Απο: M.R. Garey and D.S. Johnson,
"Computers and Intractability: A guide to
the Theory of NP-completeness"
W.H. Freeman and Co. 1979.

2

The Theory of NP-Completeness

In this chapter we present the formal details of the theory of NP-completeness. So that the theory can be defined in a mathematically rigorous way, it will be necessary to introduce formal counterparts for many of our informal notions, such as "problems" and "algorithms." Indeed, one of the main goals of this chapter is to make explicit the connection between the formal terminology and the more intuitive, informal shorthand that is commonly used in its place. Once we have this connection well in hand, it will be possible for us to pursue our discussions primarily at the informal level in later chapters, reverting to the formal level only when necessary for clarity and rigor.

The chapter begins by discussing decision problems and their representation as "languages," equating "solving" a decision problem with "recognizing" the corresponding language. The one-tape Turing machine is introduced as our basic model for computation and is used to define the class P of all languages recognizable deterministically in polynomial time. This model is then augmented with a hypothetical "guessing" ability, and the augmented model is used to define the class NP of all languages recognizable "nondeterministically" in polynomial time. After discussing the relationship between P and NP, we define the notion of a polynomial transformation from one language to another and use it to define what will be our most important class, the class of NP-complete problems. The chapter concludes with the statement and proof of Cook's fundamental theorem, which provides us with our first bona fide NP-complete problem.

2.1 Decision Problems, Languages, and Encoding Schemes

As a matter of convenience, the theory of NP-completeness is designed to be applied only to *decision problems.* Such problems, as mentioned in Chapter 1, have only two possible solutions, either the answer "yes" or the answer "no." Abstractly, a decision problem Π consists simply of a set D_Π of *instances* and a subset $Y_\Pi \subseteq D_\Pi$ of *yes-instances.* However, most decision problems of interest possess a considerable amount of additional structure, and we will describe them in a way that emphasizes this structure. The standard format we will use for specifying problems consists of two parts, the first part specifying a *generic instance* of the problem in terms of various components, which are sets, graphs, functions, numbers, etc., and the second part stating a yes-no *question* asked in terms of the generic instance. The way in which this specifies D_Π and Y_Π should be apparent. An instance belongs to D_Π if and only if it can be obtained from the generic instance by substituting particular objects of the specified types for all the generic components, and the instance belongs to Y_Π if and only if the answer for the stated question, when particularized to that instance, is "yes."

For example, the following describes a well-known decision problem from graph theory:

SUBGRAPH ISOMORPHISM
INSTANCE: Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
QUESTION: Does G_1 contain a subgraph isomorphic to G_2, that is, a subset $V' \subseteq V_1$ and a subset $E' \subseteq E_1$ such that $|V'| = |V_2|$, $|E'| = |E_2|$, and there exists a one-to-one function $f: V_2 \rightarrow V'$ satisfying $\{u,v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E'$?

A decision problem related to the traveling salesman problem can be described as follows:

TRAVELING SALESMAN
INSTANCE: A finite set $C = \{c_1, c_2, \ldots, c_m\}$ of "cities," a "distance" $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, and a bound $B \in Z^+$ (where Z^+ denotes the positive integers).
QUESTION: Is there a "tour" of all the cities in C having total length no more than B, that is, an ordering $<c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}>$ of C such that

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(m)}, c_{\pi(1)}) \leqslant B \ ?$$

The reader will find many more examples of the use of this format throughout the book, but these two should suffice for now to convey the basic idea. The second example also serves to illustrate an important point about how a decision problem can be derived from an optimization problem. If the optimization problem asks for a structure of a certain type that has minimum "cost" among all such structures (for example, a tour that has minimum length among all tours), we can associate with that problem the decision problem that includes a numerical bound B as an additional parameter and that asks whether there exists a structure of the required type having cost *no more than* B (for example, a tour of length no more than B). Decision problems can be derived from maximization problems in an analogous way, simply by replacing "no more than" by "at least."

The key point to observe about this correspondence is that, so long as the cost function is relatively easy to evaluate, the decision problem can be no harder than the corresponding optimization problem. Clearly, if we could find a minimum length tour for the traveling salesman problem in polynomial time, then we could also solve the associated decision problem in polynomial time. All we need do is find the minimum length tour, compute its length, and compare that length to the given bound B. Thus, if we could demonstrate that TRAVELING SALESMAN is NP-complete (as indeed it is), we would know that the traveling salesman optimization problem is at least as hard. In this way, even though the theory of NP-completeness restricts attention to only decision problems, we can extend the implications of the theory to optimization problems as well. (We shall see in Chapter 5 that decision problems and optimization problems are often even more closely tied: Many decision problems, including TRAVELING SALESMAN, can also be shown to be "no easier" than their corresponding optimization problems.)

The reason for the restriction to decision problems is that they have a very natural, formal counterpart, which is a suitable object to study in a mathematically precise theory of computation. This counterpart is called a "language" and is defined in the following way.

For any finite set Σ of symbols, we denote by Σ^* the set of all finite strings of symbols from Σ. For example, if $\Sigma = \{0,1\}$, then Σ^* consists of the empty string "ϵ," the strings $0,1,00,01,10,11,000,001$, and all other finite strings of 0's and 1's. If L is a subset of Σ^*, we say that L is a *language* over the alphabet Σ. Thus $\{01,001,111,1101010\}$ is a language over $\{0,1\}$, as is the set of all binary representations of integers that are perfect squares, as is the set $\{0,1\}^*$ itself.

The correspondence between decision problems and languages is brought about by the encoding schemes we use for specifying problem instances whenever we intend to compute with them. Recall that an encoding scheme e for a problem Π provides a way of describing each instance of Π by an appropriate string of symbols over some fixed alphabet Σ. Thus the problem Π and the encoding scheme e for Π partition Σ^* into three classes of strings: those that are not encodings of instances of Π, those that encode instances of Π for which the answer is "no," and those that encode instances of Π for which the answer is "yes." This third class of strings is the language we associate with Π and e, setting

$$L[\Pi,e] = \left\{ x \in \Sigma^*: \begin{array}{l} \Sigma \textit{ is the alphabet used by } e, \textit{ and } x \textit{ is the} \\ \textit{encoding under } e \textit{ of an instance } I \in Y_\Pi \end{array} \right\}$$

Our formal theory is applied to decision problems by saying that, if a result holds for the language $L[\Pi,e]$, then it holds for the problem Π under the encoding scheme e.

In fact, we shall usually follow standard practice and be a bit more informal than this. Each time we introduce a new concept in terms of languages, we will observe that the property is essentially encoding independent, so long as we restrict ourselves to "reasonable" encoding schemes. That is, if e and e' are any two reasonable encoding schemes for Π, then the property holds either for both $L[\Pi,e]$ and $L[\Pi,e']$ or for neither. This will allow us to say, informally, that the property holds (or does not hold) for the problem Π, without actually specifying any encoding scheme. However, whenever we do so, the implicit assertion will be that we could, if requested, specify a particular reasonable encoding scheme e such that the property holds for $L[\Pi,e]$.

Notice that when we operate in this encoding-independent manner, we lose contact with any precise notion of "input length." Since we need some parameter in terms of which time complexity can be expressed, it is convenient to assume that every decision problem Π has an associated, encoding-independent function Length: $D_\Pi \to Z^+$, which is "polynomially related" to the input lengths we would obtain from a reasonable encoding scheme. By *polynomially related* we mean that, for any reasonable encoding scheme e for Π, there exist two polynomials p and p' such that if $I \in D_\Pi$ and x is a string encoding the instance I under e, then Length$[I] \leqslant p(|x|)$ and $|x| \leqslant p'(\text{Length}[I])$, where $|x|$ denotes the length of the string x. In the SUBGRAPH ISOMORPHISM problem, for example, we might take

$$\text{Length}[I] = |V_1| + |V_2|$$

where $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are the graphs making up an instance. In the TRAVELING SALESMAN decision problem we might take

$$\text{Length}[I] = m + \lceil \log_2 B \rceil + \max\{\lceil \log_2 d(c_i, c_j) \rceil : c_i, c_j \in C\}$$

Since any two reasonable encoding schemes for a problem Π will yield polynomially related input lengths, a wide variety of Length functions are possible for Π, and all our results will carry through for any such function that meets the above conditions.

The usefulness of this informal, encoding-independent approach depends, of course, on there being some agreement as to what constitutes a "reasonable" encoding scheme. The generally accepted meaning of "reasonable" includes both the notion of "conciseness," as captured by the two conditions mentioned in Chapter 1, and the notion of "decodability." The intent of "conciseness" is that instances of a problem should be described with the natural brevity we would use in actually specifying those instances for a computer, without any unnatural "padding" of the input. Such padding could be used, for example, to expand the input length so much that we artificially convert an exponential time algorithm into a polynomial time algorithm. The intent of "decodability" is that, given any particular component of a generic instance, one should be able to specify a polynomial time algorithm that is capable of extracting a description of that component from any given encoded instance.

Of course, these elaborations do not provide a formal definition of "reasonable encoding scheme," and we know of no satisfactory way of making such a definition. Even though most people would agree on whether or not a particular encoding scheme for a given problem is reasonable, the absence of a formal definition can be somewhat discomforting. One way of resolving this difficulty would be to require that generic problem instances always be formed from a fixed collection of basic types of set-theoretic objects. We will not impose such a constraint here, but, as an indication of our intent when we refer to "reasonable encoding schemes," we now give a brief description (which first time readers may wish to skip) of how such a standard encoding scheme could be defined.

Our standard encoding scheme will map instances into "structured strings" over the alphabet $\Psi = \{0, 1, -, [,], (,), ,\}$. We define *structured strings* recursively, as follows:

(1) The binary representation of an integer k as a string of **0**'s and **1**'s (preceded by a minus sign "−" if k is negative) is a structured string representing the integer k.

(2) If x is a structured string representing the integer k, then $[x]$ is a structured string that can be used as a "name" (for example, for a vertex in a graph, a set element, or a city in a traveling salesman instance).

(3) If $x_1, x_2, \ldots, x_m$ are structured strings representing the objects $X_1, X_2, \ldots, X_m$, then $(x_1, x_2, \ldots, x_m)$ is a structured string representing the sequence $<X_1, X_2, \ldots, X_m>$.

To derive an encoding scheme for a particular decision problem specified in our standard format, we first note that, once we have built up a representation for each object in an instance as a structured string, the representation of the entire instance is determined using rule (3) above. Thus we need only specify how the representation for each type of object is constructed. For this we shall restrict ourselves to integers, "unstructured elements" (vertices, elements, cities, etc.), sequences, sets, graphs, finite functions, and rational numbers.

Rules (1) and (3) already tell us how to represent integers and sequences. To represent each of the unstructured elements in an instance, we merely assign it a distinct "name," as constructed by rule (2), in such a way that if the total number of unstructured elements in an instance is N, then no name with magnitude exceeding N is used. The representations for the four other object types are as follows:

A *set* of objects is represented by ordering its elements as a sequence $<X_1, X_2, \ldots, X_m>$ and taking the structured string corresponding to that sequence.

A *graph* with vertex set V and edge set E is represented by a structured string (x,y), where x is a structured string representing the set V, and y is a structured string representing the set E (the elements of E being the two-element subsets of V that are edges).

A *finite function* $f: \{U_1, U_2, \ldots, U_m\} \rightarrow W$ is represented by a structured string $((x_1,y_1),(x_2,y_2), \ldots, (x_m,y_m))$ where x_i is a structured string representing the object U_i and y_i is a structured string representing the object $f(U_i) \in W, 1 \leqslant i \leqslant m$.

A *rational number* q is represented by a structured string (x,y) where x is a structured string representing an integer a, y is a structured string representing an integer b, $a/b = q$, and the greatest common divisor of a and b is 1.

Although it might be convenient to have a wider collection of object types at our disposal, the ones above will suffice for most purposes and are enough to illustrate our notion of a reasonable encoding scheme. Furthermore, there would be no loss of generality in restricting ourselves to just these types for specifying generic instances, since other types of objects can always be expressed in terms of the ones above.

Note that our prescriptions are not sufficient to generate a *unique* string for encoding each instance but merely for ensuring that each string that does encode an instance obeys certain structural restrictions. A different choice of names for the basic elements or a different choice of order for the description of a set could lead to different strings that encode the same instance. In fact, it makes no difference how many strings encode an instance so long as we can decode each to obtain the essential components of the instance. Moreover, our definitions take this into account; for example, in $L[\Pi,e]$, the set of all strings that encode yes-instances of Π under e, each instance may be represented many times.

Before going on, we remind the reader that our standard encoding scheme is intended solely to illustrate how one might define such a standard scheme, although it also provides a reference point for what we mean by a "reasonable" encoding scheme. There is no reason why some other general scheme could not be used, or why we could not merely devise an individual encoding scheme for each problem of interest. If the chosen scheme is "equivalent" to ours, in the sense that there exist polynomial time algorithms for converting an encoding of an instance under either scheme to an encoding of that instance under the other scheme, then it, too, will be called "reasonable." If the chosen scheme is *not* equivalent to ours in this sense, then one can still prove results with respect to that scheme, but the encoding-independent terminology should not be used for describing them. Throughout this book we will restrict our attention to reasonable encoding schemes for problems.

2.2 Deterministic Turing Machines and the Class P

In order to formalize the notion of an algorithm, we will need to fix a particular model for computation. The model we choose is the *deterministic one-tape Turing machine* (abbreviated DTM), which is pictured schematically in Figure 2.1. It consists of a *finite state control*, a *read-write head*, and a *tape* made up of a two-way infinite sequence of *tape squares*, labeled $\ldots,-2,-1,0,1,2,3,\ldots$

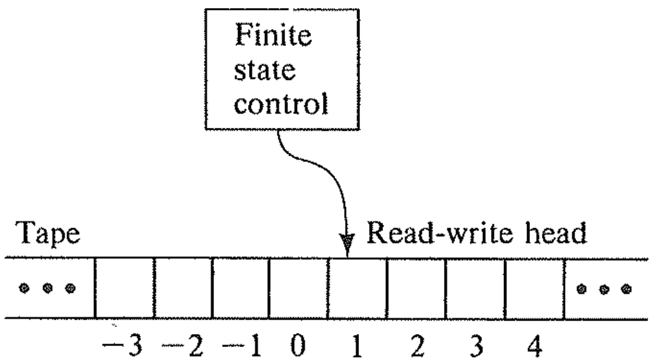

Figure 2.1 Schematic representation of a deterministic one-tape Turing machine (DTM).

A *program* for a DTM specifies the following information:

(1) A finite set Γ of tape *symbols*, including a subset $\Sigma \subset \Gamma$ of *input symbols* and a distinguished *blank symbol* $b \in \Gamma - \Sigma$;

(2) a finite set Q of *states*, including a distinguished *start-state* q_0 and two distinguished *halt-states* q_Y and q_N;

(3) a *transition function* $\delta\colon (Q - \{q_Y, q_N\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1,+1\}$.

The operation of such a program is straightforward. The *input* to the DTM is a string $x \in \Sigma^*$. The string x is placed in tape squares 1 through $|x|$, one symbol per square. All other squares initially contain the blank symbol. The program starts its operation in state q_0, with the read-write head scanning tape square 1. The computation then proceeds in a step-by-step manner. If the current state q is either q_Y or q_N, then the computation has ended, with the answer being "yes" if $q = q_Y$ and "no" if $q = q_N$. Otherwise the current state q belongs to $Q - \{q_Y, q_N\}$, some symbol $s \in \Gamma$ is in the tape square being scanned, and the value of $\delta(q,s)$ is defined. Suppose $\delta(q,s) = (q', s', \Delta)$. The read-write head then erases s, writes s' in its place, and moves one square to the left if $\Delta = -1$, or one square to the right if $\Delta = +1$. At the same time, the finite state control changes its state from q to q'. This completes one "step" of the computation, and we are ready to proceed to the next step, if there is one.

$$\Gamma = \{0,1,b\},\ \Sigma = \{0,1\}$$

$$Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$$

q	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

$$\delta(q,s)$$

Figure 2.2 An example of a DTM program $M = (\Gamma, Q, \delta)$.

An example of a simple DTM program M is shown in Figure 2.2. The transition function δ for M is described in a tabular format, where the entry in row q and column s is the value of $\delta(q,s)$. Figure 2.3 illustrates the computation of M on the input $x = 10100$, giving the state, head position, and contents of the non-blank portion of the tape before and after each step.

Note that this computation halts after eight steps, in state q_Y, so the answer for 10100 is "yes." In general, we say that a DTM program M with input alphabet Σ *accepts* $x \in \Sigma^*$ if and only if M halts in state q_Y when applied to input x. The language L_M *recognized* by the program M is given by

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$$

It is not hard to see that the DTM program of Figure 2.2 recognizes the language

$$\{x \in \{0,1\}^* : \textit{the rightmost two symbols of } x \textit{ are both } 0\}$$

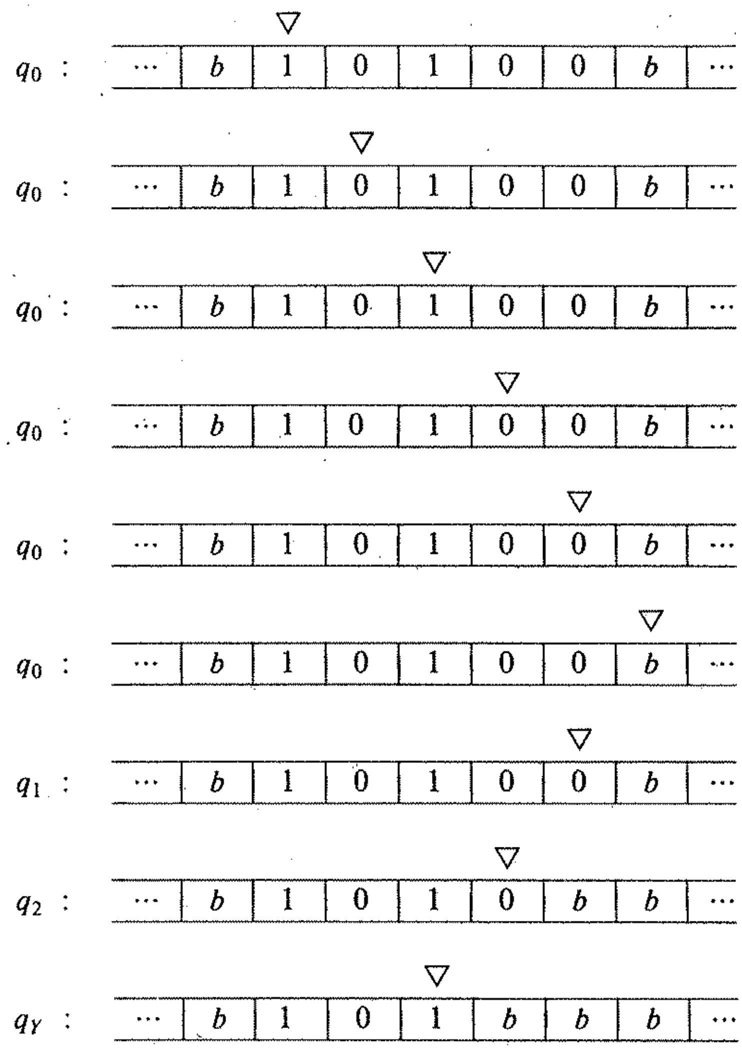

Figure 2.3 The computation of the program M from Figure 2.2 on input 10100.

Observe that this definition of language recognition does not require that M halt for *all* input strings in Σ^*, only for those in L_M. If x belongs to $\Sigma^* - L_M$, then the computation of M on x might halt in state q_N, or it might continue forever without halting. However, for a DTM program to correspond to our notion of an *algorithm*, it must halt on all possible strings over its input alphabet. In this sense, the DTM program of Figure 2.2 is algorithmic, since it will halt for any input string from $\{0,1\}^*$

The correspondence between "recognizing" languages and "solving" decision problems is straightforward. We say that a DTM program M *solves* the decision problem Π under encoding scheme e if M halts for all input strings over its input alphabet and $L_M = L[\Pi,e]$. The DTM program of Figure 2.2 once more provides an illustration. Consider the following number-theoretic decision problem:

INTEGER DIVISIBILITY BY FOUR
INSTANCE: A positive integer N.
QUESTION: Is there a positive integer m such that $N=4m$?

Under our standard encoding scheme, the integer N is represented by the string of 0's and 1's that is its binary representation. Since a positive integer is divisible by four if and only if the last two digits of its binary representation are 0, this DTM program "solves" the INTEGER DIVISIBILITY BY FOUR problem under our standard encoding scheme.

For future reference, we also point out that a DTM program can be used to compute functions. Suppose M is a DTM program with input alphabet Σ and tape alphabet Γ that halts for all input strings from Σ^*. Then M computes the *function* $f_M: \Sigma^* \rightarrow \Gamma^*$ where, for each $x \in \Sigma^*$, $f_M(x)$ is defined to be the string obtained by running M on input x until it halts and then forming a string from the symbols in tape squares 1, 2, 3, etc., in sequence, up to and including the rightmost non-blank tape square. The program M of Figure 2.2 computes the function $f_M: \{0,1\}^* \rightarrow \{0,1,b\}^*$ that maps each string $x \in \{0,1\}^*$ to the string $f_M(x)$ obtained by deleting the last two symbols of x (with $f_M(x)$ equal to the empty string if $|x|<2$).

It is well known that DTM programs are capable of performing much more complicated tasks than those illustrated by our simple example. Even though a DTM has only a single sequential tape and can perform only a very limited amount of work in a single step, a DTM program can be designed to perform any computation that can be performed on an ordinary computer, albeit more slowly. For the reader interested in how this is done, there are a number of excellent references, for example [Minsky, 1967] or [Hopcroft and Ullman, 1969]. For the reader who is *not* interested in how this is done, there is the welcome assurance that no expertise at programming DTMs will be required in this book. The reason for our introduction of the DTM model is to provide us with a formal counterpart of an algorithm upon which to base our definitions.

A formal definition of "time complexity" is now possible. The *time* used in the computation of a DTM program M on an input x is the number of steps occurring in that computation up until a halt state is entered. For a DTM program M that halts for all inputs $x \in \Sigma^*$, its *time complexity function* $T_M: Z^+ \rightarrow Z^+$ is given by

$$T_M(n) = \max \left\{ m : \begin{array}{l} \textit{there is an } x \in \Sigma^*, \textit{ with } |x|=n, \textit{ such that the} \\ \textit{computation of } M \textit{ on input } x \textit{ takes time } m \end{array} \right\}$$

Such a program M is called a *polynomial time DTM program* if there exists a polynomial p such that, for all $n \in Z^+$, $T_M(n) \leqslant p(n)$.

We are now ready to give the formal definition of the first important class of languages that we will be considering, the class P. It is defined as follows:

$$\text{P} = \{L: \textit{ there is a polynomial time DTM program } M \textit{ for which } L = L_M\}$$

We will say that a decision problem Π belongs to P under the encoding scheme e if $L[\Pi,e] \in$ P, that is, if there is a polynomial time DTM program that "solves" Π under encoding scheme e. In light of the previously mentioned equivalence between reasonable encoding schemes, we will usually omit the specification of a particular reasonable encoding scheme, simply saying that the decision problem Π belongs to P.

We also will be informal in our use of the term "polynomial time algorithm." Our formal counterpart for a polynomial time algorithm is the polynomial time DTM program. However, because of the equivalence between "realistic" computer models with respect to polynomial time pointed out in Chapter 1, the formal definition of P could have been rephrased in terms of programs for any such model and the same class of languages would have resulted. Thus we need not tie ourselves to the details of the DTM model when informally demonstrating that certain tasks can be performed by polynomial time algorithms. In fact, we will follow standard practice and discuss algorithms in an almost model-independent manner, speaking of them as operating directly on the components of an instance (the sets, graphs, numbers, etc.) rather than on their encoded descriptions. Here our implicit assertion is that one could, if one desired and had the patience, design a polynomial time DTM program corresponding to each polynomial time algorithm we discuss. Our informal demonstrations should be taken as indicating how this would be done and should be convincing to any reader familiar with the kinds of basic tasks that can be performed in polynomial time on an ordinary computer.

2.3 Nondeterministic Computation and the Class NP

In this section we introduce our second important class of languages/decision problems, the class NP. Before we proceed to the formal definitions in terms of languages and Turing machines, however, it will be useful to provide an intuitive idea of the informal notion this class is intended to capture.

Consider the TRAVELING SALESMAN problem described at the beginning of this chapter: Given a set of cities, the distances between them, and a bound B, does there exist a tour of all the cities having total length B or less? There is no known polynomial time algorithm for solving this problem. However, suppose someone claimed, for a particular instance of this problem, that the answer for that instance is "yes." If we were skeptical, we could demand that they "prove" their claim by providing us with a tour having the required properties. It would then be a simple matter for us to verify the truth or falsity of their claim merely by checking that what they provided us with is actually a tour and, if so, computing its length and comparing that quantity to the given bound B. Furthermore, we could specify our "verification procedure" as a general algorithm that has time complexity polynomial in Length$[I]$.

Another example of a problem with this property is the SUBGRAPH ISOMORPHISM problem of Section 2.1. Given an arbitrary instance I of this problem, consisting of two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, if the answer for I is "yes," then this fact can be "proved" by giving the required subsets $V' \subseteq V_1$ and $E' \subseteq E_1$ and the required one-to-one function $f: V_2 \rightarrow V'$. Again the validity of the claim can be verified easily in time polynomial in Length$[I]$, merely by checking that V', E', and f satisfy all the stated requirements.

It is this notion of polynomial time "verifiability" that the class NP is intended to isolate. Notice that polynomial time verifiability does not imply polynomial time solvability. In saying that one can verify a "yes" answer for a TRAVELING SALESMAN instance in polynomial time, we are not counting the time one might have to spend in searching among the exponentially many possible tours for one of the desired form. We merely assert that, given any tour for an instance I, we can verify in polynomial time whether or not that tour "proves" that the answer for I is "yes."

Informally we can define NP in terms of what we shall call a *nondeterministic algorithm*. We view such an algorithm as being composed of two separate stages, the first being a *guessing stage* and the second a *checking stage*. Given a problem instance I, the first stage merely "guesses" some structure S. We then provide both I and S as inputs to the checking stage, which proceeds to compute in a normal deterministic manner, either eventually halting with answer "yes," eventually halting with answer "no," or computing forever without halting (as we shall see, the latter two cases need not be distinguished). A nondeterministic algorithm "solves" a decision problem Π if the following two properties hold for all instances $I \in D_\Pi$:

1. If $I \in Y_\Pi$, then there exists some structure S that, when guessed for input I, will lead the checking stage to respond "yes" for I and S.
2. If $I \notin Y_\Pi$, then there exists *no* structure S that, when guessed for input I, will lead the checking stage to respond "yes" for I and S.

For example, a nondeterministic algorithm for TRAVELING SALESMAN could be constructed using a guessing stage that simply guesses an arbitrary sequence of the given cities and a checking stage that is identical to the aforementioned polynomial time "proof verifier" for TRAVELING SALESMAN. Clearly, for any instance I, there will exist a guess S that leads the checking stage to respond "yes" for I and S if and only if there is a tour of the desired length for I.

A nondeterministic algorithm that solves a decision problem Π is said to operate in "polynomial time" if there exists a polynomial p such that, for every instance $I \in Y_\Pi$, there is some guess S that leads the deterministic checking stage to respond "yes" for I and S within time $p(\text{Length}[I])$. Notice that this has the effect of imposing a polynomial bound on the "size" of the guessed structure S, since only a polynomially bounded amount of time can be spent examining that guess.

The class NP is defined informally to be the class of all decision problems Π that, under reasonable encoding schemes, can be solved by polynomial time nondeterministic algorithms. Our example above indicates that TRAVELING SALESMAN is one member of NP. The reader should have no difficulty in providing a similar demonstration for SUBGRAPH ISOMORPHISM.

The use of the term "solve" in these informal definitions should, of course, be taken with a grain of salt. It should be evident that a "polynomial time nondeterministic algorithm" is basically a definitional device for capturing the notion of polynomial time verifiability, rather than a realistic method for solving decision problems. Instead of having just one possible computation on a given input, it has many different ones, one for each possible guess.

There is another important way in which the "solution" of decision problems by nondeterministic algorithms differs from that for deterministic algorithms: the lack of symmetry between "yes" and "no." If the problem "Given I, is X true for I?" can be solved by a polynomial time (deterministic) algorithm, then so can the complementary problem "Given I, is X false for I?" This is because a deterministic algorithm halts for all inputs, so all we need do is interchange the "yes" and "no" responses (interchange states q_Y and q_N in a DTM program). It is not at all obvious that the same holds true for all problems solvable by polynomial time nondeterministic algorithms. Consider, for example, the complement of the TRAVELING SALESMAN problem: Given a set of cities, the intercity distances, and a bound B, is it true that *no* tour of all the cities has length B or less? There is no known way to verify a "yes" answer to this problem short of examining all possible tours (or a large proportion of them). In other words, no polynomial time nondeterministic algorithm for this complementary problem is known. The same is true of many other problems in NP. Thus, although membership in P for a problem Π implies membership in P for its complement, the analogous implication is not known to hold for NP.

We conclude this section by formalizing our definition in terms of languages and Turing machines. The formal counterpart of a nondeterministic algorithm is a program for a *nondeterministic one-tape Turing machine* (NDTM). For simplicity, we will be using a slightly non-standard NDTM model. (More standard versions are described in [Hopcroft and Ullman, 1969] and [Aho, Hopcroft, and Ullman, 1974]. The reader may find it an interesting exercise to verify the equivalence of our model to these with respect to polynomial time.)

The NDTM model we will be using has exactly the same structure as a DTM, except that it is augmented with a *guessing module* having its own *write-only head*, as illustrated schematically in Figure 2.4. The guessing module provides the means for writing down the "guess" and will be used solely for this purpose.

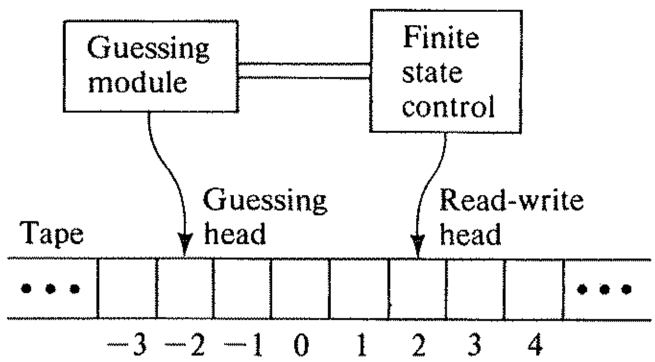

Figure 2.4 Schematic representation of a nondeterministic one-tape Turing machine (NDTM).

An *NDTM program* is specified in exactly the same way as a DTM program, including the tape alphabet Γ, input alphabet Σ, blank symbol b, state set Q, initial state q_0, halt states q_Y and q_N, and transition function $\delta\colon (Q-\{q_Y,q_N\})\times\Gamma \rightarrow Q\times\Gamma\times\{-1,+1\}$. The computation of an NDTM program on an input string $x\in\Sigma^*$ differs from that of a DTM in that it takes place in two distinct stages.

The first stage is the "guessing" stage. Initially, the input string x is written in tape squares 1 through $|x|$ (while all other squares are blank), the read-write head is scanning square 1, the write-only head is scanning square -1, and the finite state control is "inactive." The guessing module then directs the write-only head, one step at a time, either to write some symbol from Γ in the tape square being scanned and move one square to the left, or to stop, at which point the guessing module becomes inactive and the finite state control is activated in state q_0. The choice of whether to remain active, and, if so, which symbol from Γ to write, is made by the guessing module in a totally arbitrary manner. Thus the guessing module can write any string from Γ^* before it halts and, indeed, need never halt.

The "checking" stage begins when the finite state control is activated in state q_0. From this point on, the computation proceeds solely under the direction of the NDTM program according to exactly the same rules as for a DTM. The guessing module and its write-only head are no longer involved, having fulfilled their role by writing the guessed string on the tape. Of course, the guessed string can (and usually will) be examined during the checking stage. The computation ceases when and if the finite state control enters one of the two halt states (either q_Y or q_N) and is said to be an *accepting computation* if it halts in state q_Y. All other computations, halting or not, are classed together simply as *non-accepting computations.*

Notice that any NDTM program M will have an infinite number of possible computations for a given input string x, one for each possible guessed string from Γ^*. We say that the NDTM program M *accepts* x if at least one of these is an accepting computation. The language *recognized* by M is

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$$

The *time* required by an NDTM program M to accept the string $x \in L_M$ is defined to be the minimum, over all accepting computations of M for x, of the number of steps occurring in the guessing and checking stages up until the halt state q_Y is entered. The *time complexity function* $T_M: Z^+ \rightarrow Z^+$ for M is

$$T_M(n) = \max\left(\{1\} \cup \left\{m: \begin{array}{l}\text{there is an } x \in L_M \text{ with } |x| = n \text{ such} \\ \text{that the time to accept } x \text{ by } M \text{ is } m\end{array}\right\}\right)$$

Note that the time complexity function for M depends only on the number of steps occurring in *accepting* computations, and that, by convention, $T_M(n)$ is set equal to 1 whenever no inputs of length n are accepted by M.

The NDTM program M is a *polynomial time NDTM program* if there exists a polynomial p such that $T_M(n) \leqslant p(n)$ for all $n \geqslant 1$. Finally, the class NP is formally defined as follows:

$$\text{NP} = \{L : \text{there is a polynomial time NDTM program } M \text{ for which } L_M = L\}$$

It is not hard to see how these formal definitions correspond to the informal definitions that preceded them. The only point deserving special mention is that, whereas we usually envision a nondeterministic algorithm as guessing a structure S that in some way depends on the given instance I, the guessing module of an NDTM entirely disregards the given input. However, since *every* string from Γ^* is a possible guess, we can always design our NDTM program so that the checking stage begins by checking whether or not the guessed string corresponds (under the implicit interpretation our program places on strings) to an appropriate guess for the given input. If not, the program can immediately enter the halt state q_N.

A decision problem Π will be said to belong to NP under encoding scheme e if the language $L[\Pi,e] \in$ NP. As with P, we shall feel free to say that Π is in NP without giving a specific encoding scheme, so long as it is clear that *some* reasonable encoding scheme for Π will yield a language that is in NP.

Furthermore, since any realistic computer model can be augmented with an analogue of our "guessing module with write-only head," we could have rephrased our formal definitions in terms of any of the other standard models of computation. Since all these models are equivalent with respect to deterministic polynomial time, the resulting versions of NP would all be identical. Thus we will be on firm ground when, as already proposed, we identify our formally defined class NP with the class of all decision problems "solvable" by polynomial time nondeterministic algorithms.

In the next section we discuss the relationship between the two classes P and NP as a preliminary to introducing our third and, for this book, most important class, the class of NP-complete problems.

2.1 The Relationship Between P and NP

The relationship between the classes P and NP is fundamental for the theory of NP-completeness. Our first observation, which is implicit in our earlier discussions but which has not been stated explicitly until now, is that $P \subseteq NP$. Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm. To see this, one simply needs to observe that any deterministic algorithm can be used as the checking stage of a nondeterministic algorithm. If $\Pi \in$ P, and A is any polynomial time deterministic algorithm for Π, we can obtain a polynomial time nondeterministic algorithm for Π merely by using A as the checking stage and ignoring the guess. Thus $\Pi \in$ P implies $\Pi \in$ NP.

As we also hinted in our discussions, there are many reasons to believe that this inclusion is proper, that is, that P does not equal NP. Polynomial time nondeterministic algorithms certainly appear to be more powerful than polynomial time deterministic ones, and we know of no general methods for converting the former into the latter. In fact, the best general result we can state at present is given by the following:

Theorem 2.1 If $\Pi \in$ NP, then there exists a polynomial p such that Π can be solved by a deterministic algorithm having time complexity $O(2^{p(n)})$.

Proof: Suppose A is a polynomial time nondeterministic algorithm for solving Π, and let $q(n)$ be a polynomial bound on the time complexity of A. (Without loss of generality, we can assume that q can be evaluated in polynomial time, for example, by taking $q(n) = c_1 n^{c_2}$ for suitably large integer constants c_1 and c_2.) Then we know that, for every accepted input of length n, there must exist some guessed string (over the tape alphabet Γ) of length at most $q(n)$ that leads the checking stage of A to respond "yes" for that input in no more than $q(n)$ steps. Thus the number of possible guesses that need be considered is at most $k^{q(n)}$, where $k = |\Gamma|$, since guesses shorter than $q(n)$ can be regarded as guesses of length exactly $q(n)$ by filling them out with blanks. We can deterministically discover whether A has an accepting computation for a given input of length n by applying the deterministic checking stage of A, until it halts or makes $q(n)$ steps, on each of the $k^{q(n)}$ possible guesses. The simulation responds "yes" if it encounters a guessed string that leads to an accepting computation within the time bound; otherwise it responds "no." This clearly yields a deterministic algorithm for solving Π. Furthermore, its time complexity is essentially $q(n) \cdot k^{q(n)}$, which, although exponential, is $O(2^{p(n)})$ for an appropriately chosen polynomial p. ■

Of course the simulation in the proof of Theorem 2.1 could be speeded up somewhat by using branch-and-bound techniques or backtrack search and by carefully enumerating the guesses so that obviously irrelevant strings are avoided. Nevertheless, despite the considerable savings that might be achieved, there is no known way to perform this simulation in less than exponential time.

Thus the ability of a nondeterministic algorithm to check an exponential number of possibilities in polynomial time might lead one to suspect that polynomial time nondeterministic algorithms are strictly more powerful than polynomial time deterministic algorithms. Indeed, for many individual problems in NP, such as TRAVELING SALESMAN, SUBGRAPH ISOMORPHISM, and a wide variety of others, no polynomial time solution algorithms have been found despite the efforts of many knowledgeable and persistent researchers.

For these reasons, it is not surprising that there is a widespread belief that P≠NP, even though no proof of this conjecture appears on the horizon. Of course, a skeptic might say that our failure to find a proof that P≠NP is just as strong an argument in favor of P=NP as our failure to find polynomial time algorithms is an argument for the opposite view. Problems always appear to be intractable until we discover efficient algorithms for solving them. Even a skeptic would be likely to agree, however, that, given our current state of knowledge, it seems more reasonable to operate under the assumption that P≠NP than to devote one's efforts to proving the contrary. In any case, we shall adopt a tentative picture of the world of NP as shown in Figure 2.5, with the expectation (but not the certainty) that the shaded region denoting NP − P is not totally uninhabited.

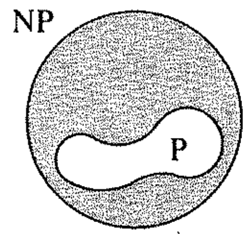

Figure 2.5 A tentative view of the world of NP.

2.5 Polynomial Transformations and NP-Completeness

If P differs from NP, then the distinction between P and NP−P is meaningful and important. All problems in P can be solved with polynomial time algorithms, whereas all problems in NP−P are intractable. Thus, given a decision problem $\Pi \in$ NP, if P$\neq$NP, we would like to know which of these two possibilities holds for Π.

Of course, until we can prove that P$\neq$NP, there is no hope of showing that any particular problem belongs to NP−P. For this reason, the theory of NP-completeness focuses on proving results of the weaker form "if P$\neq$NP, then $\Pi \in$ NP−P." We shall see that, although these conditional results might appear to be almost as difficult to prove as the corresponding unconditional results, there are techniques available that often enable us to prove them in a straightforward way. The extent to which such results should be regarded as evidence for intractability depends on how strongly one believes that P differs from NP.

The key idea used in this conditional approach is that of a polynomial transformation. A *polynomial transformation* from a language $L_1 \subseteq \Sigma_1^*$ to a language $L_2 \subseteq \Sigma_2^*$ is a function $f: \Sigma_1^* \rightarrow \Sigma_2^*$ that satisfies the following two conditions:

1. There is a polynomial time DTM program that computes f.
2. For all $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

If there is a polynomial transformation from L_1 to L_2, we write $L_1 \propto L_2$, read "L_1 transforms to L_2" (dropping the modifier "polynomial," which is to be understood).

The significance of polynomial transformations comes from the following lemma:

Lemma 2.1 If $L_1 \propto L_2$, then $L_2 \in$ P implies $L_1 \in$ P (and, equivalently, $L_1 \notin$ P implies $L_2 \notin$ P).

Proof: Let Σ_1 and Σ_2 be the alphabets of L_1 and L_2 respectively, let $f: \Sigma_1^* \to \Sigma_2^*$ be a polynomial transformation from L_1 to L_2, let M_f denote a polynomial time DTM program that computes f, and let M_2 be a polynomial time DTM program that recognizes L_2. A polynomial time DTM program for recognizing L_1 can be constructed by composing M_f with M_2. For an input $x \in \Sigma_1^*$, we first apply the portion corresponding to program M_f to construct $f(x) \in \Sigma_2^*$. We then apply the portion corresponding to program M_2 to determine if $f(x) \in L_2$. Since $x \in L_1$ if and only if $f(x) \in L_2$, this yields a DTM program that recognizes L_1. That this program operates in polynomial time follows immediately from the fact that M_f and M_2 are polynomial time algorithms. To be specific, if p_f and p_2 are polynomial functions bounding the running times of M_f and M_2, then $|f(x)| \leqslant p_f(|x|)$, and the running time of the constructed program is easily seen to be $O(p_f(|x|)+p_2(p_f(|x|)))$, which is bounded by a polynomial in $|x|$. ■

If Π_1 and Π_2 are decision problems, with associated encoding schemes e_1 and e_2, we shall write $\Pi_1 \propto \Pi_2$ (with respect to the given encoding schemes) whenever there exists a polynomial transformation from $L[\Pi_1,e_1]$ to $L[\Pi_2,e_2]$. As usual, we will omit the reference to specific encoding schemes when we are operating under our standard assumption that only reasonable encoding schemes are used. Thus, at the problem level, we can regard a polynomial transformation from the decision problem Π_1 to the decision problem Π_2 as a function $f: D_{\Pi_1} \to D_{\Pi_2}$ that satisfies the two conditions:

1. f is computable by a polynomial time algorithm; and
2. for all $I \in D_{\Pi_1}$, $I \in Y_{\Pi_1}$ if and only if $f(I) \in Y_{\Pi_2}$.

Let us obtain a more concrete idea of what this definition means by considering an example. For a graph $G=(V,E)$ with vertex set V and edge set E, a *simple circuit* in G is a sequence $< v_1, v_2, \ldots, v_k >$ of distinct vertices from V such that $\{v_i, v_{i+1}\} \in E$ for $1 \leqslant i < k$ and such that $\{v_k, v_1\} \in E$. A *Hamiltonian circuit* in G is a simple circuit that includes all the vertices of G. The HAMILTONIAN CIRCUIT problem is defined as follows:

HAMILTONIAN CIRCUIT
INSTANCE: A graph $G=(V,E)$.
QUESTION: Does G contain a Hamiltonian circuit?

The reader will no doubt recognize a certain similarity between this problem and the TRAVELING SALESMAN decision problem. We shall show that HAMILTONIAN CIRCUIT (HC) transforms to TRAVELING SALESMAN (TS). This requires that we specify a function f that maps each instance of HC to a corresponding instance of TS and that we prove that this function satisfies the two properties required of a polynomial transformation.

The function f is defined quite simply. Suppose $G=(V,E)$, with $|V|=m$, is a given instance of HC. The corresponding instance of TS has a set C of cities that is identical to V. For any two cities $v_i, v_j \in C$, the intercity distance $d(v_i,v_j)$ is defined to be 1 if $\{v_i,v_j\} \in E$ and 2 otherwise. The bound B on the desired tour length is set equal to m.

It is easy to see (informally) that this transformation f can be computed by a polynomial time algorithm. For each of the $m(m-1)/2$ distances $d(v_i,v_j)$ that must be specified, it is necessary only to examine G to see whether or not $\{v_i,v_j\}$ is an edge in E. Thus the first required property is satisfied. To verify that the second requirement is met, we must show that G contains a Hamiltonian circuit if and only if there is a tour of all the cities in $f(G)$ that has total length no more than B. First, suppose that $<v_1,v_2,\ldots,v_m>$ is a Hamiltonian circuit for G. Then $<v_1,v_2,\ldots,v_m>$ is also a tour in $f(G)$, and this tour has total length $m=B$ because each intercity distance traveled in the tour corresponds to an edge of G and hence has length 1. Conversely, suppose that $<v_1,v_2,\ldots,v_m>$ is a tour in $f(G)$ with total length no more than B. Since any two cities are either distance 1 or distance 2 apart, and since exactly m such distances are summed in computing the tour length, the fact that $B=m$ implies that each pair of successively visited cities must be exactly distance 1 apart. By the definition of $f(G)$, it follows that $\{v_i,v_{i+1}\}$, $1 \leqslant i < m$, and $\{v_m,v_1\}$ are all edges of G, and hence $<v_1,v_2,\ldots,v_m>$ is a Hamiltonian circuit for G.

Thus we have shown that $HC \propto TS$. Although this proof is much simpler than many we will be describing, it contains all the essential elements of a proof of polynomial transformability and can serve as a model for how such proofs are constructed at the informal level.

The significance of Lemma 2.1 for decision problems now can be illustrated in terms of what it says about HC and TS. In essence, we conclude that if TRAVELING SALESMAN can be solved by a polynomial time algorithm, then so can HAMILTONIAN CIRCUIT, and if HC is intractable, then so is TS. Thus Lemma 2.1 allows us to interpret $\Pi_1 \propto \Pi_2$ as meaning that Π_2 is "at least as hard" as Π_1.

The "polynomial transformability" relation is especially useful because it is transitive, a fact captured by our next lemma.

Lemma 2.2 If $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.

Proof: Let Σ_1, Σ_2, and Σ_3 be the alphabets of languages L_1, L_2, and L_3, respectively, let $f_1: \Sigma_1^* \to \Sigma_2^*$ be a polynomial transformation from L_1 to L_2, and let $f_2: \Sigma_2^* \to \Sigma_3^*$ be a polynomial transformation from L_2 to L_3. Then the function $f: \Sigma_1^* \to \Sigma_3^*$ defined by $f(x)=f_2(f_1(x))$ for all $x \in \Sigma_1^*$ is the desired transformation from L_1 to L_3. Clearly, $f(x) \in L_3$ if and only if

$x \in L_1$, and the fact that f can be computed by a polynomial time DTM program follows from an argument analogous to that used in the proof of Lemma 2.1. ■

We can define two languages L_1 and L_2 (two decision problems Π_1 and Π_2) to be *polynomially equivalent* whenever both $L_1 \propto L_2$ and $L_2 \propto L_1$ (both $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$). Lemma 2.2 tells us that this is a legitimate equivalence relation and, furthermore, that the relation "$\propto$" imposes a partial order on the resulting equivalence classes of languages (decision problems). In fact, the class P forms the "least" equivalence class under this partial order and hence can be viewed as consisting of the computationally "easiest" languages (decision problems). The class of NP-complete languages (problems) will form another such equivalence class, distinguished by the property that it contains the "hardest" languages (decision problems) in NP.

Formally, a language L is defined to be *NP-complete* if $L \in$ NP and, for all other languages $L' \in$ NP, $L' \propto L$. Informally, a decision problem Π is NP-complete if $\Pi \in$ NP and, for all other decision problems $\Pi' \in$ NP, $\Pi' \propto \Pi$. Lemma 2.1 then leads us to our identification of the NP-complete problems as "the hardest problems in NP." If any single NP-complete problem can be solved in polynomial time, then *all* problems in NP can be so solved. If any problem in NP is intractable, then so are all NP-complete problems. An NP-complete problem Π, therefore, has the property mentioned at the beginning of this section: If P$\neq$NP, then $\Pi \in$ NP$-$P. More precisely, $\Pi \in$ P if and only if P$=$NP.

Assuming that P$\neq$NP, we now can give a more detailed picture of "the world of NP," as shown in Figure 2.6. Notice that NP is not simply partitioned into "the land of P" and "the land of NP-complete." As we shall see in Chapter 7, if P differs from NP, then there must exist problems in NP that are neither solvable in polynomial time nor NP-complete.

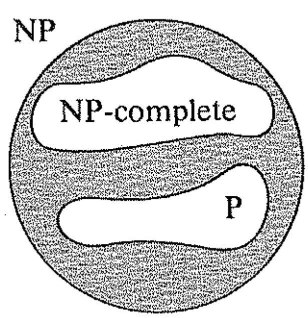

Figure 2.6 The world of NP, revisited.

Our main interest, however, is in the NP-complete problems themselves. Although we suggested at the outset of this section that there are straightforward techniques for proving that a problem is NP-complete, the requirements we have just described would appear to be rather demanding. One must show that *every* problem in NP transforms to our prospective NP-complete problem Π. It is not at all obvious how one might go about doing this. *A priori*, it is not even apparent that any NP-complete problems need exist.

The following lemma, which is an immediate consequence of our definitions and the transitivity of $\propto$, shows that matters would be simplified considerably if we possessed just one problem that we knew to be NP-complete.

Lemma 2.3 If L_1 and L_2 belong to NP, L_1 is NP-complete, and $L_1 \propto L_2$, then L_2 is NP-complete.

Proof: Since $L_2 \in$ NP, all we need to do is show that, for every $L' \in$ NP, $L' \propto L_2$. Consider any $L' \in$ NP. Since L_1 is NP-complete, it must be the case that $L' \propto L_1$. The transitivity of $\propto$ and the fact that $L_1 \propto L_2$ then imply that $L' \propto L_2$. ∎

Translated to the decision problem level, this lemma gives us a straightforward approach for proving new problems NP-complete, once we have at least one known NP-complete problem available. To prove that Π is NP-complete, we merely show that

1. Π ∈ NP, and
2. some known NP-complete problem Π′ transforms to Π.

Before we can use this approach, however, we still need some first NP-complete problem. Such a problem is provided by Cook's fundamental theorem, which we state and prove in the next section.

2.6 Cook's Theorem

The honor of being the "first" NP-complete problem goes to a decision problem from Boolean logic, which is usually referred to as the SATISFIABILITY problem (SAT, for short). The terms we shall use in describing it are defined as follows:

Let $U = \{u_1, u_2, \ldots, u_m\}$ be a set of Boolean *variables*. A *truth assignment* for U is a function $t: U \rightarrow \{T, F\}$. If $t(u) = T$ we say that u is "true" under t; if $t(u) = F$ we say that u is "false." If u is a variable in U, then u and $\bar{u}$ are *literals* over U. The literal u is true under t if and only if the variable u is true under t; the literal $\bar{u}$ is true if and only if the variable u is false.

A *clause* over U is a set of literals over U, such as $\{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. The clause above will be satisfied by t *unless* $t(u_1) = F$, $t(u_3) = T$, and $t(u_8)=F$. A collection C of clauses over U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C. Such a truth assignment is called a *satisfying truth assignment* for C. The SATISFIABILITY problem is specified as follows:

SATISFIABILITY
INSTANCE: A set U of variables and a collection C of clauses over U.
QUESTION: Is there a satisfying truth assignment for C?

For example, $U=\{u_1,u_2\}$ and $C=\{\{u_1,\bar{u}_2\}, \{\bar{u}_1,u_2\}\}$ provide an instance of SAT for which the answer is "yes." A satisfying truth assignment is given by $t(u_1)=t(u_2)=T$. On the other hand, replacing C by $C'=\{\{u_1,u_2\}, \{u_1,\bar{u}_2\}, \{\bar{u}_1\}\}$ yields an instance for which the answer is "no"; C' is not satisfiable.

The seminal theorem of Cook [1971] can now be stated:

Theorem 2.1 (Cook's Theorem) SATISFIABILITY is NP-complete.

Proof: SAT is easily seen to be in NP. A nondeterministic algorithm for it need only guess a truth assignment for the given variables and check to see whether that assignment satisfies all the clauses in the given collection C. This is easy to do in (nondeterministic) polynomial time. Thus the first of the two requirements for NP-completeness is met.

For the second requirement, let us revert to the language level, where SAT is represented by a language $L_{SAT} = L[SAT,e]$ for some reasonable encoding scheme e. We must show that, for all languages $L \in$ NP, $L \propto L_{SAT}$. The languages in NP are a rather diverse lot, and there are infinitely many of them, so we cannot hope to present a separate transformation for each one of them. However, each of the languages in NP can be described in a standard way, simply by giving a polynomial time NDTM program that recognizes it. This allows us to work with a generic polynomial time NDTM program and to derive a generic transformation from the language it recognizes to L_{SAT}. This generic transformation, when specialized to a particular NDTM program M recognizing the language L_M, will give the desired polynomial transformation from L_M to L_{SAT}. Thus, in essence, we will present a simultaneous proof for all $L \in$ NP that $L \propto L_{SAT}$.

To begin, let M denote an arbitrary polynomial time NDTM program, specified by Γ, Σ, b, Q, q_0, q_Y, q_N, and δ, which recognizes the language $L=L_M$. In addition, let $p(n)$ be a polynomial over the integers that bounds the time complexity function $T_M(n)$. (Without loss of generality, we can assume that $p(n) \geqslant n$ for all $n \in Z^+$.) The generic transformation f_L will be derived in terms of M, Γ, Σ, b, Q, q_0, q_Y, q_N, δ, and p.

It will be convenient to describe f_L as if it were a mapping from strings over Σ to instances of SAT, rather than to strings over the alphabet of our encoding scheme for SAT, since the details of the encoding scheme could be filled in easily. Thus f_L will have the property that for all $x \in \Sigma^*$, $x \in L$ if and only if $f_L(x)$ has a satisfying truth assignment. The key to the construction of f_L is to show how a set of clauses can be used to check whether an input x is accepted by the NDTM program M, that is, whether $x \in L$.

If the input $x \in \Sigma^*$ is accepted by M, then we know that there is an accepting computation for M on x such that both the number of steps in the checking stage and the number of symbols in the guessed string are bounded by $p(n)$, where $n = |x|$. Such a computation cannot involve any tape squares except for those numbered $-p(n)$ through $p(n)+1$, since the read-write head begins at square 1 and moves at most one square in any single step. The status of the checking computation at any one time can be specified completely by giving the contents of these squares, the current state, and the position of the read-write head. Furthermore, since there are no more than $p(n)$ steps in the checking computation, there are at most $p(n)+1$ distinct times that must be considered. This will enable us to describe such a computation completely using only a limited number of Boolean variables and a truth assignment to them.

The variable set U that f_L constructs is intended for just this purpose. Label the elements of Q as $q_0, q_1{=}q_Y, q_2{=}q_N, q_3, \ldots, q_r$, where $r = |Q|-1$, and label the elements of Γ as $s_0{=}b, s_1, s_2, \ldots, s_v$, where $v = |\Gamma|-1$. There will be three types of variables, each of which has an intended meaning as specified in Figure 2.7. By the phrase "at time i" we mean "upon completion of the i^{th} step of the checking computation."

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leqslant i \leqslant p(n)$ $0 \leqslant k \leqslant r$	At time i, M is in state q_k.
$H[i,j]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$	At time i, the read-write head is scanning tape square j.
$S[i,j,k]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$ $0 \leqslant k \leqslant v$	At time i, the contents of tape square j is symbol s_k.

Figure 2.7 Variables in $f_L(x)$ and their intended meanings.

A computation of M induces a truth assignment on these variables in the obvious way, under the convention that, if the program halts before time $p(n)$, the configuration remains static at all later times, maintaining the same halt-state, head position, and tape contents. The tape contents at time 0 consists of the input x, written in squares 1 through n, and the guess w, written in squares -1 through $-|w|$, with all other squares blank.

On the other hand, an arbitrary truth assignment for these variables need not correspond at all to a computation, much less to an accepting computation. According to an arbitrary truth assignment, a given tape square might contain many symbols at one time, the machine might be simultaneously in several different states, and the read-write head could be in any subset of the positions $-p(n)$ through $p(n)+1$. The transformation f_L works by constructing a collection of clauses involving these variables such that a truth assignment is a *satisfying* truth assignment if and only if it is the truth assignment induced by an accepting computation for x whose checking stage takes $p(n)$ or fewer steps and whose guessed string has length at most $p(n)$. We thus will have

$x \in L$ $\Leftrightarrow$ there is an accepting computation of M on x

$\Leftrightarrow$ there is an accepting computation of M on x with $p(n)$ or fewer steps in its checking stage and with a guessed string w of length exactly $p(n)$

$\Leftrightarrow$ there is a satisfying truth assignment for the collection of clauses in $f_L(x)$.

This will mean that f_L satisfies one of the two conditions required of a polynomial transformation. The other condition, that f_L can be computed in polynomial time, will be verified easily once we have completed our description of f_L.

The clauses in $f_L(x)$ can be divided into six groups, each imposing a separate type of restriction on any satisfying truth assignment as given in Figure 2.8.

It is straightforward to observe that if all six clause groups perform their intended missions, then a satisfying truth assignment will have to correspond to the desired accepting computation for x. Thus all we need to show is how clause groups performing these missions can be constructed.

Group G_1 consists of the following clauses:

$$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\},\ 0 \leq i \leq p(n)$$

$$\{\overline{Q[i,j]}, \overline{Q[i,j']}\},\ 0 \leq i \leq p(n),\ 0 \leq j < j' \leq r$$

The first $p(n)+1$ of these clauses can be simultaneously satisfied if and only if, for each time i, M is in *at least* one state. The remaining $(p(n)+1)(r+1)(r/2)$ clauses can be simultaneously satisfied if and only if at no time i is M in *more than one* state. Thus G_1 performs its mission.

Groups G_2 and G_3 are constructed similarly, and groups G_4 and G_5 are both quite simple, each consisting only of one-literal clauses. Figure 2.9 gives a complete specification of the first five groups. Note that the number

	Clause group	Restriction imposed
"Exclusion"	G_1	At each time i, M is in exactly one state.
	G_2	At each time i, the read-write head is scanning exactly one tape square.
	G_3	At each time i, each tape square contains exactly one symbol from Γ.
"Boundary conditions"	G_4	At time 0, the computation is in the initial configuration of its checking stage for input x.
	G_5	By time $p(n)$, M has entered state q_Y and hence has accepted x.
T.M. Function	G_6	For each time i, $0 \leqslant i < p(n)$, the configuration of M at time $i+1$ follows by a single application of the transition function δ from the configuration at time i.

Figure 2.8 Clause groups in $f_L(x)$ and the restrictions they impose on satisfying truth assignments.

of clauses in these groups, and the maximum number of literals occurring in each clause, are both bounded by a polynomial function of n (since r and v are *constants* determined by M and hence by L).

The final clause group G_6, which ensures that each successive configuration in the computation follows from the previous one by a single step of program M, is a bit more complicated. It consists of two subgroups of clauses.

The first subgroup guarantees that if the read-write head is *not* scanning tape square j at time i, then the symbol in square j does not change between times i and $i+1$. The clauses in this subgroup are as follows:

$S[i,j,l] \wedge \overline{H}[i,j] \Rightarrow S[i+1,j,l] \equiv$

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\},\ 0 \leqslant i < p(n), -p(n) \leqslant j \leqslant p(n)+1, 0 \leqslant l \leqslant v$$

For any time i, tape square j, and symbol s_l, if the read-write head is not scanning square j at time i, and square j contains s_l at time i but not at time $i+1$, then the above clause based on i, j, and l will fail to be satisfied (otherwise it *will* be satisfied). Thus the $2(p(n)+1)^2(v+1)$ clauses in this subgroup perform their mission.

Clause group	Clauses in group
G_1	$\{Q[i,0],Q[i,1],\ldots,Q[i,r]\},\ 0\leqslant i\leqslant p(n)$ $\{\overline{Q[i,j]},\overline{Q[i,j']}\},\ 0\leqslant i\leqslant p(n),0\leqslant j<j'\leqslant r$
G_2	$\{H[i,-p(n)],H[i,-p(n)+1],\ldots,H[i,p(n)+1]\},\ 0\leqslant i\leqslant p(n)$ $\{\overline{H[i,j]},\overline{H[i,j']}\},\ 0\leqslant i\leqslant p(n),-p(n)\leqslant j<j'\leqslant p(n)+1$
G_3	$\{S[i,j,0],S[i,j,1],\ldots,S[i,j,v]\},\ 0\leqslant i\leqslant p(n),-p(n)\leqslant j\leqslant p(n)+1$ $\{\overline{S[i,j,k]},\overline{S[i,j,k']}\},0\leqslant i\leqslant p(n),-p(n)\leqslant j\leqslant p(n)+1,0\leqslant k<k'\leqslant v$
G_4	$\{Q[0,0]\},\{H[0,1]\},\{S[0,0,0]\},$ $\{S[0,1,k_1]\},\{S[0,2,k_2]\},\cdots,\{S[0,n,k_n]\},$ $\{S[0,n+1,0]\},\{S[0,n+2,0]\},\ldots,\{S[0,p(n)+1,0]\},$ where $x=s_{k_1}s_{k_2}\cdots s_{k_n}$
G_5	$\{Q[p(n),1]\}$

Figure 2.9 The first five clause groups in $f_L(x)$.

The remaining subgroup of G_6 guarantees that the *changes* from one configuration to the next are in accord with the transition function δ for M. For each quadruple (i,j,k,l), $0\leqslant i<p(n)$, $-p(n)\leqslant j\leqslant p(n)+1$, $0\leqslant k\leqslant r$, and $0\leqslant l\leqslant v$, this subgroup contains the following three clauses:

$$\{\overline{H[i,j]},\ \overline{Q[i,k]},\ \overline{S[i,j,l]},\ H[i+1,j+\Delta]\}$$
$$\{\overline{H[i,j]},\ \overline{Q[i,k]},\ \overline{S[i,j,l]},\ Q[i+1,k']\}$$
$$\{\overline{H[i,j]},\ \overline{Q[i,k]},\ \overline{S[i,j,l]},\ S[i+1,j,l']\}$$

where if $q_k\in Q-\{q_Y,q_N\}$, then the values of Δ, k', and l' are such that $\delta(q_k,s_l)=(q_{k'},s_{l'},\Delta)$, and if $q_k\in\{q_Y,q_N\}$, then $\Delta=0$, $k'=k$, and $l'=l$.

Although it may require a few minutes of thought, it is not difficult to see that these $6(p(n))\,(p(n)+1)\,(r+1)\,(v+1)$ clauses impose the desired restriction on satisfying truth assignments.

Thus we have shown how to construct clause groups G_1 through G_6 performing the previously stated missions. If $x\in L$, then there is an accepting computation of M on x of length $p(n)$ or less, and this computation, given the interpretation of the variables, imposes a truth assignment that satisfies all the clauses in $C=G_1\cup G_2\cup G_3\cup G_4\cup G_5\cup G_6$.

Conversely, the construction of C is such that any satisfying truth assignment for C must correspond to an accepting computation of M on x. It follows that $f_L(x)$ has a satisfying truth assignment if and only if $x \in L$.

All that remains to be shown is that, for any fixed language L, $f_L(x)$ can be constructed from x in time bounded by a polynomial function of $n = |x|$. Given L, we choose a particular NDTM M that recognizes L in time bounded by a polynomial p (we need not find this NDTM itself in polynomial time, since we are only proving that the desired transformation f_L *exists*). Once we have a specific NDTM M and a specific polynomial p, the construction of the set U of variables and collection C of clauses amounts to little more than filling in the blanks in a standard (though complicated) formula. The polynomial boundedness of this computation will follow immediately once we show that Length$[f_L(x)]$ is bounded above by a polynomial function of n, where Length$[I]$ reflects the length of a string encoding the instance I under a reasonable encoding scheme, as discussed in Section 2.1. Such a "reasonable" Length function for SAT is given, for example, by $|U| \cdot |C|$. No clause can contain more than $2 \cdot |U|$ literals (that's all the literals there are), and the number of symbols required to describe an individual literal need only add an additional $\log |U|$ factor, which can be ignored when all that is at issue is polynomial boundedness. Since r and v are fixed in advance and can contribute only constant factors to $|U|$ and $|C|$, we have $|U| = O(p(n)^2)$ and $|C| = O(p(n)^2)$. Hence Length$[f_L(x)] = |U| \cdot |C| = O(p(n)^4)$, and is bounded by a polynomial function of n as desired.

Thus the transformation f_L can be computed by a polynomial time algorithm (although the particular polynomial bound it obeys will depend on L and on our choices for M and p), and we conclude that, for every $L \in$ NP, f_L is a polynomial transformation from L to SAT (technically, of course, from L to L_{SAT}). It follows, as claimed, that SAT is NP-complete. ■