_2 \), and we construct \( \mathcal{F}_1'' \) by including \( \neg u \) in each clause of \( \mathcal{F}'_1 \). We then let \( \mathcal{F}' = \mathcal{F}_1'' \land \mathcal{F}_2'' \). Clearly, \( \mathcal{F}' \) is in CNF. Furthermore, it is not hard to show that \( \mathcal{F}' \) satisfies properties \( P_1-P_3 \) with respect to \( \mathcal{F} \).

The algorithm for constructing \( \mathcal{F}' \) is shown in Figure 16.5. It uses a data type \texttt{MutableNat}, which contains a single readable and writable representation variable \texttt{data}, which is a \texttt{Nat}. It also uses the function \texttt{Append} specified in Figure 4.15.

It is not hard to see that \texttt{AddToClauses} operates in \( O(n) \) time, where \( n \) is the number of clauses in \( \mathcal{F} \). Furthermore, \texttt{NormalizedToCNF} only constructs a new clause when processing a literal; hence, the number of clauses in the CNF formula is no more than \( |\mathcal{F}| \). As suggested in Exercise 4.3, \texttt{Append} can be implemented to run in \( O(n) \) time, where \( n \) is the number of elements in its first argument. It follows that the time for a single call to \texttt{NormalizedToCNF}, excluding recursive calls, runs in \( O(|\mathcal{F}|) \) time.
**Precondition:** \( \mathcal{F} \) is a `BinaryTreeNode` referring to normalized boolean expression tree, and \( m \) refers to a `MutableNat` larger than the absolute value of any variable in \( \mathcal{F} \).

**Postcondition:** Returns a `ConsList` representing a CNF formula \( \mathcal{F}' \) that satisfies properties \( P_1 - P_3 \) with respect to \( \mathcal{F} \). \( m \) has a value larger than the absolute value of any variable in the returned formula.

\[
\text{NormalizedToCnf}(\mathcal{F}, m) \\
\quad \text{root} \leftarrow \mathcal{F}.\text{Root}() \\
\quad \text{if root = and or root = or} \\
\quad \quad \quad l \leftarrow \text{NormalizedToCnf}(\mathcal{F}.\text{LeftChild}(), m) \\
\quad \quad \quad r \leftarrow \text{NormalizedToCnf}(\mathcal{F}.\text{RightChild}(), m) \\
\quad \quad \quad \text{if root = or} \\
\quad \quad \quad \quad x \leftarrow m.\text{Data}(); l \leftarrow \text{AddToClauses}(l, x) \\
\quad \quad \quad \quad r \leftarrow \text{AddToClauses}(r, -x); m.\text{SetData}(x + 1) \\
\quad \quad \quad \text{return Append}(l, r) \\
\quad \quad \text{else} \\
\quad \quad c \leftarrow \text{new ConsList}(\text{root}, \text{new ConsList}()) \\
\quad \text{return new ConsList}(c, \text{new ConsList}())
\]

**Precondition:** \( \mathcal{F} \) is a (possibly empty) `ConsList` representing a CNF formula, and \( \alpha \) is a nonzero `Int`.

**Postcondition:** Returns a `ConsList` obtained by adding the literal \( \alpha \) to each clause in \( \mathcal{F} \).

\[
\text{AddToClauses}(\mathcal{F}, \alpha) \\
\quad \text{if } \mathcal{F}.\text{isEmpty}() \\
\quad \quad \text{return } \mathcal{F} \\
\quad \text{else} \\
\quad \quad h \leftarrow \text{new ConsList}(\alpha, \mathcal{F}.\text{head}()) \\
\quad \quad t \leftarrow \text{AddToClauses}(\mathcal{F}.\text{tail}(), \alpha) \\
\quad \text{return new ConsList}(h, t)
\]
**Figure 16.6** The reduction from Sat to CSat

**Precondition:** $\mathcal{F}$ is BinaryTreeNode referring to a boolean expression tree.

**Postcondition:** Returns a ConsList representing a CNF formula that is satisfiable iff $\mathcal{F}$ is satisfiable.

$\text{SatToCSat}(\mathcal{F})$

\[
\begin{align*}
\mathcal{F}' & \leftarrow \text{Normalize}(\mathcal{F}, \text{false}) \\
\text{m} & \leftarrow \text{new MutableInt}(); \text{m.SetData(MaxVar}(\mathcal{F}') + 1) \\
\text{return NormalizedToCnf}(\mathcal{F}', \text{m})
\end{align*}
\]

Because NormalizedToCnf is called once for every node in the expression tree $\mathcal{F}$, its overall running time is in $O(|\mathcal{F}|^2)$.

The reduction is implemented in Figure 16.6. It clearly runs in $O(|\mathcal{F}|^2)$ time, so that $\text{Sat} \leq^p \text{CSat}$. We can therefore show the following theorem.

**Theorem 16.11** CSat is $\mathcal{NP}$-complete.

**Proof:** By the above discussion, CSat is $\mathcal{NP}$-hard. In order to show CSat to be in $\mathcal{NP}$, we use essentially the same reasoning as we did in showing Sat to be in $\mathcal{NP}$. The only difference is that we need an algorithm to evaluate a CNF expression, rather than an expression tree. It is a straightforward matter to adapt BoolEval (Figure 16.2) to evaluate a CNF expression $\mathcal{F}$ in $O(|\mathcal{F}|)$ time — the details are left as an exercise. It follows that CSat $\in \mathcal{NP}$. CSat is therefore $\mathcal{NP}$-complete. \hfill \Box

As a second example, let us further restrict our inputs by limiting the number of literals in each clause. We say that a CNF formula is in \textit{k-conjunctive normal form} (or $k$-CNF) if no clause contains more than $k$ literals. We then define $k$-Sat to be the problem of determining satisfiability for a given $k$-CNF formula. Though we won’t show it here, it turns out that $2$-Sat $\in \mathcal{P}$. In what follows, we will show that $3$-Sat is $\mathcal{NP}$-complete.

The fact that $3$-Sat $\in \mathcal{NP}$ follows immediately from the fact that CSat $\in \mathcal{NP}$, as $3$-Sat is the same problem as CSat, only with more restrictions placed on the input. Thus, the proof that CSat $\in \mathcal{NP}$ also proves that $3$-Sat $\in \mathcal{NP}$.

In order to show that $3$-Sat is $\mathcal{NP}$-hard, we have two choices: we can
reduce either SAT or CSat to 3-Sat. Reducing CSat to 3-Sat would appear to be less work, as instances of CSat are already in CNF. All that remains is to ensure that the number of literals in each clause is no more than 3. We will therefore show that CSat \( \leq_p 3\text{-Sat} \).

As in the previous reduction, we will not produce an equivalent formula. Instead, we will again introduce new variables. In addition, we will break up clauses that are too long into clauses containing only 3 literals.

Suppose our formula contains a clause \( C = \alpha_1 \vee \cdots \vee \alpha_m \), where \( m > 3 \).

We first introduce \( m - 3 \) new variables, \( u_1, \ldots, u_{m-3} \). We then construct the following clauses to replace \( C \):

- \( \alpha_1 \vee \alpha_2 \vee u_1 \);
- \( \neg u_i \vee \alpha_{i+2} \vee u_{i+1} \) for \( 1 \leq i \leq m - 4 \); and
- \( \neg u_{m-3} \vee \alpha_{m-1} \vee \alpha_m \).

We first claim that any assignment of boolean values that satisfies \( C \) can be extended to an assignment that satisfies each of the new clauses. To see why, first observe that if \( C \) is satisfied, then \( \alpha_i \) must be true for some \( i \). We can then set \( u_1, \ldots, u_{i-2} \) to true and \( u_{i-1}, \ldots, u_{m-3} \) to false. Then each of the first \( i - 2 \) clauses is satisfied because \( u_1, \ldots, u_{i-2} \) are true. The \((i - 1)\)st clause, \( \neg u_{i-2} \vee \alpha_i \vee u_{i-1} \) is satisfied because \( \alpha_i \) is true. Finally, the remaining clauses are satisfied because \( \neg u_{i-1}, \ldots, \neg u_{m-3} \) are true.

We now claim that any assignment that satisfies the new clauses will also satisfy \( C \). Suppose to the contrary that all the new clauses are satisfied, but that \( C \) is not satisfied — i.e., that \( \alpha_1, \ldots, \alpha_m \) are all false. Then in order for the first clause to be satisfied, \( u_1 \) must be true. Likewise, it is easily shown by induction on \( i \) that each \( u_i \) must be true. Then the last clause is not satisfied — a contradiction.

If we apply the above transformation to each clause having more than 3 literals in a CNF formula \( F \) and retain those clauses with no more than 3 literals, then the resulting 3-CNF formula is satisfiable if \( F \) is satisfiable. Furthermore, it is not hard to implement this reduction in \( O(|F|) \) time — the details are left as an exercise. Hence, CSat \( \leq_p 3\text{-Sat} \). We therefore conclude that 3-Sat is \( \mathcal{NP} \)-complete.

### 16.5 Vertex Cover and Independent Set

So far, all of the problems that we have shown to be \( \mathcal{NP} \)-complete are satisfiability problems for various kinds of boolean formulas. As we have
seen in earlier chapters, it is sometimes possible to reduce a problem $A$ to another problem $B$ that at first looks nothing like problem $A$. By applying this technique to polynomial many-one reducibility, we can identify $\mathcal{NP}$-complete problems in other domains.

For example, let us consider the vertex cover problem, which we will denote $VC$. Let $G = (V, E)$ be an undirected graph. A vertex cover for $G$ is a subset $C \subseteq V$ such that for each edge $\{u, v\} \in E$, $C \cap \{u, v\} \neq \emptyset$; i.e., at least one endpoint of each edge is contained in the vertex cover. The vertex cover problem is to decide whether a given undirected graph has a vertex cover of size $k$, where $k$ is a given positive integer.

To show that $VC \in \mathcal{NP}$, we will treat bit strings as arrays $A[0..m-1]$ of boolean values. We can interpret an array $A[0..m-1]$ as describing a subset $S$ of the vertices $\{0, 1, \ldots, n-1\}$ such that for $0 \leq i < n$, $i \in S$ iff $i < m$ and $A[i]$ is true. It is then an easy matter to check, in time linear in the size of a graph $G$, whether $A[0..m-1]$ denotes a vertex cover of $G$ with size $k$ — the details are left as an exercise. Therefore, $VC \in \mathcal{NP}$.

In order to show that $VC$ is $\mathcal{NP}$-hard, we need to reduce one of the three satisfiability problems to it. We will use $3$-Sat because $3$-CNF formulas have a simpler structure than either CNF or arbitrary boolean formulas. Still, it is not immediately clear how we can construct, from a given $3$-CNF formula $F$, an undirected graph $G$ and a positive integer $k$ such that $G$ has a vertex cover of size $k$ iff $F$ is satisfiable.

One rather simplistic approach is first to decide whether $F$ is satisfiable, then to construct one of two fixed graphs — one that has a vertex cover of size 1, or one that does not. However, because $3$-Sat is $\mathcal{NP}$-hard, we cannot decide in polynomial time whether $F$ is satisfiable unless $\mathcal{P} = \mathcal{NP}$. As a result, such an approach will probably never work.

Instead, we need to construct an instance of $VC$ whose solution will give us a solution to our original instance of $3$-Sat. In order to do this, we should try to see what the two problems have in common. A particularly useful technique is to compare the proofs of membership in $\mathcal{NP}$. Often we can find a reduction that has the side-effect of transforming each proof $\phi \in B$ for one problem to a proof $\phi'$ for the other.

Let $F$ be a given $3$-CNF formula with $n$ clauses, $C_1, \ldots, C_n$. For $1 \leq i \leq n$, let $\alpha_{i1}$, $\alpha_{i2}$, and $\alpha_{i3}$ be the three literals in clause $C_i$ (if there are fewer than three literals in $C_i$, we set $\alpha_{i3}$ and, if necessary, $\alpha_{i2}$ to equal $\alpha_{i1}$). A proof for this instance of $3$-Sat represents an assignment of boolean values to the variables. A proof for an instance of $VC$ represents a set of vertices. Perhaps we can associate the selection of a boolean value to the selection of one of two possible vertices. In particular, let us construct, for each variable
NP-COMPLETENESS

Figure 16.7 The graph constructed from \((x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_3)\) in the reduction from 3-Sat to VC.

In order to complete the reduction, we need to ensure that any vertex cover of size \(k\) describes a satisfying assignment for \(\mathcal{F}\), and that for any satisfying assignment for \(\mathcal{F}\), there is a vertex cover of size \(k\) that describes it. To this end, we will add more structure to the graph we are constructing. We know that for a satisfying assignment, each clause contains at least one true literal. In order to model this constraint with a graph, let us construct, for each clause \(C_i\), the vertices \(c_{i1}, c_{i2},\) and \(c_{i3}\), along with the edges \(\{c_{i1}, c_{i2}\},\) \(\{c_{i2}, c_{i3}\},\) and \(\{c_{i3}, c_{i1}\}\). Then any vertex cover must contain at least two of these three vertices.

Finally, for \(1 \leq i \leq n\) and \(1 \leq j \leq 3\), we construct an additional edge \(\{c_{ij}, \alpha_{ij}\}\). For example, Figure 16.7 shows the graph constructed from the 3-CNF formula \((x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_3)\). By setting \(k = m + 2n\), where \(m\)
is the number of variables and $n$ is the number of clauses in $F$, we force any vertex cover of size $k$ to contain exactly one of the two vertices constructed for each variable and exactly two of the three vertices constructed for each clause. In order to cover all of the edges $\{c_{ij}, \alpha_{ij}\}$, the vertex cover must be such that in each clause $C_i$, there is at least one literal $\alpha_{ij}$ that belongs to the vertex cover. Thus, we can represent an assignment of true to a literal by including it in the vertex cover. We can now show the following lemma.

**Lemma 16.12** Let $F$ be a 3-CNF formula with $m$ variables and $n$ clauses, and let $G$ and $k$ be the graph and positive integer resulting from the above construction. Then $G$ has a vertex cover of size $k$ iff $F$ is satisfiable.

**Proof:** We must show the implication in both directions.

$\Rightarrow$: Suppose $G$ has a vertex cover $S$ of size $k = m + 2n$. Then $S$ must contain at least one of the two vertices $x_i$ and $\neg x_i$ for $1 \leq i \leq m$, plus at least two of the three vertices $c_{i1}$, $c_{i2}$, and $c_{i3}$ for $1 \leq i \leq n$. Because this gives a total of at least $k$ vertices, we conclude that $S$ must contain exactly one of $x_i$ and $\neg x_i$ and exactly two of $c_{i1}$, $c_{i2}$, and $c_{i3}$. Let us set $x_i$ to true iff $x_i \in S$. Consider any clause $C_i$. Let $j$ be such that $c_{ij} \notin S$. Because the edge $\{c_{ij}, \alpha_{ij}\}$ must be covered, $\alpha_{ij}$ must be in $S$. Therefore, at least one literal in $C_i$ is true. We conclude that $F$ is satisfiable.

$\Leftarrow$: Suppose $F$ is satisfiable. Let $A$ be a satisfying assignment. We can then construct a vertex cover $S$ as follows. First, for $1 \leq i \leq m$, if $x_i$ is true in $A$, we include $x_i$ in $S$; otherwise, we include $\neg x_i$. Thus, each of the edges $\{x_i, \neg x_i\}$ is covered. Then for $1 \leq i \leq n$, we include in $S$ two of $c_{i1}$, $c_{i2}$, and $c_{i3}$, so that the vertex that is not included is adjacent to an $\alpha_{ij} \in S$ (note that because $A$ is a satisfying assignment, such a vertex exists for each clause). Thus, $S$ is of size $m + 2n = k$ and covers all edges in $G$. □

It is easily seen that the above construction can be implemented to run in $O(m + n)$ time, or linear in the size of the formula — the details are left as an exercise. From Lemma 16.12, 3-Sat $\leq_m^{P}$ VC. Because 3-Sat is $NP$-hard, it follows that VC is $NP$-hard. Because we have shown that VC $\in NP$, we have the following theorem.

**Theorem 16.13** VC is $NP$-complete.

A problem closely related to VC is the independent set problem, which we denote IS. An independent set in an undirected graph $G$ is a subset $S$ of
the vertices such that no pair of vertices in $S$ is adjacent in $G$. The independent set problem is to decide, for a given undirected graph $G$ and natural number $k$, whether $G$ has an independent set of size $k$. The relationship between IS and VC is shown by the following theorem.

**Theorem 16.14** Let $G = (V, E)$ be an undirected graph and $S \subseteq V$. Then $S$ is an independent set iff $V \setminus S$ is a vertex cover.

**Proof:** We must show the implication in both directions.

$\Leftarrow$: Suppose $S$ is an independent set. Let $\{u, v\} \in E$. Because $u$ and $v$ cannot both be in $S$, at least one of them is in $V \setminus S$. It follows that $V \setminus S$ is a vertex cover.

$\Rightarrow$: Suppose $V \setminus S$ is a vertex cover. Let $u$ and $v$ be two vertices in $S$. Because $V \setminus S$ is a vertex cover containing neither $u$ nor $v$, $u$ cannot be adjacent to $v$. It follows that $S$ is an independent set. $\square$

Given this close relationship between the two problems, it is an easy matter to modify the proof that $VC \in \mathcal{NP}$ to show that $IS \in \mathcal{NP}$. Furthermore, it follows from Theorem 16.14 that for an undirected graph $G$ with $n$ vertices and a positive integer $k$, $G$ has an independent set of size $n - k$ iff $G$ has a vertex cover of size $k$. Clearly, we can construct $n - k$ in polynomial time; hence, $VC \leq^p IS$. We therefore have the following theorem.

**Theorem 16.15** $IS$ is $\mathcal{NP}$-complete.

### 16.6 3-Dimensional Matching

In this section, we will study a problem closely related to the bipartite matching problem of Section 14.3. The input will consist of three nonempty disjoint sets, $X$, $Y$, and $Z$, each having the same number of elements, and a set of triples $W \subseteq X \times Y \times Z$. We wish to decide if there is a subset $M \subseteq W$ such that each element of $X \cup Y \cup Z$ occurs in exactly one triple in $M$. We call this problem the 3-dimensional matching problem (3DM).

Note that if we were to use two disjoint sets instead of three, we could think of the two sets as the two vertex sets of a bipartite graph. The set of pairs (instead of triples) would then be directed edges. Our problem would then be that of deciding whether there is a matching including all
the vertices of this directed graph. 3DM is then the natural extension of this problem to 3-dimensional hypergraphs. Using the algorithm MATCHING (Figure 14.10 on page 463), we can decide the 2-dimensional version (2DM) in $O(na)$ time, where $n$ is the number of vertices and $a$ is the number of edges in the graph; thus, $2DM \in \mathcal{P}$. However, we will now show that $3DM$ is $\mathcal{NP}$-complete.

In order to show that $3DM \in \mathcal{NP}$, let us first denote an instance by

$$\begin{align*}
X &= \{x_1, \ldots, x_m\} \\
Y &= \{y_1, \ldots, y_m\} \\
Z &= \{z_1, \ldots, z_m\} \\
W &= \{w_1, \ldots, w_n\}.
\end{align*}$$

We interpret a bit string $\phi$ as encoding an array $A[1..k]$ such that each block of $b$ bits encodes an element of $A$, where $b$ is the number of bits needed to encode $n$. Any bit string that does not have length exactly $bm$ will be considered to be invalid. To verify that the array $A$ encoded by $\phi$ is a proof, we can check that

- $\phi$ is valid;
- $1 \leq A[i] \leq n$ for $1 \leq i \leq m$; and
- each element of $X \cup Y \cup Z$ belongs to some triple $w_{A[i]}$, where $1 \leq i \leq m$.

This can easily be done in $O(bm^2)$ time — the details are left as an exercise. Hence, $3DM \in \mathcal{NP}$.

In order to show that $3DM$ is $\mathcal{NP}$-hard, we need to reduce some $\mathcal{NP}$-complete problem to it. So far, we have identified five $\mathcal{NP}$-complete problems: three satisfiability problems and two graph problems. However, none of these bears much resemblance to $3DM$. We therefore make use of a principle that has proven to be quite effective over the years: when in doubt, try $3$-SAT.

As we did in showing $3$-SAT $\leq^P_{m} \mathcal{VC}$, we will begin by focusing on the proofs of membership in $\mathcal{NP}$ for two problems. Specifically, we want to relate the choice of a subset of $W$ to the choice of truth values for boolean variables. Let’s start by considering two triples, $(x, a_x, b_x)$ and $(\neg x, a_x, b_x)$, where $x$ is some boolean variable. If these are the only two triples containing $a_x$ or $b_x$, any matching must include exactly one of these triples. This choice could be used to set the value of $x$.

If we were to construct two such triples for each variable, we would then need to construct triples to represent the clauses. Using a similar idea,
we could introduce, for a given clause \( \alpha_i \lor \alpha_i \lor \alpha_i \), the triples \( \langle \alpha_{i1}, c_i, d_i \rangle \), \( \langle \alpha_{i2}, c_i, d_i \rangle \), and \( \langle \alpha_{i3}, c_i, d_i \rangle \) — one triple for each literal in the clause. Again, any matching must contain exactly one of these triples. If we let \( x \) be false when \( \langle x, a_x, b_x \rangle \) is chosen, then the triple chosen for the clause must contain a \textit{true} literal.

This construction has a couple of shortcomings, however. First, because each literal must occur exactly once in a matching, we can use a given variable to satisfy only one clause. Furthermore, if more than one literal is \textit{true} in a given clause, there may remain literals that are unmatched. These shortcomings should not be too surprising, as we could do essentially the same construction producing pairs instead of triples — the third components are redundant. Thus, if this construction had worked, we could have used the same technique to reduce 3-SAT to 2DM, which belongs to \( \mathcal{P} \). We would have therefore proved that \( \mathcal{P} = \mathcal{NP} \).

In order to overcome the first shortcoming, we need to enrich our construction so that we have several copies of each literal. To keep it simple, we will make one copy for each clause, regardless of whether the literal appears in the clause. We must be careful, however, so that when we choose the triples to set the boolean value, we must either take all triples containing \( x \) or all triples containing \( \neg x \). Because we are constructing triples rather than pairs, we can indeed accomplish these goals.

Let \( x_1, \ldots, x_n \) denote all the copies of the literal \( x \), and let \( \neg x_1, \ldots, \neg x_n \) denote all the copies of the literal \( \neg x \). We then introduce the following triples (see Figure 16.8):

- \( \langle x_i, a_{x_i}, b_{x_i} \rangle \) for \( 1 \leq i \leq n \);
- \( \langle \neg x_i, a_{x_i}, b_{x_i, i+1} \rangle \) for \( 1 \leq i \leq n - 1 \); and
- \( \langle \neg x_n, a_{x_n}, b_{x_1} \rangle \).

It is not too hard to see that in order to match all of the \( a_{x_i} \)'s and \( b_{x_i} \)'s, a matching must include either those triples containing the \( x_i \)’s or those triples containing the \( \neg x_i \)’s.

We can now use the construction described earlier for building triples from clauses, except that for clause \( i \), we include the \( i \)th copy of each literal in its triple. Thus, in any matching, there must be for each clause at least one triple containing a copy of a literal. However, there still may be unmatched copies of literals. We need to introduce more triples in order to match the remaining copies.

Suppose our 3-CNF formula \( \mathcal{F} \) has \( n \) clauses and \( m \) variables. Then our construction so far contains:
Figure 16.8 Triples for setting boolean values in the reduction from 3-Sat to 3DM, with $n = 4$

- $2mn$ copies of literals;
- $mn$ $a$s and $n$ $c$s; and
- $mn$ $b$s and $n$ $d$s.

In order to make the above three sets of equal size, we add $e_i$ to the second set and $f_i$ to the third set, for $1 \leq i \leq (m-1)n$. We then include all possible triples $\langle x_i, e_j, f_j \rangle$ and $\langle \neg x_i, e_j, f_j \rangle$ for $1 \leq i \leq n$ and $1 \leq j \leq (m-1)n$. Using this construction, we can now show the following theorem.

**Theorem 16.16** 3DM is $\mathcal{NP}$-complete.

**Proof:** We have shown that 3DM $\in \mathcal{NP}$. We will show that it is $\mathcal{NP}$-hard by showing that $3$-Sat $\leq_p$ 3DM. Specifically, we will show that the above construction is a polynomial-time many-one reduction.

We will first show that the construction can be computed in polynomial time. It is easily seen that the time required for the construction is pro-
portional to the number of triples produced. Suppose the CNF formula $F$ contains $m$ variables and $n$ clauses. The triples produced include

- $2mn$ triples for setting truth values;
- one triple for each literal in each clause, or at most $3n$ triples; and
- triples $\langle x_i, e_j, f_j \rangle$ and $\langle \neg x_i, e_j, f_j \rangle$ for each variable $x$, each $i$ such that $1 \leq i \leq n$, and each $j$ such that $1 \leq j \leq (m - 1)n$, or $2(m - 1)mn^2$ triples.

Thus, the total number of triples produced is at most

$$2mn + 3n + 2(m - 1)mn^2.$$  

Because this value is polynomial in the size of $F$, the construction can be done in polynomial time.

Let $W$ be the set of triples constructed. In order to complete the proof, we must show that $W$ contains a matching if $F$ is satisfiable.

$\Rightarrow$: Suppose $W$ contains a matching $M$. As we have argued above, for each variable $x$, $M$ must contain either those triples $\langle x_i, a_{xi}, b_{xi} \rangle$ or those triples $\langle \neg x_i, a_{xi}, b_{xi} \rangle$, for $1 \leq i \leq n$. Let us set $x$ to false iff the triples $\langle x_i, a_{xi}, b_{xi} \rangle$ belong to $M$. $M$ must also contain some triple $\langle \alpha_{ij}, c_i, d_i \rangle$ for $1 \leq i \leq n$. Because $\alpha_{ij}$ cannot also be in another triple in $M$, $\alpha_{ij}$ must be true. Thus, each clause contains at least one true literal, so that $F$ is satisfiable.

$\Leftarrow$: Suppose $F$ is satisfiable, and let $A$ denote a satisfying assignment of boolean values to the variables in $F$. We construct a matching $M$ as follows. First, if $x$ is false in $A$, we include $\langle x_i, a_{xi}, b_{xi} \rangle$ for $1 \leq i \leq n$; otherwise, we include $\langle \neg x_i, a_{xi}, b_{xi} \rangle$ for $1 \leq i \leq n$. Thus, each $a_{xi}$ and $b_{xi}$ is included exactly once. Then for clause $i$, because $A$ is a satisfying assignment there is at least one literal $\alpha_{ij}$ that is true in $A$. Because $\alpha_{ij}$ has not yet been included in $M$, we can include the triple $\langle \alpha_{ij}, c_i, d_i \rangle$ in $M$. Thus, $M$ includes each $c_i$ and $d_i$ exactly once.

At this point $M$ includes no item more than once, but does not include any of the $e_i$s or $f_i$s. Furthermore, because exactly $mn + n$ of the $x_i$s and $\neg x_i$s have been included, $(m - 1)n$ have not yet been included. Let $\beta_1, \ldots, \beta_{(m-1)n}$ denote the $x_i$s and $\neg x_i$s that have not yet been included. We complete $M$ by including $\langle \beta_i, e_i, f_i \rangle$ for $1 \leq i \leq (m - 1)n$. It is now easily seen that $M$ is a matching. $\square$
16.7 Partitioning and Strong \(NP\)-Completeness

In this section, we will look at partitioning problems related to the 0-1 knapsack problem of Section 12.4. These are the first \(NP\)-complete problems we will have seen in which numbers play a significant role. As we will see, the \(NP\)-completeness of a number problem does not always imply intractability. For this reason, we will introduce a stronger notion of \(NP\)-completeness.

The most basic partitioning problem consists of a set of items, each having a positive integer weight. The problem is to decide whether the items can be partitioned into two disjoint subsets having identical total weight. More formally, let \(w_1, \ldots, w_n\) denote the weights of the items. We wish to decide whether there is a subset \(S \subseteq \{1, \ldots, n\}\) such that

\[
\sum_{i \in S} w_i = \sum_{i \notin S} w_i.
\]

The problem is known as the partition problem, or \(\text{Part}\). We leave it as an exercise to show that \(\text{Part} \in \text{NP}\). We will show that \(\text{Part}\) is \(\text{NP}\)-hard, and therefore \(\text{NP}\)-complete.

Before showing the \(\text{NP}\)-hardness of \(\text{Part}\), however, we first observe that this problem is a special case of the 0-1 knapsack problem in which the values are equal to the weights and the weight bound \(W\) is half the total weight. In Section 12.4, we sketched an algorithm to solve this problem in \(O(nW)\) time. Clearly, this same algorithm can be applied to \(\text{Part}\). This would seem to imply that \(\text{Part} \in \text{P}\), so that showing \(\text{Part}\) to be \(\text{NP}\)-hard would amount to showing \(\text{P} = \text{NP}\).

However, the \(O(nW)\) algorithm does not prove that \(\text{Part} \in \text{P}\). The reason is that we have defined \(\text{P}\) to be the set of decision problems that can be decided in a time polynomial in the length of their inputs. We claim that \(nW\) is not necessarily polynomial in the length of the input to \(\text{Part}\). To see why, note that the number of bits required to encode an integer is logarithmic in the value of the integer; hence, the value is exponential in the length of the encoding. Because \(W\) is one of the integers given as input, \(nW\) is not bounded by a polynomial in the length of the input.

The relationship between the value of an integer and the length of its binary encoding is essential to the \(\text{NP}\)-hardness of \(\text{Part}\), as its proof will illustrate. We will now present that proof, which is a reduction from 3DM.

Let \(W, X, Y,\) and \(Z\) represent an instance of 3DM, where

- \(X = \{x_0, \ldots, x_{m-1}\}\);
- \(Y = \{y_0, \ldots, y_{m-1}\}\);
• $Z = \{z_0, \ldots, z_{m-1}\}$; and
• $W = \{w_0, \ldots, w_{n-1}\}$ such that each $w_i \in X \times Y \times Z$.

We will construct a weight for each triple, plus two additional weights.
Suppose $\langle x_i, y_j, z_k \rangle \in W$. The weight we construct for this triple will be $$(n + 1)^{2m+i} + (n + 1)^{m+j} + (n + 1)^k.$$ If we were to express this weight in radix $n + 1$, it would consist entirely of 1s and 0s and have exactly three 1s. The positions of the three 1s in this encoding determine the three components of the triple as follows:

- $i$ digits: $100 \cdots 00 \cdots 0100 \cdots 00 \cdots 0100 \cdots 0$
- $j$ digits: $m$ digits
- $k$ digits: $m$ digits

Consider any subset $S \subseteq W$. Clearly, no element of $X \cup Y \cup Z$ can occur in more than $n$ triples in $S$. Thus, when viewed in radix $n + 1$, the sum of the weights corresponding to the elements of $S$ describes the number of occurrences of each element of $X \cup Y \cup Z$ in $S$. Specifically, if we number the digits beginning with 0 for the least significant digit, then

- digit $2m+i$ gives the number of occurrences of $x_i$ in $S$, for $0 \leq i < m$;
- digit $m+j$ gives the number of occurrences of $y_j$ in $S$, for $0 \leq j < m$; and
- digit $k$ gives the number of occurrences of $z_k$ is $S$, for $0 \leq k < m$.

It follows that $S$ is a matching iff the sum of its corresponding weights is

$$M = \sum_{i=0}^{3m-1} (n + 1)^i,$$

which in radix $n + 1$ is simply $3m$ 1s.

In order to complete the construction, we need two more weights. Let $C$ denote the sum of the $n$ weights constructed so far. We construct the following two weights:

$A = 2C - M$

and

$B = C + M$.  

CHAPTER 16. \(\text{NP}-\text{COMPLETENESS}\) 534

Thus, the sum of all \(n + 2\) weights is \(4C\). Because \(A + B = 3C > 2C\), \(A\) and \(B\) cannot belong to the same subset in a partition. Furthermore, the subset containing \(A\) must also contain items corresponding to elements of \(W\) having total weight \(M\). Because these elements must form a matching, the weights we have constructed contain a partition iff \(W\) contains a matching.

To see that the time needed to construct the instance of \(\text{PART}\) is polynomial in the size of the instance of \(3\text{DM}\), we first observe that

\[
2C \leq 2 \sum_{i=0}^{3m-1} n(n+1)^i < 2(n+1)^{3m}.
\]

Therefore each weight constructed has a binary encoding with no more than \(1 + \lceil 3m \log(n + 1) \rceil\) bits. Because addition, subtraction, multiplication, and exponentiation can all be performed in a time polynomial in the number of bits in their operands (see Exercise 4.14, Section 10.1, Exercise 10.24, and Sections 15.3-15.4), the construction can clearly be performed in time polynomial in the size of the instance of \(3\text{DM}\). We therefore have the following theorem.

**Theorem 16.17** \(\text{PART}\) is \(\text{NP}\)-hard.

Note that in the above construction, the weights can become extremely large, though their lengths are all polynomial in the size of the instance of \(3\text{DM}\). It is not too hard to imagine that we might want to solve the partition problem for a large number of weights — thousands or perhaps even millions. However, when numbers represent physical quantities — including time — we don’t expect them to be very long. For example, about 300 bits are sufficient to encode in binary the estimated number of elementary particles in the universe. Thus, because there is an algorithm for \(\text{PART}\) whose running time is a low-order polynomial in the length of the input and the values encoded in the input, it seems unreasonable to consider this problem to be intractable.

In order to accommodate numbers in the input, we say that an algorithm is *pseudopolynomial* if its running time is bounded by some polynomial in the length of the input and the largest integer encoded in the input. Thus, the \(O(nW)\) algorithm for 0-1 knapsack (and hence partition) is pseudopolynomial. Whenever the numbers in a decision problem’s input refer to physical quantities, we consider the problem to be tractable if it has a pseudopolynomial algorithm. However, if the numbers are purely mathematical entities
(as, for example, in cryptographic applications), we consider the problem to be tractable only if it belongs to \( \mathcal{P} \).

We would also like to extend the notion of \( \mathcal{NP} \)-hardness to account for numbers in the input. To this end, we first define a way to restrict a decision problem so that no integer in an instance is too large. Specifically, for a decision problem \( X \) and a function \( f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \), we define \( X_f \) to be the restriction of \( X \) to instances \( x \) in which no integer has a value larger than \( f(|x|) \). We then say that \( X \) is \( \mathcal{NP} \)-hard \emph{in the strong sense} if there is a polynomial \( p \) such that \( X_p \) is \( \mathcal{NP} \)-hard. If, in addition, \( X \in \mathcal{NP} \), we say that \( X \) is \( \mathcal{NP} \)-complete \emph{in the strong sense}.

Suppose we were to find a pseudopolynomial algorithm for a strongly \( \mathcal{NP} \)-hard problem. When we restrict the problem so that its instances have integers bounded by some polynomial, the pseudopolynomial algorithm becomes truly polynomial, so that the restricted problem would be in \( \mathcal{P} \). Furthermore, this restricted problem is still \( \mathcal{NP} \)-hard in the ordinary sense. Thus, by Theorem 16.6, we would have shown that \( \mathcal{P} = \mathcal{NP} \). It therefore seems highly unlikely that there is a pseudopolynomial algorithm for any strongly \( \mathcal{NP} \)-hard problem.

In order to show a problem to be \( \mathcal{NP} \)-hard in the strong sense, we must ensure that the reduction produces numbers whose values are bounded above by some polynomial in the length of the instance we construct. The proof of Cook’s Theorem in Section 16.8 does not construct large integers; hence, \( \text{SAT} \) is \( \mathcal{NP} \)-complete in the strong sense. Furthermore, all of the \( \mathcal{NP} \)-hardness proofs we have presented so far, only the proof that \( \text{PART} \) is \( \mathcal{NP} \)-hard constructs integers whose values are not bounded by some polynomial in the length of the input. As a result, \( \text{CSAT} \), \( \text{3SAT} \), \( \text{VC} \), \( \text{IS} \), and \( \text{3DM} \) are all \( \mathcal{NP} \)-complete in the strong sense. However, these results are rather uninteresting because none of their instances contain numbers that can become large in comparison to the length of the input without rendering the problem trivial.

In what follows, we will show a problem with potentially large numbers to be \( \mathcal{NP} \)-complete in the strong sense. We will use a restricted form of polynomial many-one reduction motivated by the following theorem.

**Theorem 16.18** Let \( f \) be a polynomial many-one reduction from problem \( X \) to problem \( Y \), where \( X \) is \( \mathcal{NP} \)-hard in the strong sense. Suppose that \( f \) satisfies the following properties:

1. there is a polynomial \( p_1 \) such that \( p_1(|f(x)|) \geq |x| \) for every instance \( x \) of \( X \); and
2. there is a two-variable polynomial $p_2$ such that each integer constructed has a value no greater than $p_2(|x|, \mu(x))$, where $\mu(x)$ denotes the maximum value of any integer in $x$.

Then $Y$ is $\mathcal{NP}$-hard in the strong sense.

**Proof:** Because $X$ is $\mathcal{NP}$-hard in the strong sense, there is some polynomial $p$ such that $X_p$ is $\mathcal{NP}$-hard. If the reduction is then applied to $X_p$, all numbers constructed will have values bounded by $p_2(|x|, p(|x|))$, by Property 2. Furthermore, by Property 1, these values are no more than $p_2(p_1(|f(x)|), p(p_1(|f(x)|)))$, which is a polynomial in the length of the instance constructed. The reduction from $X_p$ to $Y$ therefore shows that $Y$ is $\mathcal{NP}$-hard in the strong sense. □

We say that a polynomial many-one reduction is a *pseudopolynomial* reduction, denoted by $\leq_{pp}^m$, if it satisfies the properties given in Theorem 16.18.

Let us now consider other partitioning problems. For a fixed natural number $k > 1$, the $k$-partition problem ($k$-PART) is defined as follows. The input consists of $kn$ items, each having a positive integer weight, such that the sum of the weights is $Bn$ for some positive integer $B$. Furthermore, each weight $w$ must satisfy

$$
\frac{B}{k+1} < w < \frac{B}{k-1}.
$$

The question we ask is whether the $kn$ items can be partitioned into $n$ disjoint subsets, each having total weight exactly $B$. Note that the constraints on the weights imply that each subset will contain exactly $k$ items.

In what follows, we will show that 4-Part is $\mathcal{NP}$-hard in the strong sense. We leave it as exercises to show that

- $k$-PART $\in \mathcal{NP}$ for all $k > 1$;
- 3-Part is $\mathcal{NP}$-complete in the strong sense; and
- 2-Part $\in \mathcal{P}$.

We will show that 3DM $\leq_{pp}^m$ 4-Part. The reduction will be somewhat similar to the reduction from 3DM to Part, but we must be careful that the weights we construct are not too large. Let us describe an instance of 3DM using the same notation as we did for the earlier reduction. We will assume that each element occurs in at least one triple. Otherwise, there is no matching, and we can create an instance with 7 items having weight 6.
and one item having weight 8, so that the total weight is 50, and $B = 25$. Clearly, $25/5 < 6 < 8 < 25/3$; hence, this is a valid instance, but there is clearly no way to form a subset with weight 25.

We will construct, for each triple $\langle x_i, y_j, z_k \rangle \in W$, four weights: one weight for each of $x_i$, $y_j$, and $z_k$, plus one weight for the triple itself. Because each element of $X \cup Y \cup Z$ can occur in several triples, we may construct several items for each element. Exactly one of these will be a matching item. All non-matching items constructed from the same element will have the same weight, which will be different from that of the matching item constructed from that element. We will construct the weights so that in any 4-partition, the item constructed from a triple must be grouped with either the matching items constructed from the elements of the triple, or three non-matching items — one corresponding to each element of the triple. In this way, a 4-partition will exist iff $W$ contains a matching.

As in the previous reduction, it will be convenient to view the weights in a particular radix $r$, which we will specify later. In this case, however, the weights will contain only a few radix-$r$ digits. We will choose $r$ to be large enough that when we add any four of the weights we construct, each column of digits will have a sum strictly less than $r$; hence, we will be able to deal with each digit position independently in order to satisfy the various constraints. Note that if we construct the weights so that for every triple, the sum of the four weights constructed is the same, then this sum will be $B$.

We will use the three low-order digits to enforce the constraint that the four items within any partition must be derived from some triple and its three components. To this end, we make the following assignments:

- For any weight constructed from $x_i \in X$, we assign $i + 1$ to the first digit and 0 to the second and third digits.

- For any weight constructed from $y_j \in Y$, we assign $j + 1$ to the second digit and 0 to the first and third digits.

- For any weight constructed from $z_k \in Z$, we assign $k + 1$ to the third digit and 0 to the first two digits.

- For a triple $\langle x_i, y_j, z_k \rangle$, we assign the first three digits the values $2m - i$, $2m - j$, and $2m - k$, respectively.

$B$ will therefore have $2m + 1$ as each of its three low-order digits.

Note that because each weight constructed from a triple has a value of at least $m + 1$ in each of its three low-order digits, no two of these weights
can be grouped together. Furthermore, because all three low-order digits of any weight constructed from an element of \(X \cup Y \cup Z\) have values no more than \(m\), and at least two of these values are 0, in order to reach a sum of \(B\) with only four weights, at least one must correspond to a triple. Thus, in any 4-partition, each group must contain exactly one weight corresponding to a triple, and the other three weights must correspond to the elements of that triple.

We will use the fourth digit to enforce the constraint that for any four items grouped together, the items corresponding to elements from \(X \cup Y \cup Z\) are either all matching items or all non-matching items. We therefore assign the fourth digits as follows:

- for each matching item: 1;
- for each non-matching item from \(X \cup Y\): 0;
- for each non-matching item from \(Z\): 3; and
- for each item from \(W\): 0.

The fourth digit of \(B\) will therefore be 3. Furthermore, because any group must have one item from each of \(W\), \(X\), \(Y\), and \(Z\), it must contain either three matching items or three non-matching items. Setting the digits in this way therefore ensures that the set of weights has a 4-partition iff \(W\) contains a matching.

Finally, in order to ensure that each weight is greater than \(B/5\) and less than \(B/3\), we set the fifth digit of all weights to 1. As a result, the fifth digit of \(B\) is 4. We will see that by choosing \(r\) to be sufficiently large, all weights will be within the proper range.

To summarize, for a triple \(\langle x_i, y_j, z_k \rangle\), we have the weight

\[
  r^4 + (2m - k)r^2 + (2m - j)r + 2m - i.
\]

For the elements \(x_i \in X\), \(y_j \in Y\), and \(z_k \in Z\), we have for the matching items the weights

\[
\begin{align*}
  r^4 & + r^3 + i + 1 \\
  r^4 & + r^3 + (j + 1)r \\
  r^4 & + r^3 + (k + 1)r^2
\end{align*}
\]

and for any non-matching item the weights

\[
\begin{align*}
  r^4 & + i + 1 \\
  r^4 & + (j + 1)r \\
  r^4 & + 3r^3 + (k + 1)r^2
\end{align*}
\]
Furthermore, 
\[ B = 4r^4 + 3r^3 + (2m + 1)r^2 + (2m + 1)r + 2m + 1. \]

To complete the reduction, we must assign a value to \( r \). As we have observed, \( r \) must be larger than the sum of any four digits occurring in the same column. Thus, \( r \) must be strictly larger than both \( 8m \) and \( 12 \). Because \( m \geq 1 \), we can satisfy these constraints by setting \( r = 13m \). We then have 
\[
\begin{align*}
B/5 &< (4r^4 + 4r^3)/5 \\
&< (4r^4 + r^4)/5 \\
&= r^4,
\end{align*}
\]
so that every weight is larger than \( B/5 \). Furthermore, each weight is less than 
\[
\begin{align*}
r^4 + 4r^3 &< r^4 + r^4/3 \\
&= 4r^4/3 \\
&< B/3.
\end{align*}
\]

We must now show that the above reduction is pseudopolynomial. We first observe that no weight is larger than \( 2r^4 = 2(13m)^4 \), which is polynomial in the size of the instance of 3DM. Furthermore, it is easily seen that all the weights can be constructed in a time linear in the size of the instance of 3DM. We therefore have the following theorem.

**Theorem 16.19** 4-PART is \( \mathcal{NP} \)-hard in the strong sense.

### 16.8 Proof of Cook’s Theorem

In this section, we will present a proof of Cook’s Theorem, namely, that SAT is \( \mathcal{NP} \)-complete. As we have already observed in Section 16.1, SAT \( \in \mathcal{NP} \). It therefore remains to be shown that SAT is \( \mathcal{NP} \)-hard. Furthermore, because we have used Cook’s Theorem either directly or indirectly to show all of our \( \mathcal{NP} \)-hardness results, we cannot use any of these results to prove Cook’s Theorem. Because we cannot show the \( \mathcal{NP} \)-hardness of SAT by reducing any known \( \mathcal{NP} \)-hard problem to it, we must directly use the definition of \( \mathcal{NP} \)-hardness — that is, we must directly show that for every problem \( X \in \mathcal{NP} \), \( X \leq_m \text{SAT} \).
Our proof will therefore involve a generic reduction. This kind of reduction is more abstract in that we begin with an arbitrary $X \in \mathcal{NP}$. Specifically, the same reduction should work for any problem $X$ that we might choose from $\mathcal{NP}$. Thus, the only assumption we can make about $X$ is that it satisfies the definition of $\mathcal{NP}$: there exist a polynomial $p(n)$ and a decision problem $Y \subseteq I \times \mathcal{B}$, where $I$ is the set of instances of $X$, such that

- $Y \in \mathcal{P}$ — that is, there exist a polynomial $p'(n)$ and an algorithm $A$ that takes an element $x \in I$ and an element $\phi \in \mathcal{B}$ as its inputs and decides within $p'(|x| + |\phi|)$ steps whether $(x, \phi) \in Y$;
- for each $x \in I$, $x \in X$ iff there is a proof $\phi \in \mathcal{B}$ such that $(x, \phi) \in Y$; and
- for each $x \in X$, there is a proof $\phi \in \mathcal{B}$ such that $(x, \phi) \in Y$ and $|\phi| \leq p(|x|)$.

For our reduction, we need to construct, for a given instance $x \in I$, a boolean formula $F$ that is satisfiable iff $x \in X$. Equivalently, $F$ must be satisfiable iff there is a $\phi \in \mathcal{B}$ such that $|\phi| \leq p(|x|)$ and $(x, \phi) \in Y$. Our reduction therefore will construct from $x$ a formula $F$ that in some sense simulates the algorithm $A$ on $x$ and some unknown $\phi$, where $|\phi| \leq p(|x|)$.

The input for our reduction is the instance $x$. However, because our reduction is generic, it must work for any algorithm $A$ and any polynomials $p(n)$ and $p'(n)$ satisfying the above constraints. Therefore, $A$, $p$, and $p'$ are in a sense additional inputs to our reduction. What makes constructing such a reduction rather difficult is the fact that one of these additional inputs is an algorithm. In order to be able to handle an algorithm as input to an algorithm, we need to define more precisely what we mean by an algorithm.

Rather than formalizing all of the constructs we have been using in our algorithms, we will instead simplify matters by defining a lower-level model of computation, which we will call a random access machine, or RAM. Thus, in constructing a boolean formula to simulate a RAM, we will in essence be defining an interpreter for a simple machine language. Such a task is much simpler than defining an interpreter for a high-level language, such as the notation we have been using to present our algorithms.

In order to maintain some consistency between this computational model and the algorithms we have designed, we will assume that a RAM consists of the following:

- a fixed program consisting of a sequence of $P > 0$ instructions numbered 0 through $P - 1$;
• a program counter, which is initially 0 and can store any natural number less than $P$;
• countably infinitely many memory locations, each of which is initially 0 and can store any natural number;
• two input streams from which values may be read one bit at a time; and
• a single output bit, which is produced when the program terminates.

Because we will be using this model only for representing an algorithm for deciding whether $(x, \phi) \in Y$, we need exactly two input streams, one for $x$ and one for $\phi$. Furthermore, we can represent a “yes” output by setting the output bit to 1.

We will assume that each memory location is addressed by a unique natural number. Each machine will then have the following instruction set:

- **INPUT($i, l$):** Stores the next bit from input stream $i$, where $i$ is either 0 or 1, in memory location $l$. If all of the input has already been read, the value 2 is stored.
- **LOAD($n, l$):** Stores the natural number $n$ at memory location $l$.
- **COPY($l_1, l_2$):** Copies the value stored at location $l_1$ into location $l_2$.
- **GOTO($p$):** Changes the value of the program counter to $p$.
- **IFLEQ($l_1, l_2, p$):** If the value at location $l_1$ is less than or equal to the value at location $l_2$, changes the value of the program counter to $p$.
- **ADD($l_1, l_2$):** Adds the value in location $l_1$ to the value in location $l_2$, saving the result in location $l_2$.
- **SUBTRACT($l_1, l_2$):** Subtracts the value in location $l_1$ from the value in location $l_2$, saving the result in location $l_2$.
- **SHIFT($l$):** Replaces the value $n$ stored in location $l$ with $\lfloor n/2 \rfloor$.
- **HALT($b$):** Terminates the program with output $b$, which must be either 0 or 1.

In addition, wherever a memory location is used, an *indirection operator* “*” may be added. The expression “*l” indicates that the location referenced
by the value stored in location $l$ should be used. Thus, for example, if 21 is stored at 0, 2 is stored at 1, and 52 is stored at 2, the instruction

$$\text{Add}(0, 1)$$

will add 21 (stored at 0) to 52 (stored at 2, the value referenced by 1), and store the sum 73 in location 2. Indirection operators cannot be nested.

Each instruction that does not explicitly change the program counter will increment it by 1. We will assume that any time an instruction cannot be executed (e.g., because a larger number is subtracted from a smaller number or the program counter would index beyond the program), the program will immediately terminate with output 0.

We will now argue somewhat informally that for any decision problem $X$, there is a deterministic polynomial-time algorithm deciding $X$ iff there is a polynomial-time RAM deciding $X$. First, if we have a deterministic polynomial-time algorithm $A$, we can build a RAM to execute $A$ using standard compiling techniques. Some statements in an algorithm may require much more time when compiled to a RAM. For example, to implement a multiplication, we can use the following top-down formulation:

$$ab = \begin{cases} 
0 & \text{if } b = 0 \\
(a + a)^{b/2} & \text{if } b \text{ is positive and even} \\
(a + a)^{b+1}/2 + a & \text{if } b \text{ is odd.}
\end{cases}$$

To determine whether $b = 0$, we can simply check whether $b \leq 0$. To determine whether $b$ is even, we can copy $b$ to $c$, shift $c$ yielding $d = \lfloor b/2 \rfloor$, and subtract $d + d$ from $b$. If the result is less than or equal to 0, then $b$ is even; otherwise, $b$ is odd. This technique can be implemented with a loop that runs in time linear in the number of bits in $b$.

In order to implement data structures, we need a memory manager for a RAM. Because we have infinitely many memory locations, only finitely many of which may be in use at any given time, we can use one memory location $l_1$ to store a value $\text{avail}$ such that for every $l \geq \text{avail}$, $l$ is unused. If we need to allocate $n$ locations, we can copy $l_1$ to some memory location $l_2$ and add $n$ to $l_1$. To access the $i$th of these $n$ locations, we add $i$ to $l_2$ and use the indirection operator on the result. Thus, it is a straightforward matter to implement the data structures in this text. Specifically, because we can implement a STACK, we can implement function and operation calls, including recursion.

It is important to realize that even though some operations are less efficient on a RAM, there is some polynomial $p(n)$ that bounds the running
time of any simulation of an algorithmic step. Thus, if we have an algorithm that runs in $p'(n)$ steps, the RAM simulation will run in no more than $p'(p(n))$ steps, which is still polynomial in $n$. Furthermore, if the values of the natural numbers in the algorithm are bounded by $p'(n)$, then so are the values in the memory locations of the RAM.

Conversely, it is not hard to write an algorithm to simulate a polynomial-time RAM. We can use variables to store the value of the program counter and the indices of the next available bits of the two input streams. In addition, we can use a VARRAY (see Section 9.5) to store the memory locations used by the RAM. Note that because the values of all memory locations must always be less than $2^{p(n)}$, where $p(n)$ is some polynomial in the number of bits in the two input streams, all memory locations that can be accessed have addresses strictly less than $2^{p(n)}$. We can therefore use a VARRAY of size $2^{p(n)}$ to keep track of the RAM’s memory. It is then a straightforward matter to simulate the RAM using constant time for each instruction, plus some constant time for initialization.

Using RAMs, we now have a slightly different characterization of $NP$. Specifically, $NP$ is the set of all decision problems $X$ such that there exist

- polynomials $p(n)$ and $p'(n)$; and
- a RAM $M$ deciding a problem $Y \subseteq I \times B$, where $I$ is the set of instances for $X$;

such that

- $M$ terminates within $p'(|x| + |\phi|)$ steps on input $(x, \phi)$;
- all memory locations of $M$ maintain values strictly less than $2^{p'(|x| + |\phi|)}$ given input $(x, \phi)$;
- for each $x \in I$, $x \in X$ iff there is a proof $\phi \in B$ such that $(x, \phi) \in Y$; and
- for each $x \in X$, there is a $\phi \in B$ such that $(x, \phi) \in Y$ and $|\phi| \leq p(|x|)$.

Thus, to reduce an arbitrary problem $X \in NP$ to SAT, we need to construct from a given $x \in I$, where $I$ is the set of instances of $X$, a boolean formula $F$ such that $F$ is satisfiable iff there is a $\phi \in B$ with $|\phi| \leq p(|x|)$ such that $M$ outputs 1 on input $(x, \phi)$. Furthermore, the running time of the construction of $F$ must be bounded by some polynomial in $|x|$. In designing the construction, we may utilize $p(n)$, $p'(n)$, and $M$, which depend only on
the problem $X$; however, the instance $x$ is the input for the construction, so that we cannot know it in advance.

Because we can use any polynomial-time $M$ that decides $Y$, we can simplify matters further by making a couple of assumptions about $M$. First, we can assume that $M$ contains at least one \texttt{HALT}(1) instruction — if there is no input yielding a “yes” answer, we can always include such an instruction at an unreachable location. Second, because statements that cannot be executed due to error conditions have the same effect as \texttt{HALT}(0), and because the instruction set is powerful enough to check any run-time error conditions, we can assume that all statements can be executed without error.

In addition, we make some simplifying assumptions regarding the polynomials $p$ and $p'$. First, we note that by removing any negative terms from $p'$, we obtain a polynomial that is nondecreasing and never less than the original polynomial. Thus, we can assume that $p'$ is nondecreasing, so that $p'(|x| + p(|x|))$ will give an upper bound on the number of steps executed by $M$ on an input $(x, \phi)$ with $|\phi| \leq p(|x|)$. Furthermore, because $p'(n + p(n))$ is a polynomial, we can assume that $p'(n)$ is an upper bound on the number of steps taken by $M$ on any input $(x, \phi)$ such that $|x| = n$ and $|\phi| \leq p(n)$. Note that with these assumptions, $p'(n) \geq p(n)$ for all $n$. We can therefore choose $p(n) = p'(n)$, so that we can use a single polynomial $p$ to bound both $|\phi|$ and the number of steps executed by $M$. Finally, we can choose $p$ so that $p(n) \geq n$ for all $n \in \mathbb{N}$.

As the first step in our construction, we need boolean variables to represent the various components of the state of $M$ at various times in its execution. First, we need variables describing the input sequences $x$ and $\phi$. For $x$, we will use the variables $\overline{x}[k]$ for $1 \leq k \leq n$, where $n$ is the length of $x$. Because $\phi$ is unknown, even during the execution of the construction, we cannot know its exact length; however, we do know that its length is no more than $p(n)$. We therefore will use the variables $\overline{\phi}[k]$ for $1 \leq k \leq p(n)$ to represent $\phi$.

We also need variables to keep track of which bits are unread at each step of the execution of $M$. For this purpose, we will use the variables $\hat{x}_i[k]$ for $0 \leq i \leq p(n)$ and $0 \leq k \leq n$, plus the variables $\hat{\phi}_i[k]$ for $0 \leq i \leq p(n)$ and $0 \leq k \leq p(n)$. We want $\hat{x}_i[k]$ to be true iff the $k$th bit of $x$ has not been read after $i$ execution steps. Likewise, we want $\hat{\phi}_i[k]$ to be true iff the $k$th bit of $\phi$ exists and has not been read after $i$ execution steps.

We then need to record the value of the program counter at each execution step. We will use the variables $p_{ij}$ for $0 \leq i \leq p(n)$ and $0 \leq j < P$ for this purpose, where $P$ denotes the number of instructions in the program of
We want \( p_{ij} \) to be \text{true} iff the program counter has a value of \( j \) after \( i \) execution steps.

Recording the values of the memory locations at each execution step presents more of a challenge. Because a memory location can contain any value less than \( 2^{p(n)} \), \( M \) can access any memory location with an address less than \( 2^{p(n)} \). If we were to construct variables for each of these locations, we would end up with exponentially many variables. We cannot hope to construct a formula containing this many variables in a polynomial amount of time. However, the number of memory locations accessed by any instruction is at most four — the number accessed by \( \text{COPY}(\ast l_1, \ast l_2) \) in the worst case. As a result, \( M \) can access a total of no more than \( 4^{p(n)} \) different memory locations. We can therefore use a technique similar to the implementation of a \texttt{VArray} (see Section 9.5) in order to keep track of the memory locations actually used.

Specifically, we will let the variables \( a_j[k] \), for \( 1 \leq j \leq 4^{p(n)} \) and \( 1 \leq k \leq p(n) \), denote the value of the \( k \)th bit of the address of some location \( l_j \), where the first bit is the least significant bit. Here, we will let \text{true} represent 1 and \text{false} represent 0. Then the variables \( v_{ij}[k] \), for \( 0 \leq i \leq p(n), 1 \leq j \leq 4^{p(n)} \), and \( 1 \leq k \leq p(n) \), will record the value of the \( k \)th bit of the value stored at location \( l_j \) after \( i \) execution steps. We will make no requirement that location \( l_j \) actually be used by \( M \), nor do we require that \( l_j \) be a different location from \( l_j' \) when \( j \neq j' \).

Finally, we will use the additional variables \( c_i[0..p(n)] \) and \( d_i[1..p(n)] \) for \( 1 \leq i \leq p(n) \). We will explain their purposes later.

Before we describe the formula \( F \) that we will construct, let us first define some abbreviations, or “macros”, that will make the description of \( F \) simpler. First, we will define the following:

\[
\text{IF}(y, z) = \neg y \lor z.
\]

This abbreviation specifies that if \( y \) is \text{true}, then \( z \) must also be \text{true}. However, if \( y \) is \text{false}, then no constraint is placed upon \( z \). Note that such an expression can be constructed in \( O(1) \) time.

We can extend the above abbreviation to specify an if-then-else construct:

\[
\text{IFELSE}(y, z_1, z_2) = \text{IF}(y, z_1) \land \text{IF}(\neg y, z_2).
\]

This specifies that if \( y \) is \text{true}, then \( z_1 \) is \text{true}, but if not, then \( z_2 \) is \text{true}. Clearly it can be constructed in \( O(1) \) time.
We will now define an abbreviation for the specification that two variables \(y\) and \(z\) are equal:

\[
\text{Eq}(y, z) = \text{IFElse}(y, z, \neg z).
\]

Again, such an expression can be constructed in \(O(1)\) time.

We can extend this abbreviation in a couple of ways. First, we can use one of the constants \textit{true} or \textit{false} in place of one of the variables. This will be useful, for example, if we want to specify that some variable has the same value as some specific bit of the input \(x\), say \(x[k]\). Because \(x[k]\) is not a variable in the formula, it cannot appear in the formula; however, when we’re designing the construction, we don’t know this value. It would therefore be convenient to be able to write \(\text{Eq}(y, x[k])\), and to let the construction fill in the appropriate value for \(x[k]\). We then define

\[
\text{Eq}(y, \text{true}) = y
\]

and

\[
\text{Eq}(y, \text{false}) = \neg y.
\]

We can also extend the abbreviation to arrays of variables as follows:

\[
\text{Eq}(y[1..n], z[1..n]) = \bigwedge_{k=1}^{n} \text{Eq}(y[k], z[k]).
\]

Such an expression can be constructed in \(O(n)\) time. For different sizes of arrays, the running time will be proportional to the number of elements. To aid in readability, we will typically drop the range of subscripts when the entire array is used.

When we specify the behavior of a program step, we need a way of checking to see if \(a_j\) records some particular memory location \(l\). Let \(l[1..p(n)]\) be the bits comprising \(l\). We can check whether a given \(a_j[1..p(n)] = l[1..p(n)]\) using the Eq abbreviation. However, we may in many cases need to check whether \(a_j\) records the memory location indirectly addressed by \(l\). For this test, we use the following abbreviation:

\[
\text{IND}(i, j, l) = \bigvee_{j' = 1}^{4p(n)} (\text{Eq}(a_{j'}, l) \land \text{Eq}(a_j, v_{ij'})).
\]

Because \(a_j, a_{j'}, l,\) and \(v_{ij'}\) are arrays of \(p(n)\) elements, this expression can be constructed in \(O(p^2(n))\) time.
Finally, we will need some abbreviations for specifying the behavior of arithmetic and comparison instructions. We can express all of these behaviors using the specification of a sum. Thus, we need to express that the sum of \( y[1..p(n)] \) and \( z[1..p(n)] \) is \( s[1..p(n)] \). In order to express this constraint, we will need to represent the “carry” bits used to compute the sum. For this purpose, we will use the variables \( c_i[0..p(n)] \) for some \( i \). Specifically, for \( 1 \leq k \leq p(n) \), \( c_i[k] \) will be the carry from the sum of \( y[k] \), \( z[k] \), and \( c_i[k-1] \), and the value of \( c_i[0] \) will be \text{false}, denoting 0.

We first observe that the low-order bit of the sum of \( y[k] \), \( z[k] \), and \( c_i[k-1] \) is the exclusive-or of the three bits. We therefore first define an abbreviation for specifying exclusive-or:

\[
\text{Xor}(y, z) = (y \lor z) \land \neg (y \land z).
\]

We then observe that the carry bit of this sum is 1 iff at least two of the three bits are 1. Stated another way, the carry bit is 1 iff either

- both \( y[k] \) and \( z[k] \) are 1; or
- \( c_i[k-1] \) is 1 and either \( y[k] \) or \( z[k] \) is 1.

We therefore define the following abbreviation:

\[
\text{Sum}(y, z, s, i) = \neg c_i[0] \land \neg c_i[p(n)] \land \bigwedge_{k=1}^{p(n)} (\text{Eq}(s[k], \text{Xor}(\text{Xor}(y[k], z[k]), c_i[k-1]))) \land \text{Eq}(c_i[k], (y[k] \land z[k]) \lor (c_i[k-1] \land (y[k] \lor z[k]))).
\]

This expression states that \( y[1..p(n)] + z[1..p(n)] = s[1..p(n)] \) and that \( c_i[0..p(n)] \) give the carry bits of this sum. This expression can be constructed in \( O(p(n)) \) time.

Let us now begin to construct our formula \( \mathcal{F} \). This formula will be the conjunction of a number of sub-formulas, each of which will specify some constraint on the values of the boolean variables. These constraints together will encode the requirement that \( M \) gives a “yes” answer on input \((x, \phi)\), where \( x \) is the given input for problem \( X \). Because we will not specify the value of \( \phi \), the formula will be satisfiable iff there is some \( \phi \) with length at most \( p(n) \) such that \( M \) yields an output of 1 on \((x, \phi)\). Furthermore, we will show that we can construct this formula within a polynomial amount of time.
For the first constraint, we will specify that for $0 \leq i \leq p(n)$, there is at most one $j$ such that $p_{ij}$ is true. Thus, there will be no ambiguity as to the value of the program counter at each execution step. We specify this constraint with the sub-formula,

$$F_1 = \bigwedge_{i=0}^{p(n)} \bigwedge_{j=0}^{P-2} \bigwedge_{j'=j+1}^{P-1} \text{IF}(p_{ij}, \lnot p_{ij'}).$$

Because $P$ is a constant depending only on the problem $X$, this sub-formula can be constructed in $O(p(n))$ time.

We now need to specify some initialization constraints. The first is simply that the program counter has an initial value of 0. We specify this constraint with the sub-formula,

$$F_2 = p_{0,0}.$$

Clearly, this sub-formula can be constructed in $O(1)$ time.

We also need to specify that the variables $\overline{x}[1..n]$ encode the input string $x$. Let the $k$th bit of $x$ be denoted by $x[k]$ for $1 \leq k \leq n$. We then construct

$$F_3 = \text{EQ}(\overline{x}, x).$$

Because $x$ and $\overline{x}$ are arrays of $n$ elements, this sub-formula can be constructed in $O(n)$ time.

In addition, we need to specify that all bits in both input streams are initially unread. Notice that for each $i$, we have defined a variable $\hat{x}_i[0]$, but there is no $\overline{x}[0]$. The purpose of these variables is so that whenever $x[k]$ is the next bit to be read, $\hat{x}_i[k-1]$ is false and $\hat{x}_i[k]$ is true. In order to enforce this constraint initially, we construct

$$F_4 = \lnot \hat{x}_0[0] \land \bigwedge_{k=1}^{n} \hat{x}_0[k].$$

Clearly, this sub-formula can be constructed in $O(n)$ time.

We need a similar specification for $\phi$; however, we don’t know the exact length of $\phi$. We want $\hat{\phi}_0[k]$ to be true iff $1 \leq k \leq |\phi|$. Thus, we need $\hat{\phi}_0[0]$ to be false. Then as $k$ increases, $\hat{\phi}_0[k]$ may be true for a while, but as soon as we encounter a false $\hat{\phi}_0[k]$, these variables must be false for all greater values of $k$. We can enforce this constraint with the following sub-formula:

$$F_5 = \lnot \hat{\phi}_0[0] \land \bigwedge_{k=1}^{p(n)-1} \text{IF}(\lnot \hat{\phi}_0[k], \lnot \hat{\phi}_0[k+1]).$$
We can clearly construct this sub-formula in $O(p(n))$ time.

As the final initialization specification, we need to specify that all memory locations are initially 0. We therefore construct

$$F_6 = \bigwedge_{j=1}^{4p(n)p(n)} \bigwedge_{k=1}^{p(n)} \neg v_{0,j}[k].$$

Clearly, we can construct this sub-formula in $O(p^2(n))$ time.

We now need a constraint specifying that at some point, a \texttt{HALT(1)} instruction is executed. Let $A$ be the set of program locations at which a \texttt{HALT(1)} instruction appears. We then construct

$$F_7 = \bigvee_{i=0}^{p(n)-1} \bigvee_{j \in A} p_{ij}.$$ 

Because the size of $A$ depends only on the problem $X$, this sub-formula can be constructed in $O(p(n))$ time.

To complete the formula, we need constraints specifying the correct behavior of $M$. To this end, we will construct one sub-formula for each instruction in the program of $M$. These sub-formulas will depend on the particular instruction. Let $0 \leq q < P$, where $P$ is the number of instructions in the program. In what follows, we will describe how the sub-formula $F'_q$ is constructed depending on the instruction at program location $q$.

Regardless of the specific instruction, the sub-formula will have the same general form. In each case, $F'_q$ must specify that some particular behavior occurs whenever the program counter has a value of $q$. $F'_q$ will therefore have the following form:

$$F'_q = \bigwedge_{i=1}^{p(n)} \text{IF}(p_i - 1, q, \psi_q(i)),$$

where $\psi_q(i)$ is a predicate specifying the result of executing the $i$th instruction.

Each $\psi_q(i)$ will be a conjunction of predicates, each specifying some aspect of the result of executing the $i$th instruction. In particular, $\psi_q(i)$ will be the conjunction of the following predicates:

- $U_q(i)$, which specifies how the memory locations are updated;

- $E_q(i)$, which specifies what memory locations must be represented in $F$ in order for this instruction to be simulated (this specification is needed to prevent $U_q(i)$ from being vacuously satisfied);
• $I_q(i)$, which specifies which input bits remain unread; and
• $P_q(i)$, which specifies the new value of the program counter.

$\psi_q(i)$ is then defined as follows:

$$\psi_q(i) = U_q(i) \wedge E_q(i) \wedge I_q(i) \wedge P_q(i).$$

(16.2)

There are some instances of the above predicates that occur for more than one type of instruction.

• If the instruction at location $q$ is not an INPUT instruction, then

$$I_q(i) = \text{Eq}(\hat{x}_i, \hat{x}_{i-1}) \wedge \text{Eq}(\hat{\phi}_i, \hat{\phi}_{i-1}).$$

(16.3)

• If this instruction is neither a GOTO, an IFLEQ, nor a HALT, then

$$P_q(i) = p_{i,q+1}.$$  

(16.4)

• If this instruction is either a GOTO or a HALT, then

$$U_q(i) = \bigwedge_{j=1}^{4p(n)} \text{Eq}(v_{ij}, v_{i-1,j}),$$

and

$$E_q(i) = \text{true.}$$

(16.5)  

(16.6)

In what follows, we will define the remaining predicates for several of the possible instructions. We leave the remaining cases as exercises.

Let us first consider an instruction LOAD($n,l$). Because $l$ is the only memory location that is accessed, we can define

$$E_q(i) = \bigvee_{j=1}^{4p(n)} \text{Eq}(a_j, l).$$

Because its value changes to $n$, we can define

$$U_q(i) = \bigwedge_{j=1}^{4p(n)} \text{IFELSE}(\text{Eq}(a_j, l), \text{Eq}(v_{ij}, n), \text{Eq}(v_{ij}, v_{i-1,j})).$$

Note that the above expression specifies that every $v_{ij}$ such that $a_j = l$ has its value changed to $n$. 
Let us now compute the time needed to construct the resulting sub-formula $F'_q$. Because the arrays $a_j$ and $l$ each contain $p(n)$ elements, $E_q(i,j)$ can be constructed in $O(p^2(n))$ time. It is not hard to verify that $U_q(i)$ can be constructed in $O(p^2(n))$ time as well. Clearly, $P_q(i)$ as defined in (16.4) can be constructed in $O(1)$ time. Finally, $I_q(i)$ as defined in (16.3) can be constructed in $O(p(n))$ time. Thus, $\psi_q(i)$ can be constructed in $O(p^3(n))$ time.

We can handle an instruction LOAD($n, \ast l$) in a similar way, but using the IND abbreviation. Thus, we define

$$E_q(i) = \bigvee_{j=1}^{4p(n)} \text{IND}(i-1,j,l)$$

and

$$U_q(i) = \bigwedge_{j=1}^{4p(n)} \text{IF}(\text{IND}(i-1,j,l), E_q(v_{ij},n), E_q(v_{ij},v_{ij-1,j})).$$

In this case, $E_q(i)$ and $U_q(i)$ can be constructed in $O(p^3(n))$ time, so that $F'_q$ can be constructed in $O(p^4(n))$ time.

Let us now consider an instruction IFLEQ($l_1, l_2, q'$). Because the memory locations $l_1$ and $l_2$ are referenced, we define

$$E_q = \bigvee_{j=1}^{4p(n)} \text{EQ}(a_j,l_1) \land \bigvee_{j=1}^{4p(n)} \text{EQ}(a_j,l_2).$$

This statement will cause the program counter to be set to $q'$ if the value stored at $l_1$ is less than or equal to the value stored at $l_2$; otherwise, the program counter will be set to $q + 1$. We first observe that for natural numbers $v_1$ and $v_2$, $v_1 \leq v_2$ iff $v_2 - v_1 \geq 0$. We can therefore use the variable $d_i$ to record $|v_2 - v_1|$ using the SUM abbreviation as follows:

$$U_q(i) = \bigwedge_{j=1}^{4p(n)} \text{EQ}(v_{ij},v_{ij-1,j}) \land$$

$$\bigwedge_{j=1}^{4p(n)} \bigwedge_{j'=1}^{4p(n)} \text{IF}(\text{EQ}(a_j,l_1) \land \text{EQ}(a_{j'},l_2),$$

$$\text{SUM}(v_{ij},d_i,v_{ij'}) \lor \text{SUM}(v_{ij'},d_i,v_{ij})).$$
We can now define $P_q(i)$ as follows:

$$P_q(i) = \text{IFELSE} \left( \bigvee_{j=1}^{4p(n)} \bigvee_{j'=1}^{4p(n)} \text{EQ}(a_j, l_1) \land \text{EQ}(a_{j'}, l_2) \land \text{SUM}(v_{ij}, d_i, v_{ij'}), 
\begin{array}{c}
P_{i+1}^q, p_{i,q+1} \\
\end{array} \right).$$

$E_q(i)$ can be constructed in $O(p^2(n))$ time, and both $U_q(i)$ and $P_q(i)$ can be constructed in $O(p^3(n))$ time. Furthermore, $I_q(i)$ as given in (16.3) can be constructed in $O(p(n))$ time. The total time needed to construct $F_q'$ is therefore in $O(p^4(n))$.

Finally, let us consider a Halt instruction. For a Halt instruction, we have already defined $I_q(i)$ (16.3), $U_q(i)$ (16.5), and $E_q(i)$ (16.6). To define $P_q(i)$, we need to specify that for all $i' > i$, each $p_{i'j}$ is false:

$$P_q(i) = \bigwedge_{i' = i+1}^{p(n)} \bigwedge_{j=0}^{P-1} \neg p_{i'j}.$$ 

Because $P$ is a constant depending only on $X$, $P_q(i)$ can be constructed in $O(p(n))$ time. Furthermore, $I_q(i)$ can be constructed in $O(p(n))$ time, $U_q(i)$ can be constructed in $O(p^2(n))$ time, and $E_q(i)$ can be constructed in $O(1)$ time. The sub-formula $F_q'$ can therefore be constructed in $O(p^3(n))$ time.

We leave it as exercises to show that the sub-formula $F_q'$ can be constructed for each of the other cases in a time in $O(p^5(n))$. We now define the formula $F$ as the conjunction of all of the sub-formulas:

$$F = \bigwedge_{q=1}^{7} F_q \land \bigwedge_{q=0}^{P-1} F_q'.$$

Because $P$ is a constant depending only on $X$, $F$ can be constructed in $O(p^3(n))$ time.

We must now show that $F$ is satisfiable if and only if there is some $\phi \in \mathcal{B}$ such that $M$ executes a Halt(1) instruction on input $(x, \phi)$. Suppose $F$ is satisfiable. Let us fix some satisfying assignment to the variables of $F$. Because $F_5$ must be true by this assignment, there must be some $k$, $1 \leq k \leq p(n)$, such that $\bar{\phi}[k']$ is true for $1 \leq k' \leq k$ and $\bar{\phi}[k']$ is false for $k < k' \leq p(n)$. Let $\phi = \bar{\phi}[1..k]$. By the above construction, for $1 \leq i \leq p(n)$ and $0 \leq j < P$, $p_{ij}$
is true iff the $i$th instruction executed by $M$ on input $(x, \phi)$ is the instruction at program location $j$. Finally, because $F_7$ must be satisfied, one of these instructions must be a $\text{HALT}(1)$ instruction.

Now suppose that for some $\phi \in B$, $M$ executes a $\text{HALT}(1)$ instruction on input $(x, \phi)$. By our choice of the polynomial $p(n)$, we can assume that $|\phi| \leq p(|x|)$. Let us now set $\overline{x} = x$ and $\overline{\phi}[1..|\phi|] = \phi$. We will also set $\hat{\phi}[k] = \text{true}$ for $1 \leq k \leq |\phi|$ and $\hat{\phi}[k] = \text{false}$ for $|\phi| < k \leq p(n)$. We can clearly assign truth values to the variables in the sub-formulas $F_q$ for $2 \leq q \leq 6$ so that all of these sub-formulas are satisfied. By the above construction, we can then assign truth values to the variables in each of the sub-formulas $F'_q$ for $1 \leq q \leq p(n)$ so that these formulas, along with $F_1$, are satisfied. Such an assignment will yield $p_{ij} = \text{true}$ iff $p_j$ is the $i$th instruction executed by $M$ on input $(x, \phi)$. Because $M$ executes a $\text{HALT}(1)$ instruction on this input, $F_7$ must also be satisfied. Therefore, $F$ is satisfied by this assignment.

We have therefore shown that $X \leq^p \text{SAT}$. Because $X$ can be any problem in $\mathcal{NP}$, it follows that $\text{SAT}$ is $\mathcal{NP}$-hard. Because $\text{SAT} \in \mathcal{NP}$, it follows that $\text{SAT}$ is $\mathcal{NP}$-complete.

16.9 Summary

The $\mathcal{NP}$-complete problems comprise a large class of decision problems for which no polynomial-time algorithms are known. Furthermore, if a polynomial time algorithm were found for any one of these problems, we would be able to construct polynomial-time algorithms for all of them. For this reason, along with many others that are beyond the scope of this book, we tend to believe that none of these problems can be solved in polynomial time. Note, however, that this conjecture has not been proven. Indeed, this question — whether $\mathcal{P} = \mathcal{NP}$ — is the most famous open question in theoretical computer science.

Proofs of $\mathcal{NP}$-completeness consist of two parts: membership in $\mathcal{NP}$ and $\mathcal{NP}$-hardness. Without knowledge of any $\mathcal{NP}$-complete problems, it is quite tedious to prove a problem to be $\mathcal{NP}$-hard. However, given one or more $\mathcal{NP}$-complete problems, the task of proving additional problems to be $\mathcal{NP}$-hard is greatly eased using polynomial-time many-one reductions.

Some general guidelines for finding a reduction from a known $\mathcal{NP}$-complete problem to a problem known to be in $\mathcal{NP}$ are as follows:

- Look for a known $\mathcal{NP}$-complete problem that has similarities with the problem in question.
If all else fails, try reducing from 3-Sat.

Look at the proofs of membership in \( \mathcal{NP} \) and try to transform proofs \( \phi \in B \) for the known \( \mathcal{NP} \)-complete problem to proofs \( \phi' \in B \) for the problem in question.

Large numbers play an interesting role in the theory of \( \mathcal{NP} \)-completeness. In particular, some problems become \( \mathcal{NP} \)-hard simply because very large numbers can be given as input using comparatively few bits. The definitions of strong \( \mathcal{NP} \)-completeness and strong \( \mathcal{NP} \)-hardness exclude such problems. A refinement of polynomial many-one reducibility, namely, pseudopolynomial reducibility, is used to prove strong \( \mathcal{NP} \)-hardness.

16.10 Exercises

Exercise 16.1 Prove that if \( X \), \( Y \), and \( Z \) are decision problems such that \( X \leq^p_m Y \) and \( Y \leq^p_m Z \), then \( X \leq^p_m Z \).

Exercise 16.2 Adapt BOOLEval (Figure 16.2) to evaluate a CNF expression \( F \) in \( O(|F|) \) time.

Exercise 16.3 Implement the reduction from CSat to 3-Sat, as outlined in Section 16.4, to run in \( O(n) \) time, where \( n \) is the size of the given CNF formula.

Exercise 16.4 Give an algorithm that takes as input an undirected graph \( G \), a natural number \( k \), and an array \( A[0..m-1] \) of booleans, and determines whether \( A \) denotes a vertex cover of \( G \) with size \( k \). Your algorithm must run in \( O(n+a) \) time, where \( n \) and \( a \) are the number of vertices and edges, respectively, of \( G \). For the purpose of analyzing the running time, you may assume that \( G \) is implemented as a ListGraph.

Exercise 16.5 Implement the reduction from 3-Sat to VC, as outlined in Section 16.5, to run in \( O(m+n) \) time, where \( m \) is the number of variables and \( n \) is the number of clauses in the given 3-CNF formula.

* Exercise 16.6 Let NotAllEqual-3-Sat be the problem of deciding, for a given 3-CNF formula \( f \), whether there is an assignment of boolean variables such that each clause in \( f \) contains at least one true literal and at least one false literal. Prove that this problem is \( \mathcal{NP} \)-complete.
Exercise 16.7 A clique is a complete undirected graph — i.e., a graph such that for every pair of distinct vertices u and v, \{u, v\} is an edge. The clique problem (CLIQUE) is the problem of deciding, for a given undirected graph G and natural number k, if G has a subgraph that is a clique with k vertices. Show that CLIQUE is \(\mathbf{NP}\)-complete.

Exercise 16.8 Two graphs G and G' are said to be isomorphic if the vertices of G can be renamed so that the resulting graph is G'. Given two graphs G and G' and a natural number k, we wish to decide whether G and G' contain isomorphic subgraphs with k vertices. Show that this problem is \(\mathbf{NP}\)-complete. You may use the result of Exercise 16.7.

** Exercise 16.9 A Hamiltonian cycle in a graph G is a cycle that contains each vertex in G exactly once. Prove that the problem of deciding whether a given undirected graph contains a Hamiltonian cycle is \(\mathbf{NP}\)-complete. [Hint: Reduce VC to this problem.]**

Exercise 16.10 Repeat Exercise 16.9 for directed graphs. You may use the result of Exercise 16.9.

Exercise 16.11 As was defined in Exercise 10.34, a Hamiltonian path in a graph G is a simple path that contains each vertex in G exactly once. Prove that problem of deciding whether a given undirected graph has a Hamiltonian path is \(\mathbf{NP}\)-complete. You may use the results of Exercises 16.9 and 16.10.

Exercise 16.12 Repeat Exercise 16.11 for directed graphs. You may use the results of Exercises 16.9, 16.10, and 16.11.

Exercise 16.13 Given a directed graph G = (V, E) and a positive integer k, we wish to determine whether there is a subset V' \(\subseteq\) V of size k such that every cycle in G contains at least one vertex in V'. Show that this problem is \(\mathbf{NP}\)-complete.

* Exercise 16.14 Given an undirected graph G = (V, E) and a positive integer k, we wish to decide whether V can be partitioned into two disjoint sets, V_1 and V_2, such that V_1 contains exactly k vertices and for every vertex \(u \in V_2\), there is a vertex \(v \in V_1\) such that \{u, v\} \(\in E\). Show that this problem is \(\mathbf{NP}\)-complete.
**Exercise 16.15** Give an algorithm that takes an instance \((X, Y, Z, W)\) of 3DM and a bit string \(\phi\) and determines whether \(\phi\) is a proof that \((X, Y, Z, W)\) has a matching, as defined in Section 16.6. You may assume that \((X, Y, Z, W)\) is represented by a natural number \(m\) and an array \(W[1..n]\) of triples of the form \((i, j, k)\) such that \(m \leq n\) and each \(i, j,\) and \(k\) is a positive integer no greater than \(m\). Your algorithm should run in \(O(m^2 \lg n)\) time.

**Exercise 16.16** Given a finite sequence of finite sets and a natural number \(k\), we wish to decide whether the sequence contains at least \(k\) mutually disjoint sets. Show that this problem is \(\text{NP}\)-complete.

**Exercise 16.17** Prove that PART, as defined in Section 16.7, is in \(\text{NP}\).

**Exercise 16.18** Suppose we modify the 0-1 knapsack problem (see Section 12.4) by including a target value \(V\) as an additional input. The problem then is to decide whether there is a subset of the items whose total weight does not exceed the weight bound \(W\) and whose total value is at least \(V\). Prove that this problem is \(\text{NP}\)-complete.

**Exercise 16.19** Suppose we are given a set of items \(a_1, \ldots, a_n\), each having a positive integer weight \(w_i\), and positive integers \(k\) and \(W\). We wish to decide whether the items can be partitioned into \(k\) mutually disjoint subsets \(A_1, \ldots, A_k\), such that
\[
\sum_{j=1}^{k} \left( \sum_{a_i \in A_j} w_i \right)^2 \leq W.
\]
Show that this problem is \(\text{NP}\)-complete.

**Exercise 16.20** Suppose we are given a sequence \(S_1, \ldots, S_n\) of finite sets. We wish to partition
\[
\bigcup_{i=1}^{n} S_i
\]
into two disjoint sets \(S\) and \(S'\) such that for \(1 \leq i \leq n\), \(S_i \not\subseteq S\) and \(S_i \not\subseteq S'\). Show that this problem is \(\text{NP}\)-complete. [**Hint:** Reduce 3-SAT to this problem.]

**Exercise 16.21** Suppose we are given an undirected graph \(G = (V, E)\) with exactly \(3k\) vertices. We wish to partition \(V\) into \(k\) disjoint subsets such that each subset forms a path of length 2 in \(G\). Show that this problem is \(\text{NP}\)-complete. [**Hint:** Reduce 3DM to this problem.]
**Exercise 16.22** Given a directed graph $G = (V, E)$, we wish to decide whether each vertex $v_i \in V$ can be assigned a label $L_i \in \mathbb{N}$ such that $L_i$ is the least natural number that is not in the set

$$\{L_j \mid (v_i, v_j) \in E\}.$$ 

Show that this problem is $\mathcal{NP}$-complete. [Hint: Reduce 3-SAT to this problem.]

**Exercise 16.23** Show that the problem of deciding whether a given undirected graph has a 3-coloring is $\mathcal{NP}$-complete. (See Exercise 13.12 for the definition of a 3-coloring.) [Hint: Reduce 3-SAT to this problem.]

**Exercise 16.24** Show that the problem of deciding whether a given undirected graph has a $k$-coloring is $\mathcal{NP}$-complete for each fixed $k \geq 4$. You may use the result of Exercise 16.23.

**Exercise 16.25** Certain aspects of the board game Axis and Allies$^\text{TM}$ can be modeled as follows. The game is played on an undirected graph. The playing pieces include fighters and aircraft carriers, each of which has a natural number range. These pieces are each assigned to a vertex of the graph. Each vertex may be assigned any number of pieces. A combat scenario is valid if it is possible to move each piece to a new vertex (possibly the same one) so that

- for each move, the distance (i.e., number of edges) from the starting vertex to the ending vertex is no more than the range of piece moved; and
- after the pieces are moved, each vertex has no more than twice as many fighters as aircraft carriers.

Prove that the problem of determining whether a combat scenario is valid is $\mathcal{NP}$-complete.

**Exercise 16.26** Let $k$-PART be as defined in Section 16.7.

a. Prove that $k$-PART $\in \mathcal{NP}$ for all $k \geq 1$.

**b. Prove that 3-PART is $\mathcal{NP}$-complete in the strong sense. [Hint: Show that 4-PART $\leq_{\text{pp}}$ 3-PART.]**

c. Prove that 2-PART $\in \mathcal{P}$.
Exercise 16.27 The bin packing problem (BP) is to decide whether a given set of items, each having a weight \( w_i \), can be partitioned into \( k \) disjoint sets each having a total weight of at most \( W \), where \( k \) and \( W \) are given positive integers. Show that BP is \( \mathcal{NP} \)-complete in the strong sense.

Exercise 16.28 Suppose we are given a complete undirected graph \( G \) with positive integer edge weights and a positive integer \( k \). The traveling salesperson problem (TSP) is to determine whether there is a Hamiltonian cycle in \( G \) with total weight no more than \( k \). Show that this problem is \( \mathcal{NP} \)-complete in the strong sense. You may use the result of Exercise 16.9.

* Exercise 16.29 We are given a set of \( n \) tasks, each having an execution time \( e_i \in \mathbb{N} \), a ready time \( r_i \in \mathbb{N} \), and a deadline \( d_i \in \mathbb{N} \). We wish to decide whether there is a nonpreemptive schedule that meets the constraints of all of the tasks. In other words, we wish to know if there is a function \( f : [1..n] \to \mathbb{N} \) such that for \( 1 \leq i \leq n \),
  
  - \( r_i \leq f(i) \);
  - \( f(i) + e_i \leq d_i \); and
  - for \( 1 \leq j \leq i \) and \( j \neq i \), either \( f(j) + e_j \leq f(i) \) or \( f(j) \geq f(i) + e_i \).

Show that this problem is \( \mathcal{NP} \)-complete in the strong sense.

* Exercise 16.30 We are given an undirected graph \( G = (V,E) \), a sequence \( \langle w_1, \ldots, w_{|E|} \rangle \) of natural numbers, and a positive integer \( k \). We wish to decide whether there is a 1-1 function \( f : E \to \{1, \ldots, |E|\} \) such that if each edge \( e \in E \) is assigned a length of \( w_{f(e)} \), then for every pair of vertices \( u \) and \( v \), there is a path from \( u \) to \( v \) with length at most \( k \). Prove that this problem is \( \mathcal{NP} \)-complete in the strong sense.

Exercise 16.31 Define the predicate \( \mathcal{P}_q(i) \) for the case in which the instruction at location \( q \) is \( \text{Goto}(q') \). Show that the resulting sub-formula \( \mathcal{F}'_q \) can be constructed in \( O(p(n)) \) time.

Exercise 16.32 Define the predicates \( \mathcal{E}_q(i) \) and \( \mathcal{U}_q(i) \) for the case in which the instruction at location \( q \) is \( \text{Copy}(l_1,l_2) \). Show that the resulting sub-formula \( \mathcal{F}'_q \) can be constructed in \( O(p^4(n)) \) time.

Exercise 16.33 Define the predicates \( \mathcal{E}_q(i) \) and \( \mathcal{U}_q(i) \) for the case in which the instruction at location \( q \) is \( \text{Copy}(*l_1,*l_2) \). Show that the resulting sub-formula \( \mathcal{F}'_q \) can be constructed in \( O(p^5(n)) \) time.
Exercise 16.34 Define the predicates $E_q(i)$, $U_q(i)$, and $P_q(i)$ for the case in which the instruction at location $q$ is $\text{IfLeq}(\ast l_1, \ast l_2, q')$. Show that the resulting sub-formula $F'_q$ can be constructed in $O(p^5(n))$ time.

Exercise 16.35 Define the predicates $E_q(i)$ and $U_q(i)$ for the case in which the instruction at location $q$ is $\text{Add}(\ast l_1, \ast l_2)$. Show that the resulting sub-formula $F'_q$ can be constructed in $O(p^5(n))$ time.

Exercise 16.36 Define the predicates $E_q(i)$ and $U_q(i)$ for the case in which the instruction at location $q$ is $\text{Subtract}(\ast l_1, \ast l_2)$. Show that the resulting sub-formula $F'_q$ can be constructed in $O(p^5(n))$ time.

Exercise 16.37 Define the predicates $E_q(i)$ and $U_q(i)$ for the case in which the instruction at location $q$ is $\text{Shift}(\ast l)$. Show that the resulting sub-formula $F'_q$ can be constructed in $O(p^4(n))$ time.

* Exercise 16.38 Define the predicates $I_q(i)$, $E_q(i)$, and $U_q(i)$ for the case in which the instruction at location $q$ is $\text{Input}(1, \ast l)$. Show that the resulting sub-formula $F'_q$ can be constructed in $O(p^5(n))$ time.

16.11 Chapter Notes

$NP$-completeness was introduced by Cook [23], who proved that SAT and CSAT are $NP$-complete. Karp [74] then demonstrated the importance of this topic by proving $NP$-completeness of 21 problems, including VC, 3DM, PART, and the problems described in Exercises 16.7, 16.9, 16.10, 16.13, 16.16, 16.18, and 16.23. The original definition of $NP$ was somewhat different from the one given here — it was based on nondeterministic Turing machines, rather than on algorithms or RAMs. The definition given in Section 16.1 is based on a definition given by Brassard and Bratley [18]. All of these definitions are equivalent.

The notion of strong $NP$-completeness was introduced by Garey and Johnson [51]. They provided the definitions of strong $NP$-completeness, pseudopolynomial algorithms, and pseudopolynomial reductions. They had earlier given $NP$-completeness proofs for $k$-PART for $k \geq 3$ [49] and for the problem described in Exercise 16.29 [50]. As it turned out, their reductions were pseudopolynomial. Their book on $NP$-completeness [52] is an excellent resource.

Exercise 16.20 is solved by Lovasz [86]. Exercise 16.21 is solved by Kirkpatrick and Hell [75]. Exercise 16.22 is solved by van Leeuwen [107]. The
solution to Exercise 16.30 is attributed to Perl and Zaks by Garey and Johnson [52].

Axis and Allies™ (mentioned in Exercise 16.25) is a registered trademark of Hasbro, Inc.