Chapter 11

$\mathcal{NP}$-HARD AND $\mathcal{NP}$-COMPLETE PROBLEMS

11.1 BASIC CONCEPTS

In this chapter we are concerned with the distinction between problems that can be solved by a polynomial time algorithm and problems for which no polynomial time algorithm is known. It is an unexplained phenomenon that for many of the problems we know and study, the best algorithms for their solutions have computing times that cluster into two groups. The first group consists of problems whose solution times are bounded by polynomials of small degree. Examples we have seen in this book include ordered searching, which is $O(\log n)$, polynomial evaluation which is $O(n)$, sorting which is $O(n \log n)$, and string editing which is $O(mn)$.

The second group is made up of problems whose best-known algorithms are nonpolynomial. Examples we have seen include the traveling salesperson and the knapsack problems for which the best algorithms given in this text have complexities $O(n^2 2^n)$ and $O(2^{n/2})$ respectively. In the quest to develop efficient algorithms, no one has been able to develop a polynomial time algorithm for any problem in the second group. This is very important because algorithms whose computing times are greater than polynomial (typically the time is exponential) very quickly require such vast amounts of time to execute that even moderate-size problems cannot be solved (see Section 1.3 for more details).

The theory of $\mathcal{NP}$-completeness which we present here does not provide a method of obtaining polynomial time algorithms for problems in the second group. Nor does it say that algorithms of this complexity do not exist. Instead, what we do is show that many of the problems for which there are

no known polynomial time algorithms are computationally related. In fact, we establish two classes of problems. These are given the names $\mathcal{NP}$-hard and $\mathcal{NP}$-complete. A problem that is $\mathcal{NP}$-complete has the property that it can be solved in polynomial time if and only if all other $\mathcal{NP}$-complete problems can also be solved in polynomial time. If an $\mathcal{NP}$-hard problem can be solved in polynomial time, then all $\mathcal{NP}$-complete problems can be solved in polynomial time. All $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard, but some $\mathcal{NP}$-hard problems are not known to be $\mathcal{NP}$-complete.

Although one can define many distinct problem classes having the properties stated above for the $\mathcal{NP}$-hard and $\mathcal{NP}$-complete classes, the classes we study are related to nondeterministic computations (to be defined later). The relationship of these classes to nondeterministic computations together with the apparent power of nondeterminism leads to the intuitive (though as yet unproved) conclusion that no $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problem is polynomially solvable.

We see that the class of $\mathcal{NP}$-hard problems (and the subclass of $\mathcal{NP}$-complete problems) is very rich as it contains many interesting problems from a wide variety of disciplines. First, we formalize the preceding discussion of the classes.

11.1.1 Nondeterministic Algorithms

Up to now the notion of algorithm that we have been using has the property that the result of every operation is uniquely defined. Algorithms with this property are termed *deterministic algorithms*. Such algorithms agree with the way programs are executed on a computer. In a theoretical framework we can remove this restriction on the outcome of every operation. We can allow algorithms to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later. This leads to the concept of a *nondeterministic algorithm*. To specify such algorithms, we introduce three new functions:

1. Choice(S) arbitrarily chooses one of the elements of set S.
2. Failure() signals an unsuccessful completion.
3. Success() signals a successful completion.

The assignment statement $x :=$ Choice$(1, n)$ could result in x being assigned any one of the integers in the range $[1, n]$. There is no rule specifying how this choice is to be made. The Failure() and Success() signals are used to define a computation of the algorithm. These statements cannot be used to effect a **return**. Whenever there is a set of choices that leads to a successful

completion, then one such set of choices is always made and the algorithm terminates successfully. *A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.* The computing times for Choice, Success, and Failure are taken to be $O(1)$. A machine capable of executing a nondeterministic algorithm in this way is called a *nondeterministic machine.* Although nondeterministic machines (as defined here) do not exist in practice, we see that they provide strong intuitive reasons to conclude that certain problems cannot be solved by fast deterministic algorithms.

Example 11.1 Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is Algorithm 11.1.

```
1  j := Choice(1, n);
2  if A[j] = x then {write (j); Success();}
3  write (0); Failure();
```

Algorithm 11.1 Nondeterministic search

From the way a nondeterministic computation is defined, it follows that the number 0 can be output if and only if there is no j such that $A[j] = x$. Algorithm 11.1 is of nondeterministic complexity $O(1)$. Note that since A is not ordered, every deterministic search algorithm is of complexity $\Omega(n)$. □

Example 11.2 [Sorting] Let $A[i]$, $1 \leq i \leq n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) (Algorithm 11.2) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1:n]$ is used for convenience. Line 4 initializes B to zero though any value different from all the $A[i]$ will do. In the **for** loop of lines 5 to 10, each $A[i]$ is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 ascertains that $B[j]$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. The **for** loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 7 for such an output order, algorithm NSort is a sorting algorithm. Its complexity is $O(n)$. Recall that all deterministic sorting algorithms must have a complexity $\Omega(n \log n)$. □

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers.
3   {
4       for i := 1 to n do B[i] := 0; // Initialize B[ ].
5       for i := 1 to n do
6       {
7           j := Choice(1, n);
8           if B[j] ≠ 0 then Failure();
9           B[j] := A[i];
10      }
11      for i := 1 to n − 1 do  // Verify order.
12          if B[i] > B[i + 1] then Failure();
13      write (B[1 : n]);
14      Success();
15  }
```

Algorithm 11.2 Nondeterministic sorting

A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation. In theory, each time a choice is to be made, the algorithm makes several copies of itself. One copy is made for each of the possible choices. Thus, many copies are executing at the same time. The first copy to reach a successful completion terminates all other computations. If a copy reaches a failure completion, then only that copy of the algorithm terminates. Although this interpretation may enable one to better understand nondeterministic algorithms, it is important to remember that a nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead, it has the ability to select a "correct" element from the set of allowable choices (if such an element exists) every time a choice is to be made. A correct element is defined relative to a shortest sequence of choices that leads to a successful termination. In case there is no sequence of choices leading to a successful termination, we assume that the algorithm terminates in one unit of time with output "unsuccessful computation." Whenever successful termination is possible, a nondeterministic machine makes a sequence of choices that is a shortest sequence leading to a successful termination. Since, the machine we are defining is fictitious, it is not necessary for us to concern ourselves with how the machine can make a correct choice at each step.

Definition 11.1 Any problem for which the answer is either zero or one is called a *decision problem*. An algorithm for a decision problem is termed

a *decision algorithm.* Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an *optimization problem.* An *optimization algorithm* is used to solve an optimization problem. □

It is possible to construct nondeterministic algorithms for which many different choice sequences lead to successful completions. Algorithm NSort of Example 11.2 is one such algorithm. If the numbers $A[i]$ are not distinct, then many different permutations will result in a sorted sequence. If NSort were written to output the permutation used rather than the $A[i]$'s in sorted order, then its output would not be uniquely defined. We concern ourselves only with those nondeterministic algorithms that generate unique outputs. In particular we consider only nondeterministic decision algorithms. A successful completion is made if and only if the output is 1. A 0 is output if and only if there is no sequence of choices leading to a successful completion. The output statement is implicit in the signals Success and Failure. No explicit output statements are permitted in a decision algorithm. Clearly, our earlier definition of a nondeterministic computation implies that the output from a decision algorithm is uniquely defined by the input parameters and algorithm specification.

Although the idea of a decision algorithm may appear very restrictive at this time, many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can. In other cases, we can at least make the statement that if the decision problem cannot be solved in polynomial time, then the optimization problem cannot either.

Example 11.3 [Maximum clique] A maximal complete subgraph of a graph $G = (V, E)$ is a *clique.* The size of the clique is the number of vertices in it. The *max clique problem* is an optimization problem that has to determine the size of a largest clique in G. The corresponding decision problem is to determine whether G has a clique of size at least k for some given k. Let DClique(G, k) be a deterministic decision algorithm for the clique decision problem. If the number of vertices in G is n, the size of a max clique in G can be found by making several applications of DClique. DClique is used once for each k, $k = n, n-1, n-2, \ldots$, until the output from DClique is 1. If the time complexity of DClique is $f(n)$, then the size of a max clique can be found in time $\leq n\ f(n)$. Also, if the size of a max clique can be determined in time $g(n)$, then the decision problem can be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time. □

Example 11.4 [0/1 knapsack] The knapsack decision problem is to determine whether there is a 0/1 assignment of values to x_i, $1 \leq i \leq n$, such that $\sum p_i x_i \geq r$ and $\sum w_i x_i \leq m$. The r is a given number. The p_i's and w_i's are

nonnegative numbers. If the knapsack decision problem cannot be solved in deterministic polynomial time, then the optimization problem cannot either. □

Before proceeding further, it is necessary to arrive at a uniform parameter n to measure complexity. We assume that n is the length of the input to the algorithm (that is, n is the input size). We also assume that all inputs are integer. Rational inputs can be provided by specifying pairs of integers. Generally, the length of an input is measured assuming a binary representation; that is, if the number 10 is to be input, then in binary it is represented as 1010. Its length is 4. In general, a positive integer k has a length of $\lfloor \log_2 k \rfloor + 1$ bits when represented in binary. The length of the binary representation of 0 is 1. The size, or length, n of the input to an algorithm is the sum of the lengths of the individual numbers being input. In case the input is given using a different representation (say radix r), then the length of a positive number k is $\lfloor \log_r k \rfloor + 1$. Thus, in decimal notation, $r = 10$ and the number 100 has a length $\log_{10} 100 + 1 = 3$. Since $\log_r k = \log_2 k / \log_2 r$, the length of any input using radix r ($r > 1$) representation is $c(r)n$, where n is the length using a binary representation and $c(r)$ is a number that is fixed for a given r.

When inputs are given using the radix $r = 1$, we say the input is in *unary form*. In unary form, the number 5 is input as 11111. Thus, the length of a positive integer k is k. It is important to observe that the length of a unary input is exponentially related to the length of the corresponding r-ary input for radix r, $r > 1$.

Example 11.5 [Max clique] The input to the max clique decision problem can be provided as a sequence of edges and an integer k. Each edge in $E(G)$ is a pair of numbers (i, j). The size of the input for each edge (i, j) is $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$ if a binary representation is assumed. The input size of any instance is

$$n = \sum_{\substack{(i,j) \in E(G) \\ i<j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

Note that if G has only one connected component, then $n \geq |V|$. Thus, if this decision problem cannot be solved by an algorithm of complexity $p(n)$ for some polynomial $p()$, then it cannot be solved by an algorithm of complexity $p(|V|)$. □

Example 11.6 [0/1 knapsack] Assuming p_i, w_i, m, and r are all integers, the input size for the knapsack decision problem is

$$q = \sum_{1 \leq i \leq n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + 2n + \lfloor \log_2 m \rfloor + \lfloor \log_2 r \rfloor + 2$$

Note that $q > n$. If the input is given in unary notation, then the input size s is $\sum p_i + \sum w_i + m + r$. Note that the knapsack decision and optimization problems can be solved in time $p(s)$ for some polynomial $p()$ (see the dynamic programming algorithm). However, there is no known algorithm with complexity $O(p(n))$ for some polynomial $p()$. □

We are now ready to formally define the complexity of a nondeterministic algorithm.

Definition 11.2 The *time required by a nondeterministic algorithm* performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. In case successful completion is not possible, then the time required is $O(1)$. A nondeterministic algorithm is of complexity $O(f(n))$ if for all inputs of size n, $n \geq n_0$, that result in a successful completion, the time required is at most $cf(n)$ for some constants c and n_0. □

In Definition 11.2 we assume that each computation step is of a fixed cost. In word-oriented computers this is guaranteed by the finiteness of each word. When each step is not of a fixed cost, it is necessary to consider the cost of individual instructions. Thus, the addition of two m-bit numbers takes $O(m)$ time, their multiplication takes $O(m^2)$ time (using classical multiplication), and so on. To see the necessity of this, consider the algorithm Sum (Algorithm 11.3). This is a deterministic algorithm for the sum of subsets decision problem. It uses an $(m+1)$-bit word s. The ith bit in s is zero if and only if no subset of the integers $A[j]$, $1 \leq j \leq n$, sums to i. Bit 0 of s is always 1 and the bits are numbered $0, 1, 2, \ldots, m$ right to left. The function Shift shifts the bits in s to the left by $A[i]$ bits. The total number of steps for this algorithm is only $O(n)$. However, each step moves $m+1$ bits of data and would take $O(m)$ time on a conventional computer. Assuming one unit of time is needed for each basic operation for a fixed word size, the complexity is $O(nm)$ and not $O(n)$.

The virtue of conceiving of nondeterministic algorithms is that often what would be very complex to write deterministically is very easy to write nondeterministically. In fact, it is very easy to obtain polynomial time nondeterministic algorithms for many problems that can be deterministically solved by a systematic search of a solution space of exponential size.

Example 11.7 [Knapsack decision problem] DKP (Algorithm 11.4) is a nondeterministic polynomial time algorithm for the knapsack decision problem. The **for** loop of lines 4 to 8 assigns 0/1 values to $x[i]$, $1 \leq i \leq n$. It also computes the total weight and profit corresponding to this choice of $x[\,]$. Line 9 checks to see whether this assignment is feasible and whether the resulting profit is at least r. A successful termination is possible iff the answer to the decision problem is yes. The time complexity is $O(n)$. If q is the input length using a binary representation, the time is $O(q)$. □

```
1   Algorithm Sum(A, n, m)
2   {
3       s := 1;
4           // s is an (m + 1)-bit word. Bit zero is 1.
5       for i := 1 to n do
6           s := s or Shift(s, A[i]);
7       if the mth bit in s is 1 then
8           write ("A subset sums to m.");
9       else write ("No subset sums to m.");
10  }
```

Algorithm 11.3 Deterministic sum of subsets

Example 11.8 [Max clique] Algorithm DCK (Algorithm 11.5) is a nondeterministic algorithm for the clique decision problem. The algorithm begins by trying to form a set of k distinct vertices. Then it tests to see whether these vertices form a complete subgraph. If G is given by its adjacency matrix and $|V| = n$, the input length m is $n^2 + \lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$. The **for** loop of lines 4 to 9 can easily be implemented to run in nondeterministic time $O(n)$. The time for the **for** loop of lines 11 and 12 is $O(k^2)$. Hence the overall nondeterministic time is $O(n + k^2) = O(n^2) = O(m)$. There is no known polynomial time deterministic algorithm for this problem. □

Example 11.9 [Satisfiability] Let $x_1, x_2, \ldots$ denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. The symbol $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) if and only if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF whereas $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The **satisfiability** problem is to determine whether a formula is true for some assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

It is easy to obtain a polynomial time nondeterministic algorithm that terminates successfully if and only if a given propositional formula $E(x_1, \ldots, x_n)$ is satisfiable. Such an algorithm could proceed by simply choosing (nondeter-

```
1   Algorithm DKP(p, w, n, m, r, x)
2   {
3       W := 0; P := 0;
4       for i := 1 to n do
5       {
6           x[i] := Choice(0, 1);
7           W := W + x[i] * w[i]; P := P + x[i] * p[i];
8       }
9       if ((W > m) or (P < r)) then Failure();
10      else Success();
11  }
```

Algorithm 11.4 Nondeterministic knapsack algorithm

```
1   Algorithm DCK(G, n, k)
2   {
3       S := ∅; // S is an initially empty set.
4       for i := 1 to k do
5       {
6           t := Choice(1, n);
7           if t ∈ S then Failure();
8           S := S ∪ {t} // Add t to set S.
9       }
10      // At this point S contains k distinct vertex indices.
11      for all pairs (i, j) such that i ∈ S, j ∈ S, and i ≠ j do
12          if (i, j) is not an edge of G then Failure();
13      Success();
14  }
```

Algorithm 11.5 Nondeterministic clique pseudocode

ministically) one of the 2^n possible assignments of truth values to $(x_1, \ldots, x_n)$ and verifying that $E(x_1, \ldots, x_n)$ is true for that assignment.

Eval (Algorithm 11.6) does this. The nondeterministic time required by the algorithm is $O(n)$ to choose the value of $(x_1, \ldots, x_n)$ plus the time needed to deterministically evaluate E for that assignment. This time is proportional to the length of E. □

```
1  Algorithm Eval(E, n)
2  // Determine whether the propositional formula E is
3  // satisfiable. The variables are x_1, x_2, ..., x_n.
4  {
5      for i := 1 to n do  // Choose a truth value assignment.
6          x_i := Choice(false, true);
7      if E(x_1, ..., x_n) then Success();
8      else Failure();
9  }
```

Algorithm 11.6 Nondeterministic satisfiability

Reference :

Text Book-

E. Horowitz and S. Sahni: Fundamental of Computer Algorithms, Galgotia Pub. /Pitman, New Delhi/London, 1987/1978.