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 nc
MAN could be constru
bitrary sequence of the
the aforementioned p
SALESMAN. Clearly,
leads the checking stag
a tour of the desired le

A nondeterministi
to operate in "polynor
for every instance $I \in$
checking stage to resp
Notice that this has
"size" of the guesse
amount of time can be

The class NP is d
lems Π that, under rea
mial time nondetermi
TRAVELING SALESI
no difficulty in provi
MORPHISM.

The use of the te
course, be taken with
mial time nondetermi
capturing the notion c
method for solving de
computation on a give
sible guess.

There is another
problems by nondeter
algorithms: the lack of
"Given I, is X true
ministic) algorithm, th
X false for I?" This
puts, so all we need d
change states q_Y and
the same holds true fo
ministic algorithms. C
ELING SALESMAN
and a bound B, is it t
There is no known w
examining all possibl
words, no polynomial

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 complemen-

tary 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.

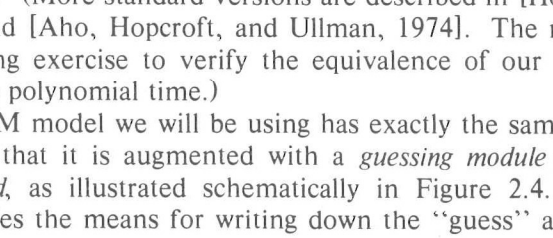

Figure 2.4 Schematic representation of a nondeterministic one-tape Turing machine (NDTM).

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 δ: $(Q-\{q_Y,q_N\})\times\Gamma \rightarrow Q\times\Gamma\times\{-1,+1\}$. 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 $L[\Pi,e] \in \text{NP}$. 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.

2.4 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 $\text{P} \subseteq \text{NP}$. 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 $\Pi \in \text{P}$, 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 $\Pi \in \text{P}$ implies $\Pi \in \text{NP}$.

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 $\Pi \in \text{NP}$, 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 solv-

ing 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 $P \neq NP$, 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 $P \neq NP$ 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 $P \neq NP$ 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 $\Pi \in \text{NP}$, if $\text{P} \neq \text{NP}$, we would like to know which of these two possibilities holds for Π.

Of course, until we can prove that $\text{P} \neq \text{NP}$, 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 $\text{P} \neq \text{NP}$, then $\Pi \in \text{NP}-\text{P}$." 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 $f: \Sigma_1^* \to \Sigma_2^*$ 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 $L_2 \in \text{P}$ implies $L_1 \in \text{P}$ (and, equivalently, $L_1 \notin \text{P}$ implies $L_2 \notin \text{P}$).

Proof: Let Σ_1 and Σ_2 be the alphabets of L_1 and L_2 respectively, let $f: \Sigma_1^* \rightarrow \Sigma_2^*$ 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 $f: D_{\Pi_1} \rightarrow D_{\Pi_2}$ 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 $f_1: \Sigma_1^* \rightarrow \Sigma_2^*$ be a polynomial transformation from L_1 to L_2, and let $f_2: \Sigma_2^* \rightarrow \Sigma_3^*$ be a polynomial transformation from L_2 to L_3. Then the function $f: \Sigma_1^* \rightarrow \Sigma_3^*$ 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.

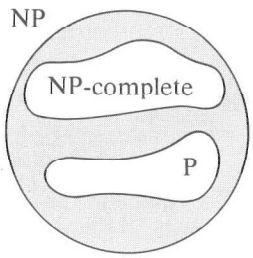

Figure 2.6 The world of NP, revisited.

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 $L_2 \in \text{NP}$, all we need to do is show that, for every $L' \in \text{NP}$, $L' \propto L_2$. Consider any $L' \in \text{NP}$. 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. $\Pi \in \text{NP}$, 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* $t(u_1) = F$, $t(u_3) = T$,