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

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

The reader will fi
throughout the book,
basic idea. The second
about how a decision
lem. If the optimizatio
has minimum "cost"
has minimum length a
the decision problem
parameter and that asks
having cost *no more th*
B). Decision problem
analogous way, simply

The key point to
the cost function is rel
no harder than the c
could find a minimum
lynomial time, then w
polynomial time. All
its length, and compa
could demonstrate th
indeed it is), we would
lem is at least as ha
completeness restricts
the implications of the
see in Chapter 5 that d
even more closely tie
SALESMAN, can also
optimization problems

The reason for th
very natural, formal
mathematically precise
"language" and is defi

For any finite set
strings of symbols fro
the empty string "ϵ,"
finite strings of 0's a
language over the alp
over $\{0,1\}$, as is the se
fect squares, as is the

The corresponde
brought about by the
stances whenever we
scheme e for a proble
by an appropriate stri
problem Π and the en

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

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

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 $\text{Length}[I] \leqslant p(|x|)$ 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 $\langle X_1, X_2, \ldots, X_m \rangle$.

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
rithms for converting a
encoding of that instar
called "reasonable." If
sense, then one can sti
encoding-independent t
Throughout this book
schemes for problems.

2.2 Deterministic Tur

In order to formal
particular model for co
one-tape Turing machine
in Figure 2.1. It consis
made up of a two-
$\ldots,-2,-1,0,1,2,3,\ldots$

Tape
• • •

Figure 2.1 Schematic re
(DTM).

A *program* for a D

(1) A finite set
symbols and

(2) a finite set
two distingu

(3) a *transition f*

The operation of
DTM is a string $x \in$
$|x|$, one symbol per

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 $\ldots,-2,-1,0,1,2,3,\ldots$

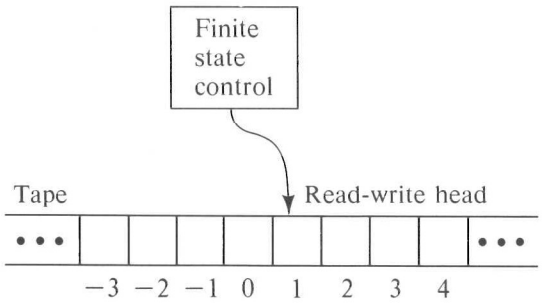

Figure 2.1 Schematic representation of a deterministic one-tape Turing machine (DTM).

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* $\delta: (Q-\{q_Y,q_N\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1,+1\}$.

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

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

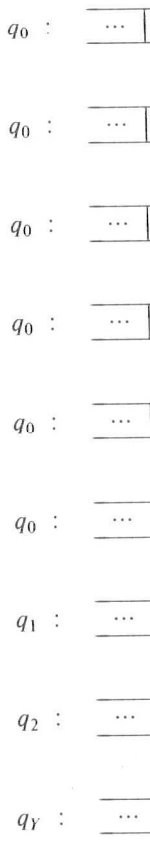

Figure 2.3 The computa

Observe that this
that M halt for *all* inpu
to $\Sigma^* - L_M$, then the c
might continue forever
correspond to our notio
over its input alphabet.
gorithmic, since it will h

The correspondenc
decision problems is str
the decision problem Π

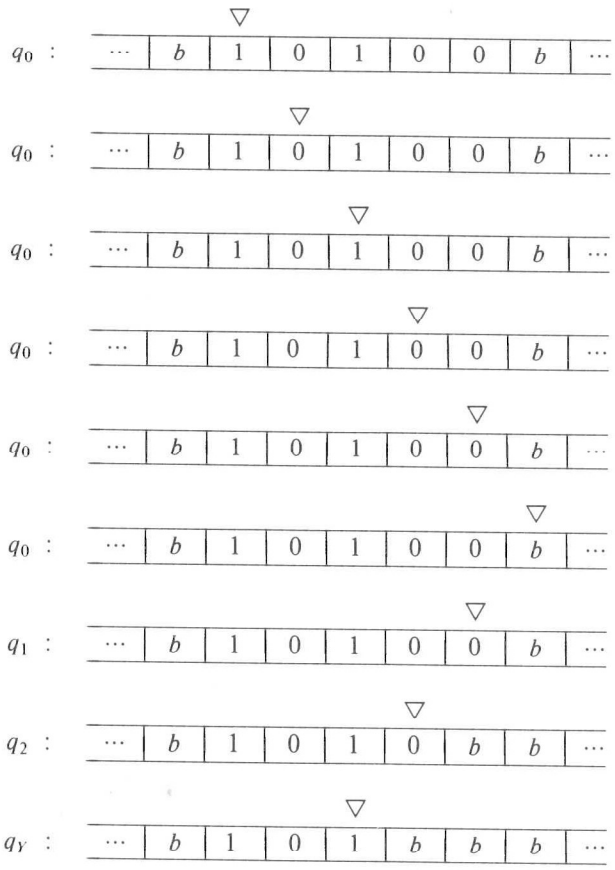

Figure 2.3 The computation of the program M from Figure 2.2 on input 10100.

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 call
polynomial p such that, f

We are now ready t
class of languages that w
follows:

$$P = \{ L : \textit{there is a pol}$$

We will say that a decis
scheme e if $L[\Pi, e] \in P$,
that "solves" Π under e
tioned equivalence betw
omit the specification of
saying that the decision p

We also will be infor
rithm." Our formal cou
lynomial time DTM p
between "realistic" comp
ed out in Chapter 1, the
in terms of programs fo
would have resulted. Th
DTM model when infor
formed by polynomial t
practice and discuss alg
speaking of them as op
(the sets, graphs, numbe
Here our implicit asserti
tience, design a polynom
nomial time algorithm w
taken as indicating how
reader familiar with the
nomial time on an ordin

2.3 Nondeterministic C

In this section
languages/decision prob
mal definitions in terms
be useful to provide an
tended to capture.

Consider the TRA
ginning of this chapter:

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:

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

We will say that a decision problem Π belongs to P under the encoding scheme e if $L[\Pi, e] \in \mathrm{P}$, 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,