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 phenomenon 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 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*. 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 $A(i)$, $1 \leq i \leq n$ 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 $A(j)$, $1 \leq j \leq n$ 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 $X(i)$, $1 \leq i \leq n$. 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
  for i ← 1 to n do
    X(i) ← choice (0, 1)
  repeat
  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
  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 no 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 some assignment of truth values to the variables. *CFN-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 ← 1 to n do     //choose a truth value assignment//
      $x_i$ ← 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 $P \overset{?}{=} NP$ 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 $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$).

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, ..., $p(n)$. 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, ..., $2p(n)$ and solves the same decision problem as does A. A' simulates the behavior of A. However, A' maps the addresses $f(1)$, $f(2)$, ..., $f(k)$ 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, exactly 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

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

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, $2 \le i \le l$. 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. $E = \bigwedge_{1 < t \le p(n)} E_t$.

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

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

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

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

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 $1 \le j \le w$.

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

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

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

$G_{i,t}$ 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 $Y \leftarrow V$; $Y \leftarrow V \textcircled{op} Z$ for $\textcircled{op}$ 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

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

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

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

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

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 $a_l = B(y, l, t)$ if bit l is 1 in S_l and $a_i = \bar{B}(y, l, t)$ if bit l is zero in S_l. If S is of the form $r{:}u$ then M is just the formula that asserts $r \le Y \le u$. 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 $O(l^2p(n))$ 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 $\propto$ 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: $V = \{\langle \sigma, i \rangle \mid \sigma$ is a literal in clause $C_i\}$; $E = \{(\langle \sigma, i \rangle, \langle \delta, j \rangle) \mid i \neq j$ and $\sigma \neq \bar{\delta}\}$. 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 $S = \{\langle \sigma, i \rangle \mid \sigma$ 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 $S' = \{\sigma \mid \langle \sigma, i \rangle \in S$ 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:

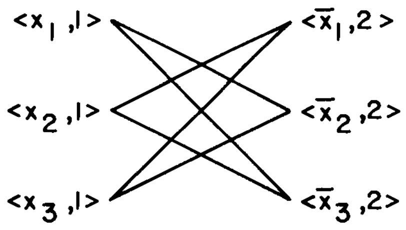


Figure 11.1 A sample graph and satisfiability

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:

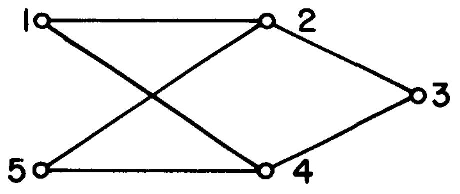


Figure 11.2 A sample graph and node cover

$S = \{2, 4\}$ 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 $G' = (V, \overline{E})$ where $\overline{E} = \{(u, v) | u \in V, v \in V \text{ and } (u, v) \notin E\}$.

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 $\overline{E}$ connecting vertices in K, the remaining $n - |K|$ vertices in G' must cover all edges in $\overline{E}$. 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 $f: V \rightarrow \{1, 2, \ldots, k\}$ 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 satisfiability 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. $\square$

Directed Hamiltonian Cycle (DHC)

A directed Hamiltonian cycle in a directed graph $G = (V, E)$ is a directed cycle of length $n = |V|$. So, the cycle goes through every vertex exactly once and then returns to the starting vertex. The DHC problem is to determine if G has a directed Hamiltonian cycle.

Example 11.14 1, 2, 3, 4, 5, 1 is a directed Hamiltonian cycle in the graph of Figure 11.3.

If the edge $\langle 5, 1\rangle$ is deleted from this graph then it has no directed Hamiltonian cycle.

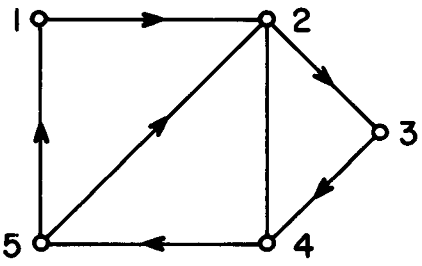


Figure 11.3 A sample graph and Hamiltonian cycle

Theorem 11.5 CNF-satisfiability $\propto$ directed hamiltonian cycle (DHC).
Proof: Let F be a propositional formula in CNF. We shall show how to construct a directed graph G such that F is satisfiable iff G has a directed Hamiltonian cycle. Since this construction can be carried out in time polynomial in the size of F, it will follow that CNF-satisfiability $\propto$ DHC. Understanding of the construction of G is greatly facilitated by the use of an example. The example we shall use is $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ where

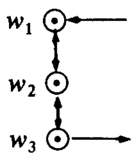
$$\begin{aligned} C_1 &= x_1 \vee \bar{x}_2 \vee x_4 \vee \bar{x}_5 \\ C_2 &= \bar{x}_1 \vee x_2 \vee x_3 \\ C_3 &= \bar{x}_1 \vee \bar{x}_3 \vee x_5 \\ C_4 &= \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4 \vee \bar{x}_5 \end{aligned}$$

Assume that F has r clauses $C_1, C_2, \ldots, C_r$ and n variables $x_1, x_2, \ldots, x_n$. Draw an array with r rows and $2n$ columns. Row i will denote clause C_i. Each variable x_i will be represented by two adjacent columns, one for each of the literals x_i and $\bar{x}_i$. Figure 11.4 shows the array for the example formula. Insert a $\odot$ into column x_i and row C_j iff x_i is a literal in C_j. Insert a $\odot$ into column $\bar{x}_i$ and row C_j iff $\bar{x}_i$ is a literal in C_j. Between each pair of columns x_i and $\bar{x}_i$ introduce two vertices u_i and v_i: u_i at the top and v_i at the bottom of the column. For each i, draw two chains of edges upwards from v_i to u_i one connecting together all $\odot$'s in column x_i and the other connecting all $\odot$'s in column $\bar{x}_i$ (see Figure 11.4). Now, draw edges $\langle u_i, v_{i+1}\rangle$, $1 \le i < n$. Introduce a box $\boxed{i}$ at the right end of each row C_i, $1 \le i \le r$. Draw the edges $\langle u_n, \boxed{1}\rangle$ and $\langle \boxed{r}, v_1\rangle$. Draw edges $\langle \boxed{i}, \boxed{i+1}\rangle$, $1 \le i < r$ (see Figure 11.4).

To complete the graph we shall replace each $\odot$ and $\boxed{i}$ by a subgraph. Each $\odot$ is replaced by the subgraph of Figure 11.5(a) (of course, unique vertex labelings are needed for each copy of the subgraph). Each box $\boxed{i}$ is replaced by the subgraph of Figure 11.6. In this subgraph A_i is an entrance node and B_i an exit node. The edges $\langle \boxed{i}, \boxed{i+1}\rangle$ referred to earlier are really $\langle B_i, A_{i+1}\rangle$. Edge $\langle u_n, \boxed{1}\rangle$ is $\langle u_n, A_1\rangle$ and $\langle \boxed{r}, v_1\rangle$ is $\langle B_r, v_1\rangle$. j_i is the number of literals in clause C_i. In the subgraph of Figure 11.6 an edge

$$R_{i,a} \longrightarrow \odot \longrightarrow R_{i,a+1}$$

indicates a connection to a $\odot$ subgraph in row C_i. $R_{i,a}$ is connected to the "1" vertex of the $\odot$ and $R_{i,a+1}$ (or $R_{i,1}$ if $a = j_i$) is entered from the "3" vertex. Thus in the $\odot$ subgraph



of Figure 11.5(b) w_1 and w_3 are the "1" and "3" vertices respectively. The incoming edge is $\langle R_{i,1}, w_1\rangle$ and the outgoing edge is $\langle w_3, R_{i,2}\rangle$. This completes the construction of G.

If F is satisfiable then let S be an assignment of truth values for which F is true. A Hamiltonian cycle for G can start at v_1, go to u_1 then to v_2, then u_2, then v_3, then u_3, $\ldots$, u_n. In going from v_i up to u_i this cycle will use the column corresponding to x_i if x_i is true in S. Otherwise it will go up the column corresponding to $\bar{x}_i$. From u_n this cycle will go to A_1 and then through $R_{1,1}$, $R_{1,2}$, $R_{1,3}$, $\ldots$, R_{1,j_1}, B_1 to A_2 $\cdots$ to v_1. In going from $R_{i,a}$ to $R_{i,a+1}$ in any subgraph $\boxed{i}$ a diversion will be made to a $\odot$ subgraph in row i iff the vertices of that $\odot$ subgraph are not already on the path from v_1 to $R_{i,a}$. Note that if C_i has i_j literals then the construction of $\boxed{i}$ allows a diversion to at most $i_j - 1$ $\odot$ subgraphs. This is adequate as at least one $\odot$ subgraph must already have been traversed in row C_i (as at least one such subgraph must correspond to a true literal). So, if F is satisfiable then G has a directed Hamiltonian cycle. It remains to show that if G has a directed Hamiltonian cycle then F is satisfiable. This may be seen by starting at vertex v_1 on any Hamiltonian cycle for G. Because of the construction of the $\odot$ and $\boxed{i}$ subgraphs, such a cycle must proceed by going up exactly one column of each pair $(x_i, \bar{x}_i)$. In addition, this part of the cycle must traverse at least one $\odot$ subgraph in each row. Hence the columns used in going from v_i to u_i, $1 \le i \le n$ define a truth assignment for which F is true.

We conclude that F is satisfiable iff G has a Hamiltonian cycle. The theorem now follows from the observation that G may be obtained from F in polynomial time. □

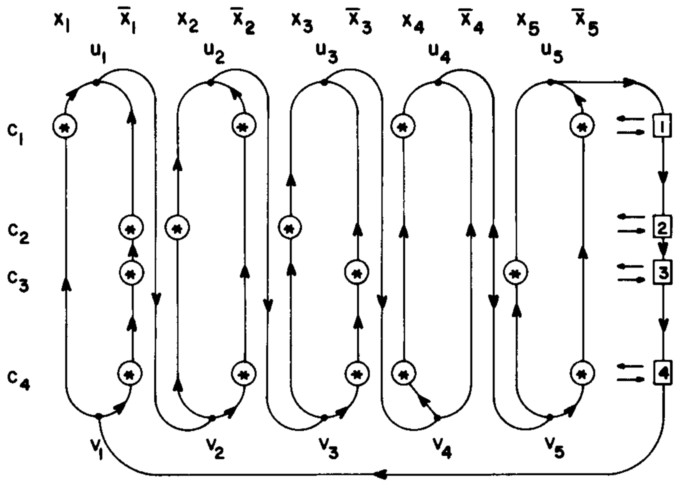


Figure 11.4 Array structure for formula in Theorem 11.5

Traveling Salesperson Decision Problem (TSP)

The traveling salesperson problem was introduced in Chapter 5. The corresponding decision problem is to determine if a complete directed graph $G = (V, E)$ with edge costs, $c(u, v)$, has a tour of cost at most M.

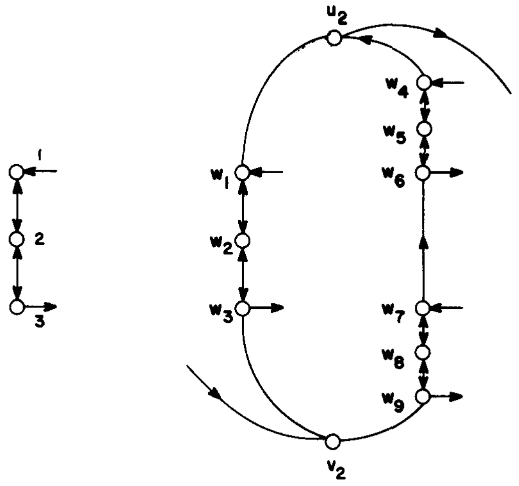


Figure 11.5 The ⊙ subgraph and its insertion into column 2

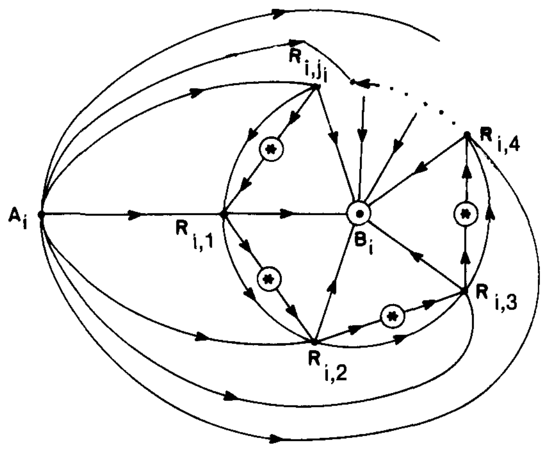


Figure 11.6 The $\boxed{i}$ subgraph

Theorem 11.6 Directed Hamiltonian cycle (DHC) $\propto$ traveling salesperson decision problem (TSP)

Proof: From the directed graph $G = (V, E)$ construct the complete directed graph $G' = (V, E'), E = \{\langle i, j\rangle \,|\, i \neq j\}$ and $c(i, j) = 1$ if $\langle i, j\rangle$

$\in E$; $c(i, j) = 2$ if $i \neq j$ and $\langle i, j \rangle \notin E$. Clearly, G' has a tour of cost at most n iff G has a directed Hamiltonian cycle. □

AND/OR Graph Decision Problem (AOG)

AND/OR graphs were introduced in Section 6.3. Let us assume that there is a cost associated with each edge in the graph. The *cost* of a solution graph H of an AND/OR graph G is the sum of the costs of the edges in H. The *AND/OR graph decision problem* (AOG) is to determine if G has a solution graph of cost at most k, for k a given input.

Example 11.15 Consider the directed graph of Figure 11.7. The problem to be solved is P_1. To do this, one may solve either nodes P_2, P_3 or P_7, as P_1 is an OR node. The cost incurred is then either 2, 2 or 8 (i.e., cost in addition to that of solving one of P_2, P_3 or P_7). To solve P_2, both P_4 and P_5 have to be solved, as P_2 is an AND node. The total cost to do this is 2. To solve P_3, we may solve either P_5 or P_6. The minimum cost to do this is 1. P_7 is free. In this example, then, the optimal way to solve P_1 is first solve P_6, then P_3 and finally P_1. The total cost for this solution is 3. □

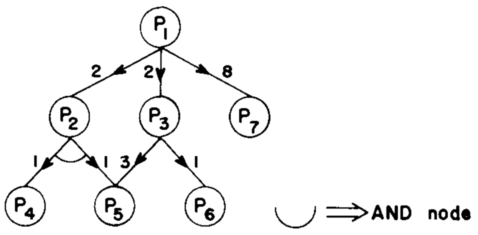


Figure 11.7 AND/OR graph

Theorem 11.7 CNF-satisfiability $\propto$ AND/OR graph decision problem

Proof: Let P be a propositional formula in CNF. We show how to transform a formula P in CNF into an AND/OR graph such that the AND/OR graph so obtained has a certain minimum cost solution iff P is satisfiable. Let

$$P = \bigwedge_{i=1}^{k} C_i, \quad C_i = \vee\, l_j,$$

where the l_j's are literals. The variables of P, $V(P)$ are $x_1, x_2, \ldots, x_n$. The AND/OR graph will have nodes as follows:

1. There is a special node, S, with no incoming arcs. This node represents the problem to be solved.
2. S is an AND node with descendent nodes $P, x_1, x_2, \ldots, x_n$.
3. Each node x_i represents the corresponding variable x_i in the formula P. Each x_i is an OR node with two descendents denoted Tx_i and Fx_i respectively. If Tx_i is solved, then this will correspond to assigning a truth value of "true" to the variable x_i. Solving node Fx_i will correspond to assigning a truth value of "false" to x_i.
4. The node P represents the formula P, and is an AND node. It has k descendents $C_1, C_2, \ldots, C_k$. Node C_i corresponds to the clause C_i in the formula P. The nodes C_i are OR nodes.
5. Each node of type Tx_i or Fx_i has exactly one descendent node which is terminal (i.e., has no edges leaving it). These terminal nodes shall be denoted $v_1, v_2, \ldots, v_{2n}$.

To complete the construction of the AND/OR graph, the following edges and costs are added:

1. From each node C_i an edge $\langle C_i, Tx_j \rangle$ is added if x_j occurs in clause C_i. An edge $\langle C_i, Fx_j \rangle$ is added if $\bar{x}_j$ occurs in the clause C_i. This is done for all variables x_j appearing in the clause C_i. C_i is designated an OR node.
2. Edges from nodes of type Tx_i or Fx_i to their respective terminal nodes are assigned a weight or cost 1.
3. All other edges have a cost 0.

In order to solve S, each of the nodes $P, x_1, x_2, \ldots, x_n$ must be solved. Solving nodes $x_1, x_2, \ldots, x_n$ costs n. To solve P, we must solve all the nodes $C_1, C_2, \ldots, C_k$. The cost of a node C_i is at most 1. However, if one of its descendent nodes was solved while solving the nodes $x_1, x_2, \ldots, x_n$, then the additional cost to solve C_i is 0, as the edges to its descendent nodes have cost 0 and one of its descendents has already been solved. I.e., a node C_i can be solved at no cost if one of the literals occurring in the clause C_i has been assigned a value "true." From this it follows that the entire graph (i.e., node S) can be solved at a cost n if there is some assignment of truth values to the x_i's such that at least one literal in each clause is true under that assignment, i.e., if the formula P is satisfiable. If P is not satisfiable, then the cost is more than n.

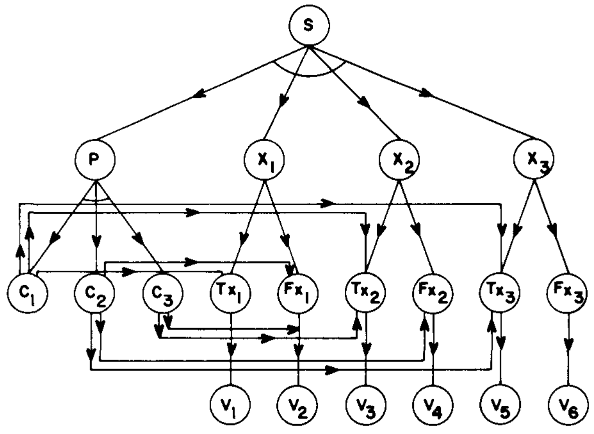
We have now shown how to construct an AND/OR graph from a formula P such that the AND/OR graph so constructed has a solution of cost n iff P is satisfiable. Otherwise the cost is more than n. The construction clearly takes only polynomial time. This completes the proof. □

Example 11.16 Consider the formula:

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2); \quad V(P) = x_1, x_2, x_3; \; n = 3.$$

Figure 11.8 shows the AND/OR graph obtained by applying the construction of Theorem 11.7.

The nodes Tx_1, Tx_2, Tx_3 can be solved at a total cost of 3. The node P then costs nothing extra. The node S can then be solved by solving all its descendent nodes and the nodes Tx_1, Tx_2 and Tx_3. The total cost for this solution is 3 (which is n). Assigning the truth value "true" to the variables of P results in P being "true." □



AND nodes marked ◡
All other nodes are OR

Figure 11.8 AND/OR graph for Example 11.16

11.4 NP-HARD SCHEDULING PROBLEMS

To prove the results of this section we shall need to use the NP-hard problem called partition. This problem requires us to decide whether a given multiset $A = \{a_1, a_2, \ldots, a_n\}$ of n positive integers has a partition P such that $\Sigma_{i \in P}\, a_i = \Sigma_{i \notin P}\, a_i$. We can show this problem NP-hard by first showing the sum of subsets problem (Chapter 7) NP-hard. Recall that in the sum of subsets problem we have to determine if $A = \{a_1, a_2, \ldots, a_n\}$ has a subset S that sums to a given integer M.

Theorem 11.8 Exact cover $\propto$ sum of subsets.

Proof: The exact cover problem is shown NP-hard in the exercises. In this problem we are given a family of sets $F = \{S_1, S_2, \ldots, S_k\}$ and are required to determine if there is a subset $T \subseteq F$ of disjoint sets such that

$$\bigcup_{S_i \in T} S_i = \bigcup_{S_i \in F} S_i = \{u_1, u_2, \ldots, u_n\}.$$

From any given instance of this problem construct the sum of subset problem $A = \{a_1, \ldots, a_k\}$ with $a_j = \Sigma_{1 \le i \le n}\, \epsilon_{ji}(k + 1)^{i-1}$ where $\epsilon_{ji} = 1$ if $u_i \in S_j$ and $\epsilon_{ji} = 0$ otherwise and $M = \Sigma_{0 \le i < n}\, (k + 1)^i = ((k + 1)^n - 1)/k$. Clearly, F has an exact cover iff $A = \{a_1, \ldots, a_k\}$ has a subset with sum M. Since A and M may be constructed from F in polynomial time, exact cover $\propto$ sum of subsets. □

Theorem 11.9 Sum of subsets $\propto$ partition

Proof: Let $A = \{a_1, \ldots, a_n\}$ and M define an instance of the sum of subsets problem. Construct the set $B = \{b_1, b_2, \ldots, b_{n+2}\}$ with $b_i = a_i$, $1 \le i \le n$, $b_{n+1} = M + 1$ and $b_{n+2} = (\Sigma_{1 \le i \le n}\, a_i) + 1 - M$. B has a partition iff A has a subset with sum M. Since B may be obtained from A and M in polynomial time, sum of subsets $\propto$ partition. □

One may easily show Partition $\propto$ 0/1-Knapsack and Partition $\propto$ Job sequencing with deadlines. Hence, these problems are also NP-hard.

Scheduling Identical Processors

Let P_i, $1 \le i \le m$ be m identical processors (or machines). The P_i could for example be line printers in a computer output room. Let J_i, $1 \le i \le n$ be n jobs. Job J_i requires t_i processing time. A schedule S is an assignment of jobs to processors. For each job J_i, S specifies the time intervals and the processor(s) on which this job is to be processed. A job cannot be processed by more than one processor at any given time. Let f_i be the time at which the processing of job J_i is completed. The *mean finish time* (mft) of schedule S is:

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i$$

Let w_i be a weight associated with each job J_i. The *weighted mean finish time* (wmft) of schedule S is:

$$\text{WMFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} w_i f_i.$$

Let T_i be the time at which P_i finishes processing all jobs (or job segments) assigned to it. The *finish time* of S is:

$$\text{FT}(S) = \max_{1 \le i \le m} \{T_i\}.$$

S is a *non-preemptive schedule* iff each job J_i is processed continuously from start to finish on the same processor. In a *preemptive* schedule each job need not be processed continuously to completion on one processor.

At this point it is worth noting the similarity between the optimal tape storage problem of Section 4.2 and non-preemptive schedules. Mean retrieval time, weighted mean retrieval time and maximum retrieval time respectively correspond to mean finish time, weighted mean finish time and finish time. Minimum finish time schedules can therefore be obtained using the algorithm developed in Section 4.2. Obtaining minimum weighted mean finish time and minimum finish time non-preemptive schedules is NP-hard.

Theorem 11.10 Partition $\propto$ minimum finish time non-preemptive schedule

Proof: We shall prove this for $m = 2$. The extension to $m > 2$ is trivial. Let a_i, $1 \le i \le n$ be an instance of the partition problem. Define n jobs with processing requirements $t_i = a_i$, $1 \le i \le n$. There is a non-preemptive schedule for this set of jobs on two processors with finish time at most $\Sigma\, t_i/2$ iff there is a partition of the a_i's. □

Theorem 11.11 Partition $\propto$ minimum WMFT non-preemptive schedule

Proof: Once again we prove this for $m = 2$ only. The extension to $m > 2$

is trivial. Let a_i, $1 \leq i \leq n$ define an instance of the partition problem. Construct a two processor scheduling problem with n jobs and $w_i = t_i = a_i$, $1 \leq i \leq n$. For this set of jobs there is a non-preemptive schedule S with weighted mean flow time at most $1/2 \; \Sigma \, a_i^2 + 1/4 \; (\Sigma \, a_i)^2$ iff the a_i's have a partition. To see this let the weights and times of jobs on P_1 be $(\bar{w}_1, \bar{t}_1), \ldots, (\bar{w}_k, \bar{t}_k)$ and on P_2 be $(\bar{\bar{w}}_1, \bar{\bar{t}}_1), \ldots, (\bar{\bar{w}}_l, \bar{\bar{t}}_l)$. Assume this is the order in which the jobs are processed on their respective processors. Then, for this schedule S we have:

$$\begin{aligned} \text{n*WMFT}(S) &= \bar{w}_1\bar{t}_1 + \bar{w}_2(\bar{t}_1 + \bar{t}_2) + \ldots + \bar{w}_k(\bar{t}_1 + \ldots + \bar{t}_k) \\ &\quad + \bar{\bar{w}}_1\bar{\bar{t}}_1 + \bar{\bar{w}}_2(\bar{\bar{t}}_1 + \bar{\bar{t}}_2) + \cdots + \bar{\bar{w}}_l(\bar{\bar{t}}_1 + \cdots + \bar{\bar{t}}_l) \\ &= \frac{1}{2}\,\Sigma\, w_i^2 + \frac{1}{2}\,(\Sigma\,\bar{w}_i)^2 + \frac{1}{2}\,(\Sigma\, w_i - \Sigma\,\bar{w}_i)^2. \end{aligned}$$

Thus, n*WMFT$(S) \geq (1/2)\,\Sigma\, w_i^2 + (1/4)\,(\Sigma\, w_i)^2$. This value is obtainable iff the w_i's (and so also the a_i's) have a partition. □

Flow Shop Scheduling

We shall use the flow shop terminology developed in Section 5.8. When $m = 2$, minimum finish time schedules can be obtained in $O(n \log n)$ time if n jobs are to be scheduled. When $m = 3$ obtaining minimum finish time schedules (whether preemptive or non-preemptive) is NP-hard. For the case of non-preemptive schedules this is easy to see (exercise 30). We shall prove the result for preemptive schedules. The proof we shall give is also valid for the non-preemptive case. However, a much simpler proof exists for the non-preemptive case.

Theorem 11.12 Partition $\propto$ minimum finish time preemptive flow shop schedule ($m > 2$).

Proof: We shall use only three processors. Let $A = \{a_1, a_2, \ldots, a_n\}$ define an instance of the partition problem. Construct the following preemptive flow shop instance, FS, with $n + 2$ jobs, $m = 3$ machines and at most 2 nonzero tasks per job:

$$t_{1,i} = a_i;\; t_{2,i} = 0;\; t_{3,i} = a_i, \quad 1 \leq i \leq n$$

$$t_{1,n+1} = T/2;\; t_{2,n+1} = T; \quad t_{3,n+1} = 0$$

$$t_{1,n+2} = 0;\; t_{2,n+2} = T; \quad t_{3,n+2} = T/2$$

where

$$T = \sum_{1}^{n} a_i.$$

We now show that the above flow shop instance has a preemptive schedule with finish time at most $2T$ iff A has a partition.

(a) If A has a partition u then there is a non-preemptive schedule with finish time $2T$. One such schedule is shown in Figure 11.9.

(b) If A has no partition then all preemptive schedules for FS must have a finish time greater than $2T$. This can be shown by contradiction. Assume that there is a preemptive schedule for FS with finish time at most $2T$. We make the following observations regarding this schedule:

 (i) task $t_{1,n+1}$ must finish by time T (as $t_{2,n+1} = T$ and cannot start until $t_{1,n+1}$ finishes)

 (ii) task $t_{3,n+2}$ cannot start before T units of time have elapsed as $t_{2,n+2} = T$.

Observation (i) implies that only $T/2$ of the first T time units are free on processor one. Let V be the set of indices of tasks completed on processor 1 by time T (excluding task $t_{1,n+1}$). Then,

$$\sum_{i \in V} t_{1,i} < T/2$$

as A has no partition. Hence

$$\sum_{\substack{i \notin V \\ 1 \le i \le n}} t_{3,i} > T/2.$$

The processing of jobs not included in V cannot commence on processor 3 until after time T since their processor 1 processing is not completed until after T. This together with observation (ii) implies that the total amount of processing left for processor 3 at time T is

$$t_{3,n+2} + \sum_{\substack{i \notin V \\ 1 \le i \le n}} t_{3,i} > T.$$

The schedule length must therefore be more than $2T$. □

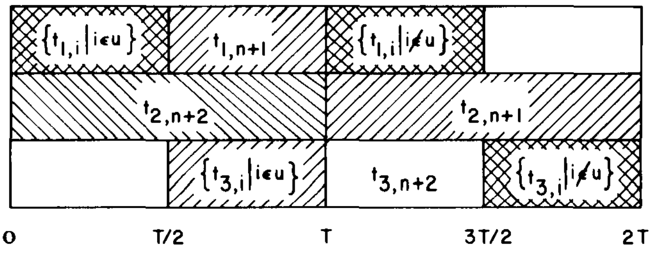


Figure 11.9 A possible schedule

Job Shop Scheduling

A job shop, like a flow shop, has m different processors. The n jobs to be scheduled require the completion of several tasks. The time of the jth task for job J_i is $t_{k,i,j}$. Task j is to be performed on processor P_k. The tasks for any job J_i are to be carried out in the order 1, 2, 3, ..., etc. Task j cannot begin until task $j - 1$ (if $j > 1$) has been completed. Note that it is quite possible for a job to have many tasks that are to be performed on the same processor. In a non-preemptive schedule, a task once begun is processed without interruption until it is completed. The definitions of FT(S) and MFT(S) extend to this problem in a natural way. Obtaining either a minimum finish time preemptive or minimum finish time non-preemptive schedule is NP-hard even when $m = 2$. The proof for the nonpreemptive case is very simple (use partition). We shall present the proof for the preemptive case. This proof will also be valid for the non-preemptive case but will not be the simplest proof for this case.

Theorem 11.13 Partition $\propto$ minimum finish time preemptive job shop schedule ($m > 1$).

Proof: We shall use only two processors. Let $A = \{a_1, a_2, \ldots, a_n\}$ define an instance of the partition problem. Construct the following job shop instance JS, with $n + 1$ jobs and $m = 2$ processors.

$$\text{Jobs } 1, \ldots, n: t_{1,i,1} = t_{2,i,2} = a_i \quad \text{for} \quad 1 \le i \le n$$
$$\text{Job } n + 1: \quad t_{2,n+1,1} = t_{1,n+1,2} = t_{2,n+1,3} = t_{1,n+1,4} = T/2$$

where

$$T = \sum_1^n a_i$$

We show that the above job shop problem has a preemptive schedule with finish time at most $2T$ iff S has a partition.

a) If A has a partition u then there is a schedule with finish time $2T$ (see Figure 11.10).

b) If A has no partition then all schedules for JS must have a finish time greater than $2T$. To see this assume that there is a schedule S for JS with finish time at most $2T$. Then, job $n + 1$ must be scheduled as in Figure 11.10. Also, there can be no idle time on either P_1 or P_2. Let R be the set of jobs scheduled on P_1 in the interval $[0, T/2]$. Let R' be the subset of R representing jobs whose first task is completed on P_1 in this interval. Since the a_i's have no partition, $\Sigma_{j \in R'} t_{i,j,1} < T/2$. Consequently, $\Sigma_{j \in R'} t_{2,j,2} < T/2$. Since only the second tasks of jobs in R' may be scheduled on P_2 in the interval $[T/2, T]$, it follows that there is some idle time on P_2 in this interval. Hence, S must have finish time greater than $2T$. □

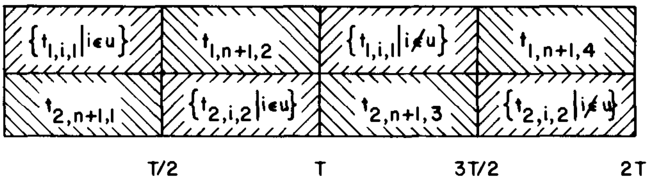

$\{t_{1,i,1} | i \in u\}$
$t_{1,n+1,2}$
$\{t_{1,i,1} | i \notin u\}$
$t_{1,n+1,4}$
$t_{2,n+1,1}$
$\{t_{2,i,2} | i \in u\}$
$t_{2,n+1,3}$
$\{t_{2,i,2} | i \notin u\}$
T/2
T
3T/2
2T

Figure 11.10 Another schedule

11.5 NP-HARD CODE GENERATION PROBLEMS

Code Generation With Common Subexpressions

When arithmetic expressions have common subexpressions they may be represented by a directed acyclic graph (dag). Every internal node (node with nonzero out-degree) in the dag represents an operator. Assuming the expression contains only binary operators, each internal node, P, has out-degree two. The two nodes adjacent from P will be called the left and right children of P respectively. The children of P are the roots of the dags for the left and right operands of P. P is the parent of its children. In case the expression contains no common subexpressions, its dag representation is identical to the tree representation of Section 6.2. Figure 11.11 shows some expressions and their dag representations.

Definition: A *leaf* is a node with out-degree zero. A *level one* node is a node both of whose children are leaves. A *shared node* is a node with more than one parent. A *leaf dag* is a dag in which all shared nodes are leaves. A *level one* dag is a dag in which all shared nodes are level one nodes.

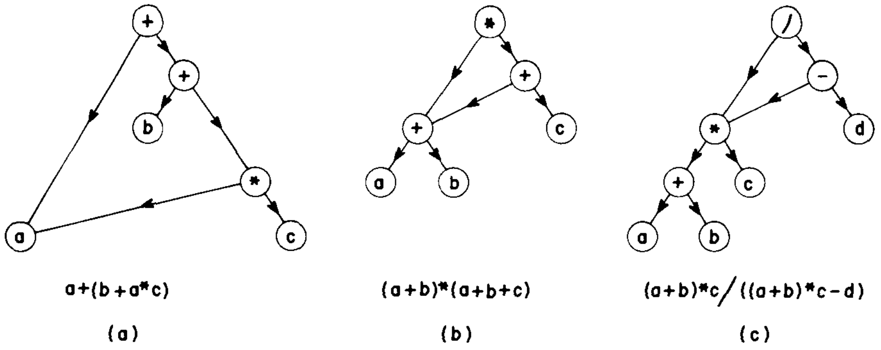


Figure 11.11 Expressions and their dags

Example 11.17 The dag of Figure 11.11(a) is a leaf dag. Figure 11.11(b) is a level one dag. Figure 11.11(c) is neither a leaf dag nor a level one dag. □

A leaf dag results from an arithmetic expression in which the only common subexpressions are simple variables or constants. A level one dag results from an expression in which the only common subexpressions are of the form a $\textcircled{\scriptsize op}$ b where a and b are simple variables or constants and $\textcircled{\scriptsize op}$ is an operator.

The problem of generating optimal code for level one dags is NP-hard even when the machine for which code is being generated has only one register. Determining the minimum number of registers needed to evaluate a dag with no STOREs is also NP-hard. Note that both these problems can be solved in linear time when there are no common subexpressions (Section 6.2).

Example 11.18 The optimal codes for the dag of Figure 11.11(b) for one and two registers machines is given in Figure 11.12.

The minimum number of registers needed to evaluate this dag without any STOREs is 2. □

```
LOAD   a,R1                    LOAD   a,R1
ADD    R1,b,R1                 ADD    R1,b,R1
STORE  T1,R1                   ADD    R1,c,R2
ADD    R1,c,R1                 MUL    R1,R2,R1
STORE  T2,R1
LOAD   T1,R1
MUL    R1,T2,R1

        (a)                            (b)
```

Figure 11.12 Optimal codes for one and two register machines

In order to prove the above statements we shall use the feedback node set (FNS) problem that is shown to be NP-hard in the exercises.

FNS: Given a directed graph $G = (V, E)$ and an integer k determine if there exists a subset V' of vertices $V' \subseteq V$ and $|V'| \leq k$ such that the graph $H = (V - V', E - \{\langle u, v\rangle \mid u \in V' \text{ or } v \in V'\})$ obtained from G by deleting all vertices in V' and all edges incident to a vertex in V' contains no directed cycles.

We shall explicitly prove only that generating optimal code is NP-hard. Using the construction of this proof one can also show that determining the minimum number of registers needed to evaluate a dag with no STOREs is also NP-hard. The proof assumes that expressions may contain commutative operators and that shared nodes may be computed only once. It is easily extended to allow recomputation of shared nodes. Using an idea due to Ravi Sethi, the proof is easily extended to the case when only noncommutative operators are allowed (see Exercise 41).

Theorem 11.14 FNS $\propto$ optimal code generation for level one dags on a one register machine.

Proof: Let G, k be an instance of FNS. Let n be the number of vertices in G. We shall construct a dag A with the property that the optimal code for the expression corresponding to A has at most $n + k$ LOADs iff G has a feedback node set of size at most R.

The dag A consists of three kinds of nodes: leaf nodes, chain nodes and tree nodes. All chain and tree nodes are internal nodes representing commutative operators (e.g., '+'). Leaf nodes represent distinct variables. We shall use d_v to denote the out-degree of vertex v of G. Corresponding to each vertex v of G there is a directed chain of chain nodes $v_1, v_2, \ldots, v_{d_v+1}$ in A. Node v_{d_v+1} is the *head node* of the chain for v and is the parent of two leaf nodes v_L and v_R (see Example 11.19 and Figure 11.13). v_1 is the *tail* of the chain. From each of the chain nodes corresponding to vertex v, except the head node, there is one directed edge to the head node of one of the chains corresponding to a vertex w such that $\langle v, w \rangle$ is an edge in G. Each such edge goes to a distinct head. Note that as a result of the addition of these edges, each chain node now has out-degree two. Since each chain node represents a commutative operator, it does not matter which of its two children is regarded as the left child.

At this point we have a dag in which the tail of every chain has in-degree zero. We now introduce tree nodes to combine all the heads together so that we are left with only one node (the root) with in-degree zero. Since G has n vertices, we need $n - 1$ tree nodes (note that every binary tree with $n - 1$ internal nodes has n external nodes). These $n - 1$ nodes are connected together to form a binary tree (any binary tree with $n - 1$ nodes will do). In place of the external nodes we connect the tails of the n chains (see Figure 11.13(b)). This yields a dag A corresponding to an arithmetic expression.

It is easy to see that every optimal code for A will have exactly n LOADs of leaf nodes. Also, there will be exactly one instruction of type ⓞⓟ for every chain node and tree node (we assume that a shared node is computed only once). Hence, the only variable is the number of LOADs and STOREs of chain and tree nodes. If G has no directed cycles then its vertices may be arranged in topological order (vertex u precedes vertex v in a topological ordering only if there is no directed path from u to v in G). Let $v_1, v_2, \ldots, v_n$ be a topological ordering of the vertices in G. The expression A can be computed using no LOADs of chain and tree nodes by first computing all nodes on the chain for v_n and storing the result of the tail node. Next, all nodes on the chain for v_{n-1} may be computed. In addition, we can compute any nodes on the path from the tail for v_{n-1} to the root for which both operands are available. Finally, one result needs to be stored. Next, the chain for v_{n-2} may be computed. Again, we can compute all nodes on the path from this chain tail to the root for which both operands are available. Continuing in this way, the entire expression may be computed.

If G contains at least one cycle: $v_1, v_2, \ldots, v_i, v_1$ then every code for A must contain at least one LOAD of a chain node on a chain for one of $v_1, v_2, \ldots, v_i$. Further, if none of these vertices is on any other cycle then all their chain nodes may be computed using only one load of a chain node. This argument is readily generalized to show that if the size of a minimum feedback node set is p then every optimal code for A contains exactly $n + p$ LOADs. The p LOADs correspond to a combination of tail nodes corresponding to a minimum feedback node set and the siblings of these tail nodes. In case we had used non-commutative operators for chain nodes and made each successor on a chain the left child of its parent then the p LOADs will correspond to the tails of the chains of any minimum feedback set. Furthermore, if the optimal code contains p LOADs of chain nodes then G has a feedback node set of size p. □

Example 11.19 Figure 11.13(b) shows the dag A corresponding to the graph G of Figure 11.13(a). $\{r, s\}$ is a minimum feedback node set for G. The operator in each chain and tree node may be assumed to be '+'. Every code for A has a load corresponding to one of (p_L, p_R), (q_L, q_R), ... and (u_L, u_R). The expression A can be computed using only two additional LOADs by computing nodes in the order r_4, s_2, q_2, q_1, p_2, p_1, c, u_3, u_2, u_1, t_2, t_1, e, s_1, r_3, r_2, r_1, d, b, a. Note that a LOAD is needed to compute s_1 and also to compute r_3. □

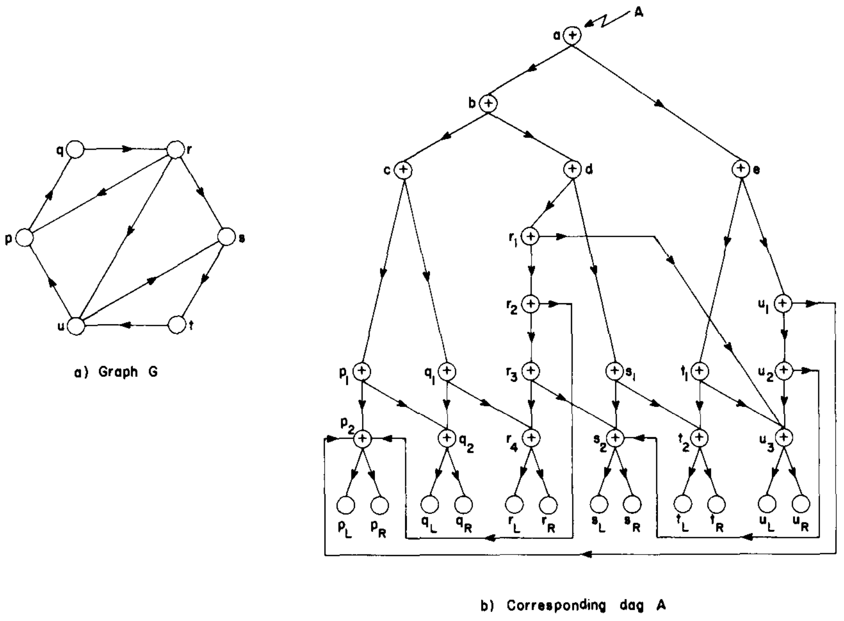


Figure 11.13 A graph and its corresponding dag.

Implementing Parallel Assignment Instructions

A SPARKS *parallel assignment* instruction has the format $(v_1, v_2, \ldots, v_n) \leftarrow (e_1, e_2, \ldots, e_n)$ where the v_is are distinct variable names and the e_is are expressions. The semantics of this statement is that the value of v_i is updated to be the value of the expression e_i, $1 \le i \le n$. The value of the expression e_i is to be computed using the values the variables in e_i have before this instruction is executed.

Example 11.20

(i) $(A, B) \leftarrow (B, C)$ is equivalent to $A \leftarrow B; B \leftarrow C$
(ii) $(A, B) \leftarrow (B, A)$ is equivalent to $T \leftarrow A; A \leftarrow B; B \leftarrow T$
(iii) $(A, B) \leftarrow (A + B, A - B)$ is equivalent to $T1 \leftarrow A$; $T2 \leftarrow B$; $A \leftarrow T1 + T2; B \leftarrow T1 - T2$ and also to $T1 \leftarrow A; A \leftarrow A + B$; $B \leftarrow T1 - B$. □

As the above example indicates, it may be necessary to store some of the v_is into temporary locations when executing a parallel assignment. These stores are needed only when some of the v_is appear in the expressions e_j, $1 \le j \le n$. A variable v_i is *referenced* by expression e_j iff v_i appears in e_j. It should be clear that only referenced variables need to be copied into temporary locations. Further, Examples 11.20 (ii) and (iii) show that not all referenced variables need to be copied.

An implementation of a parallel assignment statement is a sequence of instructions of types $T_j \leftarrow v_i$ and $v_i \leftarrow e'_i$ where e'_i is obtained from e_i by replacing all occurrences of a v_i that has already been updated with a reference to the temporary location in which the old value of v_i has been saved. Let $R = (\tau(1), \ldots, \tau(n))$ be a permutation of $(1, 2, \ldots, n)$. R is a *realization* of an assignment statement. It specifies the order in which statements of type $v_i \leftarrow e'_i$ appear in an implementation of a parallel assignment statement. The order is $v_{\tau(1)} \leftarrow e'_{\tau(1)}$; $v_{\tau(2)} \leftarrow e'_{\tau(2)}$; etc. The implementation also has statements of type $T_j \leftarrow v_i$ interspersed. Without loss of generality we may assume that the statement $T_j \leftarrow v_i$ (if it appears in the implementation) immediately precedes the statement $v_i \leftarrow e'_i$. Hence, a realization completely characterizes an implementation. The minimum number of instructions of type $T_j \leftarrow v_i$ for any given realization is easy to determine. This number is the cost of the realization. The *cost* $C(R)$ of a realization R is the number of v_i that are referenced by an e_j that corresponds to an instruction $v_j \leftarrow e'_j$ that appears after the instruction $v_i \leftarrow e'_i$.

Example 11.21 Consider the statement $(A, B, C) \leftarrow (D, A + B, A - B)$ The $3! = 6$ different realizations and their costs are:

R	$C(R)$
1, 2, 3	2
1, 3, 2	2
2, 1, 3	2
2, 3, 1	1
3, 1, 2	1
3, 2, 1	0

The realization 3, 2, 1 corresponding to the implementation $C \leftarrow A - B$; $B \leftarrow A + B$; $A \leftarrow D$ needs no temporary stores ($C(R) = 0$). □

An optimal realization for a parallel assignment statement is one with minimum cost. When the expressions e_i are all variable names or constants, an optimal realization can be found in linear time ($O(n)$). When the e_i are allowed to be expressions with operators then finding an optimal realization is NP-Hard. We shall prove this latter statement using the feedback node set problem.

Theorem 11.15 FNS $\propto$ minimum cost realization.

Proof: Let $G = (V, E)$ be any n vertex directed graph. Construct the parallel assignment statement P: $(v_1, v_2, \ldots, v_n) \leftarrow (e_1, e_2, \ldots, e_n)$ where the v_i's correspond to the n vertices in V and e_i is the expression $v_{i_1} + v_{i_2} + \cdots + v_{i_j}$. $\{v_{i_1}, v_{i_2}, \ldots, v_{i_j}\}$ is the set of vertices adjacent from v_i (i.e. $\langle v_i, v_{i_l}\rangle \in E(G)$, $1 \le l \le j$). This construction requires at most $O(n^2)$ time.

Let U be any feedback node set for G. Let $G' = (V', E') = (V - U, E - \{\langle x, y\rangle \mid x \in U \text{ or } y \in U\})$ be the graph obtained by deleting vertex set U and all edges incident to vertices in U. From the definition of a feedback node set it follows that G' is acyclic. So, the vertices in $V - U$ may be arranged in a sequence $s_1, s_2, \ldots, s_m$ where $m = |V - U|$ and E' contains no edge $\langle s_j, s_i\rangle$ for any i, j, $1 \le i < j \le m$. Hence, an implementation of P in which variables corresponding to vertices in U are first stored in temporary locations followed by the instructions $v_i \leftarrow e'_i$ corresponding to $v_i \in U$, followed by the corresponding instructions for $s_1, s_2, \ldots, s_m$ (in that order), will be a correct implementation. (e'_i is e_i with all occurrences of $v_i \in U$ replaced by the corresponding temporary location). The realization, R, corresponding to this implementation has $C(R) = |U|$. Hence, if G has a feedback node set of size at most k then P has an optimal realization of cost at most k.

Suppose P has a realization R of cost k. Let U be the set of k variables that have to be stored in temporary locations and let $R = (q_1, q_2, \ldots, q_n)$. From the definition of $C(R)$ it follows that no e_{q_i} references a v_{q_j} with $j < i$ unless $v_{q_j} \in U$. Hence, the deletion of vertices in U from G leaves G acyclic. Thus, U defines a feedback node set of size k for G.

G has a feedback node set of size at most k iff P has a realization of cost at most k. Thus we can solve the feedback node set problem in polynomial time if we have a polynomial time algorithm that determines a minimum cost realization. □

11.6 SOME SIMPLIFIED NP-HARD PROBLEMS

Once we have shown a problem L to be NP-hard we would be inclined to dismiss the possibility that L can be solved in deterministic polynomial time. At this point, however, one may naturally ask the question: Can a suitably restricted version (i.e., some subclass) of an NP-hard problem be solved in deterministic polynomial time? It should be easy to see that by placing enough restrictions on any NP-hard problem (or by defining a sufficiently small subclass) we can arrive at a polynomially solvable problem. As examples, consider the following:

i) CNF-satisfiability with at most three literals per clause is NP-hard. If each clause is restricted to have at most two literals then CNF-satisfiability is polynomially solvable.
ii) Generating optimal code for a parallel assignment statement is NP-hard. However, if the expressions e_i are restricted to be simple variables then optimal code can be generated in polynomial time.
iii) Generating optimal code for level one dags is NP-hard but optimal code for trees can be generated in polynomial time.
iv) Determining if a planar graph is three colorable is NP-hard. To determine if it is two colorable we only have to see if it is bipartite.

Since it is very unlikely that NP-hard problems are polynomially solvable, it is important to determine the weakest restrictions under which we can solve a problem in polynomial time.

To narrow the gap between subclasses for which polynomial time algorithms are known and those for which such algorithms are not known, it is desirable to obtain as strong a set of restrictions under which a problem remains NP-hard or NP-complete.

We state without proof the severest restrictions under which certain problems are known to be NP-hard or NP-complete. We shall state these simplified or restricted problems as decision problems. For each problem we shall specify only the input and the decision to be made.

Theorem 11.16 The following decision problems are NP-complete:

1. **Node Cover**
   Input: An undirected graph G with node degree at most 3 and an integer k.
   Decision: Does G have a node cover of size at most k?

2. **Planar Node Cover**
   Input: A planar undirected graph G with node degree at most 6 and an integer k.
   Decision: Does G have a node cover of size at most k?

3. **Colorability**
   Input: A planar undirected graph G with node degree at most four.
   Decision: Is G 3-colorable?

4. **Undirected Hamiltonian Cycle**
   Input: An undirected graph G with node degree at most three.
   Decision: Does G have a Hamiltonian cycle?

5. **Planar Undirected Hamiltonian Cycle**
   Input: A planar undirected graph.
   Decision: Does G have a Hamiltonian cycle?

6. **Planar Directed Hamiltonian Path**
   Input: A planar directed graph G with in-degree at most 3 and out-degree at most 4.
   Decision: Does G have a directed Hamiltonian path?

7. **Unary Input Partition**
   Input: Positive integers a_i, $1 \le i \le m$, n, and B such that

   $$\sum_{1 \le i \le m} a_i = nB, \; \frac{B}{4} < a_i < \frac{B}{2}, \; 1 \le i \le m \text{ and } m = 3n.$$

   Input is in unary notation.
   Decision: Is there a partition $\{A_1, \ldots, A_n\}$ of the a_i's such that each A_i contains three elements and

$$\sum_{a \in A_i} a = B, \quad 1 \le i \le n?$$

8. **Unary Flow Shop**
Input: Task times in unary notation and an integer T.
Decision: Is there a two processor non-preemptive schedule with mean finish time at most T?

9. **Simple Max Cut**
Input: A graph $G = (V, E)$ and an integer k.
Decision: Does V have a subset V_1 such that there are at least k edges $(u, v) \in E$ with $u \in V_1$ and $v \notin V_1$?

10. **SAT2**
Input: A propositional formula F in CNF. Each clause in F has at most two literals. An integer k.
Decision: Can at least k clauses of F be satisfied?

11. **Minimum Edge Deletion Bipartite Subgraph**
Input: An undirected graph G and an integer k.
Decision: Can G be made bipartite by the deletion of at most k edges?

12. **Minimum Node Deletion Bipartite Subgraph**
Input: An undirected graph G and an integer k.
Decision: Can G be made bipartite by the deletion of at most k vertices.

13. **Minimum Cut Into Equal-Sized Subsets**
Input: An undirected graph $G = (V, E)$, two distinguished vertices s and t and a positive integer W.
Decision: Is there a partition $V = V_1 \cup V_2$, $V_1 \cap V_2 = \phi$, $|V_1| = |V_2|$, $s \in V_1$, $t \in V_2$ and $|\{(u, v) | u \in V_1, v \in V_2 \text{ and } (u, v) \in E\}| \le W$?

14. **Simple Optimal Linear Arrangement**
Input: An undirected graph $G = (V, E)$ and an integer k. $|V| = n$.
Decision: Is there a one to one function $f: V \rightarrow \{1, 2, \ldots, n\}$ such that

$$\sum_{(u,v) \in E} |f(u) - f(v)| \le k$$

REFERENCES AND SELECTED READINGS

A comprehensive treatment of NP-hard and NP-complete problems may be found in the book:

Computers and intractability: A guide to the theory of NP-Completeness, by M. Garey and D. Johnson, Freeman and Co., San Francisco, 1979.

Cook's theorem (Section 11.2) appears in:
"The complexity of theorem-proving procedures," by S. A. Cook, *Proc. of the Third ACM Symposium on Theory of Computing,* 1971, pp. 151–158.

The above paper also shows satisfiability $\propto$ clique. Cook's original proof is in terms of Turing machines. The proof given in the text was adapted by S. Sahni. We are grateful to R. Kain for pointing out an error in the original adaptation. J. Ullman has adapted Cook's proof to a somewhat different machine model.

Karp showed the importance of the class of NP-complete problems by exhibiting 21 problems that are NP-complete. His list of problems includes node cover, feedback arc set, feedback node set, Hamiltonian cycle, partition, sum of subsets, job sequencing with deadlines, max cut etc. Karp's work appears in:

"Reducibility among combinatorial problems," by R. Karp, *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104. [exercises 6, 10, 11, 14, 19, 20, 21, 23, 24, 29, 37, 40].

Our proof satisfiability $\propto$ directed Hamiltonian cycle is from:

"On reducibility among combinatorial problems," by P. Hermann, MIT MAC Report TR-113, December 1973. [exercises 8, 9]

The proof satisfiability $\propto$ AND/OR Graphs is from:

"Computationally related problems," by S. Sahni, *SIAM Journal on Computing,* 3:4(1974), pp. 262–279. [exercises 18, 25, 35]

This paper also contains reductions to many network flow, n-person game theory and optimization problems. Theorem 11.11 is due to Bruno, Coffman and Sethi. It appears in the following paper:

"Scheduling independent tasks to reduce mean finishing-time," by J. Bruno, E. G. Coffman, Jr. and R. Sethi, *Comm. ACM,* 17:7, July 1974, pp. 382–387.

The proof used in the text for Theorem 11.11 is due to S. Sahni and appears in:

"Algorithms for scheduling independent tasks," by S. Sahni, *JACM*, 23, 1976, pp. 114–127.

Theorems 11.12 and 11.13 are due to Gonzalez and Sahni. The reference is:

"Flow shop and job shop schedules: complexity and approximation," by T. Gonzalez and S. Sahni, *Op. Res.*, 26(1), pp. 36–52, 1978.

The proof used in the text for Theorem 11.13 is due to D. Nassimi. Many other scheduling problems are known to be NP-hard. Some references are:

"Machine scheduling problems," by A. Rinnooy Kan, Ph.D. thesis, Mathematical Centrum, Amsterdam, 1976.

"Sequencing by enumerative methods," by J. K. Lenstra, Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1976.

"Polynomial complete scheduling problems," by J. D. Ullman, *JCSS*, June 1975, pp. 384–393.

Computer and Job Shop Scheduling Theory, by E. G. Coffman, J. Wiley, New York 1976.

"The Complexity of Flowshop and Jobshop Scheduling," by M. Garey, D. Johnson, and R. Sethi, *Math. of Operations Research*, 1:2(1976), pp. 117–129 [exercises 30, 31].

"Complexity results for multiprocessor scheduling under resource constraints," by M. Garey and D. Johnson, *SIAM Journal on Computing*, 4:4(1975), pp. 397–411.

"On the complexity of a timetable and multicommodity flow problems," by S. Even, *SIAM Jr. on Computing*, 5, 691–703 (1976).

"Algorithms for minimizing mean flow time," by J. Bruno, E. G. Coffman, and R. Sethi, *Proc. IFIP Congr. 74*, 1974, pp. 504–510.

"On the complexity of mean flow time scheduling," by R. Sethi, *Math. of Op. Res.*, 2(4), 320–330 (1977).

"Open Shop Scheduling to Minimize Finish Time," by T. Gonzalez and S. Sahni, *JACM*, 23(4), pp. 665–679 (1976).

"Optimization and approximation in deterministic sequencing and scheduling: a survey," by R. Graham, E. Lawler, J. Lenstra and A. Rinnooy Kan, Department of Operations Research, Mathematisch Centrum, Amsterdam, Report #BW 82/77, 1977.

"Complexity of machine scheduling problems," by P. Brucker, J. Lenstra and A.

Rinnooy Kan, Math. Centrum, Amsterdam, Report #BW 43/75, 1975. [exercises 30, 31, 32, 33].

"Complexity of scheduling shops with no wait in process," by S. Sahni and Y. Cho, University of Minnesota, Technical Report #77-20, 1977 (to appear in Math. of Oper. Res.).

"Preemptive shop scheduling of independent job with release times," by Y. Cho and S. Sahni, University of Minnesota, Technical Report #78-5, 1978. [exercise 36].

The proof of Theorem 11.14 is an adapation of a proof that appears in:

"Code generation for expressions with common subexpressions," by A. Aho, S. Johnson and J. Ullman, *JACM,* 24(1), pp. 146–160 (1977). [exercise 41].

The fact that the code generation problem for one register machines is NP-hard was first proved by Bruno and Sethi in:

"Code generation for a one-register machine," by J. Bruno and R. Sethi, *J.ACM,* 23(3), pp. 502-510 (1976).

The result of the above paper is stronger than Theorem 11.14 as it applies even to expressions containing no commutative operators. Theorem 11.15 is due to R. Sethi. The reference is:

"A note on implementing parallel assignment instructions," by R. Sethi, *Info. Proc. Let.,* 2, pp. 91-95 (1973).

Further results on NP-Hard code generation problems appear in:

"Complete register allocation problems," by R. Sethi, *SIAM Jr. on Comp.,* 4(3), pp. 226–248 (1975).

"Code generation for short/long address machines," by E. Robertson, University of Wisconsin, MRC report #1779, August 1977.

The results stated in Section 11.6 may be found in:

"Some simplified NP-Complete graph problems," by M. Garey, D. Johnson and L. Stockmeyer, *Jr. Theo. Comp. Sci.,* 1, pp. 237-267 (1976).

"The planar Hamiltonian circuit problem is NP-Complete," by M. Garey, D. Johnson and R. Tarjan, *SIAM Jr. on Computing*, 5(4), pp. 704–714, 1976.

"The complexity of flowshop and jobshop scheduling," by M. Garey, D. Johnson and R. Sethi, *Math. of Oper. Res.,* 1(2), pp. 117-129 (1976).

Other interesting papers on NP-hard and NP-complete problems are:

"Some complexity results for the traveling salesman problem," by C. Papadimitriou and K. Steiglitz, *Proc. Eighth Annual ACM Symposium on Theory of Computing,* May 1976, pp. 1-9.

"On the computational complexity of combinatorial problems," by R. Karp, *Networks,* 5, 1975, pp. 45-68.

"Polynomially complete fault detection problems," by O. Ibarra and S. Sahni, *IEEE Trans. Comp.,* 24(3), pp. 242-249(1975). [exercises 26, 27].

"Generalizing NP-Completeness to permit different input measures," by M. Garey and D. Johnson, Bell Laboratories, New Jersey, 1976.

"Strong NP-Completeness results: motivation, examples and implications," by M. Garey and D. Johnson, Bell Laboratories, New Jersey, 1976.

"Constructing optimal binary decision trees is NP-Complete," by L. Hyafil and R. Rivest, *Info. Proc. Let.,* 5(1), pp. 15-17 (1976).

"Assignment commands with array references," by P. Downey and R. Sethi, *Proc. 17th Annual Symp. on Found. of Comp.,* pp. 57-66 (1976) (to appear in *JACM*).

"On the computational complexity of schema equivalence," by R. Constable, H. Hunt and S. Sahni, *8th Annual Princeton Conference on Information Sciences and Systems,* pp. 15-20 (1974).

"Complexity of trie index construction," by D. Comer and R. Sethi, *JACM,* 24(3), 1977, pp. 428-440.

"Combinatorial problems: reducibility and approximation," by S. Sahni and E. Horowitz, *Operations Research*, to appear.

"Complexity of decision problems based on finite two-person perfect-information games," by T. Schaefer, *Proc. Eighth Annual ACM Symposium on Theory of Computing,* May 1976, pp. 41-49.

"Some polynomial and integer divisibility problems are NP-Hard," by D. Plaisted, *Proc. 17th Annual Symp. on Found. of Comp.,* pp. 264-267 (1976).

"Traversal marker placement problems are NP-Complete," by S. Maheshwari, University of Colorado, Computer Science Technical Report #CU-CS-092-76, May 1976.

"A note on reductions to directed HC," by J. Seiferas, *8th Annual Princeton Conference on Information Sciences and Systems,* pp. 24-28, 1974.

"Two NP-complete problems in nonnegative integer programming," by G. Leuker, Princeton University Computer Science Laboratory, Technical Report TR-178, 1975. [exercise 44].

"The complexity of satisfiability problems," by T. Schaefer, *10th ACM Symposium on Theory of Computing*, pp. 216–226, 1978.

EXERCISES

1. Obtain a nondeterministic algorithm of complexity $O(n)$ to determine whether or not there is a subset of the n numbers a_i, $1 \leq i \leq n$ that sums to M.

2. (i) Show that the knapsack optimization problem reduces to the knapsack decision problem when all the p's, w's and M are integer and the complexity is measured as a function of input length. (Hint: if the input length is m then $\Sigma\, p_i \leq n2^m$ where n is the number of objects. Use a binary search to determine the optimal solution value).
   (ii) Let DK be an algorithm for the knapsack decision problem. Let R be the value of an optimal solution to the knapsack optimization problem. Show how to obtain a 0/1 assignment for the x_i, $1 \leq i \leq n$ such that $\Sigma\, p_ix_i = R$ and $\Sigma\, w_ix_i \leq M$ by making n applications of DK.

3. In conjunction with formula G in the proof of Cook's theorem (Section 11.2), obtain M for the following cases for instruction i. Note that M can contain at most $O(p(n))$ literals (as a function of n). Obtain M under the assumption that negative numbers are represented in ones complement. Show how the corresponding $G_{i,t}$'s may be transformed into CNF. The length of $G_{i,t}$ must increase by no more than a constant factor (say w^2) during this transformation.
   i) $Y \leftarrow Z$
   ii) $Y \leftarrow V - Z$
   iii) $Y \leftarrow V + Z$
   iv) $Y \leftarrow V * Z$
   v) $Y \leftarrow$ **choice** (0, 1)
   vi) $Y \leftarrow$ **choice** $(r{:}u)$ where r and u are variables.

4. Show that the clique optimization problem reduces to the clique decision problem.

5. Let SAT(E) be an algorithm to determine whether or not a propositional formula E in CNF is satisfiable. Show that if E is satisfiable and has n variables $x_1, x_2, \ldots, x_n$ then using SAT(E) n times one can determine a truth value assignment for the x_i's for which E is true.

6. Let SATY be the problem of determining whether a propositional formula in CNF having at most three literals per clause is satisfiable. Show that CNF satisfiability $\propto$ SATY (Hint: Show how to write a clause with more than three literals as the **and** of several clauses each containing at most three literals. For this you will have to introduce some new variables. Any assignment that satisfies the original clause must satisfy all the new clauses created).

7. Let SAT3 be similar to SATY (Exercise 6) except that each clause has exactly three literals. Show that SATY $\propto$ SAT3.

8. Let F be a propositional formula in CNF. Two literals x and y in F are *compatible* iff they are not in the same clause and $x \neq \bar{y}$. x and y are *incompatible* iff x and y are not compatible. Let SATINC be the problem of determining if a formula F in which each literal is incompatible with at most three other literals is satisfiable. Show that SAT3 $\propto$ SATINC.

9. Let 3-NODE COVER be the node cover decision problem of Section 11.3 restricted to graphs of degree 3. Show that SATINC $\propto$ 3-NODE COVER (see Exercise 8).

10. [Feedback Node Set]
    (a) Let $G = (V, E)$ be a directed graph. Let $S \subseteq V$ be a subset of vertices such that deletion of S and all edges incident to vertices in S results in a graph G' with no directed cycles. Such an S is a feedback node set. The size of S is the number of vertices in S. The feedback node set decision problem (FNS) is to determine for a given input k if G has a feedback node set of size at most k. Show that node cover decision problem $\propto$ FNS.
    (b) Write a polynomial time nondeterministic algorithm for FNS.

11. [Feedback Arc Set] Let $G = (V, E)$ be a directed graph. $S \subseteq E$ is a feedback arc set of G iff every directed cycle in G contains an edge in S. The feedback arc set decision problem (FAS) is to determine if G has a feedback arc set of size at most k. Show that node cover decision problem $\propto$ FAS.
    (b) Write a polynomial time nondeterministic algorithm for FAS.

12. The feedback node set optimization problem is to find a minimum feedback node set (see Exercise 10). Show that this problem reduces to FNS.

13. Show that the feedback arc set minimization problem reduces to FAS (Exercise 11).

14. [Hamiltonian Cycle] Let UHC be the problem of determining if in any given undirected graph G there exists an undirected cycle going through each vertex exactly once and returning to the start vertex. Show that DHC $\propto$ UHC (DHC is defined in Section 11.3).

15. Show UHC $\propto$ CNF satisfiability.

16. Show DHC $\propto$ CNF satisfiability.

17. [Hamiltonian Path] An i to j Hamiltonian path in a graph G is a path from

vertex i to vertex j that includes each vertex exactly once. Show that UHC is reducible to the problem of determining if G has an i to j hamiltonian path.

18. [Minimum Equivalent Graph] A directed graph $G = (V, E)$ is an equivalent graph of the directed graph $G' = (V, E')$ iff $E \subseteq E'$ and the transitive closures of G and G' are the same. G is a minimum equivalent graph iff $|E|$ is minimum amongst all equivalent graphs of G'. The minimum equivalent graph decision problem (MEG) is to determine if G' has a minimum equivalent graph with $|E| \leq k$ where k is some given input.
    (a) Show that DHC $\propto$ MEG.
    (b) Write a nondeterministic polynomial time algorithm for MEG.

19. [Clique Cover] The clique cover decision problem (CC) is to determine if G is the union of l or fewer cliques. Show that chromatic number decision problem $\propto$ CC.

20. [Set Cover] Let $F = \{S_j\}$ be a finite family of sets. Let $T \subseteq F$ be a subset of F. T is a cover of F iff

$$\bigcup_{S_i \in T} S_i = \bigcup_{S_i \in F} S_i.$$

The set cover decision problem is to determine if F has a cover T containing no more than k sets. Show that the node cover decision problem is reducible to this problem.

21. [Exact Cover] Let $F = \{S_j\}$ be as above. $T \subseteq F$ is an exact cover of F iff T is a cover of F and the sets in T are pairwise disjoint. Show that the chromatic number decision problem reduces to the problem of determining if F has an exact cover.

22. Show that SAT3 $\propto$ EXACT COVER (see Exercise 21).

23. [Hitting Set] Let F be as in Exercise 21. The hitting set problem is to determine if there exists a set H such that $|H \cap S_j| = 1$ for all $S_j \in F$. Show that exact cover $\propto$ hitting set.

24. [Tautology] A propositional formula is a tautology iff it is true for all possible truth assignments to its variables. The tautology problem is to determine whether or not a DNF formula is a tautology.
    (a) Show that CNF satisfiability $\propto$ DNF tautology.
    (b) Write a polynomial time nondeterministic algorithm TAUT(F) that terminates successfully iff F is not a tautology.

25. [Minimum Boolean Form] Let the length of a propositional formula be equal

to the sum of the number of literals in each clause. Two formulas F and G on variables $x_1, \ldots, x_n$ are equivalent if for all assignments to $x_1, \ldots, x_n$ F is true iff G is true. Show that deciding if F has an equivalent formula of length no more than k is NP-Hard. (Hint: Show DNF tautology reduces to this problem).

26. [Circuit Realization] Let C be a circuit made up of **and, or** and **not** gates. Let $x_1, \ldots, x_n$ be the inputs and f the output. Show that deciding if $f(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)$ where F is a propositional formula is NP-hard.

27. Show that determining if C is a minimum circuit (i.e. has a minimum number of gates, see Exercise 26) realizing a formula F is NP-hard.

28. [0/1-knapsack] Show that Partition $\propto$ 0/1-knapsack decision problem.

29. [Job Sequencing] Show that the job sequencing with deadlines problem (Chapter 8) is NP-hard.

30. Show that partition $\propto$ minimum finish time non-preemptive 3 processor flow shop schedule. Use only one job that has three nonzero tasks. All other jobs have only one nonzero task.

31. Show that partition $\propto$ minimum finish time non-preemptive 2 processor job shop schedule. Use only one job that has three nonzero tasks. All other jobs have only one nonzero task.

32. Let $J_1, \ldots, J_n$ be n jobs. Job i has a processing time t_i and a deadline d_i. Job i is not available for processing until time r_i. Show that deciding whether all n jobs can be processed on one machine without violating any deadline is NP-Hard. (Hint: Use partition).

33. Let J_i, $1 \leq i \leq n$ be n jobs as in the above problem. Assume $r_i = 0$, $1 \leq i \leq n$. Let f_i be the finish time of J_i in a one processor schedule S. The *tardiness* T_i of J_i is $\max\{0, f_i - d_i\}$. Let w_i, $1 \leq i \leq n$ be nonnegative weights associated with the J_i's. The total weighted tardiness is $\Sigma\, w_iT_i$. Show that finding a schedule minimizing $\Sigma\, w_iT_i$ is NP-hard. (Hint: Use partition).

34. Let J_i, $1 \leq i \leq n$ be n jobs. Job J_i has a processing time of t_i. Its processing cannot begin until time r_i. Let w_i be a weight associated with J_i. Let f_i be the finish time of J_i in a one processor schedule S. Show that finding a one processor schedule that minimizes $\Sigma\, w_if_i$ is NP-hard.

35. [Quadratic Programming] Show that finding the maximum of a function $f(x_1, \ldots, x_n)$ subject to the linear constraints $\Sigma_{1 \leq j \leq n}\, a_{ij}x_j \leq b_i$, $1 \leq i \leq n$

and $x_i \geq 0$, $1 \leq i \leq n$ is NP-hard. The function f is restricted to be of the form $\Sigma\, c_i x_i^2 + \Sigma\, d_i x_i$.

36. Show that the problem of obtaining optimal finish time preemptive schedules for a two processer flow shop is NP-hard when jobs are released at two different times R_1 and R_2. Jobs released at R_i cannot be scheduled before R_i.

37. Let $G = (V, E)$ be a graph. Let $w(i, j)$ be a weighting function for the edges of G. A *cut* of G is a subset $S \subseteq V$. The *weight* of a cut is

$$\sum_{i \in S, j \notin S} w(i, j).$$

A *max-cut* is a cut of maximum weight. Show that the problem of determining the weight of a max-cut is NP-hard.

38. [Plant Location] Let S_i, $1 \leq i \leq n$ be n possible sites at which plants may be located. At each site at most one plant can be located. If a plant is located at site S_i then a fixed cost F_i is incurred. This is the cost of setting up the plant. A plant located at S_i will have a maximum production capacity of C_i. There are n destinations, D_i, $1 \leq i \leq m$, to which products have to be shipped. The demand at D_i is d_i, $1 \leq i \leq m$. The per unit cost of shipping a product from site i to destination j is c_{ij}. A destination may be supplied from many plants. Define $y_i = 0$ if no plant is located site i and $y_i = 1$ otherwise. Let x_{ij} be the number of units of the product shipped from S_i to D_j. Then, the total cost is

$$\sum_i F_i y_i + \sum_i \sum_j c_{ij} x_{ij}, \qquad \sum_i x_{ij} = d_j \quad \text{and} \quad \sum_j x_{ij} \leq C_i y_i.$$

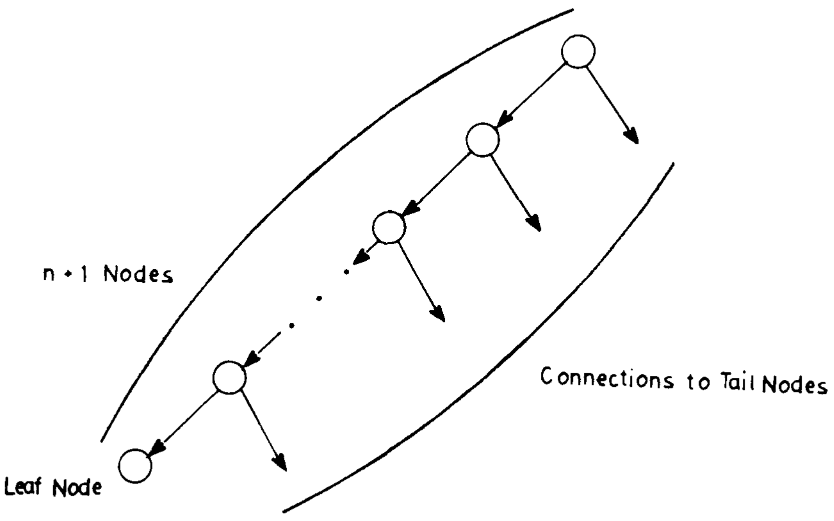
All x_{ij} are non-negative integers. We may assume that $\Sigma\, C_{ij} \geq \Sigma\, d_i$. Show that finding y_i and x_{ij} so that the total cost is minimized is NP-hard.

39. [Concentrator Location] This problem is very similar to the plant location problem (Exercise 38). The only difference is that each destination may be supplied by only 1 plant. When this restriction is imposed, the plant location problem becomes the concentrator location problem arising in computer network design. The destinations represent computer terminals. The plants represent the concentration of information from the terminals which it supplies. Show that the concentrator location problem is NP-hard under each of the following conditions:

i) $n = 2, C_1 = C_2, F_1 = F_2$ (Hint: use Partition)
ii) $F_i/C_i = F_{i+1}/C_{i+1}$, $1 \leq i < n$, $d_i = 1$ (Hint: use Exact Cover)

40. [Steiner Trees] Let T be a tree and R a subset of the vertices in T. Let $w(i, j)$ be the weight of edge (i, j) in T. If (i, j) is not an edge in T then $w(i, j) = \infty$. A Steiner tree is a subtree of T that includes the vertex set R. It may include other vertices too. Its cost is the sum of the weights of the edges in it. Show that finding a minimum cost Steiner tree is NP-hard.

41. a) How should the proof of Theorem 11.14 be modified to permit recomputation of shared nodes.
    b) [Ravi Sethi] Modify the proof of Theorem 11.14 so that it holds for level 1 dags representing expressions in which all operators are noncommutative. Hint: designate the sucessor vertex on a chain to be the left child of its predecessor vertex and use the following $n + 1$ node binary tree to connect together the tail nodes of the n chains:



    c) Show that optimal code generation is NP-hard for leaf dags on an infinite register machine. (Hint: Use FNS).

42. Assume that P is a parallel assignment statement $(v_1, \ldots, v_n) \leftarrow (e_1, \ldots, e_n)$ where each e_i is a simple variable and the v_i's are distinct. For convenience, assume that the distinct variables in P are $v_1, \ldots, v_m$ with $m \geq n$ and that $E = (i_1, i_2, \ldots, i_n)$ is a set of indices such that $e_{i_j} = v_{i_j}$. Write an $O(n)$ algorithm to find an optimal realization for P.

43. Let $F = \{S_j\}$ be a finite family of sets. Let $T \leq F$ be a subfamily of F. The

size of T, $|T|$, is the number of sets in T. Let S_i, S_j be two sets in T. S_i and S_j are disjoint iff $S_i \cap S_j = \phi$. T is a disjoint subset of F iff every pair of sets in T are disjoint. The set packing problem is to determine a disjoint subfamily T of maximum size. Show that clique α set packing.

44. Show that the following decision problem is NP-complete.
Input: Positive integers n; w_i, $1 \le i \le n$ and M.
Decision: Do there exist nonnegative integers $x_i \ge 0$, $1 \le i \le n$ such that

$$\sum_{1 \le i \le n} w_i x_i = M$$