ALGORITMI 2 2009/10

PROBLEMI DIFFICILI

DAL TESTO: HOROWITZ - SAHNI
FUNDAMENTALS OF COMPUTER ALGORITHMS
COMPUTER SCIENCE PRESS

Chapter 11

NP-HARD AND NP-COMPLETE PROBLEMS

11.1 BASIC CONCEPTS

This chapter contains what is perhaps the most important theoretical development in algorithms research in the past decade. Its importance arises from the fact that the results have meaning for all researchers who are developing computer algorithms, not only computer scientists but electrical engineers, operations researchers, etc. Thus we believe that many people will turn immediately to this chapter. In recognition of this we have tried to make the chapter self-contained. Also, we have organized the later sections according to different areas of interest.

There are however some basic ideas which one should be familiar with before reading on. The first is the idea of analyzing apriori the computing time of an algorithm by studying the frequency of execution of its statements given various sets of data. A second notion is the concept of the order of magnitude of the time complexity of an algorithm and its expression by asymptotic notation. If $T(n)$ is the time for an algorithm on n inputs, then, we write $T(n) = O(f(n))$ to mean that the time is bounded *above* by the function $f(n)$, and $T(n) = \Omega(g(n))$ to mean that the time is bounded *below* by the function $g(n)$. Precise definitions and greater elaboration of these ideas can be found in Section 1.4.

Another important idea is the distinction between problems whose solution is by a polynomial time algorithm ($f(n)$ is a polynomial) and problems for which no polynomial time algorithm is known ($g(n)$ is larger than any polynomial). It is an unexplained phenomena that for many of the problems we know and study, the best algorithms for their solution have computing times which cluster into two groups. The first group consists of problems whose solution is bounded by a polynomial of small degree. Examples we have seen in this book include ordered searching which is $O(\log n)$, polynomial evaluation is $O(n)$, sorting is $O(n \log n)$, and matrix multiplication which is $O(n^{2.81})$.

The second group contains problems whose best known algorithms are nonpolynomial. Examples we have seen include the traveling salesperson and the knapsack problem for which the best algorithms given in this text have a complexity $O(n^2 2^n)$ 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 time is 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 1.4 for more details.)

The theory of 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 shall do is show that many of the problems for which there is no known polynomial time algorithm are computationally related. In fact, we shall establish two classes of problems. These will be given the names NP-hard and NP-complete. A problem which is NP-complete will have the property that it can be solved in polynomial time iff all other NP-complete problems can also be solved in polynomial time. If an NP-hard problem can be solved in polynomial time then all NP-complete problems can be solved in polynomial time. As we shall see all NP-complete problems are NP-hard but all NP-hard problems are not NP-complete.

While one can define many distinct problem classes having the properties stated above for the NP-hard and 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 NP-complete or NP-hard problem is polynomially solvable.

We shall see that the class of NP-hard problems (and the subclass of 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.

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 outcome is not uniquely defined but is limited to a specified set 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 one new function and two new statements into SPARKS:

(i) **choice** (S)...arbitrarily chooses one of the elements of set S
(ii) **failure** ...signals an unsuccessful completion
(iii) **success** ...signals a successful completion.

The assignment statement $X \leftarrow$ **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 are equivalent to a **stop** statement and 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 unsuccessfuly 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.* While nondeterministic machines (as defined here) do not exist in practice, we shall see that they will 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

```
j ← choice(1:n)
if A(j) = x then print(j); success endif
print('0'); failure
```

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 $A(j) = x$.

The above algorithm 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 $A(i)$, $1 \le i \le n$ be an unsorted set of positive integers. The nondeterministic algorithm NSORT(A, n) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B(1:n)$ is used for convenience. Line 1 initializes B to zero though any value different from all the $A(i)$ will do. In the loop of lines 2-6 each $A(i)$ is assigned to a position in B. Line 3 nondeterministically determines this position. Line 4 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. Lines 7 to 9 verify that B is sorted in nondecreasing order. A successful completion is achieved iff the numbers are output in nondecreasing order. Since there is always a set of choices at line 3 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 $\Omega(n \log n)$. □

```
    procedure NSORT(A, n)
      //sort n positive integers//
      integer A(n), B(n), n, i, j
1     B ← 0  //initialize B to zero//
2     for i ← 1 to n do
3       j ← choice(1:n)
4       if B(j) ≠ 0 then failure endif
5       B(j) ← A(i)
6     repeat
7     for i ← 1 to n − 1 do   //verify order//
8       if B(i) > B(i + 1) then failure endif
9     repeat
10    print(B)
11    success
12  end NSORT
```

Algorithm 11.1 Nondeterministic sorting

A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation. 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. Recall that the **success** and **failure** signals are equivalent to **stop** statements in deterministic algorithms. They may not be used in place of **return** statements. While 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 shall 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 which 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.

It is possible to construct nondeterministic algorithms for which many different choice sequences lead to a successful completion. Procedure 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 shall concern ourselves only with those nondeterministic algorithms that generate a unique output. In particular we shall consider only *nondeterministic decision algorithms*. Such algorithms generate only a zero or one as their output. A binary decision is made. A successful completion is made iff the output is '1'. A '0' is output iff 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 the algorithm specification.

While 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 iff 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 [Max 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 to determine the size of a largest clique in G. The corresponding decision problem is to determine if 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 $n^*f(n)$. Also, if the size of a max clique can be determined in time $g(n)$ then the decision problem may be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time iff the clique decision problem can be solved in polynomial time. □

Example 11.4 [0/1-Knapsack] The knapsack decision problem is to determine if there is a 0/1 assignment of values to x_i, $1 \leq i \leq n$ such that $\Sigma p_i x_i \geq R$ and $\Sigma w_i x_i \leq M$. R is a given number. The p_i's and w_i's are nonnegative numbers. Clearly, 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 shall assume that n is the length of the input to the algorithm. We shall 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. I.e., 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 $\log_{10} 100 + 1 = 3$ digits. Since $\log_r k = \log_2 k / \log_2 r$, the length of any input using radix $r (r > 1)$ representation is $c(r) \cdot n$ where n is the length using a binary representation and $c(r)$ is a number which is fixed for a given r.

When inputs are given using the radix $r = 1$, we shall 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 may 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

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

Note that $m \geq n$. If the input is given in unary notation then the input size s is $\Sigma p_i + \Sigma w_i + M + R$. 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 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 $c \cdot f(n)$ for some constants c and n_0.

In the above definition 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 additon of two m bit numbers takes $O(m)$ time, their multiplication takes $O(m^2)$ time (using classical multiplication) etc. To see the necessity of this consider procedure SUM (Algorithm 11.2). This is a deterministic algorithm for the sum of subsets decision problem. It uses an $M + 1$ bit word S. The i'th bit in S is zero iff no subset of the integers $A(j)$, $1 \le j \le n$ sums to i. Bit 0 of S is always 1 and the bits are numbered 0, 1, 2, ..., 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 really 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 true complexity is $O(nM)$ and not $O(n)$.

```
procedure SUM(A, n, M)
  integer A(n), S, n, M
  S ← 1   //S is an M + 1 bit word. Bit zero is 1//
  for i ← 1 to n do
    S ← S or SHIFT(S, A(i))
  repeat
  if Mth bit in S = 0 then print ('no subset sums to M')
                       else print ('a subset sums to M')
  endif
end SUM
```

Algorithm 11.2 Deterministic sum of subsets

The virtue of conceiving of nondeterministic algorithms is that often what would be very complex to write down 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] Procedure DKP (Algorithm 11.3) is a nondeterministic polynomial time algorithm for the knapsack decision problem. Lines 1 to 3 assign 0/1 values to $X(i)$, $1 \le i \le n$. Line 4 checks to see if this assignment is feasible and if 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 m is the input length using a binary representation, the time is $O(m)$. □

```
   procedure DKP(P, W, n, M, R, X)
     integer P(n), W(n), R, X(n), n, M, i
1    for i ← 1 to n do
2      X(i) ← choice(0, 1)
3    repeat
4    if $\sum_{1\le i\le n}$ (W(i)*X(i)) > M or $\sum_{1\le i\le n}$ (P(i)*X(i)) < R then failure
                                                                      else success
5    endif
   end DKP
```

Algorithm 11.3 Nondeterministic Knapsack problem

Example 11.8 [Max Clique] Procedure DCK (Algorithm 11.4) 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 if these vertices form a complete subgraph. If G is given by its adjacency matrix and $|V| = n$, the input length m is $n^2 + \lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$. Lines 2 to 6 can easily be implemented to run in nondeterministic time $O(n)$. The time for lines 7-10 is $O(k^2)$. Hence the overall nondeterministic time is $O(n + k^2) = O(n^2) = O(m)$. There is known polynomial time deterministic algorithm for this problem. □

```
    procedure DCK(G, n, k)
1     S ← φ   //S is an initially empty set//
2     for i ← 1 to k do   //select k distinct vertices//
3       t ← choice (1:n)
4       if t ∈ S then failure endif
5       S ← S ∪ t   //add t to set S//
6     repeat
      //at this point S contains k distinct vertex indices//
7     for all pairs (i, j) such that i ∈ S, j ∈ S and i ≠ j do
8       if (i, j) is not an edge of the graph
9         then failure endif
10    repeat
11    success
    end DCK
```

Algorithm 11.4 Nondeterministic clique

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 $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$; $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) iff 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) iff 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 while $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The *satisfiability* problem is to determine if a formula is true for any 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 (nondeterministically) 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.

Procedure EVAL (Algorithm 11.5) 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. □

```
procedure EVAL(E, n)
  //Determine if the propositional formula $E$ is satisfiable. The variables//
  //are $x_i$, $1 \le i \le n$//
  boolean x(n)
  for $i \leftarrow 1$ to $n$ do   //choose a truth value assignment//
    $x_i \leftarrow$ choice (true, false)
  repeat
  if $E(x_1, \ldots, x_n)$ is true then success   //satisfiable//
                         else failure
  endif
end EVAL
```

Algorithm 11.5 Nondeterministic satisfiability

The Classes NP-hard and NP-complete

In measuring the complexity of an algorithm we shall use the input length

as the parameter. An algorithm A is of *polynomial complexity* if there exists a polynomial $p(\ )$ such that the computing time of A is $O(p(n))$ for every input of size n.

Definition P is the set of all decision problems solvable by a deterministic algorithm in polynomial time. NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic ones, we can conclude that $P \subseteq NP$. What we do not know, and what has become perhaps the most famous unsolved problem in computer science is whether $P = NP$ or $P \neq NP$.

Is it possible that for all of the problems in NP there exist polynomial time deterministic algorithms which have remained undiscovered? This seems unlikely, at least because of the tremendous effort which has already been expended by so many people on these problems. Nevertheless, a proof that $P \neq NP$ is just as elusive and seems to require as yet undiscovered techniques. But as with many famous unsolved problems, they serve to generate other useful results, and the $P \stackrel{?}{=} NP$ question is no exception.

In considering this problem S. Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that $P = NP$. Cook answered his own question in the affirmative with the following theorem.

Theorem 11.1 (Cook) Satisfiability is in P if and only if $P = NP$.

Proof: See Section 11.2 □

We are now ready to define the NP-hard and NP-complete classes of problems. First we define the notion of reducibility.

Definition Let L_1 and L_2 be problems. L_1 *reduces to* L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

This definition implies that if we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time. One may readily verify that $\propto$ is a transitive relation (i.e. if $L_1 \alpha L_2$ and $L_2 \alpha L_3$ then $L_1 \alpha L_3$).

Definition A problem L is *NP-hard* if and only if satisfiability reduces to L (satisfiability $\propto L$). A problem L is *NP-complete* if and only if L is *NP-hard* and $L \in NP$.

It is easy to see that there are *NP*-hard problems that are not *NP*-complete. Only a decision problem can be NP-complete. However, an optimization problem may be NP-hard. Furthermore if L_1 is a decision problem and L_2 an optimization problem, it is quite possible that $L_1 \propto L_2$. One may trivially show that the knapsack decision problem reduces to the knapsack optimization problem. For the clique problem one may easily show that the clique decision problem reduces to the clique optimization problem. In fact, we can also show that these optimization problems reduce to their corresponding decision problems (see exercises). Yet, optimization problems cannot be NP-complete while decision problems can. There also exist NP-hard decision problems that are not NP-complete.

Example 11.10 As an extreme example of an NP-hard decision problem that is not NP-complete consider the halting problem for deterministic algorithms. The ***halting problem*** is to determine for an arbitrary deterministic algorithm A and an input I whether algorithm A with input I ever terminates (or enters an infinite loop). It is well known that this problem is undecidable. Hence, there exists no algorithm (of any complexity) to solve this problem. So, it clearly cannot be in NP. To show satisfiability $\propto$ halting problem simply construct an algorithm A whose input is a propositional formula X. If X has n variables then A tries out all 2^n possible truth assignments and verifies if X is satisfiable. If it is then A stops. If X is not satisfiable then A enters an infinite loop. Hence, A halts on input X iff X is satisfiable. If we had a polynomial time algorithm for the halting problem then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for the halting problem. Hence, the halting problem is an NP-hard problem which is not in NP. □

Definition Two problems L_1 and L_2 are said to be *polynomially equivalent* iff $L_1 \propto L_2$ and $L_2 \propto L_1$.

In order to show that a problem, L_2 is NP-hard it is adequate to show $L_1 \propto L_2$ where L_1 is some problem already known to be NP-hard. Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto L_1$ and $L_1 \propto L_2$ then satisfiability $\propto L_2$. *To show an NP-hard decision problem NP-complete we have just to exhibit a polynomial time nondeterministic algorithm for it.* Later sections will show many problems to be NP-hard. While we shall restrict ourselves to decision problems, it should be clear that the corresponding optimization problems are also NP-hard. The NP-completeness proofs will be left as exercises (for those problems that are NP-complete).

11.2 COOK'S THEOREM

Cook's theorem (Theorem 11.1) states that satisfiability is in P iff P = NP. We shall now prove this important theorem. We have already seen that satisfiability is in NP (Example 11.9). Hence, if P = NP then satisfiability is in P. It remains to be shown that if satisfiability is in P then P = NP. In order to prove this latter statement, we shall show how to obtain from any polynomial time nondeterministic decision algorithm A and input I a formula $Q(A, I)$ such that Q is satisfiable iff A has a successful termination with input I. If the length of I is n and the time complexity of A is $p(n)$ for some polynomial $p(\ )$ then the length of Q will be $O(p^3(n) \log n) = O(p^4(n))$. The time needed to construct Q will also be $O(p^3(n) \log n)$. A deterministic algorithm Z to determine the outcome of A on any input I may be easily obtained. Z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether or not Q is satisfiable. If $O(q(m))$ is the time needed to determine if a formula of length m is satisfiable then the complexity of Z is $O(p^3(n) \log n + q(p^3(n) \log n))$. If satisfiability is in P then $q(m)$ is a polynomial function of m and the complexity of Z becomes $O(r(n))$ for some polynomial $r(\ )$. Hence, if satisfiability is in P then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P. So, the above construction will show that if satisfiability is in P then P = NP.

Before going into the construction of Q from A and I, we shall make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions will not in any way alter the class of decision problems in NP or P. The simplifying assumptions are:

i) The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction etc. between numbers one word long take one unit of time. In case numbers are longer than a word then the corresponding operations take at least as many units as the number of words making up the longest number.
ii) A *simple expression* is an expression that contains at most one operator and all operands are simple variables (i.e., no array variables are used). Some sample simple expressions are $-B$, $B + C$, D **or** E, F.

We shall assume that all assignment statements in A are of one of the following forms:

a) (simple variable) $\leftarrow$ (simple expression)
b) (array variable) $\leftarrow$ (simple variable)
c) (simple variable) $\leftarrow$ (array variable)
d) (simple variable) $\leftarrow$ **choice** (S) where S may be a finite set $\{S_1, S_2, \ldots, S_k\}$ or S may be $l{:}u$. In the latter case the function chooses an integer in the range $[l{:}u]$.

Indexing within an array is done using a simple integer variable and all index values are positive. Only one dimensional arrays are allowed. Clearly, all assignment statements not falling into one of the above categories may be replaced by a set of statements of these types. Hence, this restriction does not alter the class NP.

iii) All variables in A are of type integer or boolean.

iv) A contains no **read** or **print** statements. The only input to A is via its parameters. At the time A is invoked all variables (other than the parameters) have value zero (or **false** if boolean).

v) A contains no constants. Clearly, all constants in any algorithm may be replaced by new variables. These new variables may be added to the parameter list of A and the constants associated with them can be part of the input.

vi) In addition to simple assignment statements, A is allowed to contain only the following types of statements:

a) **go to** k where k is an instruction number
b) **if** c **then go to** a **endif**. c is a simple boolean variable (i.e., not an array) and a is an instruction number
c) **success, failure, end**
d) A may contain type declaration and dimension statments. These are not used during execution of A and so need not be translated into Q. The dimension information is used to allocate array space. It is assumed that successive elements in an array are assigned to consective words in memory.

It is assumed that the instructions in A are numbered sequentially from 1 to l (if A has l instructions). Every statement in A has a number. The **go to** instructions in a) and b) use this numbering scheme to effect a branch. It should be easy to see how to rewrite '**while-repeat**', '**repeat-until**', '**case-endcase**', '**for-repeat**', etc. statements in terms of **go to** and **if** c **then go to** a **endif** statements. Also, note that the **go to** k statement can be replaced by the statement **if true then go to** k **endif**. So, this may also be eliminated.

vii) Let $p(n)$ be a polynomial such that A takes no more than $p(n)$ time units on any input of length n. Because of the complexity assumptions of (i), A cannot change or use more than $p(n)$ words of memory. We may assume that A uses some subset of the words indexed $1, 2, 3, \ldots, p(n)$. This assumption does not restrict the class of decision problems in NP. To see this let $f(1), f(2), \ldots, f(k)$, $1 \le k \le p(n)$, be the distinct words used by A while working on input I. We can construct another polynomial time nondeterministic algorithm A' which uses $2p(n)$ words indexed $1, 2, \ldots, 2p(n)$ and solves the same decision problem as does A. A' simulates the behavior of A. However, A' maps the addresses $f(1), f(2), \ldots, f(k)$ onto the set $\{1, 2, \ldots, k\}$. The mapping function used is determined dynamically and is stored as a table in words $p(n) + 1$ through $2p(n)$. If the entry at word $p(n) + i$ is j then A' uses word i to hold the same value that A stored in word j. The simulation of A proceeds as follows: Let k be the number of distinct words referenced by A up to this time. Let j be a word referenced by A in the current step. A' searches its table to find word $p(n) + i$, $1 \le i \le k$ such that the contents of this word is j. If no such i exists then A' sets $k \leftarrow k + 1$, $i \leftarrow k$ and word $p(n) + k$ is given the value j. A' makes use of the word i to do whatever A would have done with word j. Clearly, A' and A solve the same decision problem. The complexity of A' is $O(p^2(n))$ as it takes A' $p(n)$ time to search its table and simulate a step of A. Since $p^2(n)$ is also a polynomial in n, restricting our algorithms to use only consecutive words does not alter the classes P and NP.

Formula Q will make use of several boolean variables. We state the semantics of two sets of variables used in Q:

i) $B(i, j, t)$, $1 \le i \le p(n)$, $1 \le j \le w$, $0 \le t < p(n)$.

$B(i, j, t)$ represents the status of bit j of word i following t steps (or time units) of computation. The bits in a word are numbered from right to left. The rightmost bit is numbered 1. Q will be constructed such that in any truth assignment for which Q is true, $B(i, j, t)$ is true iff the corresponding bit has value 1 following t steps of some successful computation of A on input I.

ii) $S(j, t), 1 \leq j \leq l, 1 \leq t \leq p(n)$.
Recall that l is the number of instructions in A. $S(j, t)$ represents the instruction to be executed at time t. Q will be constructed such that in any truth assignment for which Q is true, $S(j, t)$ is true iff the instruction executed by A at time t is instruction j.

Q will be made up of six subformulas C, D, E, F, G and H. $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$. These subformulas will make the following assertions:

C: The initial status of the $p(n)$ words represents the input I. All non-input variables are zero.
D: Instruction 1 is the first instruction to execute.
E: At the end of the i'th step, there can be only one next instruction to execute. Hence, for any fixed i, at most one of the $S(j, i)$, $1 \leq j \leq l$ can be true.
F: If $S(j, i)$ is true then $S(j, i + 1)$ is also true if instruction j is a **success**, **failure** or **end** statement. $S(j + 1, i + 1)$ is true if j is an assignment statement. If j is a **go to** k statement then $S(k, i + 1)$ is true. The last possibility for j is the **if** c **then** a **endif** statement. In this case $S(a, i + 1)$ is true if c is true and $S(j + 1, i + 1)$ is true if c is false.
G: If the instruction executed at step t is not an assignment statement then the $B(i, j, t)$s are unchanged. If this instruction is an assignment and the variable on the left hand side is X, then only X may change. This change is determined by the right hand side of the instruction.
H: The instruction to be executed at time $p(n)$ is a **success** instruction. Hence the computation terminates successfully.

Clearly, if C through H make the above assertions, then $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$ is satisfiable iff there is a successful computation of A on input I. We now give the formulas C through H. While presenting these formulas we shall also indicate how each may be transformed into CNF. This transformation will increase the length of Q by an amount independent of n (but dependent on w and l). This will enable us to show that CNF-satisfiability is NP-complete.

1. Formula C describes the input I. We have:

$$C = \bigwedge_{\substack{1 \leq i \leq p(n) \\ 1 \leq j \leq w}} T(i, j, 0)$$

$T(i, j, 0)$ is $B(i, j, 0)$ if the input calls for bit $B(i, j, 0)$ (i.e. bit j of word i) to be 1. $T(i, j, 0)$ is $\bar{B}(i, j, 0)$ otherwise. Thus, if there is no input then

$$C = \bigwedge_{\substack{1 \leq i \leq p(n) \\ 1 \leq j \leq w}} \bar{B}(i, j, 0).$$

Clearly, C is uniquely determined by I and is in CNF. Also, C is satisfiable only by a truth assignment representing the initial values of all variables in A.

2. $D = S(1, 1) \wedge \bar{S}(2, 1) \wedge \bar{S}(3, 1) \wedge \ldots \wedge \bar{S}(l, 1)$.
Clearly, D is satisfiable only by the assignment $S(1, 1) =$ true and $S(i, 1) =$ false, $2 \leq i \leq l$. Using our interpretation of $S(i, 1)$, this means that D is true iff instruction 1 is the first to be executed. Note that D is in CNF.

3. $E = \bigwedge_{1 < t \leq p(n)} E_t$.

Each E_t will assert that there is a unique instruction for step t. We may define E_t to be:

$$E_t = (S(1, t) \vee S(2, t) \vee \ldots \vee S(l, t)) \wedge (\bigwedge_{\substack{1 \leq j \leq l \\ 1 \leq k \leq l \\ j \neq k}} (\bar{S}(j, t) \vee \bar{S}(k, t))$$

One may verify that E_t is true iff exactly one of the $S(j, t)$s, $1 \leq j \leq l$ is true. Also, note that E is in CNF.

4. $F = \bigwedge_{\substack{1 \leq i \leq l \\ 1 \leq t < p(n)}} F_{i,t}$.

Each $F_{i,t}$ asserts that either instruction i is not the one to be executed at time t, or if it is then the instruction to be executed at time $t + 1$ is correctly determined by instruction i. Formally, we have

$$F_{i,t} = \bar{S}(i, t) \vee L$$

where L is defined as follows:

i) if instruction i is **success, failure** or **end** then L is $S(i, t + 1)$. Hence the program cannot leave such an instruction.

ii) if instruction i is **go to** k then L is $S(k, t+1)$.
iii) if instruction i is **if** X **then go to** k **endif** and variable X is represented by word j then L is $((B(j, 1, t-1) \wedge S(k, t+1)) \vee (\bar{B}(j, 1, t-1) \wedge S(i+1, t+1)))$. This assumes that bit 1 of X is 1 iff X is true.
iv) if instruction i is not any of the above then L is $S(i+1, t+1)$.

The $F_{i,t}$s defined in cases (i), (ii) and (iv) above are in CNF. The $F_{i,t}$ in case (iii) may be transformed into CNF using the boolean identity $a \vee (b \wedge c) \vee (d \wedge e) \equiv (a \vee b \vee d) \wedge (a \vee c \vee d) \wedge (a \vee b \vee e) \wedge (a \vee c \vee e)$.

5. $G = \bigwedge_{\substack{1 \le i \le l \\ 1 \le t < p(n)}} G_{i,t}$.

Each $G_{i,t}$ asserts that at time t either (i) instruction i is not executed or (ii) it is and the status of the $p(n)$ words after step t is correct with respect to the status before step t and the changes resulting from instruction i. Formally, we have

$$G_{i,t} = \bar{S}(i, t) \vee M$$

where M is defined as follows:

i) if instruction i is a **go to, if—then go to—endif, success, failure,** or **end** statement then M asserts that the status of the $p(n)$ words is unchanged. I.e., $B(k, j, t-1) = B(k, j, t)$, $1 \le k \le p(n)$ and $1 \le j \le w$.

$$M = \bigwedge_{\substack{1 \le k \le p(n) \\ 1 \le j \le w}} ((B(k,j,t-1) \wedge B(k,j,t)) \vee (\bar{B}(k,j,t-1) \wedge \bar{B}(kj, t))$$

In this case, $G_{i,t}$ may be rewritten as

$$G_{i,t} = \bigwedge_{\substack{1 \le k \le p(n) \\ 1 \le j \le w}} (\bar{S}(i, t) \vee (B(k, j, t-1) \wedge B(k, j, t)) \\ \vee (\bar{B}(k, j, t-1) \wedge \bar{B}(k, j, t)))$$

Each clause in $G_{i,t}$ is of the form $z \vee (x \wedge s) \vee (\bar{x} \wedge \bar{s})$ where z is $\bar{S}(i, t)$, x represents a $B(\ ,\ , t-1)$ and s a $B(\ ,\ , t)$. Note that $z \vee (x \wedge s) \vee (\bar{x} \wedge \bar{s})$ is equivalent to $(x \vee \bar{s} \vee z) \wedge (\bar{x} \vee s \vee z)$. Hence, $G_{i,t}$ may be transformed into CNF easily.

ii) if i is an assignment statement of type a) then M depends on the operator (if any) on the right hand side. We shall first describe the form of M for the case when instruction i is of the type $Y \leftarrow V + Z$. Let Y, V and Z be respectively represented in words y, v and z. We shall make the simplifying assumption that all numbers are non-negative. The exercises examine the case when negative numbers are allowed and 1's complement arithmetic is being used. In order to get a formula asserting that the bits $B(y, j, t)$, $1 \le j \le w$ represent the sum of $B(v, j, t-1)$ and $B(z, j, t-1)$ $1 \le j \le w$, we shall have to make use of w additional bits $C(j, t)$, $1 \le j \le w$. $C(j, t)$ will represent the carry from the addition of the bits $B(v, j, t-1)$, $B(z, j, t-1)$ and $C(j-1, t)$, $1 < j \le w$. $C(1, t)$ is the carry from the addition of $B(v, 1, t-1)$ and $B(z, 1, t-1)$. Recall that a bit is 1 iff the corresponding variable is true. Performing a bit wise addition of V and Z, we obtain $C(1, t) = B(v, 1, t-1) \wedge B(z, 1, t-1)$ and $B(y, 1, t) = B(v, 1, t-1) \oplus B(z, 1, t-1)$ where $\oplus$ is the exclusive or operation ($a \oplus b$ is true iff exactly one of a and b is true). Note that $a \oplus b \equiv (a \vee b) \wedge (\overline{a \wedge b}) \equiv (a \vee b) \wedge (\bar{a} \vee \bar{b})$. Hence, the right hand side of the expression for $B(y, 1, t)$ may be transformed into CNF using this identity. For the other bits of Y, one may verify that

$$B(y, j, t) = B(v, j, t-1) \oplus (B(z, j, t-1) \oplus C(j-1, t))$$

and

$$C(j, t) = (B(v, j, t-1) \wedge B(z, j, t-1)) \vee (B(v, j, t-1) \\ \wedge C(j-1, t)) \vee (B(z, j, t-1) \wedge C(j-1, t)).$$

Finally, we require that $C(w, t) =$ false. (i.e. there is no overflow). Let M' be the **and** of all the equations for $B(y, j, t)$ and $C(j, t)$, $1 \le j \le w$. M is given by

$$M = (\bigwedge_{\substack{1 \le k \le p(n) \\ k \ne y \\ 1 \le j \le w}} ((B(k, j, t-1) \wedge B(k, j, t)) \\ \vee (\bar{B}(k, j, t-1) \wedge \bar{B}(k, j, t))) \wedge M'$$

$G_{i,t}$ may be converted into CNF using the idea of 5 (i). This transformation will increase the length of $G_{i,t}$ by a constant factor independent of n. We leave it to the reader to figure out what M is when instruction i is either of the form $Y \leftarrow V$; $Y \leftarrow V \circledcirc Z$ for $\circledcirc$ one of $-$, $/$, $*$, $<$, $>$, $\leq$, $=$, etc.

When i is an assignment statement of types b) or c) then it necessary to select the correct array element. Consider an instruction of type b): $R(m) \leftarrow X$. In this case the formula M may be written as:

$$M = W \wedge (\bigwedge_{1 \leq j \leq u} M_j)$$

where u is the dimension of R. Note that because of restriction (vii) on the algorithm A, $u \leq p(n) \cdot W$ asserts that $1 \leq m \leq u$. The specification of W is left as an exercise. Each M_j asserts that either $m \neq j$ or $m = j$ and only the jth element of R changes. Let us assume that the values of X and m are respectively stored in words x and m and that $R(1{:}u)$ is stored in words $\alpha, \alpha + 1, \ldots, \alpha + u - 1$. M_j is given by:

$$M_j = \bigvee_{1 \leq k \leq w} T(m, k, t - 1) \vee Z$$

where T is B if the k'th bit in the binary representation of j is O and T is $\bar{B}$ otherwise. Z is defined as

$$Z = \bigwedge_{\substack{1 \leq k \leq w \\ 1 \leq r \neq p(n) \\ r \neq \alpha + j - 1}} ((B(r, k, t - 1) \wedge B(r, k, t)) \vee (\bar{B}(r, k, t - 1) \wedge \bar{B}(r, k, t - 1)))$$

$$\bigwedge_{1 \leq k \leq w} ((B(\alpha + j - 1, k, t) \wedge B(x, k, t - 1)) \vee (\bar{B}(\alpha + j - 1, k, t) \wedge \bar{B}(x, k, t - 1))).$$

Note that the number of literals in M is $O(p^2(n))$. Since j is w bits long it can represent only numbers smaller than 2^w. Hence, for $u \geq 2^w$ we need a different indexing scheme. A simple generalization is to allow multiprecision arithmetic. The index variable j could use as many words as needed. The number of words used would depend on u. At most $\log(p(n))$ words are needed. This calls for a slight change in M_j but the number of literals in M remains $O(p^2(n))$. There is no need to explicitly incorporate multiprecision arithmetic as by giving the

program access to individual words in a multiprecision index j we can require the program to simulate multiprecision arithmetic.

When i is an instruction of type c) the form of M is similar to that obtained for instructions of type b). Next, we describe how to construct M for the case i is of the form $Y \leftarrow$ **choice** (S) where S is either a set of the form $S = \{S_1, S_2, \ldots, S_k\}$ or S is of the form $r{:}u$. Assume Y is represented by word y. Is S is a set then we define

$$M = \bigvee_{1 \leq j \leq k} M_j.$$

M_j asserts that Y is S_j. This is easily done by choosing $M_j = a_1 \wedge a_2 \wedge \cdots \wedge a_w$ where $a_i = B(y, i, t)$ if bit i is 1 in S_j and $a_i = \bar{B}(y, i, t)$ if bit l is zero in S_j. If S is of the form $r{:}u$ then M is just the formula that asserts $r \leq Y \leq u$. This is left as an exercise. In both cases, $G_{i,t}$ may be transformed into CNF increasing the length of $G_{i,t}$ by at most a constant amount.

6. Let $i_1, i_2, \ldots, i_k$ be the statement numbers corresponding to the success statements in A. H is given by:

$$H = S(i_1, p(n)) \vee S(i_2, p(n)) \vee \cdots \vee S(i_k, p(n)).$$

One may readily verify that $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$ is satisfiable iff the computation of algorithm A with input I terminates successfully. Further, Q may be transformed into CNF as described above. Formula C contains $wp(n)$ literals, D contains l literals, E contains $O(l^2 p(n))$ literals, F contains $O(lp(n))$ literals, G contains $O(lwp^3(n))$ literals and H contains at most l literals. The total number of literals appearing in Q is $O(lwp^3(n)) = O(p^3(n))$ as lw is constant. Since, there are $O(wp^2(n) + lp(n))$ distinct literals in Q, each literal can be written down using $O(\log(wp^2(n) + lp(n))) = O(\log n)$ bits. The length of Q is therefore $O(p^3(n) \log n) = O(p^4(n))$ as $p(n)$ is at least n. The time to construct Q from A and I is also $O(p^3(n) \log n)$.

The above construction, shows that every problem in NP reduces to satisfiability and also to CNF-satisfiability. Hence, if either of these two problems is in P then NP $\subseteq P$ and so $P =$ NP. Also, since satisfiability is in NP, the construction of a CNF formula Q shows that satisfiability $\propto$ CNF-satisfiability. This together with the knowledge that CNF-satisfiability is in NP, implies that CNF-satisfiability is NP-complete. Note that satisfiability is also NP-complete as satisfiability α satisfiability and satisfiability is in NP.

11.3 NP-HARD GRAPH PROBLEMS

The strategy we shall adopt to show that a problem L_2 is NP-hard is:

i) Pick a problem L_1 already known to be NP-hard.
ii) Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I to L_1.
iii) Conclude from (ii) that $L_1 \propto L_2$.
iv) Conclude from (i), (iii) and the transitivity of $\propto$ that L_2 is NP-hard.

For the first few proofs we shall go through all the above steps. Later proofs will explicitly deal only with steps (i) and (ii). An NP-hard decision problem L_2 can be shown NP-complete by exhibiting a polynomial time nondeterministic algorithm for L_2. All the NP-hard decision problems we shall deal with here are also NP-complete. The construction of polynomial time nondeterministic algorithms for these problems is left as an exercise.

Clique Decision Problem (CDP)

The clique decision problem was introduced in Section 11.1. We shall show in Theorem 11.2 that CNF-satisfiability $\propto$ CDP. Using this result, the transitivity of $\propto$ and the knowledge that satisfiability $\propto$ CNF-satisfiability (Section 11.2) we can readily establish that satisfiability $\propto$ CDP. Hence, CDP is NP-hard. Since, CDP $\in$ NP, CDP is also NP-complete.

Theorem 11.2 CNF-satisfiability $\propto$ clique decision problem (CDP)

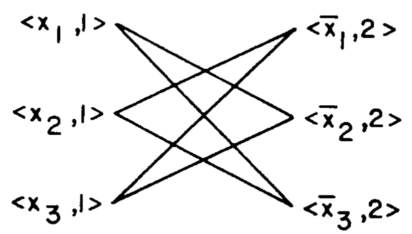
Proof: Let $F = \bigwedge_{1 \le i \le k} C_i$ be a propositional formula in CNF. Let x_i, $1 \le i \le n$ be the variables in F. We shall show how to construct from F a graph $G = (V, E)$ such that G will have a clique of size at least k iff F is satisfiable. If the length of F is m, then G will be obtainable from F in $O(m)$ time. Hence, if we have a polynomial time algorithm for CDP, then we can obtain a polynomial time algorithm for CNF-satisfiability using this construction.

For any F, $G = (V, E)$ is defined as follows: $V = \{\langle\sigma, i\rangle \mid \sigma \text{ is a literal in clause } C_i\}$; $E = \{(\langle\sigma, i\rangle, \langle\delta, j\rangle) \mid i \ne j \text{ and } \sigma \ne \bar{\delta}\}$. A sample construction is given in Example 11.11.

If F is satisfiable then there is a set of truth values for x_i, $1 \le i \le n$ such that each clause is true with this assignment. Thus, with this assignment there is at least one literal σ in each C_i such that σ is true. Let $S = \{\langle\sigma, i\rangle \mid \sigma \text{ is true in } C_i\}$ be a set containing exactly one $\langle\sigma, i\rangle$ for each i.

S forms a clique in G of size k. Similarly, if G has a clique $K = (V', E')$ of size at least k then let $S = \{\langle\sigma, i\rangle \mid \langle\sigma, i\rangle \in V'\}$. Clearly, $|S| = k$ as G has no clique of size more than k. Furthermore, if $S' = \{\sigma \mid \langle\sigma, i\rangle \in S \text{ for some } i\}$ then S' cannot contain both a literal δ and its complement $\bar{\delta}$ as there is no edge connecting $\langle\delta, i\rangle$ and $\langle\bar{\delta}, j\rangle$ in G. Hence by setting x_i = true if $x_i \in S'$ and x_i = false if $\bar{x}_i \in S'$ and choosing arbitrary truth values for variables not in S', we can satisfy all clauses in F. Hence, F is satisfiable iff G has a clique of size at least k. □

Example 11.11 Consider $F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$. The construction of Theorem 11.2 yields the graph:



$\langle x_1, 1\rangle$
$\langle \bar{x}_1, 2\rangle$
$\langle x_2, 1\rangle$
$\langle \bar{x}_2, 2\rangle$
$\langle x_3, 1\rangle$
$\langle \bar{x}_3, 2\rangle$

Figure 11.1 A sample graph and satisfiability

This graph contains six cliques of size two. Consider the clique with vertices $\{\langle x_1, 1\rangle, \langle \bar{x}_2, 2\rangle\}$. By setting x_1 = true and $\bar{x}_2$ = true (i.e. x_2 = false) F is satisfied. x_3 may be set either to true or false. □

Node Cover Decision Problem

A set $S \subseteq V$ is a *node cover* for a graph $G = (V, E)$ iff all edges in E are incident to at least one vertex in S. The size of the cover, $|S|$, is the number of vertices in S.

Example 11.12 Consider the graph:

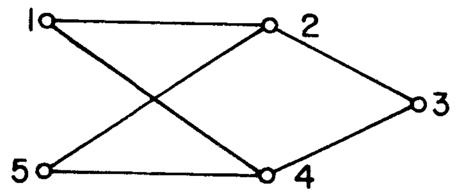


Figure 11.2 A sample graph and node cover

$S = \{2, 4\}$ is a node cover of size 2. $S = \{1, 3, 5\}$ is a node cover of size 3. □

In the node cover decision problem (NCDP) we are given a graph G and an integer k. We are required to determine if G has a node cover of size at most k.

Theorem 11.3 Clique decision problem (CDP) $\propto$ node cover decision problem (NCDP)

Proof: Let $G = (V, E)$ and k define an instance of CDP. Assume that $|V| = n$. We shall construct a graph G' such that G' has a node cover of size at most $n - k$ iff G has a clique of size at least k. Graph G' is given by $G' = (V, \bar{E})$ where $\bar{E} = \{(u, v) | u \in V, v \in V \text{ and } (u, v) \notin E\}$.

Now, we shall show that G has a clique of size at least k iff G' has a node cover of size at most $n - k$. Let K be any clique in G. Since there are no edges in $\bar{E}$ connecting vertices in K, the remaining $n - |K|$ vertices in G' must cover all edges in $\bar{E}$. Similarly, if S is a node cover of G' then $V - S$ must form a complete subgraph in G.

Since G' can be obtained from G in polynomial time, CDP can be solved in polynomial deterministic time if we have a polynomial time deterministic algorithm for NCDP. □

Note that since CNF-satisfiability $\propto$ CDP, CDP $\propto$ NCDP and $\propto$ is transitive, it follows that NCDP is NP-hard.

Chromatic Number Decision Problem (CN)

A coloring of a graph $G = (V, E)$ is a function $f: V \to \{1, 2, \ldots, k\}$ defined for all $i \in V$. If $(u, v) \in E$ then $f(u) \neq f(v)$. The *chromatic number decision problem* (CN) is to determine if G has a coloring for a given k.

Example 11.13 A possible 2-coloring of the graph of Figure 11.2 is: $f(1) = f(3) = f(5) = 1$ and $f(2) = f(4) = 2$. Clearly, this graph has no 1-coloring. □

In proving CN to be NP-hard we shall make use of the NP-hard problem SATY. This is the CNF staisfiability problem with the restriction that each clause has at most three literals. The reduction CNF-satisfiability $\propto$ SATY is left as an exercise.

Theorem 11.4 Satisfiability with at most three literals per clause (SATY) $\propto$ chromatic number (CN)

Proof: Let F be a CNF formula having at most three literals per clause and having r clauses. Let x_i, $1 \leq i \leq n$ be the n variables in F. We may assume $n \geq 4$. If $n < 4$ then we can determine if F is satisfiable by trying out all eight possible truth value assignments to x_1, x_2 and x_3. We shall construct, in polynomial time, a graph G that is $n + 1$ colorable iff F is satisfiable. The graph $G = (V, E)$ is defined by:

$$V = \{x_1, x_2, \ldots, x_n\} \cup \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \cup \{y_1, y_2, \ldots, y_n\} \cup \{C_1, C_2, \ldots, C_r\}$$

and

$$E = \{(x_1, \bar{x}_1), 1 \leq i \leq n\} \cup \{(y_i, y_j) | i \neq j\} \cup \{(y_i, x_j) | i \neq j\} \cup \{(y_i, \bar{x}_j) | i \neq j\} \cup \{(x_i, C_j) | x_i \notin C_j\} \cup \{\bar{x}_i, C_j) | \bar{x}_i \notin C_j\}$$

To see that G is $n + 1$ colorable iff F is satisfiable, we first observe that the y_i's form a complete subgraph on n vertices. Hence, each y_i must be assigned a distinct color. Without loss of generality we may assume that in any coloring of G y_i is given the color i. Since y_i is also connected to all the x_j's and $\bar{x}_j$'s except x_i and $\bar{x}_i$, the color i can be assigned to only x_i and $\bar{x}_i$. However $(x_i, \bar{x}_i) \in E$ and so a new color $n + 1$, is needed for one of these vertices. The vertex that is assigned the new color, $n + 1$, will be called the *false vertex*. The other vertex is a *true* vertex. The only way to color G using $n + 1$ colors is to assign color $n + 1$ to one of $\{x_i, \bar{x}_i\}$ for each i, $1 \leq i \leq n$.

Under what conditions can the remaining vertices be colored using no new colors? Since $n \geq 4$ and each clause has at most three literals, each C_i is adjacent to a pair of vertices x_j, $\bar{x}_j$ for at least one j. Consequently, no C_i may be assigned the color $n + 1$. Also, no C_i may be assigned a color corresponding to an x_j or $\bar{x}_j$ not in clause C_i. The last two statements imply that the only colors that can be assigned to C_i correspond to vertices x_j or $\bar{x}_j$ that are in clause C_i and are true vertices. Hence, G is $n + 1$ colorable iff there is a true vertex corresponding to each C_i. So, G is n + 1 colorable iff F is satisfiable. □

Chapter 12

APPROXIMATION ALGORITHMS FOR NP-HARD PROBLEMS

12.1 INTRODUCTION

In the preceding chapter we saw strong evidence to support the claim that no NP-hard problem can be solved in polynomial time. Yet, many NP-hard optimization problems have great practical importance and it is desirable to solve large instances of these problems in a "reasonable" amount of time. The best known algorithms for NP-hard problems have a worst case complexity that is exponential in the number of inputs. While the results of the last chapter may favor abandoning the quest for polynomial time algorithms, there is still plenty of room for improvement in an exponential algorithm. We may look for algorithms with subexponential complexity, say $2^{n/c}$ (for $c > 1$), $2^{\sqrt{n}}$ or $n^{\log n}$. In the exercises of Chapter 5 an $O(2^{n/2})$ algorithm for the knapsack problem was developed. This algorithm can also be used for the partition, sum of subsets and exact cover problem. Tarjan and Trojanowski ("Finding a maximum independent set," *SIAM Computing,* 6(3), pp. 537–546, 1977.) have obtained an $O(2^{n/3})$ algorithm for the max-clique, max-independent set and minimum node cover problems. The discovery of a subexponential algorithm for an NP-hard problem increases the maximum problem size that can actually be solved. However, for large problem instances, even an $O(n^4)$ algorithm requires too much computational effort. Clearly, what is needed is an algorithm of low polynomial complexity (say $O(n)$ or $O(n^2)$).

The use of heuristics in an existing algorithm may enable it to quickly solve a large instance of a problem provided the heuristic "works" on that instance. This was clearly demonstrated in the chapters on bactracking and branch-and-bound. A heuristic, however, does not "work" equally effectively on all problem instances. Exponential time algorithms, even coupled with heuristics will still show exponential behavior on some set of inputs.

If we are to produce an algorithm of low polynomial complexity to solve an NP-hard optimization problem, then it will be necessary to relax the meaning of solve. In this chapter we shall discuss two relaxations of the meaning of solve. In the first we shall remove the requirement that the algorithm that solves the optimization problem P must always generate an optimal solution. This requirement will be replaced by the requirement that the algorithm for P must always generate a feasible solution with value "close" to the value of an optimal solution. A feasible solution with value close to the value of an optimal solution is called an *approximate solution.* An *approximation algorithm* for P is an algorithm that generates approximate solutions for P.

While at first one may discount the virtue of an approximate solution, one should bear in mind that often, the data for the problem instance being solved is only known approximately. Hence, an approximate solution (provided its value is "sufficiently" close to that of an exact solution) may be no less meaningful than an exact solution. In the case of NP-hard problems approximate solutions have added importance as it may be true that exact solutions (i.e. optimal solutions) cannot be obtained in a feasible amount of computing time. An approximate solution may be all one can get using a reasonable amount of computing time.

In the second relaxation we shall look for an algorithm for P that *almost always* generates optimal solutions. Algorithms with this property are called *probabilistically good* algorithms. These are considered in Section 12.6. In the remainder of this section we develop the terminology to be used in discussing approximation algorithms.

Let P be a problem such as the knapsack or the traveling salesperson problem. Let I be an instance of problem P and let $F^*(I)$ be the value of an optimal solution to I. An approximation algorithm will in general produce a feasible solution to I whose value $\hat{F}(I)$ is less than (greater than) $F^*(I)$ in case P is a maximization (minimization) problem. Several categories of approximation algorithms may be defined.

Let $\mathcal{A}$ be an algorithm which generates a feasible solution to every instance I of a problem P. Let $F^*(I)$ be the value of an optimal solution to I and let $\hat{F}(I)$ be the value of the feasible solution generated by $\mathcal{A}$.

Definition $\mathcal{A}$ is an *absolute approximation* algorithm for problem P if and only if for every instance I of P, $|F^*(I) - \hat{F}(I) \leq k$ for some constant k.

Definition $\mathcal{A}$ is an *f(n)-approximate* algorithm if and only if for every instance I of size n, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq f(n)$. It is assumed that $F^*(I) > 0$.

Definition An *ϵ-approximate* algorithm is an $f(n)$-approximate algorithm for which $f(n) \leq \epsilon$ for some constant ϵ.

Note that for a maximization problem, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1$ for every feasible solution to I. Hence, for maximization problems we will normally require $\epsilon < 1$ for an algorithm to be judged ϵ-approximate. In the next few definitions we consider algorithms $\mathcal{A}(\epsilon)$ with ϵ an input to $\mathcal{A}$.

Definition $\mathcal{A}(\epsilon)$ is an *approximation scheme* iff for every given $\epsilon > 0$ and problem instance I, $\mathcal{A}(\epsilon)$ generates a feasible solution such that $|F^*(I) - \hat{F}(I)|/F^*(I) \leq \epsilon$. Again, we assume $F^*(I) > 0$.

Definition An approximation scheme is a *polynomial time approximation scheme* iff for every fixed $\epsilon > 0$ it has a computing time that is polynomial in the problem size.

Definition An approximation scheme whose computing time is a polynomial both in the problem size and in $1/\epsilon$ is a *fully polynomial time approximation scheme.*

Clearly, the most desirable kind of approximation algorithm is an absolute approximation algorithm. Unfortunately, for most NP-hard problems it can be shown that fast algorithms of this type exist only if P = NP. Surprisingly, this statement is true even for the existence of $f(n)$-approximate algorithms for certain NP-hard problems.

Example 12.1 Consider the knapsack instance $n = 3$, $M = 100$, $\{p_1, p_2, p_3\} = \{20, 10, 19\}$ and $\{w_1, w_2, w_3\} = \{65, 20, 35\}$. $(x_1, x_2, x_3) = (1, 1, 1)$ is not a feasible solution as $\Sigma\, w_i x_i > M$. The solution $(x_1, x_2, x_3) = (1, 0, 1)$ is an optimal solution. Its value $\Sigma\, p_i x_i$ is 39. Hence, $F^*(I) = 39$ for this instance. The solution $(x_1, x_2, x_3) = (1, 1, 0)$ is suboptimal. Its value is $\Sigma\, p_i x_i = 30$. This is a candidate for a possible output from an approximation algorithm. In fact, every feasible solution (in this case all three element 0/1 vectors other than (1, 1, 1) are feasible) is a candidate for output by an approximation algorithm. If the solution (1, 1, 0) is generated by an approximation algorithm on this instance then $\hat{F}(I) = 30$. $|F^*(I) - \hat{F}(I)| = 9$ and $|F^*(I) - \hat{F}(I)|/F^*(I) = 0.3$. □

Example 12.2 Consider the following approximation algorithm for the 0/1 knapsack problem: consider the objects in nonincreasing order of p_i/w_i. If object i fits then set $x_i = 1$ otherwise set $x_i = 0$. When this algorithm is used on the instance of Example 12.1, the objects are considered in the order 1, 3, 2. The result is $(x_1, x_2, x_3) = (1, 0, 1)$. The optimal solution is obtained. Now, consider the following instance: $n = 2$, $(p_1, p_2) = (2, r)$, $(w_1, w_2) = (1, r)$ and $M = r$. When $r > 1$, the optimal solution is $(x_1, x_2) = (0, 1)$. Its value, $F^*(I)$, is r. The solution generated by the approximation algorithm is $(x_1, x_2) = (1, 0)$. Its value, $\hat{F}(I)$, is 2. Hence, $|F^*(I) - \hat{F}(I)| = r - 2$. Our approximation algorithm is not an absolute approximation algorithm as there exists no constant k such that $|F^*(I) - \hat{F}(I)| \leq k$ for all instances I. Furthermore, note that $|F^*(I) - \hat{F}(I)|/F^*(I) = 1 - 2/r$. This approaches 1 as r becomes large. $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1$ for every feasible solution to every knapsack instance. Since the above algorithm always generates a feasible solution it is a 1-approximate algorithm. It is, however, not an ϵ-approximate algorithm for any ϵ, $\epsilon < 1$. □

Corresponding to the notions of absolute approximation algorithm and $f(n)$-approximate algorithm, we may define approximation problems in the obvious way. So, we can speak of k-absolute approximate problems and $f(n)$-approximate problems. The .5-approximate knapsack problem is to find any 0/1 feasible solution with $|F^*(I) - \hat{F}(I)|/F^*(I) \leq .5$.

As we shall see, approximation algorithms are usually just heuristics or rules that on the surface look like they might solve the optimization problem exactly. However, they do not. Instead, they only guarantee to generate feasible solutions with value within some constant or some factor of the optimal value. Being heuristic in nature, these algorithms are very much dependent on the individual problem being solved.

12.2 ABSOLUTE APPROXIMATIONS

Planar Graph Coloring

There are very few NP-hard optimization problems for which polynomial time absolute approximation algorithms are known. One problem is that of determining the minimum number of colors needed to color a planar graph $G = (V, E)$. It is known that every planar graph is four colorable. One may easily determine if a graph is 0, 1 or 2 colorable. It is zero colorable iff $V = \phi$. It is 1 colorable iff $E = \phi$. G is two colorable iff it is bi partite (see Exercise 6.41). Determining if a planar graph is three colorable

is NP-hard. However, all planar graphs are four colorable. An absolute approximation algorithm with $|F^*(I) - \hat{F}(I)| \leq 1$ is easy to obtain. Algorithm 12.1 is such an algorithm. It finds an exact answer when the graph can be colored using at most two colors. Since we can determine whether or not a graph is bipartite in time $O(|V| + |E|)$, the complexity of the algorithm is $O(|V| + |E|)$.

```
procedure ACOLOR(V, E)
  //determine an approximation to the minimum number of colors//
  //needed to color the planar graph G = (V, E)//

    case
      : V = φ: return (0)
      : E = φ: return (1)
      : G is bipartite: return (2)
      : else: return (4)
    endcase
end ACOLOR
```

Algorithm 12.1 Approximate coloring

Maximum Programs Stored Problem

Assume that we have n programs and two storage devices (say disks or tapes). We shall assume the devices are disks. Our discussion applies to any kind of storage device. Let l_i be the amount of storage needed to store the ith program. Let L be the storage capacity of each disk. Determining the maximum number of these n programs that can be stored on the two disks (without splitting a program over the disks) is NP-hard.

Theorem 12.1 Partition α Maximum Programs Stored.

Proof: Let $\{a_1, a_2, \ldots, a_n\}$ define an instance of the partition problem. We may assume $\Sigma\ a_i = 2T$. Define an instance of the maximum programs stored problem as follows: $L = T$ and $l_i = a_i$, $1 \leq i \leq n$. Clearly, $\{a_i, \ldots, a_n\}$ has a partition iff all n programs can be stored on the two disks. □

By considering programs in order of nondecreasing storage requirement l_i, we can obtain a polynomial time absolute approximation algorithm. Procedure PSTORE assumes $l_1 \leq l_2 \leq \cdots \leq l_n$ and assigns programs to disk 1 so long as enough space remains on this tape. Then it begins assigning programs to disk 2. In addition to the time needed to initially sort the programs into nondecreasing order of l_i, $O(n)$ time is needed to obtain the storage assignment.

```
procedure PSTORE(l, n, L)
  //assume $l_i \le l_{i+1}, 1 \le i < n$//
   $i \leftarrow 1$
   for $j \leftarrow 1$ to 2 do
     $sum \leftarrow 0$  //amount of disk $j$ already assigned//
     while $sum + l_i \le L$ do
       print ('store program', i, 'on disk', j)
       $sum \leftarrow sum + l_i$
       $i \leftarrow i + 1$
       if $i > n$ then return endif
     repeat
   repeat
end PSTORE
```

Algorithm 12.2 Approximation algorithm to store programs

Example 12.3 Let $L = 10$, $n = 4$ and $(l_1, l_2, l_3, l_4) = (2, 4, 5, 6)$. Procedure PSTORE will store programs 1 and 2 on disk 1 and only program 3 on disk 2. An optimal storage scheme stores all four programs. One way to do this is to store programs 1 and 4 on disk 1 and the other two on disk 2. □

Theorem 12.2 Let I be any instance of the maximum programs stored problem. Let $F^*(I)$ be the maximum number of programs that can be stored on two disks of length L each. Let $\hat{F}(I)$ be the number of programs stored using procedure PSTORE. Then, $|F^*(I) - \hat{F}(I)| \le 1$.

Proof: Assume that k programs are stored when Algorithm 12.2 is used. Then, $\hat{F}(I) = k$. Consider the program storage problem when only one disk of capacity $2L$ is available. In this case, considering programs in order of nondecreasing storage requirement maximizes the number of programs stored. Assume that p programs get stored when this strategy is used on a single disk of length $2L$. Clearly, $p \ge F^*(I)$ and $\Sigma_1^p l_i \le 2L$. Let j be the largest index such that $\Sigma_1^j l_i \le L$. It is easy to verify that $j \le p$ and that PSTORE assigns the first j programs to disk 1. Also,

$$\sum_{i=j+1}^{p-1} l_i \le \sum_{i=j+2}^{p} l_i \le L.$$

Hence, PSTORE assigns at least programs $j + 1, j + 2, \ldots, p - 1$ to disk 2. So, $\hat{F}(I) \ge p - 1$ and $|F^*(I) - \hat{F}(I)| \le 1$. □

Algorithm PSTORE may be extended in the obvious way to obtain a $k - 1$ absolute approximation algorithm for the case of k disks.

NP-hard Absolute Approximations

The absolute approximation algorithms for the planar graph coloring and the maximum program storage problems are very simple and straightforward. Thus, one may expect that polynomial time absolute approximation algorithms exist for most other NP-hard problems. Unfortunately, for the majority of NP-hard problems one can provide very simple proofs to show that a polynomial time absolute approximation algorithm exists iff a polynomial time exact algorithm does. Let us look at some sample proofs.

Theorem 12.3 The absolute approximate knapsack problem is NP-hard.

Proof: We shall show that the 0/1 knapsack problem with integer profits reduces to the absolute approximate knapsack problem. The theorem then follows from the observation that the knapsack problem with integer profits is NP-hard. Assume there is a polynomial time algorithm $\mathcal{A}$ that guarantees feasible solutions such that $|F^*(I) - \hat{F}(I)| \le k$ for every instance I and a fixed k. Let (p_i, w_i), $1 \le i \le n$ and M define an instance of the knapsack problem. Assume the p_i are integer. Let I' be the instance defined by $((k + 1)p_i, w_i)$, $1 \le i \le n$ and M. Clearly, I and I' have the same set of feasible solutions. Further, $F^*(I') = (k + 1)F^*(I)$ and I and I' have the same optimal solutions. Also, since all the p_i are integer, it follows that all feasible solutions to I' either have value $F^*(I')$ or have value at most $F^*(I') - (k + 1)$. If $\hat{F}(I')$ is the value of the solution generated by $\mathcal{A}$ for instance I' then $F^*(I') - \hat{F}(I')$ is either 0 or at least $k + 1$. Hence if $F^*(I') - \hat{F}(I') \le k$ then $F^*(I') = \hat{F}(I')$. So, $\mathcal{A}$ can be used to obtain an optimal solution for I' and hence I. Since the length of I' is at most (log k)*(length of I), it follows that using the above construction we can obtain a polynomial time algorithm for the knapsack problem with integer profits. □

Example 12.4 Consider the knapsack instance $n = 3$, $M = 100$, $(p_1, p_2,$

$p_3) = (1, 2, 3)$ and $(w_1, w_2, w_3) = (50, 60, 30)$. The feasible solutions are (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 0, 1) and (0, 1, 1). The values of these solutions are 1, 2, 3, 4 and 5 respectively. If we multiply the p's by 5 then $(\hat{p}_1, \hat{p}_2, \hat{p}_3) = (5, 10, 15)$. The feasible solutions are unchanged. Their values are now 5, 10, 15, 20 and 25 respectively. If we had an absolute approximation algorithm for $k = 4$ then, this algorithm will have to output the solution (0, 1, 1) as no other solution is within 4 of the optimal solution value. □

Now, consider the problem of obtaining a maximum clique of an undirected graph. The following theorem shows that obtaining a polynomial time absolute approximation algorithm for this problem is as hard as obtaining a polynomial time algorithm for the exact problem.

Theorem 12.4 Max clique $\propto$ absolute approximation max clique.

Proof: Assume that the algorithm for the absolute approximation problem finds solutions such that $|F^*(I) - \hat{F}(I)| \le k$. From any given graph $G = (V, E)$, we construct another graph $G' = (V', E')$ such that G' consists of $k + 1$ copies of G connected together such that there is an edge between every two vertices in distinct copies of G. I.e., if $V = \{v_1, v_2, \ldots, v_n\}$ then

$$V' = \bigcup_{i=1}^{k+1} \{v_1^i, v_2^i, \ldots, v_n^i\}$$

and

$$E' = \left(\bigcup_{i=1}^{k+1} \{(v_p^i, v_r^i) \mid (v_p, v_r) \in E\} \right) \cup \{(v_p^i, v_r^j) \mid i \ne j\}.$$

Clearly, the maximum clique size in G is q iff the maximum clique size in G' is $(k + 1)q$. Further, any clique in G' which is within k of the optimal clique size in G' must contain a sub-clique of size q which is a clique of size q in G. Hence, we can obtain a maximum clique for G from a k-absolute approximate maximum clique for G'. □

Example 12.5 Figure 12.1(b) shows the graph G' that results when the construction of Theorem 12.4 is applied to the graph of Figure 12.1(a). We have assumed $k = 1$. The graph of Figure 12.1(a) has two cliques.

One consists of the vertex set {1, 2} and the other {2, 3, 4}. Thus, an absolute approximation algorithm for $k = 1$ could output either of the two as solution cliques. In the graph of Figure 12.1(b), however, the two cliques are {1, 2, 1′, 2′} and {2, 3, 4, 2′, 3′, 4′}. Only the latter may be output. Hence, an absolute approximation algorithm with $k = 1$ will output the maximum clique. □

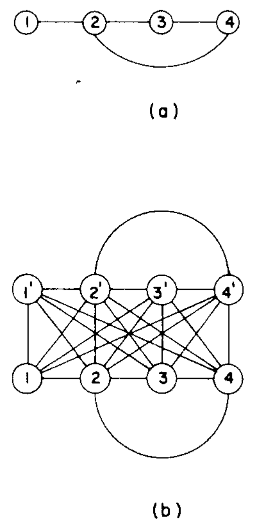


Figure 12.1 Graphs for Example 12.5

12.3 ϵ-APPROXIMATIONS

Scheduling Independent Tasks

Obtaining minimum finish time schedules on m, $m > 2$ identical processors is NP-hard. There exists a very simple scheduling rule that generates schedules with a finish time very close to that of an optimal schedule. An instance I of the scheduling problem is defined by a set of n task times, t_i, $1 \le i \le n$, and m, the number of processors. The scheduling rule we are about to describe is known as the LPT (longest processing time) rule. An LPT schedule is a schedule that results from this rule.

Definition An *LPT schedule* is one that is the result of an algorithm which, whenever a processor becomes free, assigns to that processor a task whose time is the largest of those tasks not yet assigned. Ties are broken in an arbitrary manner.

Example 12.6 Let $m = 3$, $n = 6$ and $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 4, 3)$. In an LPT schedule tasks 1, 2 and 3 are assigned to processors 1, 2 and 3 respectively. Tasks 4, 5 and 6 are respectively assigned to processors 3, 2 and 1. Figure 12.2 shows this LPT schedule. The finish time is 11. Since, $\Sigma\ t_i/3 = 11$, the schedule is also optimal.

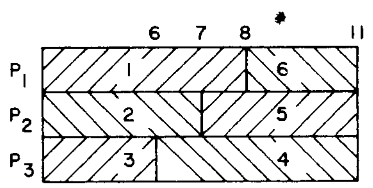


Figure 12.2 LPT schedule for Example 12.6

Example 12.7 Let $m = 3$, $n = 7$ and $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (5, 5, 4, 4, 3, 3, 3)$. Figure 12.3(a) shows the LPT schedule. This has a finish time of 11. Figure 12.3(b) shows an optimal schedule. Its finish time is 9. Hence, for this instance $|F^*(I) - \hat{F}(I)|/F^*(I) = (11 - 9)/9 = 2/9$. □

It is possible to implement the LPT rule so that at most 0(n log n) time is needed to generate an LPT schedule for n tasks on m processors. An exercise examines this. The preceding examples show that while the LPT rule may generate optimal schedules for some problem instances, it does not do so for all instances. How bad can LPT schedules be relative to optimal schedules? This question is answered by the following theorem.

Theorem 12.5 [Graham] Let $F^*(I)$ be the finish time of an optimal m processor schedule for instance I of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then,

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \le \frac{1}{3} - \frac{1}{3m}$$

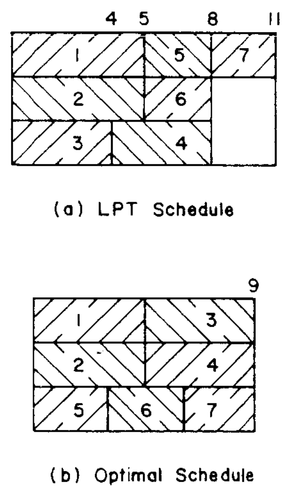


Figure 12.3 LPT and optimal schedules for Example 12.7

Proof: The theorem is clearly true for $m = 1$. So, assume $m \ge 2$. Assume that for some m, $m > 1$, there exists a set of tasks for which the theorem is not true. Then, let $(t_1, t_2, \ldots, t_n)$ define an instance I with the fewest number of tasks for which the theorem is violated. We may assume $t_1 \ge t_2 \ge \cdots \ge t_n$ and that an LPT schedule is obtained by assigning tasks in the order $1, 2, 3, \ldots, n$.

Let S be the LPT schedule obtained by assigning these n tasks in this order. Let $\hat{F}(I)$ be its finish time. Let k be the index of a task with latest completion time. Then, $k = n$. To see this, suppose $k < n$. Then, the finish time $\hat{f}$ of the LPT schedule for tasks $1, 2, \ldots, k$ is also $\hat{F}(I)$. The finish time, f^*, of an optimal schedule for these k tasks is no more than $F^*(I)$. Hence, $|f^* - \hat{f}|/f^* \ge |F^*(I) - \hat{F}(I)|/F^*(I) > 1/3 - 1/(3m)$. (The latter inequality follows from the assumption on I.) $|f^* - \hat{f}|/f^* > 1/3 - 1/(3m)$ contradicts the assumption that I is the smallest m processor instance for which the theorem does not hold. Hence, $k = n$.

Now, we show that in no optimal schedule for I can more than two tasks be assigned to any processor. Hence, $n \le 2m$. Since task n has the latest completion time in the LPT schedule for I, it follows that this task is started at time $\hat{F}(I) - t_n$ in this schedule. Further, no processor can have any idle time until this time. Hence, we obtain:

$$\hat{F}(I) - t_n \le \frac{1}{m}\sum_{1}^{n-1} t_i$$

So,

$$\hat{F}(I) \le \frac{1}{m}\sum_{1}^{n} t_i + \frac{m-1}{m} t_n.$$

Since,

$$F^*(I) \ge \frac{1}{m}\sum_{1}^{n} t_i,$$

we can conclude that

$$\hat{F}(I) - F^*(I) \le \frac{m-1}{m} t_n$$

or

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \le \frac{m-1}{m}\frac{t_n}{F^*(I)}$$

But, from the assumption on I, the left hand side of the above inequality is greater than $1/3 - 1/(3m)$. So,

$$\frac{1}{3} - \frac{1}{3m} < \frac{m-1}{m}\frac{t_n}{F^*(I)}$$

or

$$m - 1 < 3(m-1)t_n/F^*(I)$$

or

$$F^*(I) < 3t_n.$$

Hence, in an optimal schedule for I, no more than two tasks can be assigned to any processor. When the optimal schedule contains at most two tasks on any processor then it may be shown that the LPT schedule is also optimal. We leave this part of the proof as an exercise. Hence, $|F^*(I) - \hat{F}(I)|/F^*(I) = 0$ for this case. This contradicts the assumption on I. So, there can be no I that violates the theorem. □

Theorem 12.5 establishes the LPT rule as a $(1/3 - 1/(3m))$-approximate rule for task scheduling. As remarked earlier, this rule can be implemented to have complexity $O(n \log n)$. The following example shows that $1/3 - 1/(3m)$ is a tight bound on the worst case performance of the LPT rule.

Example 12.8 Let $n = 2m + 1$, $t_i = 2m - \lfloor (i+1)/2 \rfloor$, $i = 1, 2, \ldots, 2m$ and $t_{2m+1} = m$. Figure 12.4(a) shows the LPT schedule. This has a finish time of $4m - 1$. Figure 12.4(b) shows an optimal schedule. Its finish time is $3m$. Hence, $|F^*(I) - \hat{F}(I)|/F^*(I) = 1/3 - 1/(3m)$. □

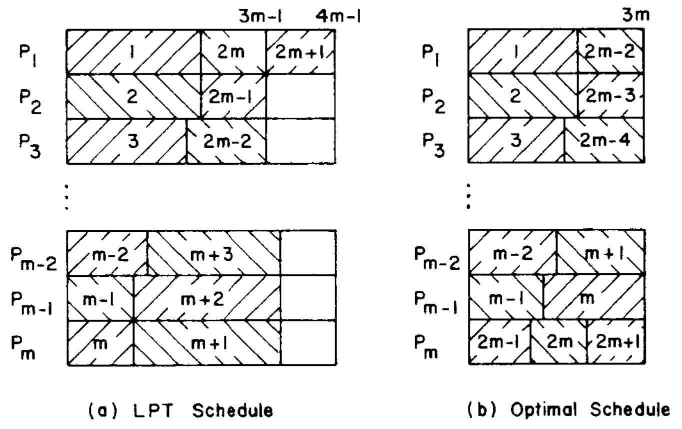


Figure 12.4 Schedules for Example 12.8

For LPT schedules, the worst case error bound of $1/3 - 1/(3m)$ is not very indicative of the expected closeness of LPT finish times to optimal finish times. When $m = 10$, the worst case error bound is .3. Two experiments were conducted ("An application of bin-packing to multiprocessor scheduleing," by E. Coffman, M. Garey and D. Johnson, *SIAM Computing*, 7(1), pp. 1-17, 1978.) to see what kind of error one might expect on a random problem for $m = 10$. In the first experiment, 30 tasks with task times chosen according to a uniform distribution between 0 and 1 were generated. $F^*(I)$ was estimated to be $\Sigma_1^{30} t_i/10$ and $\hat{F}(I)$ was the length of the LPT schedule generated. The experiment was repeated ten times and the average value of $|F^*(I) - \hat{F}(I)|/F^*(I)$ computed. This value was 0.074. In the second experiment task times were chosen according to a normal distribution. The average $|F^*(I) - \hat{F}(I)|/F^*(I)$ was 0.023 this time. These figures are probably a little inflated as $\Sigma_1^{30} t_i/10$ is probably an underestimation of the true $F^*(I)$.

Efficient ϵ-approximate algorithms exist for many scheduling problems. The references at the end of this chapter point to some of the better known ϵ-approximate scheduling algorithms. Some of these algorithms are also discussed in the exercises.

Bin Packing

In this problem we are given n objects which have to be placed in bins of equal capacity L. Object i requires l_i units of bin capacity. The objective is to determine the minimum number of bins needed to accommodate all n objects. No object may be placed partly in one bin and partly in another.

Example 12.9 Let $L = 10$, $n = 6$ and $(l_1, l_2, l_3, l_4, l_5, l_6) = (5, 6, 3, 7, 5, 4)$. Figure 12.5 shows a packing of the 6 objects using only three bins. Numbers in bins are object indices. It is easy to see that at least 3 bins are needed.

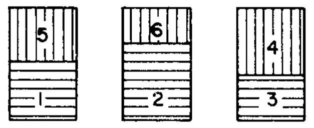


Figure 12.5 Optimal packing for Example 12.9

The bin packing problem may be regarded as a variation of the scheduling problem considered earlier. The bins represent processors and L is the time by which all tasks must be completed. l_1 is the processing requirement of task i. The problem is to determine the minimum number of processors needed to accomplish this. An alternative interpretation is to regard the bins as tapes. L is the length of a tape and l_i the tape length needed to store program i. The problem is to determine the minimum number of tapes needed to store all n programs. Clearly, many interpretations exist for this problem.

Theorem 12.6 The bin packing problem is NP-hard.

Proof: To see this consider the partition problem. Let $\{a_1, a_2, \ldots, a_n\}$ be an instance of the partition problem. Define an instance of the bin packing problem as follows: $l_i = a_i$, $1 \le i \le n$ and $L = \Sigma a_i/2$. Clearly, the minimum number of bins needed is 2 iff there is a partition for $\{a_1, a_2, \ldots, a_n\}$. □

One can devise many simple heuristics for the bin packing problem. These will not, in general, obtain optimal packings. They will, however, obtain packings that use only a "small" fraction of bins more than an optimal packing. Four simple heuristics are:

I. *First Fit* (FF)

Index the bins 1, 2, 3, ... All bins are initially filled to level zero. Objects are considered for packing in the order $1, 2, \ldots, n$. To pack object i, find the least index j such that bin j is filled to a level r, $r \le L - l_i$. Pack i into bin j. Bin j is now filled to level $r + l_i$.

II. *Best Fit* (BF)

The initial conditions on the bins and objects are the same as for FF. When object i is being considered, find the least j such that bin j is filled to a level r, $r \le L - l_i$ and r is as large as possible. Pack i into bin j. Bin j is now filled to level $r + l_i$.

III. *First Fit Decreasing* (FFD)

Reorder the objects so that $l_i \ge l_{i+1}$, $1 \le i < n$. Now use First Fit to pack the objects.

IV. *Best Fit Decreasing* (BFD)

Reorder the objects so that $l_i \ge l_{i+1}$, $1 \le i < n$. Now use Best Fit to pack the objects.

Example 12.10 Consider the problem instance of Example 12.9. Figure 12.6 shows the packings resulting when each of the above four packing rules is used. For FFD and BFD the six objects are considered in the order

(4, 2, 1, 5, 6, 3). As is evident from the figure, FFD and BFD do better than either FF or BF on this instance. While FFD and BFD obtain optimal packings on this instance, they do not in general obtain such a packing. □

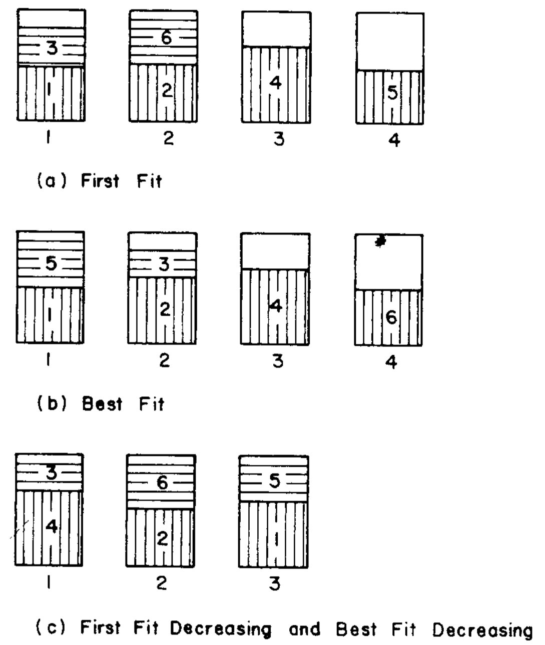


Figure 12.6 Packings resulting from the four heuristics

Theorem 12.7 Let I be an instance of the bin packing problem and let $F^*(I)$ be the minimum number of bins needed for this instance. The packing generated by either FF or BF uses no more than $(17/10)\ F^*(I) + 2$ bins. The packing generated by either FFD or BFD uses no more than $(11/9)\ F^*(I) + 4$ bins. These bounds are the best possible bounds for the respective algorithms.

Proof: The proof of this theorem is rather long and complex. It may be found in the paper: "Worst-Case Performance Bounds For Simple One-Dimensional Packing Algorithms," by Johnson, Demers, Ullman, Garey and Graham, *SIAM Jr. On Computing*, 3(4), pp. 299-325 (1974). □

NP-hard ϵ-Approximation Problems

As in the case of absolute approximations, there exist many NP-hard optimization problems for which the corresponding ϵ-approximation problems are also NP-hard. Let us look at some of these. To begin, consider the traveling salesperson problem.

Theorem 12.8 Hamiltonian cycle $\propto$ ϵ-approximate traveling salesperson.

Proof: Let $G(N,A)$ be any graph. Construct the complete graph $G_1(V, E)$ such that $V = N$ and $E = \{(u, v) | u, v \in V \text{ and } u \neq v\}$. Define the edge weighting function w to be

$$w(u, v) = \begin{cases} 1 & \text{if } (u, v) \in A \\ k & \text{otherwise} \end{cases}$$

Let $n = |N|$. For $k > 1$, the traveling salesperson problem on G_1 has a solution of length n if and only if G has a Hamiltonian cycle. Otherwise, all solutions to G_1 have length $\geq k + n - 1$. If we choose $k \geq (1 + \epsilon)n$, then the only solutions approximating a solution with value n (if there was a Hamiltonian cycle in G_1) also have length n. Consequently, if the ϵ-approximate solution has length $\leq (1 + \epsilon)n$ then it must be of length n. If it has length $>(1 + \epsilon)n$ then G has no Hamiltonian cycle. □

Another NP-hard ϵ-approximation problem is the 0/1 integer programming problem. In the optimization version of this problem we are provided with a linear optimization function $f(x) = \Sigma\, p_i x_i + p_0$. We are required to find a 0/1 vector $(x_1, x_2, \ldots, x_n)$ such that $f(x)$ is optimized (either maximized or minimized) subject to the constraints that $\Sigma\, a_{ij} x_j \leq b_i$, $1 \leq i \leq k$. k is the number of constraints. Note that the 0/1-knapsack problem is a special case of the 0/1 integer programming problem just described. Hence, the integer programming problem is also NP-hard. We shall now show that the corresponding ϵ-approximation problem is NP-hard for all ϵ, $\epsilon > 0$. This is true even when there is only one constraint (i.e., $k = 1$).