FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ALGORITHMS

Ellis Horowitz
University of Southern California

Sartaj Sahni
University of Florida

Sanguthevar Rajasekaran
University of Florida

Computer Science Press
An imprint of W. H. Freeman and Company
New York

Chapter 7

BACKTRACKING

7.1 THE GENERAL METHOD

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. The name backtrack was first coined by D. H. Lehmer in the 1950s. Early workers who studied the process were R. J. Walker, who gave an algorithmic account of it in 1960, and S. Golomb and L. Baumert who presented a very general description of it as well as a variety of applications.

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function* $P(x_1, \ldots, x_n)$. Sometimes it seeks all vectors that satisfy P. For example, sorting the array of integers in $a[1 : n]$ is a problem whose solution is expressible by an n-tuple, where x_i is the index in a of the ith smallest element. The criterion function P is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i < n$. The set S_i is finite and includes the integers 1 through n. Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n-tuple. In this chapter we study a collection of problems whose solutions are best done using backtracking.

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P. The *brute force approach* would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called

bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0, 1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

Example 7.1 [8-queens] A classic combinatorial problem is to place eight queens on an 8×8 chessboard so that no two "attack," that is, so that no two of them are on the same row, column, or diagonal. Let us number the rows and columns of the chessboard 1 through 8 (Figure 7.1). The queens can also be numbered 1 through 8. Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i. All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1, \ldots, x_8)$, where x_i is the column on which queen i is placed. The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples. The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal. The first of these two constraints implies that all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8). This realization reduces the size of the solution space from 8^8 tuples to 8! tuples. We see later how to formulate the second constraint in terms of the x_i. Expressed as an 8-tuple, the solution in Figure 7.1 is (4, 6, 8, 2, 7, 1, 3, 5). □

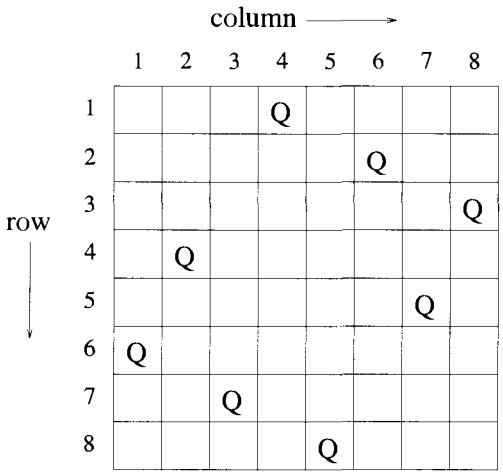


Figure 7.1 One solution to the 8-queens problem

Example 7.2 [Sum of subsets] Given positive numbers w_i, $1 \leq i \leq n$, and m, this problem calls for finding all subsets of the w_i whose sums are m. For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, and $m = 31$, then the desired subsets are (11, 13, 7) and (24, 7). Rather than represent the solution vector by the w_i which sum to m, we could represent the solution vector by giving the indices of these w_i. Now the two solutions are described by the vectors (1, 2, 4) and (3, 4). In general, all solutions are k-tuples $(x_1, x_2, \ldots, x_k)$, $1 \leq k \leq n$, and different solutions may have different-sized tuples. The explicit constraints require $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$. The implicit constraints require that no two be the same and that the sum of the corresponding w_i's be m. Since we wish to avoid generating multiple instances of the same subset (e.g., (1, 2, 4) and (1, 4, 2) represent the same subset), another implicit constraint that is imposed is that $x_i < x_{i+1}$, $1 \leq i < k$.

In another formulation of the sum of subsets problem, each solution subset is represented by an n-tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0, 1\}$, $1 \leq i \leq n$. Then $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen. The solutions to the above instance are (1, 1, 0, 1) and (0, 0, 1, 1). This formulation expresses all solutions using a fixed-sized tuple. Thus we conclude that there may be several ways to formulate a problem so that all solutions are tuples that satisfy some constraints. One can verify that for both of the above formulations, the solution space consists of 2^n distinct tuples. □

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a *tree organization* for the solution space. For a given solution space many tree organizations may be possible. The next two examples examine some of the ways to organize a solution into a tree.

Example 7.3 [n-queens] The n-queens problem is a generalization of the 8-queens problem of Example 7.1. Now n queens are to be placed on an $n \times n$ chessboard so that no two attack; that is, no two queens are on the same row, column, or diagonal. Generalizing our earlier discussion, the solution space consists of all $n!$ permutations of the n-tuple $(1, 2, \ldots, n)$. Figure 7.2 shows a possible tree organization for the case $n = 4$. A tree such as this is called a *permutation tree*. The edges are labeled by possible values of x_i. Edges from level 1 to level 2 nodes specify the values for x_1. Thus, the leftmost subtree contains all solutions with $x_1 = 1$; its leftmost subtree contains all solutions with $x_1 = 1$ and $x_2 = 2$, and so on. Edges from level i to level $i+1$ are labeled with the values of x_i. The solution space is defined by all paths from the root node to a leaf node. There are $4! = 24$ leaf nodes in the tree of Figure 7.2. □

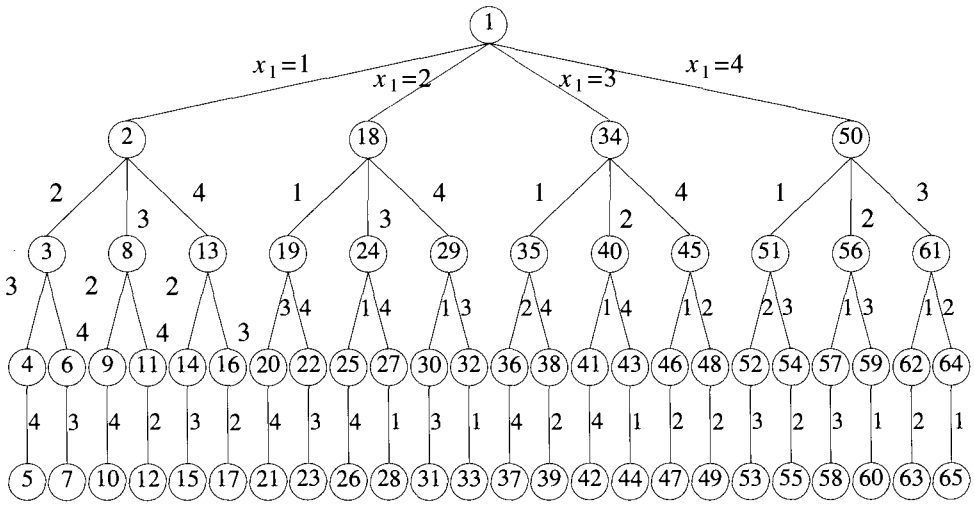


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Example 7.4 [Sum of subsets] In Example 7.2 we gave two possible formulations of the solution space for the sum of subsets problem. Figures 7.3 and 7.4 show a possible tree organization for each of these formulations for the case $n = 4$. The tree of Figure 7.3 corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i. At each node, the solution space is partitioned into subsolution spaces. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are () (this corresponds to the empty path from the root to itself), (1), (1,2), (1,2,3), (1,2,3,4), (1,2,4), (1,3,4), (2), (2,3), and so on. Thus, the leftmost subtree defines all subsets containing w_1, the next subtree defines all subsets containing w_2 but not w_1, and so on.

The tree of Figure 7.4 corresponds to the fixed tuple size formulation. Edges from level i nodes to level $i+1$ nodes are labeled with the value of x_i, which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_1, the right subtree defines all subsets not containing w_1, and so on. Now there are 2^4 leaf nodes which represent 16 possible tuples. □

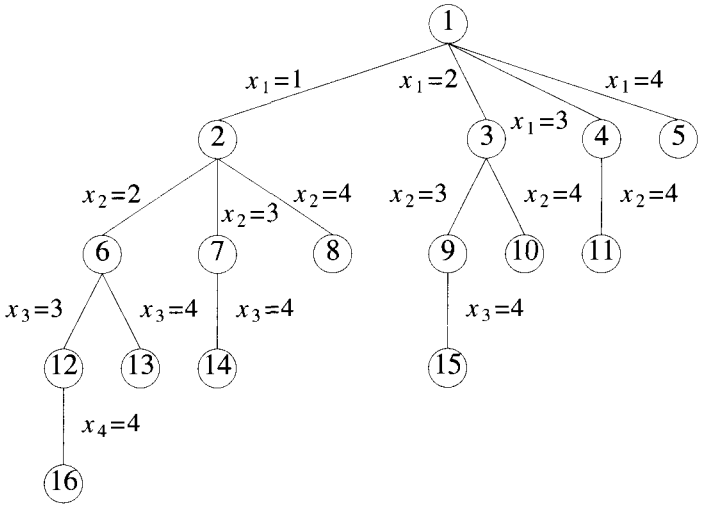


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

At this point it is useful to develop some terminology regarding tree organizations of solution spaces. Each node in this tree defines a *problem*

state. All paths from the root to other nodes define the *state space* of the problem. *Solution states* are those problem states s for which the path from the root to s defines a tuple in the solution space. In the tree of Figure 7.3 all nodes are solution states whereas in the tree of Figure 7.4 only leaf nodes are solution states. *Answer states* are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem. The tree organization of the solution space is referred to as the *state space tree*.

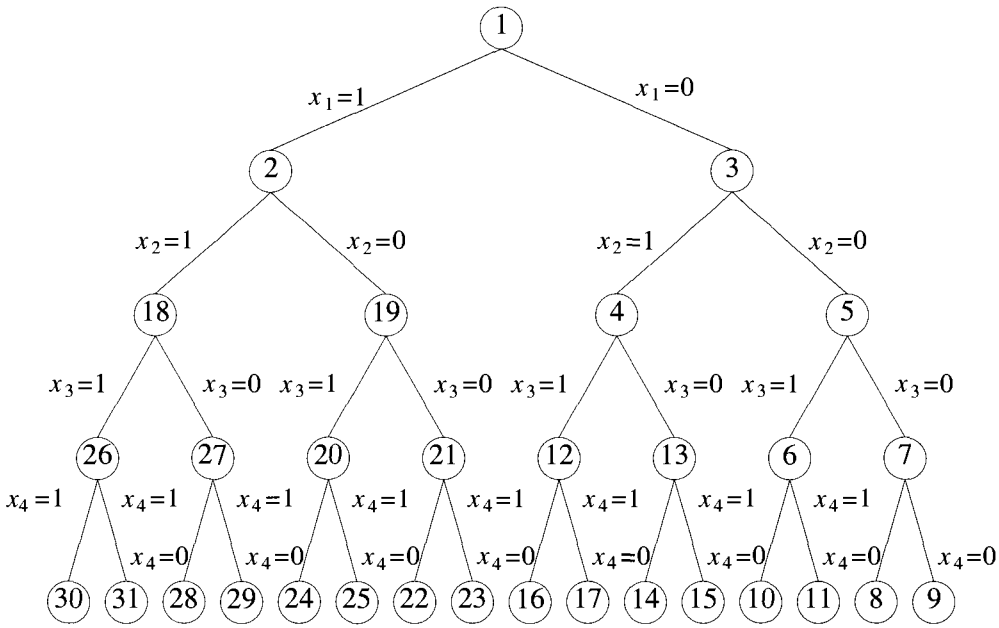


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

At each internal node in the space tree of Examples 7.3 and 7.4 the solution space is partitioned into disjoint sub-solution spaces. For example, at node 1 of Figure 7.2 the solution space is partitioned into four disjoint sets. Subtrees 2, 18, 34, and 50 respectively represent all elements of the solution space with $x_1 = 1$, 2, 3, and 4. At node 2 the sub-solution space with $x_1 = 1$ is further partitioned into three disjoint sets. Subtree 3 represents all solution space elements with $x_1 = 1$ and $x_2 = 2$. For all the state space trees we study in this chapter, the solution space is partitioned into disjoint sub-solution spaces at each internal node. It should be noted that this is

not a requirement on a state space tree. The only requirement is that every element of the solution space be represented by at least one node in the state space tree.

The state space tree organizations described in Example 7.4 are called *static trees*. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instances. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called *dynamic trees*. As an example, consider the fixed tuple size formulation for the sum of subsets problem (Example 7.4). Using a dynamic tree organization, one problem instance with $n = 4$ can be solved by means of the organization given in Figure 7.4. Another problem instance with $n = 4$ can be solved by means of a tree in which at level 1 the partitioning corresponds to $x_2 = 1$ and $x_2 = 0$. At level 2 the partitioning could correspond to $x_1 = 1$ and $x_1 = 0$, at level 3 it could correspond to $x_3 = 1$ and $x_3 = 0$, and so on. We see more of dynamic trees in Sections 7.6 and 8.3.

Once a state space tree has been conceived of for any problem, this problem can be solved by systematically generating the problem states, determining which of these are solution states, and finally determining which solution states are answer states. There are two fundamentally different ways to generate the problem states. Both of these begin with the root node and generate other nodes. A node which has been generated and all of whose children have not yet been generated is called a *live node*. The live node whose children are currently being generated is called the E-node (node being expanded). A *dead node* is a generated node which is not to be expanded further or all of whose children have been generated. In both methods of generating problem states, we have a list of live nodes. In the first of these two methods as soon as a new child C of the current E-node R is generated, this child will become the new E-node. Then R will become the E-node again when the subtree C has been fully explored. This corresponds to a depth first generation of the problem states. In the second state generation method, the E-node remains the E-node until it is dead. In both methods, *bounding functions* are used to kill live nodes without generating all their children. This is done carefully enough that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions. Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the E-node remains the E-node until it is dead lead to *branch-and-bound* methods. The branch-and-bound technique is discussed in Chapter 8.

The nodes of Figure 7.2 have been numbered in the order they would be generated in a depth first generation process. The nodes in Figures 7.3 and

7.4 have been numbered according to two generation methods in which the E-node remains the E-node until it is dead. In Figure 7.3 each new node is placed into a queue. When all the children of the current E-node have been generated, the next node at the front of the queue becomes the new E-node. In Figure 7.4 new nodes are placed into a stack instead of a queue. Current terminology is not uniform in referring to these two alternatives. Typically the queue method is called breadth first generation and the stack method is called D-search (depth search).

Example 7.5 [4-queens] Let us see how backtracking works on the 4-queens problem of Example 7.3. As a bounding function, we use the obvious criteria that if $(x_1, x_2, \ldots, x_i)$ is the path to the current E-node, then all children nodes with parent-child labelings x_{i+1} are such that $(x_1, \ldots, x_{i+1})$ represents a chessboard configuration in which no two queens are attacking. We start with the root node as the only live node. This becomes the E-node and the path is (). We generate one child. Let us assume that the children are generated in ascending order. Thus, node number 2 of Figure 7.2 is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). Figure 7.5 shows the board configurations as backtracking proceeds. Figure 7.5 shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen which were tried and rejected because another queen was attacking. In Figure 7.5(b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In Figure 7.5(c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In Figure 7.5(d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure 7.5 (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.

Figure 7.6 shows the part of the tree of Figure 7.2 that is generated. Nodes are numbered in the order in which they are generated. A node that gets killed as a result of the bounding function has a B under it. Contrast this tree with Figure 7.2 which contains 31 nodes. □

With this example completed, we are now ready to present a precise formulation of the backtracking process. We continue to treat backtracking in a general way. We assume that all answer nodes are to be found and not just one. Let $(x_1, x_2, \ldots, x_i)$ be a path from the root to a node in a state space tree. Let $T(x_1, x_2, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \ldots, x_{i+1})$ is also a path to a problem state. $T(x_1, x_2, \ldots, x_n) = \emptyset$.

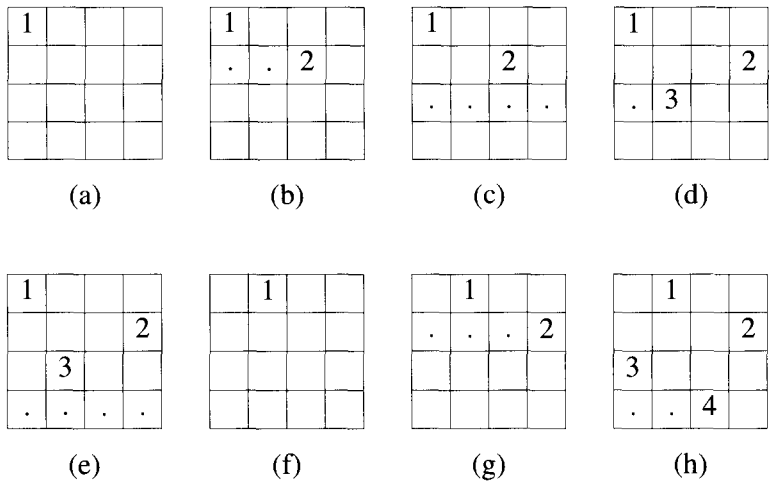


Figure 7.5 Example of a backtrack solution to the 4-queens problem

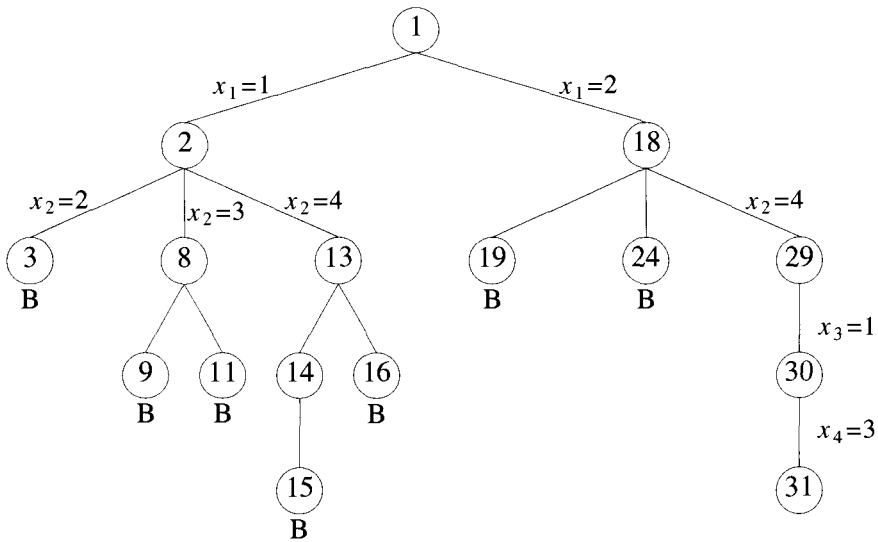


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

We assume the existence of bounding function B_{i+1} (expressed as predicates) such that if $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ is false for a path $(x_1, x_2, \ldots, x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node. Thus the candidates for position $i+1$ of the solution vector $(x_1, \ldots, x_n)$ are those values which are generated by T and satisfy B_{i+1}. Algorithm 7.1 presents a recursive formulation of the backtracking technique. It is natural to describe backtracking in this way since it is essentially a postorder traversal of a tree (see Section 6.1). This recursive version is initially invoked by

```
Backtrack(1);
```

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

Algorithm 7.1 Recursive backtracking algorithm

The solution vector $(x_1, \ldots, x_n)$, is treated as a global array $x[1:n]$. All the possible elements for the kth position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector $(x_1, \ldots, x_{k-1})$. Each time x_k is attached, a check is made to determine whether a solution has been found. Then the algorithm is recursively invoked. When the **for** loop of line 7 is exited, no more values for x_k exist and the current copy of Backtrack ends. The last unresolved call now resumes, namely, the one that continues to examine the remaining elements assuming only $k-2$ values have been set.

Note that this algorithm causes *all* solutions to be printed and assumes that tuples of various sizes may make up a solution. If only a single solution is desired, then a flag can be added as a parameter to indicate the first occurrence of success.

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1])  and B_k(x[1], ..., x[k]) is true) then
11          {
12                  if (x[1], ..., x[k] is a path to an answer node)
13                      then write (x[1 : k]);
14                  k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

Algorithm 7.2 General iterative backtracking method

An iterative version of Algorithm 7.1 appears in Algorithm 7.2. Note that $T()$ will yield the set of all possible values that can be placed as the first component x_1 of the solution vector. The component x_1 will take on those values for which the bounding function $B_1(x_1)$ is true. Also note how the elements are generated in a depth first manner. The variable k is continually incremented and a solution vector is grown until either a solution is found or no untried value of x_k remains. When k is decremented, the algorithm must resume the generation of possible elements for the kth position that have not yet been tried. Therefore one must develop a procedure that generates these values in some order. If only one solution is desired, replacing **write** $(x[1 : k])$; with **{write** $(x[1 : k])$; **return;}** suffices.

The efficiency of both the backtracking algorithms we've just seen depends very much on four factors: (1) the time to generate the next x_k, (2) the number of x_k satisfying the explicit constraints, (3) the time for the bounding functions B_k, and (4) the number of x_k satisfying the B_k. Bound-

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                or (Abs(x[j] − i) = Abs(j − k)))
10                   // or in the same diagonal
11               then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

At this point we might wonder how effective function NQueens is over the brute force approach. For an 8×8 chessboard there are $\binom{64}{8}$ possible ways to place 8 pieces, or approximately 4.4 billion 8-tuples to examine. However, by allowing only placements of queens on distinct rows and columns, we require the examination of at most 8!, or only 40,320 8-tuples.

We can use Estimate to estimate the number of nodes that will be generated by NQueens. Note that the assumptions that are needed for Estimate do hold for NQueens. The bounding function is static. No change is made to the function as the search proceeds. In addition, all nodes on the same level of the state space tree have the same degree. In Figure 7.8 we see five 8×8 chessboards that were created using Estimate.

As required, the placement of each queen on the chessboard was chosen randomly. With each choice we kept track of the number of columns a queen could legitimately be placed on. These numbers are listed in the vector beneath each chessboard. The number following the vector represents the value that function Estimate would produce from these sizes. The average of these five trials is 1625. The total number of nodes in the 8-queens state space tree is

$$1 + \sum_{j=0}^{7} \left[\Pi_{i=0}^{j} (8 - i) \right] = 69,281$$

So the estimated number of unbounded nodes is only about 2.34% of the total number of nodes in the 8-queens state space tree. (See the exercises for more ideas about the efficiency of NQueens.)

EXERCISES

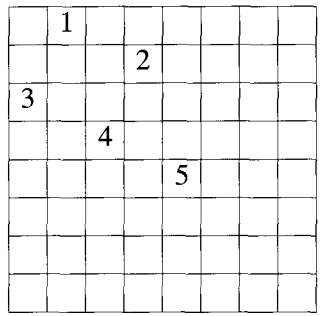
1. Algorithm NQueens can be made more efficient by redefining the function Place(k, i) so that it either returns the next legitimate column on which to place the kth queen or an illegal value. Rewrite both functions (Algorithms 7.4 and 7.5) so they implement this alternate strategy.

2. For the n-queens problem we observe that some solutions are simply reflections or rotations of others. For example, when $n = 4$, the two solutions given in Figure 7.9 are equivalent under reflection.

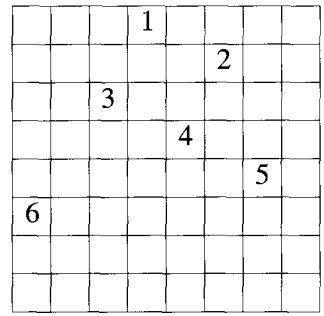
   Observe that for finding inequivalent solutions the algorithm need only set $x[1] = 2, 3, \ldots, \lceil n/2 \rceil$.

   (a) Modify NQueens so that only inequivalent solutions are computed.

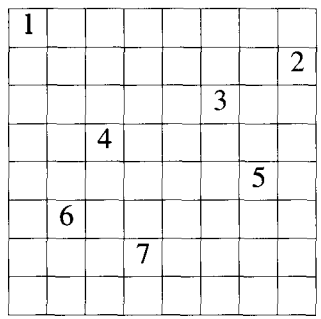
   (b) Run the n-queens program devised above for $n = 8$, 9, and 10. Tabulate the number of solutions your program finds for each value of n.



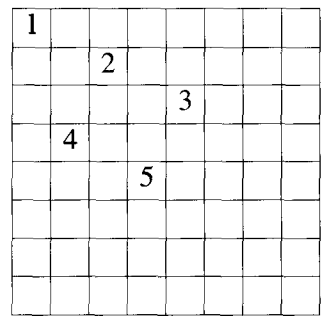
(8,5,4,3,2) = 1649



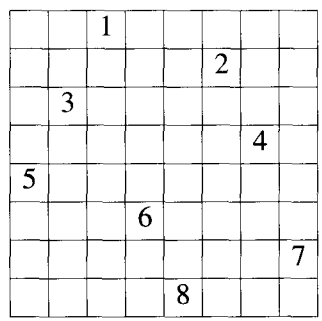
(8,5,3,1,2,1) = 769



(8,6,4,2,1,1,1) = 1401



(8,6,4,3,2) = 1977



(8,5,3,2,2,1,1,1) = 2329

Figure 7.8 Five walks through the 8-queens problem plus estimates of the tree size

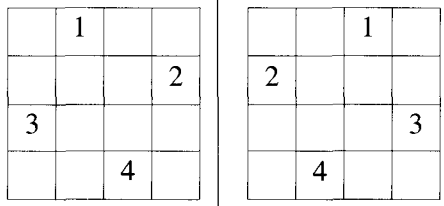


Figure 7.9 Equivalent solutions to the 4-queens problem

3. Given an $n \times n$ chessboard, a knight is placed on an arbitrary square with coordinates (x, y). The problem is to determine $n^2 - 1$ knight moves such that every square of the board is visited once if such a sequence of moves exists. Present an algorithm to solve this problem.

7.3 SUM OF SUBSETS

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem. Examples 7.2 and 7.4 showed how we could formulate this problem using either fixed- or variable-sized tuples. We consider a backtracking solution using the fixed tuple size strategy. In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.

The children of any node in Figure 7.4 are easily generated. For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Clearly $x_1, \ldots, x_k$ cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in nondecreasing order. In this case $x_1, \ldots, x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

The bounding functions we use are therefore

$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m \tag{7.1}$$

Since our algorithm will not make use of B_n, we need not be concerned by the appearance of w_{n+1} in this function. Although we have now specified all that is needed to directly use either of the backtracking schemas, a simpler algorithm results if we tailor either of these schemas to the problem at hand. This simplification results from the realization that if $x_k = 1$, then

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

For simplicity we refine the recursive schema. The resulting algorithm is SumOfSub (Algorithm 7.6).

Algorithm SumOfSub avoids computing $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variables s and r respectively. *The algorithm assumes* $w_1 \leq m$ *and* $\sum_{i=1}^{n} w_i \geq m$. The initial call is SumOfSub$(0, 1, \sum_{i=1}^{n} w_i)$. It is interesting to note that the algorithm does not explicitly use the test $k > n$ to terminate the recursion. This test is not needed as on entry to the algorithm, $s \neq m$ and $s + r \geq m$. Hence, $r \neq 0$ and so k can be no greater than n. Also note that in the **else if** statement (line 11), since $s + w_k < m$ and $s + r \geq m$, it follows that $r \neq w_k$ and hence $k + 1 \leq n$. Observe also that if $s + w_k = m$ (line 9), then $x_{k+1}, \ldots, x_n$ must be zero. These zeros are omitted from the output of line 9. In line 11 we do not test for $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$, as we already know $s + r \geq m$ and $x_k = 1$.

Example 7.6 Figure 7.10 shows the portion of the state space tree generated by function SumOfSub while working on the instance $n = 6$, $m = 30$, and $w[1 : 6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s, k, and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes A, B, and C the output is respectively (1, 1, 0, 0, 1), (1, 0, 1, 1), and (0, 0, 1, 0, 0, 1). Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). □

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

EXERCISES

1. Prove that the size of the set of all subsets of n elements is 2^n.

2. Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Do this using SumOfSub. Draw the portion of the state space tree that is generated.

3. With $m = 35$, run SumOfSub on the data (a) $w = \{5, 7, 10, 12, 15, 18, 20\}$, (b) $w = \{20, 18, 15, 12, 10, 7, 5\}$, and (c) $w = \{15, 7, 20, 5, 18, 10, 12\}$. Are there any discernible differences in the computing times?

4. Write a backtracking algorithm for the sum of subsets problem using the state space tree corresponding to the variable tuple size formulation.

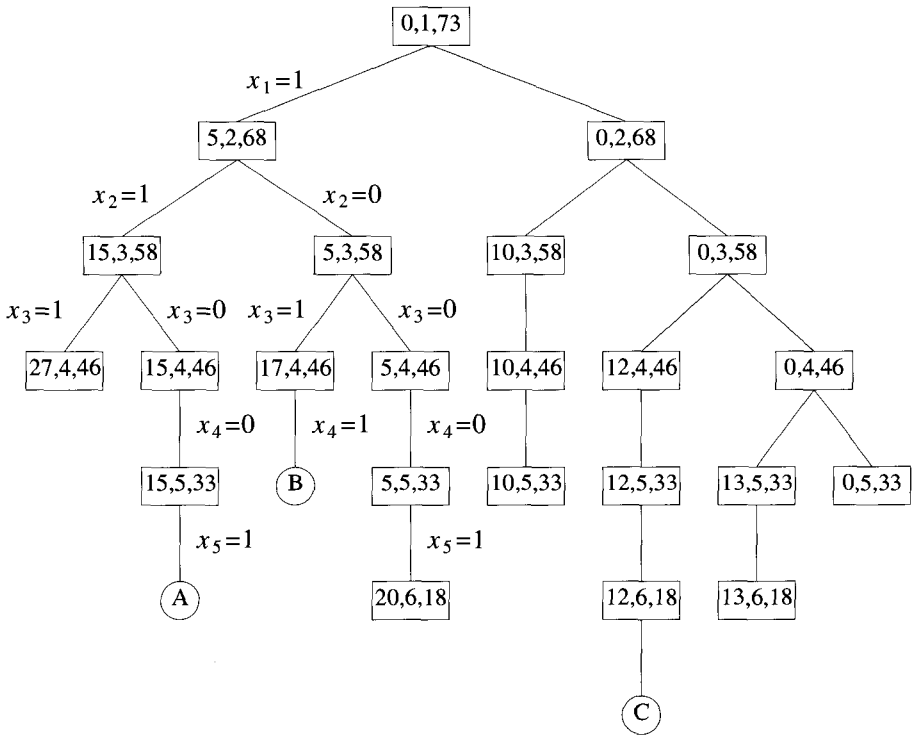


Figure 7.10 Portion of state space tree generated by SumOfSub

7.4 GRAPH COLORING

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m-colorability decision* problem and it is discussed again in Chapter 11. Note that if d is the degree of the given graph, then it can be colored with $d + 1$ colors. The *m-colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

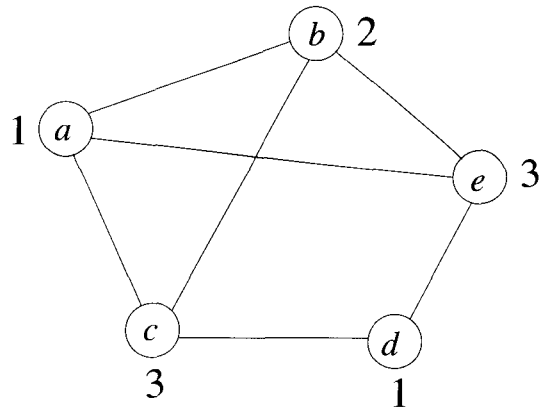


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

Suppose we represent a graph by its adjacency matrix $G[1 : n, 1 : n]$, where $G[i, j] = 1$ if (i, j) is an edge of G, and $G[i, j] = 0$ otherwise. The colors are represented by the integers $1, 2, \ldots, m$ and the solutions are given by the n-tuple $(x_1, \ldots, x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a tree of degree m and height $n + 1$. Each node at level i has m children corresponding to the m possible assignments to x_i, $1 \leq i \leq n$. Nodes at

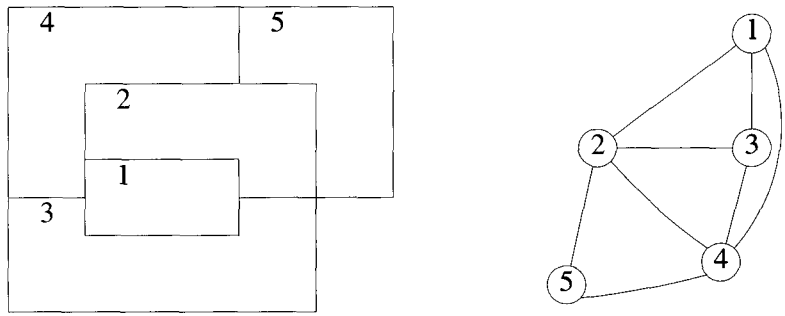


Figure 7.12 A map and its planar graph representation

level $n + 1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array* $x[\,]$ *to zero*, and then invoking the statement mColoring(1);.

Notice the similarity between this algorithm and the general form of the recursive backtracking schema of Algorithm 7.1. Function NextValue (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k, and then calls mColoring recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Algorithm 7.7 Finding all m-colorings of a graph

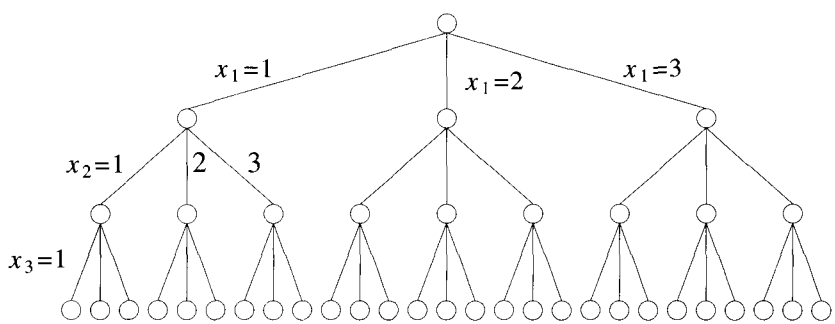


Figure 7.13 State space tree for mColoring when n =3 and m = 3

```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k − 1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

Algorithm 7.8 Generating a next color

EXERCISE

1. Program and run mColoring (Algorithm 7.7) using as data the complete graphs of size $n = 2$, 3, 4, 5, 6, and 7. Let the desired number of colors be $k = n$ and $k = n/2$. Tabulate the computing times for each value of n and k.

7.5 HAMILTONIAN CYCLES

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices

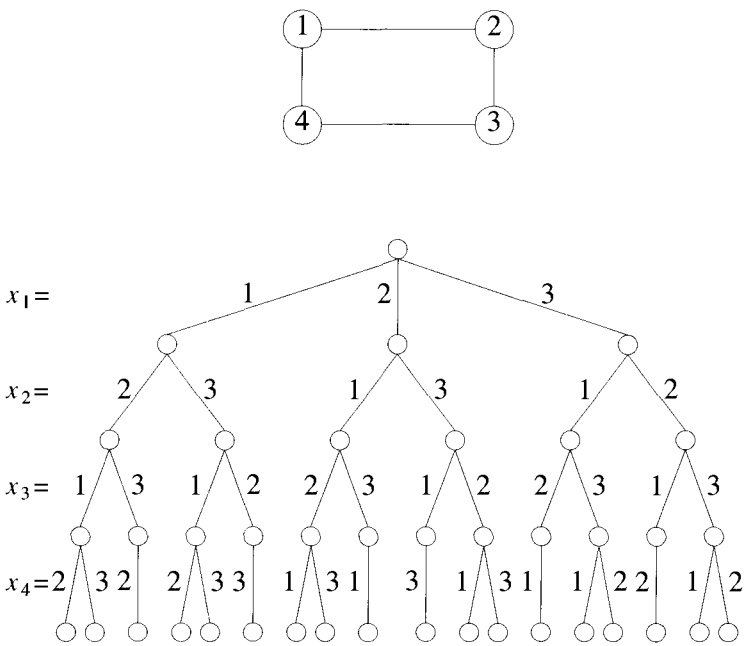


Figure 7.14 A 4-node graph and all possible 3-colorings

of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle. We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots, x_{k-1}$ have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1. We begin by presenting function NextValue(k) (Algorithm 7.9), which determines a possible next

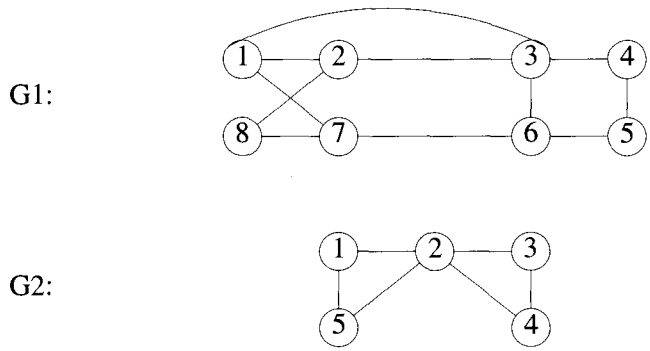


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

vertex for the proposed cycle.

Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian cycles (Algorithm 7.10). This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

Recall from Section 5.9 the traveling salesperson problem which asked for a tour that has minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists. If the common edge cost is c, the cost of a tour is cn since there are n edges in a Hamiltonian cycle.

EXERCISES

1. Determine the order of magnitude of the worst-case computing time for the backtracking procedure that finds all Hamiltonian cycles.

2. Draw the portion of the state space tree generated by Algorithm 7.10 for the graph $G1$ of Figure 7.15.

3. Generalize Hamiltonian so that it processes a graph whose edges have costs associated with them and finds a Hamiltonian cycle with minimum cost. You can assume that all edge costs are positive.

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                                  // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```

Algorithm 7.9 Generating a next vertex

```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Algorithm 7.10 Finding all Hamiltonian cycles

7.6 KNAPSACK PROBLEM

In this section we reconsider a problem that was defined and solved by a dynamic programming algorithm in Chapter 5, the 0/1 knapsack optimization problem. Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized} \tag{7.2}$$

The x_i's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's. Thus the solution space is the same as that for the sum of subsets problem. Two possible tree organizations are possible. One corresponds to the fixed tuple size formulation (Figure 7.4) and the other to the variable tuple size formulation (Figure 7.3). Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than

the value of the best solution determined so far, then that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \leq i \leq k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$ and using the greedy algorithm of Section 4.2 to solve the relaxed problem. Function Bound(cp, cw, k) (Algorithm 7.11) determines an upper bound on the best solution obtainable by expanding any node Z at level $k+1$ of the state space tree. The object weights and profits are $w[i]$ and $p[i]$. It is assumed that $p[i]/w[i] \geq p[i+1]/w[i+1]$, $1 \leq i < n$.

```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
9           if (c < m) then b := b + p[i];
10          else return b + (1 − (c − m)/w[i]) ∗ p[i];
11      }
12      return b;
13  }
```

Algorithm 7.11 A bounding function

From Bound it follows that the bound for a feasible left child of a node Z is the same as that for Z. Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is BKnap (Algorithm 7.12). It was obtained from the recursive backtracking schema. Initially set $fp := -1$;. This algorithm is invoked as

```
BKnap(1, 0, 0);
```

When $fp \neq -1$, $x[i]$, $1 \leq i \leq n$, is such that $\sum_{i=1}^{n} p[i]x[i] = fp$. In lines 8 to 18 left children are generated. In line 20, Bound is used to test whether a

```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

right child should be generated. The path $y[i]$, $1 \leq i \leq k$, is the path to the current node. The current weight $cw = \sum_{i=1}^{k-1} w[i]y[i]$ and $cp = \sum_{i=1}^{k-1} p[i]y[i]$. In lines 13 to 17 and 23 to 27 the solution vector is updated if need be.

So far, all our backtracking algorithms have worked on a static state space tree. We now see how a dynamic state space tree can be used for the knapsack problem. One method for dynamically partitioning the solution space is based on trying to obtain an optimal solution using the greedy algorithm of Section 4.2. We first replace the integer constraint $x_i = 0$ or 1 by the constraint $0 \leq x_i \leq 1$. This yields the relaxed problem

$$\max \sum_{1 \leq i \leq n} p_i x_i \quad \text{subject to} \quad \sum_{1 \leq i \leq n} w_i x_i \leq m \tag{7.3}$$

$$0 \leq x_i \leq 1, \qquad 1 \leq i \leq n$$

If the solution generated by the greedy method has all x_i's equal to zero or one, then it is also an optimal solution to the original 0/1 knapsack problem. If this is not the case, then exactly one x_i will be such that $0 < x_i < 1$. We partition the solution space of (7.2) into two subspaces. In one $x_i = 0$ and in the other $x_i = 1$. Thus the left subtree of the state space tree will correspond to $x_i = 0$ and the right to $x_i = 1$. In general, at each node Z of the state space tree the greedy algorithm is used to solve (7.3) under the added restrictions corresponding to the assignments already made along the path from the root to this node. In case the solution is all integer, then an optimal solution for this node has been found. If not, then there is exactly one x_i such that $0 < x_i < 1$. The left child of Z corresponds to $x_i = 0$, and the right to $x_i = 1$.

The justification for this partitioning scheme is that the noninteger x_i is what prevents the greedy solution from being a feasible solution to the 0/1 knapsack problem. So, we would expect to reach a feasible greedy solution quickly by forcing this x_i to be integer. Choosing left branches to correspond to $x_i = 0$ rather than $x_i = 1$ is also justifiable. Since the greedy algorithm requires $p_j/w_j \geq p_{j+1}/w_{j+1}$, we would expect most objects with low index (i.e., small j and hence high density) to be in an optimal filling of the knapsack. When x_i is set to zero, we are not preventing the greedy algorithm from using any of the objects with $j < i$ (unless x_j has already been set to zero). On the other hand, when x_i is set to one, some of the x_j's with $j < i$ will not be able to get into the knapsack. Therefore we expect to arrive at an optimal solution with $x_i = 0$. So we wish the backtracking algorithm to try this alternative first. Hence the left subtree corresponds to $x_i = 0$.

Example 7.7 Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: $p = \{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. The greedy

solution corresponding to the root node (i.e., Equation (7.3)) is $x = \{1, 1, 1, 1, 1, 21/45, 0, 0\}$. Its value is 164.88. The two subtrees of the root correspond to $x_6 = 0$ and $x_6 = 1$, respectively (Figure 7.16). The greedy solution at node 2 is $x = \{1, 1, 1, 1, 1, 0, 21/45, 0\}$. Its value is 164.66. The solution space at node 2 is partitioned using $x_7 = 0$ and $x_7 = 1$. The next E-node is node 3. The solution here has $x_8 = 21/55$. The partitioning now is with $x_8 = 0$ and $x_8 = 1$. The solution at node 4 is all integer so there is no need to expand this node further. The best solution found so far has value 139 and $x = \{1, 1, 1, 1, 1, 0, 0, 0\}$. Node 5 is the next E-node. The greedy solution for this node is $x = \{1, 1, 1, 22/23, 0, 0, 0, 1\}$. Its value is 159.56. The partitioning is now with $x_4 = 0$ and $x_4 = 1$. The greedy solution at node 6 has value 156.66 and $x_5 = 2/3$. Next, node 7 becomes the E-node. The solution here is $\{1, 1, 1, 0, 0, 0, 0, 1\}$. Its value is 128. Node 7 is not expanded as the greedy solution here is all integer. At node 8 the greedy solution has value 157.71 and $x_3 = 4/7$. The solution at node 9 is all integer and has value 140. The greedy solution at node 10 is $\{1, 0, 1, 0, 1, 0, 0, 1\}$. Its value is 150. The next E-node is 11. Its value is 159.52 and $x_3 = 20/21$. The partitioning is now on $x_3 = 0$ and $x_3 = 1$. The remainder of the backtracking process on this knapsack instance is left as an exercise. □

Experimental work due to E. Horowitz and S. Sahni, cited in the references, indicates that backtracking algorithms for the knapsack problem generally work in less time when using a static tree than when using a dynamic tree. The dynamic partitioning scheme is, however, useful in the solution of integer linear programs. The general integer linear program is mathematically stated in (7.4).

$$\begin{aligned} &\text{minimize} \quad \textstyle\sum_{1 \le j \le n} c_j \, x_j \\ &\text{subject to} \quad \textstyle\sum_{1 \le j \le n} a_{ij} \, x_j \le b_i, \quad 1 \le i \le m \\ &\qquad x_j's \text{ are nonnegative integers} \end{aligned} \tag{7.4}$$

If the integer constraints on the x_i's in (7.4) are replaced by the constraint $x_i \ge 0$, then we obtain a linear program whose optimal solution has a value at least as large as the value of an optimal solution to (7.4). Linear programs can be solved using the simplex methods (see the references). If the solution is not all integer, then a noninteger x_i is chosen to partition the solution space. Let us assume that the value of x_i in the optimal solution to the linear program corresponding to any node Z in the state space is v and v is not an integer. The left child of Z corresponds to $x_i \le \lfloor v \rfloor$ whereas the right child of Z correspond to $x_i \ge \lceil v \rceil$. Since the resulting state space tree has a potentially infinite depth (note that on the path from the root to a node Z

the solution space can be partitioned on one x_i many times as each x_i can have as value any nonnegative integer), it is almost always searched using a branch-and-bound method (see Chapter 8).

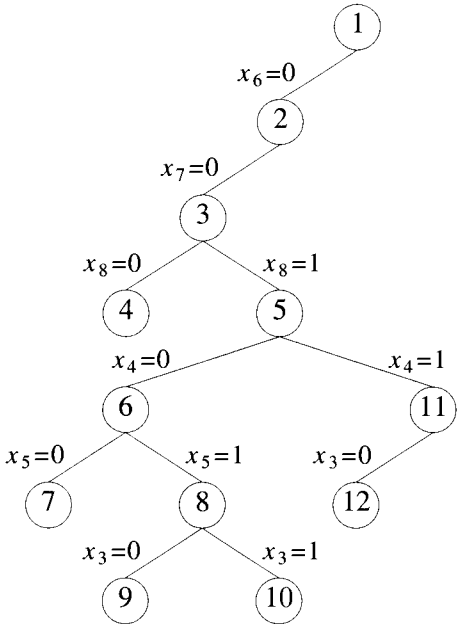


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

EXERCISES

1. (a) Present a backtracking algorithm for solving the knapsack optimization problem using the variable tuple size formulation.
   (b) Draw the portion of the state space tree your algorithm will generate when solving the knapsack instance of Example 7.7.
2. Complete the state space tree of Figure 7.16.
3. Give a backtracking algorithm for the knapsack problem using the dynamic state space tree discussed in this section.
4. [Programming project] (a) Program the algorithms of Exercises 1 and 3. Run these two programs and BKnap using the following data: $p =$

$\{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. Which algorithm do you expect to perform best?

(b) Now program the dynamic programming algorithm of Section 5.7 for the knapsack problem. Use the heuristics suggested at the end of Section 5.7. Obtain computing times and compare this program with the backtracking programs.

5. (a) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a dynamic tree than using a static tree.

   (b) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a static tree than using a dynamic tree.

   (c) Strengthen the backtracking algorithms with the following heuristic: Build an array $minw[\ ]$ with the property that $minw[i]$ is the index of the object that has least weight among objects $i, i+1, \ldots, n$. Now any E-node at which decisions for $x_1, \ldots, x_{i-1}$ have been made and at which the unutilized knapsack capacity is less than $w[minw[i]]$ can be terminated provided the profit earned up to this node is no more than the maximum determined so far. Incorporate this into your programs of Exercise 4(a). Rerun the new programs on the same data sets and see what (if any) improvements result.

7.7 REFERENCES AND READINGS

An early modern account of backtracking was given by R. J. Walker. The technique for estimating the efficiency of a backtrack program was first proposed by M. Hall and D. E. Knuth and the dynamic partitioning scheme for the 0/1 knapsack problem was proposed by H. Greenberg and R. Hegerich. Experimental results showing static trees to be superior for this problem can be found in "Computing partitions with applications to the knapsack problem," by E. Horowitz and S. Sahni, *Journal of the ACM* 21, no. 2 (1974): 277–292.

Data presented in the above paper shows that the divide-and-conquer dynamic programming algorithm for the knapsack problem is superior to BKnap.

For a proof of the four-color theorem see *Every Planar Map is Four Colorable*, by K. I. Appel, American Mathematical Society, Providence, RI, 1989.

A discussion of the simplex method for solving linear programs may be found in:

Linear Programming: An Introduction with Applications, by A. Sultan, Academic Press, 1993.

Linear Optimization and Extensions, by M. Padberg, Springer-Verlag, 1995.

7.8 ADDITIONAL EXERCISES

1. Suppose you are given n men and n women and two $n \times n$ arrays P and Q such that $P(i, j)$ is the preference of man i for woman j and $Q(i, j)$ is the preference of woman i for man j. Given an algorithm that finds a pairing of men and women such that the sum of the product of the preferences is maximized.

2. Let $A(1 : n, 1 : n)$ be an $n \times n$ matrix. The *determinant* of A is the number

$$\det(A) = \sum_s \text{sgn}(s) a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$$

where the sum is taken over all permutations $s(1), \ldots, s(n)$ of $\{1, 2, \ldots, n\}$ and $\text{sgn}(s)$ is $+1$ or -1 according to whether s is an even or odd permutation. The *permanent* of A is defined as

$$\text{per}(A) = \sum_s a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$$

The determinant can be computed as a by-product of Gaussian elimination requiring $O(n^3)$ operations, but no polynomial time algorithm is known for computing permanents. Write an algorithm that computes the permanent of a matrix by generating the elements of s using backtracking. Analyze the time of your algorithm.

3. Let MAZE$(1 : n, 1 : n)$ be a zero- or one-valued, two-dimensional array that represents a maze. A one means a blocked path whereas a zero stands for an open position. You are to develop an algorithm that begins at MAZE(1, 1) and tries to find a path to position MAZE(n, n). Once again backtracking is necessary here. See if you can analyze the time complexity of your algorithm.

4. The *assignment problem* is usually stated this way: There are n people to be assigned to n jobs. The cost of assigning the ith person to the jth job is $cost(i, j)$. You are to develop an algorithm that assigns every job to a person and at the same time minimizes the total cost of the assignment.

5. This problem is called the postage stamp problem. Envision a country that issues n different denominations of stamps but allows no more than m stamps on a single letter. For given values of m and n, write an algorithm that computes the greatest consecutive range of postage values, from one on up, and all possible sets of denominations that realize that range. For example, for $n = 4$ and $m = 5$, the stamps with values (1, 4, 12, 21) allow the postage values 1 through 71. Are there any other sets of four denominations that have the same range?

6. Here is a game called Hi-Q. Thirty-two pieces are arranged on a board as shown in Figure 7.17. Only the center position is unoccupied. A piece is only allowed to move by jumping over one of its neighbors into an empty space. Diagonal jumps are not permitted. When a piece is jumped, it is removed from the board. Write an algorithm that determines a series of jumps so that all the pieces except one are eventually removed and that final piece ends up at the center position.

7. Imagine a set of 12 plane figures each composed of five equal-size squares. Each figure differs in shape from the others, but together they can be arranged to make different-sized rectangles. In Figure 7.18 there is a picture of 12 pentominoes that are joined to create a 6×10 rectangle. Write an algorithm that finds all possible ways to place the pentominoes so that a 6×10 rectangle is formed.

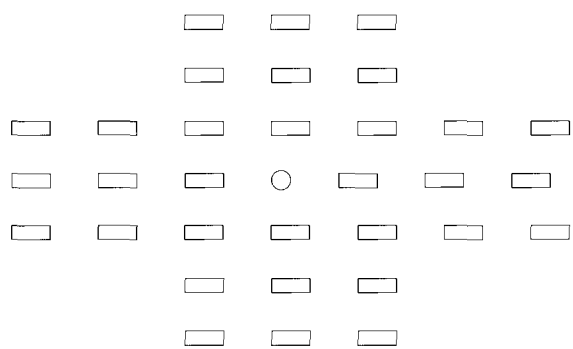


Figure 7.17 A Hi-Q board in its initial state

8. Suppose a set of electric components such as transistors are to be placed on a circuit board. We are given a connection matrix CONN, where CONN(i, j) equals the number of connections between component i and component j, and a matrix DIST, where DIST (r, s) is

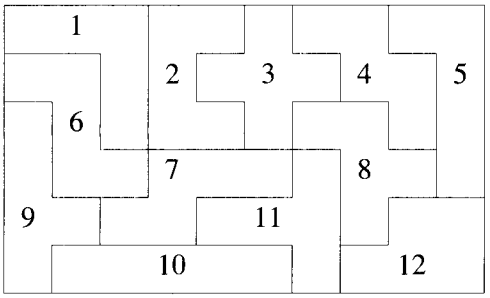


Figure 7.18 A pentomino configuration

the distance between position r and position s on the circuit board. The wiring of the board consists of placing each of n components at some location. The cost of a wiring is the sum of the products of $CONN(i,j) * DIST(r,s)$, where component i is placed at location r and component j is placed at location s. Compose an algorithm that finds an assignment of components to locations that minimizes the total cost of the wiring.

9. Suppose there are n jobs to be executed but only k processors that can work in parallel. The time required by job i is t_i. Write an algorithm that determines which jobs are to be run on which processors and the order in which they should be run so that the finish time of the last job is minimized.

10. Two graphs $G(V,E)$ and $H(A,B)$ are called *isomorphic* if there is a one-to-one onto correspondence of the vertices that preserves the adjacency relationships. More formally if f is a function from V to A and (v,w) is an edge in E, then $(f(v),f(w))$ is an edge in H. Figure 7.19 shows two directed graphs that are isomorphic under the mapping that $1,2,3,4$, and 5 and go to a,b,c,d, and e. A brute force algorithm to test two graphs for isomorphism would try out all $n!$ possible correspondences and then test to see whether adjacency was preserved. A backtracking algorithm can do better than this by applying some obvious pruning to the resultant state space tree. First of all we know that for a correspondence to exist between two vertices, they must have the same degree. So we can select at an early stage vertices of degree k for which the second graph has the fewest number of vertices of degree k. This exercise calls for devising an isomorphism algorithm that is based on backtracking and makes use of these ideas.

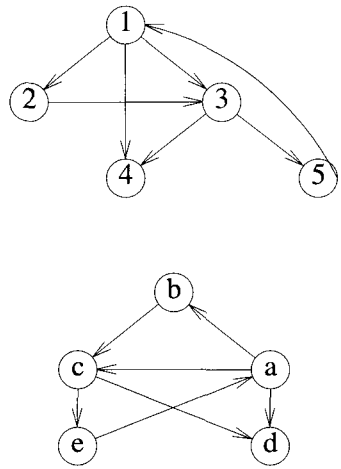


Figure 7.19 Two isomorphic graphs (Exercise 10)

11. A graph is called *complete* if all its vertices are connected to all the other vertices in the graph. A maximal complete subgraph of a graph is called a *clique*. By "maximal" we mean that this subgraph is contained within no other subgraph that is also complete. A clique of size k has $\binom{k}{i}$ subcliques of size i, $1 \leq i \leq k$. This implies that any algorithm that looks for a maximal clique must be careful to generate each subclique the fewest number of times possible. One way to generate the clique is to extend a clique of size m to size $m + 1$ and to continue this process by trying out all possible vertices. But this strategy generates the same clique many times; this can be avoided as follows. Given a clique X, suppose node v is the first node that is added to produce a clique of size one greater. After the backtracking process examines all possible cliques that are produced from X and v, then no vertex adjacent to v need be added to X and examined. Let X and Y be cliques and let X be properly contained in Y. If all cliques containing X and vertex v have been generated, then all cliques with Y and v can be ignored. Write a backtracking algorithm that generates the maximal cliques of an undirected graph and makes use of these last rules for pruning the state space tree.

Chapter 8

BRANCH-AND-BOUND

8.1 THE METHOD

This chapter makes extensive use of terminology defined in Section 7.1. The reader is urged to review this section before proceeding.

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. We have already seen (in Section 7.1) two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (**F**irst **I**n **F**irst **O**ut) search as the list of live nodes is a first-in-first-out list (or queue). A D-search-like state space search will be called LIFO (**L**ast **I**n **F**irst **O**ut) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

Example 8.1 [4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (Figure 7.2) for the 4-queens problem. Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node. It is expanded and its children, nodes 2, 18, 34, and 50, are generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively. The only live nodes now are nodes 2, 18, 34, and 50. If the nodes are generated in this order, then the next E-node is node 2. It is expanded and nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function of Example 7.5. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live

nodes. The E-node is node 34. Figure 8.1 shows the portion of the tree of Figure 7.2 that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions have a "B" under them. Numbers inside the nodes correspond to the numbers in Figure 7.2. Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound. At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54. A comparison of Figures 7.6 and 8.1 indicates that backtracking is a superior search method for this problem. □

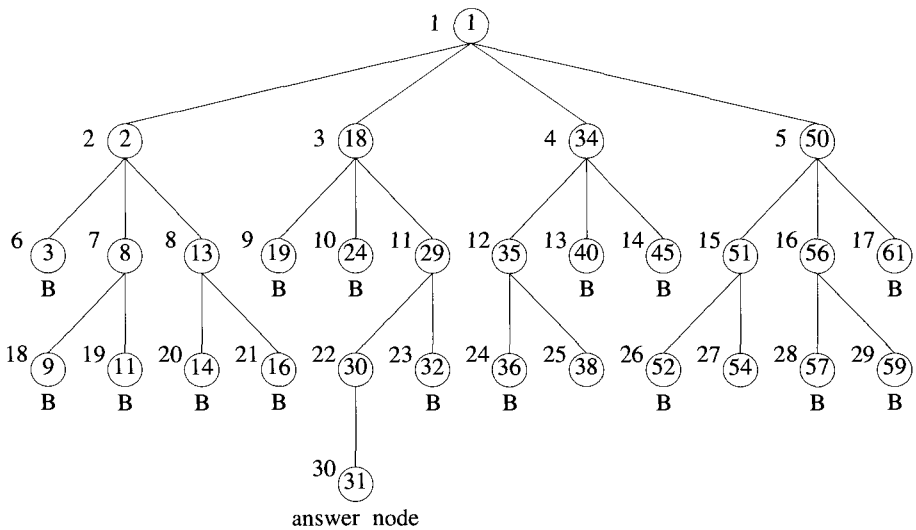


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

8.1.1 Least Cost (LC) Search

In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to an answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become the E-node following node 29. The remaining live nodes will never become E-nodes as the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node x, this cost could be (1) the number of nodes in the subtree x that need to be generated before an answer node is generated or, more simply, (2) the number of levels the nearest answer node (in the subtree x) is from x. Using cost measure 2, the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1). The costs of nodes 18 and 34, 29 and 35, and 30 and 38 are respectively 3, 2, and 1. The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1. Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 18, 29, and 30 (in that order). The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32, and 31. It should be easy to see that if cost measure 1 is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure 2 is used, then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node. The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node. Hence, by the time the cost of a node is determined, that subtree has been searched and there is no need to explore x again. For this reason, search algorithms usually rank nodes only on the basis of an estimate $\hat{g}(\cdot)$ of their cost.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function. At first, we may doubt the usefulness of using an $f(\cdot)$ other than $f(h(x)) = 0$ for all $h(x)$. We can justify such an $f(\cdot)$ on the grounds that the effort already expended in reaching the live nodes cannot be reduced and all we are concerned with now is minimizing the additional effort we spend to find an answer node. Hence, the effort already expended need not be considered.

Using $f(\cdot) \equiv 0$ usually biases the search algorithm to make deep probes into the search tree. To see this, note that we would normally expect $\hat{g}(y) \leq \hat{g}(x)$ for y, a child of x. Hence, following x, y will become the E-node, then one of y's children will become the E-node, next one of y's grandchildren will become the E-node, and so on. Nodes in subtrees other than the subtree x will not get generated until the subtree x is fully searched. This would not

be a cause for concern if $\hat{g}(x)$ were the true cost of x. Then, we would not wish to explore the remaining subtrees in any case (as x is guaranteed to get us to an answer node quicker than any other existing live node). However, $\hat{g}(x)$ is only an estimate of the true cost. So, it is quite possible that for two nodes w and z, $\hat{g}(w) < \hat{g}(z)$ and z is much closer to an answer node than w. It is therefore desirable not to overbias the search algorithm in favor of deep probes. By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

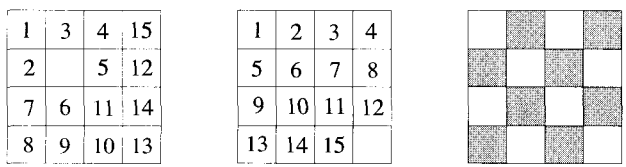
A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with bounding functions is called an LC branch-and-bound search.

In discussing LC-searches, we sometimes make reference to a cost function $c(\cdot)$ defined as follows: if x is an answer node, then $c(x)$ is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then $c(x) = \infty$ providing the subtree x contains no answer node; otherwise $c(x)$ equals the cost of a minimum-cost answer node in the subtree x. It should be easy to see that $\hat{c}(\cdot)$ with $f(h(x)) = h(x)$ is an approximation to $c(\cdot)$. From now on $c(x)$ is referred to as the cost of x.

8.1.2 The 15-puzzle: An Example

The 15-puzzle (invented by Sam Loyd in 1878) consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Figure 8.2). We are given an initial arrangement of the tiles, and the objective is to transform this arrangement into the goal arrangement of Figure 8.2(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Figure 8.2(a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Following this move, other moves can be made. Each move creates a new arrangement of the tiles. These arrangements are called the *states* of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable from the initial state iff there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the

path from the initial state to the goal state as the answer. It is easy to see that there are 16! ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position i is the frame position containing tile numbered i in the goal arrangement of Figure 8.2(b). Position 16 is the empty spot. Let $position(i)$ be the position number in the initial state of the tile numbered i. Then $position(16)$ will denote the position of the empty spot.



(a) An arrangement (b) Goal arrangement (c)

Figure 8.2 15-puzzle arrangements

For any state let $less(i)$ be the number of tiles j such that $j < i$ and $position(j) > position(i)$. For the state of Figure 8.2(a) we have, for example, $less(1) = 0$, $less(4) = 1$, and $less(12) = 6$. Let x =1 if in the initial state the empty spot is at one of the shaded positions of Figure 8.2(c) and $x = 0$ if it is at one of the remaining positions. Then, we have the following theorem:

Theorem 8.1 The goal state of Figure 8.2(b) is reachable from the initial state iff $\sum_{i=1}^{16} less(i) + x$ is even.

Proof: Left as an exercise. □

Theorem 8.1 can be used to determine whether the goal state is in the state space of the initial state. If it is, then we can proceed to determine a sequence of moves leading to the goal state. To carry out this search, the state space can be organized into a tree. The children of each node x in this tree represent the states reachable from state x by one legal move. It is convenient to think of a move as involving a move of the empty space rather than a move of a tile. The empty space, on each move, moves either up, right, down, or left. Figure 8.3 shows the first three levels of the state

space tree of the 15-puzzle beginning with the initial state shown in the root. Parts of levels 4 and 5 of the tree are also shown. The tree has been pruned a little. *No node p has a child state that is the same as p's parent.* The subtree eliminated in this way is already present in the tree and has root *parent*(*p*). As can be seen, there is an answer node at level 4.

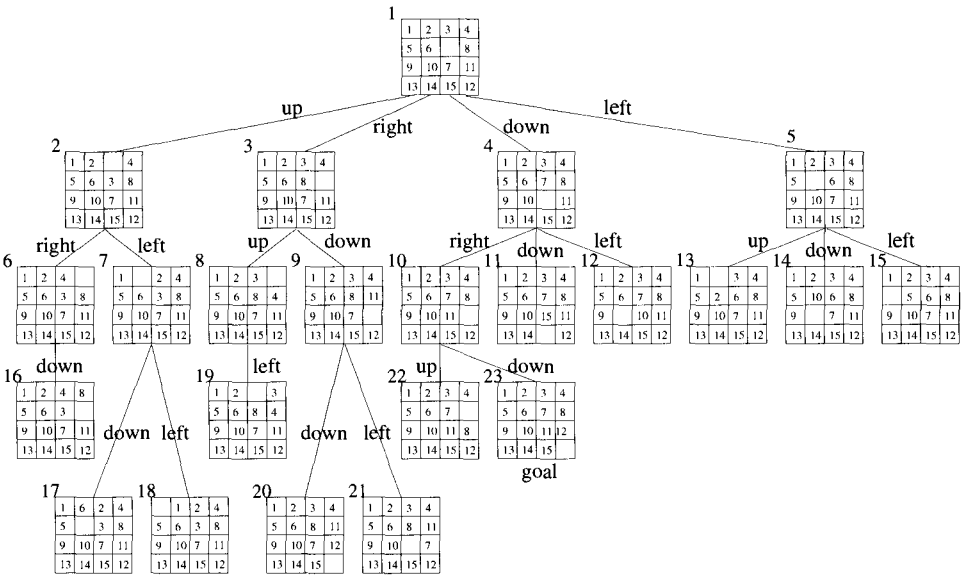


Figure 8.3 Part of the state space tree for the 15-puzzle

A depth first state space tree generation will result in the subtree of Figure 8.4 when the next moves are attempted in the order: move the empty space up, right, down, and left. Successive board configurations reveal that each move gets us farther from the goal rather than closer. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, an answer node may never be found (unless the leftmost path ends in such a node). In a FIFO search of the tree of Figure 8.3, the nodes will be generated in the order numbered. A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves. A FIFO search always generates the state space tree by levels.

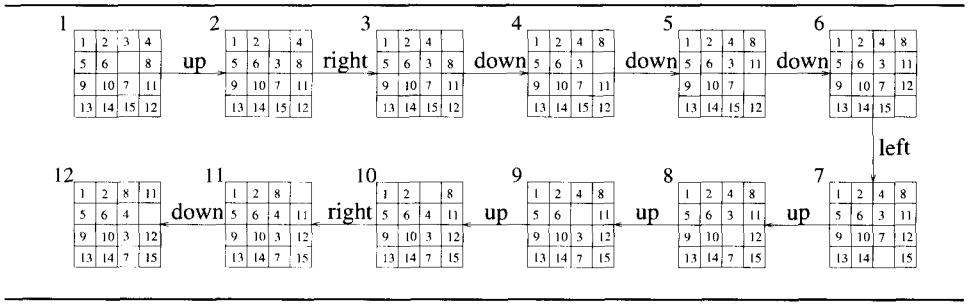


Figure 8.4 First ten steps in a depth first search

What we would like, is a more "intelligent" search method, one that seeks out an answer node and adapts the path it takes through the state space tree to the specific problem instance being solved. We can associate a cost $c(x)$ with each node x in the state space tree. The cost $c(x)$ is the length of a path from the root to a nearest goal node (if any) in the subtree with root x. Thus, in Figure 8.3, $c(1) = c(4) = c(10) = c(23) = 3$. When such a cost function is available, a very efficient search can be carried out. We begin with the root as the E-node and generate a child node with $c()$-value the same as the root. Thus children nodes 2, 3, and 5 are eliminated and only node 4 becomes a live node. This becomes the next E-node. Its first child, node 10, has $c(10) = c(4) = 3$. The remaining children are not generated. Node 4 dies and node 10 becomes the E-node. In generating node 10's children, node 22 is killed immediately as $c(22) > 3$. Node 23 is generated next. It is a goal node and the search terminates. In this search strategy, the only nodes to become E-nodes are nodes on the path from the root to a nearest goal node. Unfortunately, this is an impractical strategy as it is not possible to easily compute the function $c(\cdot)$ specified above.

We can arrive at an easy to compute estimate $\hat{c}(x)$ of $c(x)$. We can write $\hat{c}(x) = f(x) + \hat{g}(x)$, where $f(x)$ is the length of the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. One possible choice for $\hat{g}(x)$ is

$$\hat{g}(x) = \text{number of nonblank tiles not in their goal position}$$

Clearly, at least $\hat{g}(x)$ moves have to be made to transform state x to a goal state. More than $\hat{g}(x)$ moves may be needed to achieve this. To see this, examine the problem state of Figure 8.5. There $\hat{g}(x) = 1$ as only tile 7 is not in its final spot (the count for $\hat{g}(x)$ excludes the blank tile). However, the number of moves needed to reach the goal state is many more than $\hat{g}(x)$. So $\hat{c}(x)$ is a *lower bound* on the value of $c(x)$.

An LC-search of Figure 8.3 using $\hat{c}(x)$ will begin by using node 1 as the E-node. All its children are generated. Node 1 dies and leaves behind the live nodes 2, 3, 4, and 5. The next node to become the E-node is a live node with least $\hat{c}(x)$. Then $\hat{c}(2) = 1+4$, $\hat{c}(3) = 1+4$, $\hat{c}(4) = 1+2$, and $\hat{c}(5) = 1+4$. Node 4 becomes the E-node. Its children are generated. The live nodes at this time are 2, 3, 5, 10, 11, and 12. So $\hat{c}(10) = 2+1$, $\hat{c}(11) = 2+3$, and $\hat{c}(12) = 2+3$. The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is determined to be a goal node and the search terminates. In this case LC-search was almost as efficient as using the exact function $c()$. It should be noted that with a suitable choice for $\hat{c}()$, an LC-search will be far more selective than any of the other search methods we have discussed.

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

Figure 8.5 Problem state

8.1.3 Control Abstractions for LC-Search

Let t be a state space tree and $c()$ a cost function for the nodes in t. If x is a node in t, then $c(x)$ is the minimum cost of any answer node in the subtree with root x. Thus, $c(t)$ is the cost of a minimum-cost answer node in t. As remarked earlier, it is usually not possible to find an easily computable function $c()$ as defined above. Instead, a heuristic $\hat{c}$ that estimates $c()$ is used. This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$. LCSearch (Algorithm 8.1) uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add(x) to delete and add a live node from or to the list of live nodes, respectively. Least() finds a live node with least $\hat{c}()$. This node is deleted from the list of live nodes and returned. Add(x) adds the new live node x to the list of live nodes. The list of live nodes will usually be implemented as a min-heap (Section 2.4). Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When an answer node g is found, the path from

g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
    listnode = record {
        listnode *next, *parent; float cost;
    }

1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

Algorithm 8.1 LC-search

The correctness of algorithm LCSearch is easy to establish. Variable E always points to the current E-node. By the definition of LC-search, the root node is the first E-node (line 5). Line 6 initializes the list of live nodes. At any time during the execution of LCSearch, this list contains all live nodes except the E-node. Thus, initially this list should be empty (line 6). The **for** loop of line 9 examines all the children of the E-node. If one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If a child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes (line 13) and its *parent* field set to

E (line 14). When all the children of E have been generated, E becomes a dead node and line 16 is reached. This happens only if none of E's children is an answer node. So, the search must continue further. If there are no live nodes left, then the entire state space tree has been searched and no answer nodes found. The algorithm terminates in line 18. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

From the preceding discussion, it is clear that LCSearch terminates only when either an answer node is found or the entire state space tree has been generated and searched. Thus, termination is guaranteed only for finite state space trees. Termination can also be guaranteed for infinite state space trees that have at least one answer node provided a "proper" choice for the cost function $\hat{c}()$ is made. This is the case, for example, when $\hat{c}(x) > \hat{c}(y)$ for every pair of nodes x and y such that the level number of x is "sufficiently" higher than that of y. For infinite state space trees with no answer nodes, LCSearch will not terminate. Thus, it is advisable to restrict the search to find answer nodes with a cost no more than a given bound C.

One should note the similarity between algorithm LCSearch and algorithms for a breadth first search and D-search of a state space tree. If the list of live nodes is implemented as a queue with Least() and Add(x) being algorithms to delete an element from and add an element to the queue, then LCSearch will be transformed to a FIFO search schema. If the list of live nodes is implemented as a stack with Least() and Add(x) being algorithms to delete and add elements to the stack, then LCSearch will carry out a LIFO search of the state space tree. Thus, the algorithms for LC, FIFO, and LIFO search are essentially the same. The only difference is in the implementation of the list of live nodes. This is to be expected as the three search methods differ only in the selection rule used to obtain the next E-node.

8.1.4 Bounding

A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. (Another method, heuristic search, is discussed in the exercises.) A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If *upper* is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > upper$ may be killed as all answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > upper$. The starting value for *upper* can be obtained by some heuristic or can be set to ∞. Clearly, so long as the initial value for *upper* is no less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of

a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of *upper* can be updated.

Let us see how these ideas can be used to arrive at branch-and-bound algorithms for optimization problems. In this section we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function. We need to be able to formulate the search for an optimal solution as a search for a least-cost answer node in a state space tree. To do this, it is necessary to define the cost function $c(\cdot)$ such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(\cdot)$. For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution. For nodes representing infeasible solutions, $c(x) = \infty$. For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x. Since $c(x)$ is in general as hard to compute as the original optimization problem is to solve, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x. In general then, the $\hat{c}(\cdot)$ function used in a branch-and-bound solution to optimization functions will estimate the objective function value and not the computational difficulty of reaching an answer node. In addition, to be consistent with the terminology used in connection with the 15-puzzle, any node representing a feasible solution (a solution node) will be an answer node. However, only minimum-cost answer nodes will correspond to an optimal solution. Thus, answer nodes and solution nodes are indistinguishable.

As an example optimization problem, consider the job sequencing with deadlines problem introduced in Section 4.4. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Consider the following instance: $n = 4$, $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6,\ 2,\ 1)$, and $(p_4, d_4, t_4) = (3,\ 1,\ 1)$. The solution space for this instance consists of all possible subsets of the job index set $\{1, 2, 3, 4\}$. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem (Example 7.2). Figure 8.6 corresponds to the variable tuple size formulation while Figure 8.7 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In Figure 8.6 all nonsquare nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node $J = \{2, 3\}$ and the penalty (cost)

is 8. In Figure 8.7 only nonsquare leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a minimum-cost answer node. This node corresponds to $J = \{2, 3\}$ and a penalty of 8. The costs of the answer nodes of Figure 8.7 are given below the nodes.

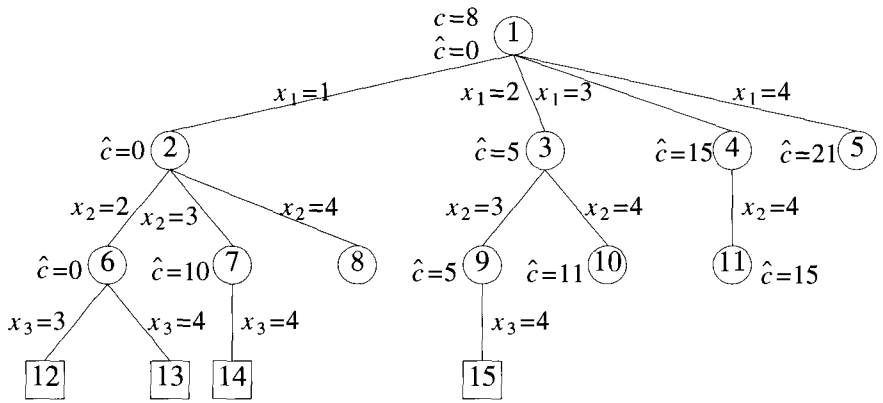


Figure 8.6 State space tree corresponding to variable tuple size formulation

We can define a cost function $c()$ for the state space formulations of Figures 8.6 and 8.7. For any circular node x, $c(x)$ is the minimum penalty corresponding to any node in the subtree with root x. The value of $c(x) = \infty$ for a square node. In the tree of Figure 8.6, $c(3) = 8$, $c(2) = 9$, and $c(1) = 8$. In the tree of Figure 8.7, $c(1) = 8$, $c(2) = 9$, $c(5) = 13$, and $c(6) = 8$. Clearly, $c(1)$ is the penalty corresponding to an optimal selection J.

A bound $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x is easy to obtain. Let S_x be the subset of jobs selected for J at node x. If $m = \max\{i | i \in S_x\}$, then $\hat{c}(x) = \sum_{\substack{i<m \\ i \notin S_x}} p_i$ is an estimate for $c(x)$ with the property $\hat{c}(x) \leq c(x)$. For each circular node x in Figures 8.6 and 8.7, the value of $\hat{c}(x)$ is the number outside node x. For a square node, $\hat{c}(x) = \infty$. For example, in Figure 8.6 for node 6, $S_6 = \{1, 2\}$ and hence $m = 2$. Also, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = 0$. For node 7, $S_7 = \{1, 3\}$ and $m = 3$. Therefore, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = p_2 = 10$. And so on. In Figure 8.7, node 12 corresponds to the omission of job 1 and hence a penalty of 5; node 13 corresponds to the omission of jobs 1 and 3 and hence a penalty of 11; and so on.

A simple upper bound $u(x)$ on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \notin S_x} p_i$. Note that $u(x)$ is the cost of the solution S_x corresponding to node x.

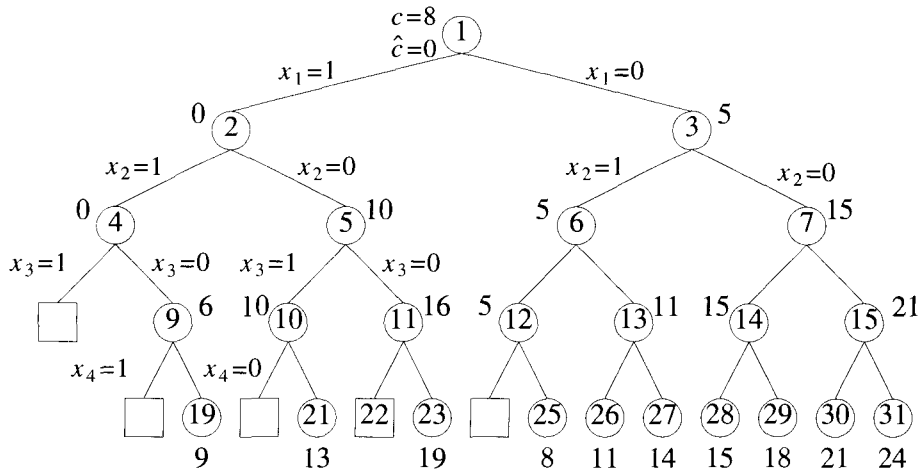


Figure 8.7 State space tree corresponding to fixed tuple size formulation

8.1.5 FIFO Branch-and-Bound

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with $upper = \infty$ (or $upper = \sum_{1 \le i \le n} p_i$) as an upper bound on the cost of a minimum-cost answer node. Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.6, nodes 2, 3, 4, and 5 are generated (in that order). Then $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$. For example, node 2 corresponds to the inclusion of job 1. Thus $u(2)$ is obtained by summing the penalties of all the other jobs. The variable *upper* is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than *upper*, nodes 4 and 5 get killed (or bounded). Only nodes 2 and 3 remain alive. Node 2 becomes the next E-node. Its children, nodes 6, 7, and 8 are generated. Then $u(6) = 9$ and so *upper* is updated to 9. The cost $\hat{c}(7) = 10 > upper$ and node 7 gets killed. Node 8 is infeasible and so it is killed. Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then $u(9) = 8$ and so *upper* becomes 8. The cost $\hat{c}(10) = 11 > upper$, and this node is killed. The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x) > upper$ each time *upper* is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x) > upper$ are distributed in some

random way in the queue. Instead, live nodes with $\hat{c}(x) > upper$ can be killed when they are about to become E-nodes.

From here on we shall refer to the FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as FIFOBB.

8.1.6 LC Branch-and-Bound

An LC branch-and-bound search of the tree of Figure 8.6 will begin with $upper = \infty$ and node 1 as the first E-node. When node 1 is expanded, nodes 2, 3, 4, and 5 are generated in that order. As in the case of FIFOBB, *upper* is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4) > upper$ and $\hat{c}(5) > upper$. Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7, and 8 are generated and *upper* is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10 > upper$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6. Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible. Node 3 becomes the next E-node. When node 9 is generated, *upper* is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed on generation. Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

From here on we refer to the LC(LIFO)-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as LCBB (LIFOBB).

EXERCISES

1. Prove Theorem 8.1.

2. Present an algorithm schema FifoBB for a FIFO branch-and-bound search for a least-cost answer node.

3. Give an algorithm schema LcBB for a LC branch-and-bound search for a least-cost answer node.

4. Write an algorithm schema LifoBB, for a LIFO branch-and-bound search for a least-cost answer node.

5. Draw the portion of the state space tree generated by FIFOBB, LCBB, and LIFOBB for the job sequencing with deadlines instance $n = 5$, $(p_1, p_2, \ldots, p_5) = (6, 3, 4, 8, 5)$, $(t_1, t_2, \ldots, t_5) = (2, 1, 2, 1, 1)$, and $(d_1, d_2, \ldots, d_5) = (3, 1, 4, 2, 4)$. What is the penalty corresponding to an optimal solution? Use a variable tuple size formulation and $\hat{c}(\cdot)$ and $u(\cdot)$ as in Section 8.1.

6. Write a branch-and-bound algorithm for the job sequencing with deadlines problem. Use the fixed tuple size formulation.

7. (a) Write a branch-and-bound algorithm for the job sequencing with deadlines problem using a dominance rule (see Section 5.7). Your algorithm should work with a fixed tuple size formulation and should generate nodes by levels. Nodes on each level should be kept in an order permitting easy use of your dominance rule.

   (b) Convert your algorithm into a program and, using randomly generated problem instances, determine the worth of the dominance rule as well as the bounding functions. To do this, you will have to run four versions of your program: ProgA$\cdots$ bounding functions and dominance rules are removed, ProgB$\cdots$ dominance rule is removed, ProgC$\cdots$ bounding function is removed, and ProgD$\cdots$ bounding functions and dominance rules are included. Determine computing time figures as well as the number of nodes generated.

8.2 0/1 KNAPSACK PROBLEM

To use the branch-and-bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem. We have already seen two possible state space tree organizations for the knapsack problem (Section 7.6). Still, we cannot directly apply the techniques of Section 8.1 since these were discussed with respect to minimization problems whereas the knapsack problem is a maximization problem. This difficulty is easily overcome by replacing the objective function $\sum p_i x_i$ by the function $-\sum p_i x_i$. Clearly, $\sum p_i x_i$ is maximized iff $-\sum p_i x_i$ is minimized. This modified knapsack problem is stated as (8.1).

$$\text{minimize} \quad -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq m \tag{8.1}$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

We continue the discussion assuming a fixed tuple size formulation for the solution space. The discussion is easily extended to the variable tuple size formulation. Every leaf node in the state space tree representing an assignment for which $\sum_{1 \leq i \leq n} w_i x_i \leq m$ is an answer (or solution) node. All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{1 \leq i \leq n} p_i x_i$ for every

answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(x)$ is recursively defined to be min $\{c(lchild(x)), c(rchild(x))\}$.

We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \leq c(x) \leq u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \leq j \leq n+1$. At node x assignments have already been made to x_i, $1 \leq i < j$. The cost of these assignments is $-\sum_{1 \leq i < j} p_i x_i$. So, $c(x) \leq -\sum_{1 \leq i < j} p_i x_i$ and we may use $u(x) = -\sum_{1 \leq i < j} p_i x_i$. If $q = -\sum_{1 \leq i < j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \mathsf{UBound}(q, \sum_{1 \leq i < j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2. As for $c(x)$, it is clear that $\mathsf{Bound}(-q, \sum_{1 \leq i < j} w_i x_i, j-1) \leq c(x)$, where Bound is as given in Algorithm 7.11.

```
1   Algorithm UBound(cp, cw, k, m)
2   // cp, cw, k, and m have the same meanings as in
3   // Algorithm 7.11. w[i] and p[i] are respectively
4   // the weight and profit of the ith object.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           if (c + w[i] ≤ m) then
10          {
11              c := c + w[i]; b := b − p[i];
12          }
13      }
14      return b;
15  }
```

Algorithm 8.2 Function $u(\cdot)$ for knapsack problem

8.2.1 LC Branch-and-Bound Solution

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

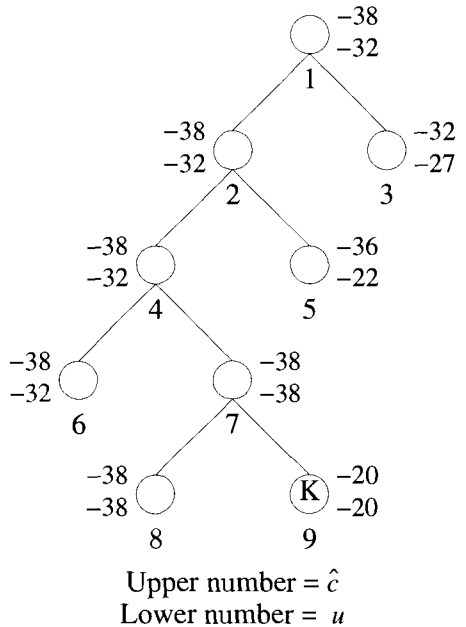


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound$(0,0,0,15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and $upper$ is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then $upper$ is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The tag sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

To use LCBB to solve the knapsack problem, we need to specify (1) the structure of nodes in the state space tree being searched, (2) how to generate the children of a given node, (3) how to recognize a solution node, and (4) a representation of the list of live nodes and a mechanism for adding a node into the list as well as identifying the least-cost node. The node structure needed depends on which of the two formulations for the state space tree is being used. Let us continue with a fixed size tuple formulation. Each node

x that is generated and put onto the list of live nodes must have a *parent* field. In addition, as noted in Example 8.2, each node should have a one bit *tag* field. This field is needed to output the x_i values corresponding to an optimal solution. To generate x's children, we need to know the level of node x in the state space tree. For this we shall use a field *level*. The left child of x is chosen by setting $x_{level(x)} = 1$ and the right child by setting $x_{level(x)} = 0$. To determine the feasibility of the left child, we need to know the amount of knapsack space available at node x. This can be determined either by following the path from node x to the root or by explicitly retaining this value in the node. Say we choose to retain this value in a field *cu* (capacity unused). The evaluation of $\hat{c}(x)$ and $u(x)$ requires knowledge of the profit $\sum_{1 \le i < level(x)} p_i x_i$ earned by the filling corresponding to node x. This can be computed by following the path from x to the root. Alternatively, this value can be explicitly retained in a field *pe*. Finally, in order to determine the live node with least $\hat{c}$ value or to insert nodes properly into the list of live nodes, we need to know $\hat{c}(x)$. Again, we have a choice. The value $\hat{c}(x)$ may be stored explicitly in a field *ub* or may be computed when needed. Assuming all information is kept explicitly, we need nodes with six fields each: *parent*, *level*, *tag*, *cu*, *pe*, and *ub*.

Using this six-field node structure, the children of any live node x can be easily determined. The left child y is feasible iff $cu(x) \ge w_{level(x)}$. In this case, $parent(y) = x$, $level(y) = level(x) + 1$, $cu(y) = cu(x) - w_{level(x)}$, $pe(y) = pe(x) + p_{level(x)}$, $tag(y) = 1$, and $ub(y) = ub(x)$. The right child can be generated similarly. Solution nodes are easily recognized too. Node x is a solution node iff $level(x) = n + 1$.

We are now left with the task of specifying the representation of the list of live nodes. The functions we wish to perform on this list are (1) test if the list is empty, (2) add nodes, and (3) delete a node with least *ub*. We have seen a data structure that allows us to perform these three functions efficiently: a min-heap. If there are m live nodes, then function (1) can be carried out in $\Theta(1)$ time, whereas functions (2) and (3) require only $O(\log m)$ time.

8.2.2 FIFO Branch-and-Bound Solution

Example 8.3 Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in Example 8.2. Initially the root node, node 1 of Figure 8.9, is the E-node and the queue of live nodes is empty. Since this is not a solution node, *upper* is initialized to $u(1) = -32$.

We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of *upper* remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue. Node 3, the next

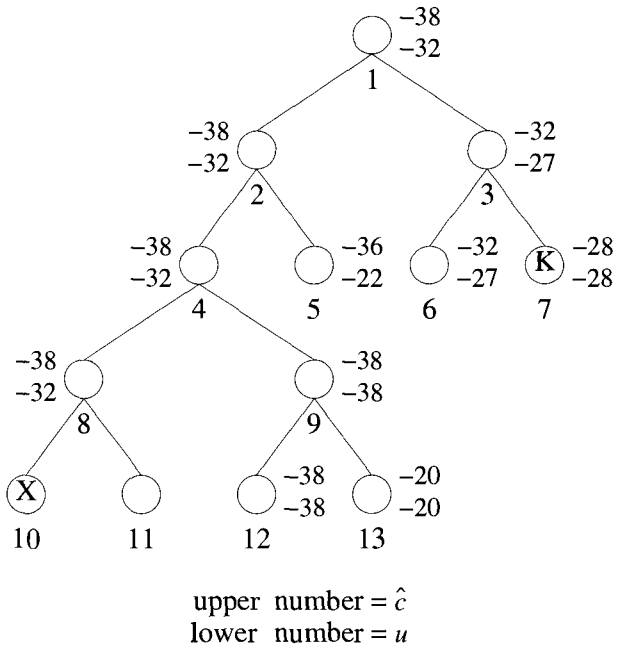


Figure 8.9 FIFO branch-and-bound tree for Example 8.3

E-node, is expanded. Its children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) > upper$. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then $upper$ is updated to $u(9) = -38$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}() > upper$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) > upper$ and so is also killed. Node 9 is expanded next. When node 12 is generated, $upper$ and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as $\hat{c}(13) > upper$. The only remaining live node is node 12. It has no children and the search terminates. The value of $upper$ and the path from node 12 to the root is output. As in the case of Example 8.2, additional information is needed to determine the x_i values on this path. □

At first we may be tempted to discard FIFOBB in favor of LCBB in solving knapsack. Our intuition leads us to believe that LCBB will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions form a heap are far more expensive (proportional to the logarithm of the heap size) than the corresponding operations on a queue ($\Theta(1)$). Consequently, the work done for each E-node is more in LCBB than in FIFOBB. Unless LCBB uses far fewer E-nodes than FIFOBB, FIFOBB will outperform (in terms of real computation time) LCBB.

We have now seen four different approaches to solving the knapsack problem: dynamic programming, backtracking, LCBB, and FIFOBB. If we compare the dynamic programming algorithm DKnap (Algorithm 5.7) and FIFOBB, we see that there is a correspondence between generating the $S^{(i)}$'s and generating nodes by levels. $S^{(i)}$ contains all pairs (P, W) corresponding to nodes on level $i+1$, $0 \leq i \leq n$. Hence, both algorithms generate the state space tree by levels. The dynamic programming algorithm, however, keeps the nodes on each level ordered by their profit earned (P) and capacity used (W) values. No two tuples have the same P or W value. In FIFOBB we may have many nodes on the same level with the same P or W value. It is not easy to implement the dominance rule of Section 5.7 into FIFOBB as nodes on a level are not ordered by their P or W values. However, the bounding rules can easily be incorporated into DKnap. Toward the end of Section 5.7 we discussed some simple heuristics to determine whether a pair $(P, W) \in S^{(i)}$ should be killed. These heuristics are readily seen to be bounding functions of the type discussed here. Let the algorithm resulting from the inclusion of the bounding functions into DKnap be DKnap1. DKnap1 is expected to be superior to FIFOBB as it uses the dominance rule in addition to the bounding functions. In addition, the overhead incurred each time a node is generated is less.

To determine which of the knapsack algorithms is best, it is necessary to program them and obtain real computing times for different data sets. Since the effectiveness of the bounding functions and the dominance rule is highly data dependent, we expect a wide variation in the computing time for different problem instances having the same number of objects n. To get representative times, it is necessary to generate many problem instances for a fixed n and obtain computing times for these instances. The generation of these data sets and the problem of conducting the tests is discussed in a programming project at the end of this section. The results of some tests can be found in the references to this chapter.

Before closing our discussion of the knapsack problem, we briefly discuss a very effective heuristic to reduce a knapsack instance with large n to an equivalent one with smaller n. This heuristic, Reduce, uses some of the ideas developed for the branch-and-bound algorithm. It classifies the objects $\{1, 2, \ldots, n\}$ into one of three categories $I1, I2$, and $I3$. $I1$ is a set of objects for which x_i must be 1 in every optimal solution. $I2$ is a set for which x_i must be 0. $I3$ is $\{1, 2, \ldots, n\} - I1 - I2$. Once $I1$, $I2$, and $I3$ have been determined, we need to solve only the reduced knapsack instance

$$\text{maximize} \sum_{i \in I3} p_i x_i$$

$$\text{subject to} \sum_{i \in I3} w_i x_i \leq m - \sum_{i \in I1} w_i x_i \tag{8.2}$$

$$x_i = 0 \text{ or } 1$$

From the solution to (8.2) an optimal solution to the original knapsack instance is obtained by setting $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$.

Function Reduce (Algorithm 8.3) makes use of two functions Ubb and Lbb. The bound Ubb($I1$, $I2$) is an upper bound on the value of an optimal solution to the given knapsack instance with added constraints $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$. The bound Lbb($I1$, $I2$) is a lower bound under the constraints of $I1$ and $I2$. Algorithm Reduce needs no further explanation. It should be clear that $I1$ and $I2$ are such that from an optimal solution to (8.2), we can easily obtain an optimal solution to the original knapsack problem.

The time complexity of Reduce is $O(n^2)$. Because the reduction procedure is very much like the heuristics used in DKnap1 and the knapsack algorithms of this chapter, the use of Reduce does not decrease the overall computing time by as much as may be expected by the reduction in the number of objects. These algorithms do dynamically what Reduce does. The exercises explore the value of Reduce further.

```
1   Algorithm Reduce(p, w, n, m, I1, I2)
2   // Variables are as described in the discussion.
3   // p[i]/w[i] ≥ p[i+1]/w[i+1], 1 ≤ i < n.
4   {
5       I1 := I2 := ∅;
6       q := Lbb(∅, ∅);
7       k := largest j such that w[1] + ··· + w[j] < m;
8       for i := 1 to k do
9       {
10          if (Ubb(∅, {i}) < q) then I1 := I1 ∪ {i};
11          else if (Lbb(∅, {i}) > q) then q := Lbb(∅, {i});
12      }
13      for i := k + 1 to n do
14      {
15          if (Ubb({i}, ∅) < q) then I2 := I2 ∪ {i};
16          else if (Lbb({i}, ∅) > q) then q := Lbb({i}, ∅);
17      }
18  }
```

Algorithm 8.3 Reduction pseudocode for knapsack problem

EXERCISES

1. Work out Example 8.2 using the variable tuple size formulation.
2. Work out Example 8.3 using the variable tuple size formulation.
3. Draw the portion of the state space tree generated by LCBB for the following knapsack instances:

   (a) $n = 5$, $(p_1, p_2, \ldots, p_5) = (10, 15, 6, 8, 4)$, $(w_1, w_2, \ldots, w_5) = (4, 6, 3, 4, 2)$, and $m = 12$.
   (b) $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5) = (4, 4, 5, 8, 9)$ and $m = 15$.

4. Do Exercise 3 using LCBB on a dynamic state space tree (see Section 7.6). Use the fixed tuple size formulation.
5. Write a LCBB algorithm for the knapsack problem using the ideas given in Example 8.2.
6. Write a LCBB algorithm for the knapsack problem using the fixed tuple size formulation and the dynamic state space tree of Section 7.6.
7. [Programming project] Program the algorithms DKnap (Algorithm 5.7), DKnap1 (page 399), LCBB for knapsack, and Bknap (Algorithm 7.12). Compare these programs empirically using randomly generated data as below:

   (a) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = \sum_1^n w_i/2$.
   (b) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = 2 \max \{w_i\}$.
   (c) Random w_i, $w_i \in [1, 100]$, $p_i = w_i + 10$, and $m = \sum_1^n w_i/2$.
   (d) Same as (c) except $m = 2 \max \{w_i\}$.
   (e) Random p_i, $p_i \in [1, 100]$, $w_i = p_i + 10$, and $m = \sum_1^n w_i/2$.
   (f) Same as (e) except $m = 2 \max \{w_i\}$.

   Obtain computing times for $n = 5, 10, 20, 30, 40, \ldots$. For each n, generate (say) ten problem instances from each of the above data sets. Report average and worst-case computing times for each of the above data sets. From these times can you say anything about the expected behavior of these algorithms?

   Now, generate problem instances with $p_i = w_i$, $1 \le i \le n$, $m = \sum w_i/2$, and $\sum w_i x_i \neq m$ for any 0, 1 assignment to the x_i's. Obtain computing times for your four programs for $n = 10$, 20, and 30. Now study the effect of changing the range to [1, 1000] in data sets (a) through (f). In sets (c) to (f) replace $p_i = w_i + 10$ by $p_i = w_i + 100$ and $w_i = p_i + 10$ by $w_i = p_i + 100$.

8. [Programming project]

   (a) Program the reduction heuristic Reduce of Section 8.2. Generate several problem instances from the data sets of Exercise 7 and determine the size of the reduced problem instances. Use n = 100, 200, 500, and 1000.

   (b) Program DKnap and the backtracking algorithm Bknap for the knapsack problem. Compare the effectiveness of Reduce by running several problem instances (as in Exercise 7). Obtain average and worst-case computing times for DKnap and Bknap for the generated problem instances and also for the reduced instances. To the times for the reduced problem instances, add the time required by Reduce. What conclusion can you draw from your experiments?

8.3 TRAVELING SALESPERSON (*)

An $O(n^2 2^n)$ dynamic programming algorithm for the traveling salesperson problem was arrived at in Section 5.9. We now investigate branch-and-bound algorithms for this problem. Although the worst-case complexity of these algorithms will not be any better than $O(n^2 2^n)$, the use of good bounding functions will enable these branch-and-bound algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

Let $G = (V, E)$ be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} equal the cost of edge $\langle i, j \rangle$, $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$, and let $|V| = n$. Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi$ is a permutation of $(2, 3, \ldots, n)\}$. Then $|S| = (n-1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n-1$, and $i_0 = i_n = 1$. S can be organized into a state space tree similar to that for the n-queens problem (see Figure 7.2). Figure 8.10 shows the tree organization for the case of a complete graph with $|V| = 4$. Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 2$, and $i_4 = 1$.

To use LCBB to search the traveling salesperson state space tree, we need to define a cost function $c(\cdot)$ and two other functions $\hat{c}(\cdot)$ and $u(\cdot)$ such that $\hat{c}(r) \leq c(r) \leq u(r)$ for all nodes r. The cost $c(\cdot)$ is such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One choice for $c(\cdot)$ is

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A, & \text{if } A \text{ is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree } A, & \text{if } A \text{ is not a leaf} \end{cases}$$

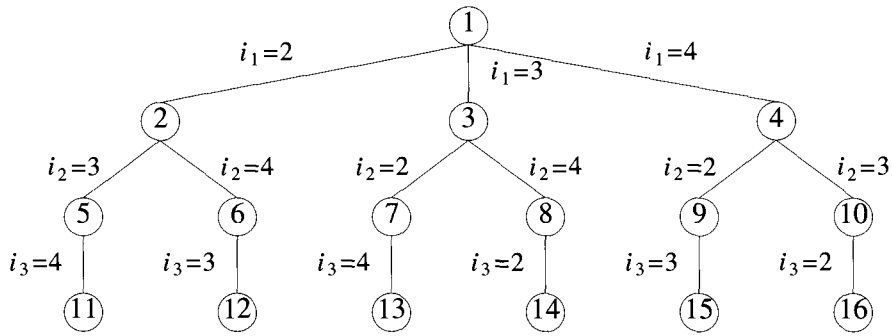


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

A simple $\hat{c}(\cdot)$ such that $\hat{c}(A) \leq c(A)$ for all A is obtained by defining $\hat{c}(A)$ to be the length of the path defined at node A. For example, the path defined at node 6 of Figure 8.10 is $i_0, i_1, i_2 = 1, 2, 4$. It consists of the edges $\langle 1, 2\rangle$ and $\langle 2, 4\rangle$. A better $\hat{c}(\cdot)$ can be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to be *reduced* iff it contains at least one zero and all remaining entries are non-negative. A matrix is *reduced* iff every row and column is reduced. As an example of how to reduce the cost matrix of a given graph G, consider the matrix of Figure 8.11(a). This corresponds to a graph with five vertices. Since every tour on this graph includes exactly one edge $\langle i, j\rangle$ with $i = k$, $1 \leq k \leq 5$, and exactly one edge $\langle i, j\rangle$ with $j = k$, $1 \leq k \leq 5$, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. A minimum-cost tour remains a minimum-cost tour following this subtraction operation. If t is chosen to be the minimum entry in row i (column j), then subtracting it from all entries in row i (column j) introduces a zero into row i (column j). Repeating this as often as needed, the cost matrix can be reduced. The total amount subtracted from the columns and rows is a lower bound on the length of a minimum-cost tour and can be used as the $\hat{c}$ value for the root of the state space tree. Subtracting 10, 2, 2, 3, 4, 1, and 3 from rows 1, 2, 3, 4, and 5 and columns 1 and 3 respectively of the matrix of Figure 8.11(a) yields the reduced matrix of Figure 8.11(b). The total amount subtracted is 25. Hence, all tours in the original graph have a length at least 25.

We can associate a reduced cost matrix with every node in the traveling salesperson state space tree. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including

edge $\langle i, j \rangle$ in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows: (1) Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j. (2) Set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1 \rangle$. (3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞. Let the resulting matrix be B. Steps (1) and (2) are valid as no tour in the subtree s can contain edges of the type $\langle i, k \rangle$ or $\langle k, j \rangle$ or $\langle j, 1 \rangle$ (except for edge $\langle i, j \rangle$). If r is the total amount subtracted in step (3) then $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$. For leaf nodes, $\hat{c}(\cdot) = c()$ is easily computed as each leaf defines a unique tour. For the upper bound function u, we can use $u(R) = \infty$ for all nodes R.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

Figure 8.11 An example

Let us now trace the progress of the LCBB algorithm on the problem instance of Figure 8.11(a). We use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.11(b) and *upper* $= \infty$. The portion of the state space tree that gets generated is shown in Figure 8.12. Starting with the root node as the E-node, nodes 2, 3, 4, and 5 are generated (in that order). The reduced matrices corresponding to these nodes are shown in Figure 8.13. The matrix of Figure 8.13(b) is obtained from that of 8.11(b) by (1) setting all entries in row 1 and column 3 to ∞, (2) setting the element at position (3, 1) to ∞, and (3) reducing column 1 by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (the cost of edge $\langle 1, 3 \rangle$ in the reduced matrix) + 11 = 53. The matrices and $\hat{c}$ value for nodes 2, 4, and 5 are obtained similarly. The value of *upper* is unchanged and node 4 becomes the next E-node. Its children 6, 7, and 8 are generated. The live nodes at this time are nodes 2, 3, 5, 6, 7, and 8. Node 6 has least $\hat{c}$ value and becomes the next E-node. Nodes 9 and 10 are generated. Node 10 is the next E-node. The solution node, node 11, is generated. The tour length for this node is $\hat{c}(11) = 28$ and *upper* is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 >$ *upper*. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

An exercise examines the implementation considerations for the LCBB algorithm. A different LCBB algorithm can be arrived at by considering

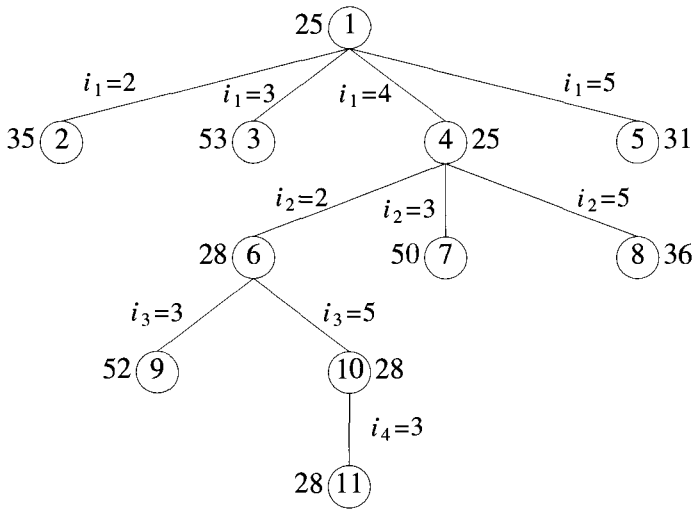


Figure 8.12 State space tree generated by procedure LCBB

a different tree organization for the solution space. This organization is reached by regarding a tour as a collection of n edges. If $G = (V, E)$ has e edges, then every tour contains exactly n of the e edges. However, for each $i, 1 \leq i \leq n$, there is exactly one edge of the form $\langle i, j \rangle$ and one of the form $\langle k, i \rangle$ in every tour. A possible organization for the state space is a binary tree in which a left branch represents the inclusion of a particular edge while the right branch represents the exclusion of that edge. Figure 8.14(b) and (c) represents the first two levels of two possible state space trees for the three vertex graph of Figure 8.14(a). As is true of all problems, many state space trees are possible for a given problem formulation. Different trees differ in the order in which decisions are made. Thus, in Figure 8.14(c) we first decide the fate of edge $\langle 1, 2 \rangle$. Rather than use a static state space tree, we now consider a dynamic state space tree (see Section 7.1). This is also a binary tree. However, the order in which edges are considered depends on the particular problem instance being solved. We compute $\hat{c}$ in the same way as we did using the earlier state space tree formulation.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

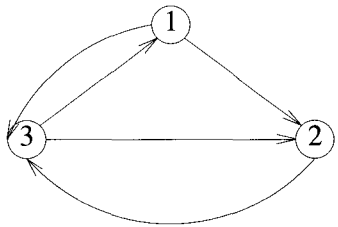
(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

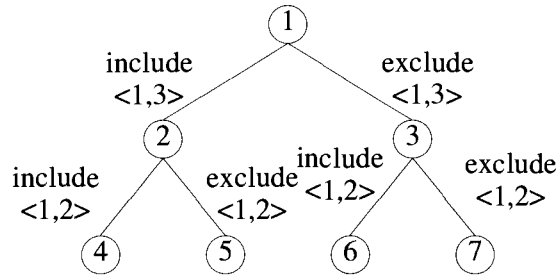
(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

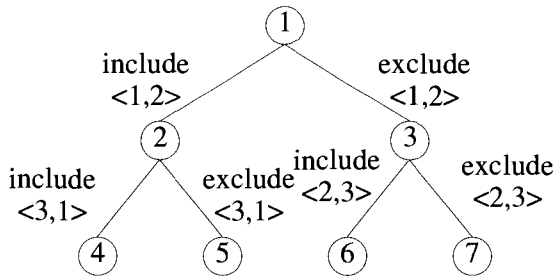
As an example of how LCBB would work on the dynamic binary tree formulation, consider the cost matrix of Figure 8.11(a). Since a total of 25



(a) Graph



(b) Part of a state space tree



(c) Part of a state space tree

Figure 8.14 An example

needs to be subtracted form the rows and columns of this matrix to obtain the reduced matrix of Figure 8.11(b), all tours have a length at least 25. This fact is represented by the root of the state space tree of Figure 8.15. Now, we must decide which edge to use to partition the solution space into two subsets. If edge $\langle i, j \rangle$ is used, then the left subtree of the root represents all tours including edge $\langle i, j \rangle$ and the right subtree represents all tours that do not include edge $\langle i, j \rangle$. If an optimal tour is included in the left subtree, then only $n - 1$ edges remain to be selected. If all optimal tours lie in the right subtree, then we have still to select n edges. Since the left subtree selects fewer edges, it should be easier to find an optimal solution in it than to find one in the right subtree. Consequently, we would like to choose as the partitioning edge an edge $\langle i, j \rangle$ that has the highest probability of being

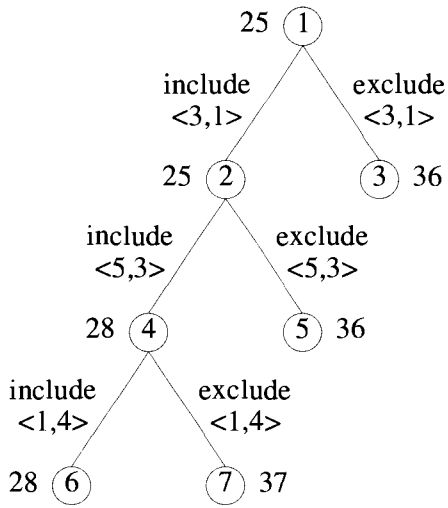
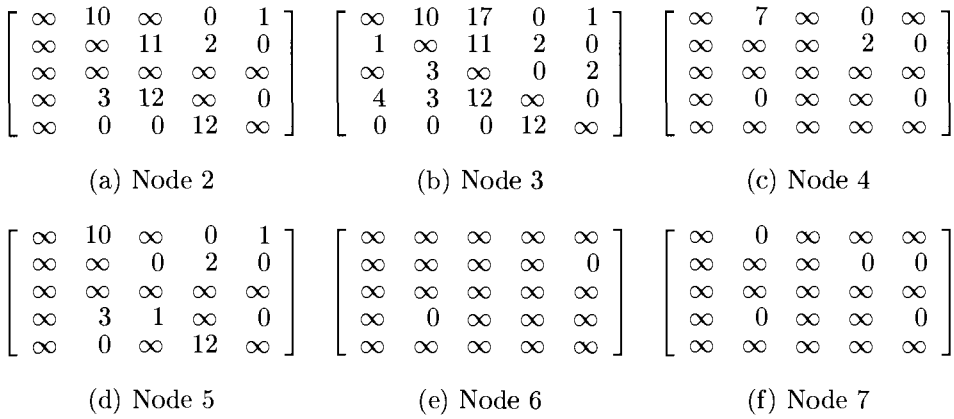


Figure 8.15 State space tree for Figure 8.11(a)

in an optimal tour. Several heuristics for determining such an edge can be formulated. A selection rule that is commonly used is select that edge which results in a right subtree that has highest $\hat{c}$ value. The logic behind this is that we soon have right subtrees (perhaps at lower levels) for which the $\hat{c}$ value is higher than the length of an optimal tour. Another possibility is to choose an edge such that the difference in the $\hat{c}$ values for the left and right subtrees is maximum. Other selection rules are also possible.

When LCBB is used with the first of the two selection rules stated above and the cost matrix of Figure 8.11(a), the tree of Figure 8.15 is generated. At the root node, we have to determine an edge $\langle i, j \rangle$ that will maximize the $\hat{c}$ value of the right subtree. If we select an edge $\langle i, j \rangle$ whose cost in the reduced matrix (Figure 8.11(b)) is positive, then the $\hat{c}$ value of the right subtree will remain 25. This is so as the reduced matrix for the right subtree will have $B(i, j) = \infty$ and all other entries will be identical to those in Figure 8.11(b). Hence B will be reduced and $\hat{c}$ cannot increase. So, we must choose an edge with reduced cost 0. If we choose $\langle 1, 4 \rangle$, then $B(1, 4) = \infty$ and we need to subtract 1 from row 1 to obtain a reduced matrix. In this case $\hat{c}$ will be 26. If $\langle 3, 1 \rangle$ is selected, then 11 needs to be subtracted from column 1 to obtain the reduced matrix for the right subtree. So, $\hat{c}$ will be 36. If A is the reduced cost matrix for node R, then the selection of edge $\langle i, j \rangle$ $(A(i, j) = 0)$ as the next partitioning edge will increase the $\hat{c}$ of the



(a) Node 2 (b) Node 3 (c) Node 4

(d) Node 5 (e) Node 6 (f) Node 7

Figure 8.16 Reduced cost matrices for Figure 8.15

right subtree by $\triangle = \min_{k \neq j}\{A(i,k)\} + \min_{k \neq i}\{A(k,j)\}$ as this much needs to be subtracted from row i and column j to introduce a zero into both. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 3,1\rangle$ $\langle 3,4\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle$, $\triangle$ = 1, 2, 11, 0, 3, 3, and 11 respectively. So, either of the edges $\langle 3,1\rangle$ or $\langle 5,3\rangle$ can be used. Let us assume that LCBB selects edge $\langle 3,1\rangle$. The $\hat{c}(2)$ (Figure 8.15) can be computed in a manner similar to that for the state space tree of Figure 8.12. In the corresponding reduced cost matrix all entries in row 3 and column 1 will be ∞. Moreover the entry (1, 3) will also be ∞ as inclusion of this edge will result in a cycle. The reduced matrices corresponding to nodes 2 and 3 are given in Figure 8.16(a) and (b). The $\hat{c}$ values for nodes 2 and 3 (as well as for all other nodes) appear outside the respective nodes.

Node 2 is the next E-node. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle$, $\triangle = 3, 2, 3, 3$, and 11 respectively. Edge $\langle 5,3\rangle$ is selected and nodes 4 and 5 generated. The corresponding reduced matrices are given in Figure 8.16(c) and (d). Then $\hat{c}(4)$ becomes 28 as we need to subtract 3 from column 2 to reduce this column. Note that entry (1, 5) has been set to ∞ in Figure 8.16(c). This is necessary as the inclusion of edge $\langle 1,5\rangle$ to the collection $\{\langle 3,1\rangle, \langle 5,3\rangle\}$ will result in a cycle. In addition, entries in column 3 and row 5 are set to ∞. Node 4 is the next E-node. The $\triangle$ values corresponding to edges $\langle 1,4\rangle, \langle 2,5\rangle$, and $\langle 4,2\rangle$ are 9, 2, and 0 respectively. Edge $\langle 1,4\rangle$ is selected and nodes 6 and 7 generated. The edge selection at node 6 is $\{\langle 3,1\rangle, \langle 5,3\rangle, \langle 1,4\rangle\}$. This corresponds to the path 5, 3, 1, 4. So, entry (4, 5) is set to ∞ in Figure 8.16(e). In general if edge $\langle i,j\rangle$ is selected, then the entries in row i and column j are set to ∞ in the left subtree. In addition, one more entry needs to be set to ∞. This is an entry whose inclusion in

the set of edges would create a cycle (Exercise 4 examines how to determine this). The next E-node is node 6. At this time three of the five edges have already been selected. The remaining two may be selected directly. The only possibility is $\{\langle 4,2\rangle, \langle 2,5\rangle\}$. This gives the path $5,3,1,4,2,5$ with length 28. So *upper* is updated to 28. Node 3 is the next E-node. Now LCBB terminates as $\hat{c}(3) = 36 > upper$.

In the preceding example, LCBB was modified slightly to handle nodes close to a solution node differently from other nodes. Node 6 is only two levels from a solution node. Rather than evaluate $\hat{c}$ at the children of 6 and then obtain their grandchildren, we just obtained an optimal solution for that subtree by a complete search with no bounding. We could have done something similar when generating the tree of Figure 8.12. Since node 6 is only two levels from the leaf nodes, we can simply skip computing $\hat{c}$ for the children and grandchildren of 6, generate all of them, and pick the best. This works out to be quite efficient as it is easier to generate a subtree with a small number of nodes and evaluate all the solution nodes in it than it is to compute $\hat{c}$ for one of the children of 6. This latter statement is true of many applications of branch-and-bound. Branch-and-bound is used on large subtrees. Once a small subtree is reached (say one with 4 or 6 nodes in it), then that subtree is fully evaluated without using the bounding functions.

We have now seen several branch-and-bound strategies for the traveling salesperson problem. It is not possible to determine analytically which of these is the best. The exercises describe computer experiments that determine empirically the relative performance of the strategies suggested.

EXERCISES

1. Consider the traveling salesperson instance defined by the cost matrix

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

   (a) Obtain the reduced cost matrix

   (b) Using a state space tree formulation similar to that of Figure 8.10 and $\hat{c}$ as described in Section 8.3, obtain the portion of the state space tree that will be generated by LCBB. Label each node by its $\hat{c}$ value. Write out the reduced matrices corresponding to each of these nodes.

   (c) Do part (b) using the reduced matrix method and the dynamic state space tree approach discussed in Section 8.3.

2. Do Exercise 1 using the following traveling salesperson cost matrix:

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

3. (a) Describe an efficient implementation for a LCBB traveling salesperson algorithm using the reduced cost matrix approach and (i) a dynamic state space tree and (ii) a static tree as in Figure 8.10.
   (b) Are there any problem instances for which the LCBB will generate fewer nodes using a static tree than using a dynamic tree? Prove your answer.

4. Consider the LCBB traveling salesperson algorithm described using the dynamic state space tree formulation. Let A and B be nodes. Let B be a child of A. If the edge (A, B) represents the inclusion of edge $\langle i, j \rangle$ in the tour, then in the reduced matrix for B all entries in row i and column j are set to ∞. In addition, one more entry is set to ∞. Obtain an efficient way to determine this entry.

5. [Programming project] Write computer programs for the following traveling salesperson algorithms:
   (a) The dynamic programming algorithm of Chapter 5
   (b) A backtracking algorithm using the static tree formulation of Section 8.3
   (c) A backtracking algorithm using the dynamic tree formulation of Section 8.3
   (d) A LCBB algorithm corresponding to (b)
   (e) A LCBB algorithm corresponding to (c)

   Design data sets to be used to compare the efficiency of the above algorithms. Randomly generate problem instances from these data sets and obtain computing times for your programs. What conclusions can you draw from your computing times?

8.4 EFFICIENCY CONSIDERATIONS

One can pose several questions concerning the performance characteristics of branch-and-bound algorithms that find least-cost answer nodes. We might ask questions such as:

1. Does the use of a better starting value for *upper* always decrease the number of nodes generated?

2. Is it possible to decrease the number of nodes generated by expanding some nodes with $\hat{c}() > upper$?

3. Does the use of a better $\hat{c}$ always result in a decrease in (or at least not an increase in) the number of nodes generated? (A $\hat{c}_2$ is better than $\hat{c}_1$ iff $\hat{c}_1(x) \leq \hat{c}_2(x) \leq c(x)$ for all nodes x.)

4. Does the use of dominance relations ever result in the generation of more nodes than would otherwise be generated?

In this section we answer these questions. Although the answers to most of the questions examined agree with our intuition, the answers to others are contrary to intuition. However, even in cases in which the answer does not agree with intuition, we can expect the performance of the algorithm to generally agree with the intuitive expectations. All the following theorems assume that the branch-and-bound algorithm is to find a minimum-cost solution node. Consequently, $c(x)$ = cost of minimum-cost solution node in subtree x.

Theorem 8.2 Let t be a state space tree. The number of nodes of t generated by FIFO, LIFO, and LC branch-and-bound algorithms cannot be decreased by the expansion of any node x with $\hat{c}(x) \geq upper$, where *upper* is the current upper bound on the cost of a minimum-cost solution node in the tree t.

Proof: The theorem follows from the observation that the value of *upper* cannot be decreased by expanding x (as $\hat{c}(x) \geq upper$). Hence, such an expansion cannot affect the operation of the algorithm on the remainder of the tree. □

Theorem 8.3 Let U_1 and $U_2, U_1 < U_2$, be two initial upper bounds on the cost of a minimum-cost solution node in the state space tree t. Then FIFO, LIFO, and LC branch-and-bound algorithms beginning with U_1 will generate no more nodes than they would if they started with U_2 as the initial upper bound.

Proof: Left as an exercise. □

Theorem 8.4 The use of a better $\hat{c}$ function in conjunction with FIFO and LIFO branch-and-bound algorithms does not increase the number of nodes generated.

Proof: Left as an exercise. □

Theorem 8.5 If a better $\hat{c}$ function is used in a LC branch-and-bound algorithm, the number of nodes generated may increase.

Proof: Consider the state space tree of Figure 8.17. All leaf nodes are solution nodes. The value outside each leaf is its cost. From these values it follows that $c(1) = c(3) = 3$ and $c(2) = 4$. Outside each of nodes 1, 2, and 3 is a pair of numbers $\binom{\hat{c}_1}{\hat{c}_2}$. Clearly, $\hat{c}_2$ is a better function than $\hat{c}_1$. However, if $\hat{c}_2$ is used, node 2 can become the E-node before node 3, as $\hat{c}_2(2) = \hat{c}_2(3)$. In this case all nine nodes of the tree will get generated. When $\hat{c}_1$ is used, nodes 4, 5, and 6 are not generated. □

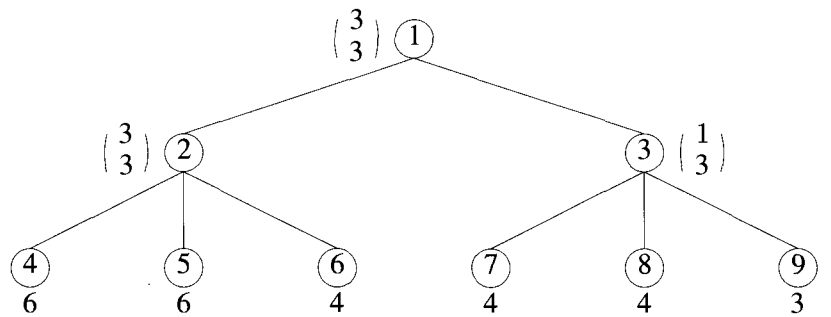


Figure 8.17 Example tree for Theorem 8.5

Now, let us look at the effect of dominance relations. Formally, a dominance relation D is given by a set of tuples, $D = \{(i_1, i_2), (i_3, i_4), (i_5, i_6), \ldots\}$. If $(i, j) \in D$, then node i is said to dominate node j. By this we mean that subtree i contains a solution node with cost no more than the cost of a minimum-cost solution node in subtree j. Dominated nodes can be killed without expansion.

Since every node dominates itself, $(i, i) \in D$ for all i and D. The relation (i, i) should not result in the killing of node i. In addition, it is quite possible for D to contain tuples (i_1, i_2), $(i_2, i_3), (i_3, i_4), \ldots, (i_n, i_1)$. In this case, the transitivity of D implies that each node i_k dominates all nodes $i_j, 1 \leq j \leq n$. Care should be taken to leave at least one of the i_j's alive. A dominance relation D_2 is said to be *stronger* than another dominance relation D_1 iff $D_1 \subset D_2$. In the following theorems I denotes the identity relation $\{(i, i) | 1 \leq i \leq n\}$.

Theorem 8.6 The number of nodes generated during a FIFO or LIFO branch-and-bound search for a least-cost solution node may increase when a stronger dominance relation is used.

Proof: Consider the state space tree of Figure 8.18. The only solution nodes are leaf nodes. Their cost is written outside the node. For the remaining nodes the number outside each node is its $\hat{c}$ value. The two dominance relations to use are $D_1 = I$ and $D_2 = I \cup \{(5,2),(5,8)\}$. Clearly, D_2 is stronger than D_1 and fewer nodes are generated using D_1 rather than D_2. □

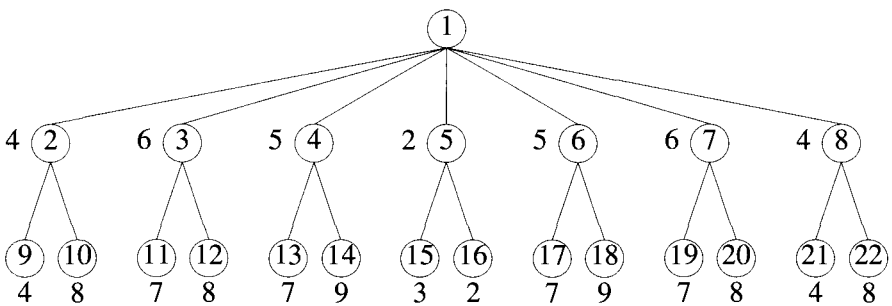


Figure 8.18 Example tree for Theorem 8.6

Theorem 8.7 Let D_1 and D_2 be two dominance relations. Let D_2 be stronger than D_1 and such that $(i,j) \in D_2, i \neq j$, implies $\hat{c}(i) < \hat{c}(j)$. An LC branch-and-bound using D_1 generates at least as many nodes as one using D_2.

Proof: Left as an exercise. □

Theorem 8.8 If the condition $\hat{c}(i) < \hat{c}(j)$ in Theorem 8.7 is removed then an LC branch-and-bound using the relation D_1 may generate fewer nodes than one using D_2.

Proof: Left as an exercise. □

EXERCISES

1. Prove Theorem 8.3.
2. Prove Theorem 8.4.
3. Prove Theorem 8.7.
4. Prove Theorem 8.8.
5. [Heuristic search] Heuristic search is a generalization of FIFO, LIFO, and LC searches. A heuristic function $h(\cdot)$ is used to evaluate all live nodes. The next E-node is the live node with least $h(\cdot)$. Discuss the advantages of using a heuristic function $h(\cdot)$ different from $\hat{c}(\cdot)$ in the search for a least-cost answer node. Consider the knapsack and traveling salesperson problems as two example problems. Also consider any other problems you wish. For these problems devise reasonable functions $h(\cdot)$ (different from $\hat{c}(\cdot)$). Obtain problem instances on which heuristic search performs better than LC-search.

8.5 REFERENCES AND READINGS

LC branch-and-bound algorithms have been extensively studied by researchers in areas such as artificial intelligence and operations research.

Branch-and-bound algorithms using dominance relations in a manner similar to that suggested by FIFOKNAP (resulting in DKnap1) were given by M. Held and R. Karp.

The reduction technique for the knapsack problem is due to G. Ingargiola and J. Korsh.

The reduced matrix technique to compute $\hat{c}$ is due to J. Little, K. Murty, D. Sweeny, and C. Karel. They employed the dynamic state space tree approach.

The results of Section 8.4 are based on the work of W. Kohler, K. Steiglitz, and T. Ibaraki.

The application of branch-and-bound and other techniques to the knapsack and related problems is discussed extensively in *Knapsack Problems: Algorithms and Computer Implementations*, by S. Martello and P. Toth, John Wiley and Sons, 1990.

Chapter 11

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

11.1 BASIC CONCEPTS

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

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

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

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

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

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

11.1.1 Nondeterministic Algorithms

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

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

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

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

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

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

Algorithm 11.1 Nondeterministic search

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

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

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

Algorithm 11.2 Nondeterministic sorting

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm 11.3 Deterministic sum of subsets

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

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

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

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

Algorithm 11.4 Nondeterministic knapsack algorithm

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

Algorithm 11.5 Nondeterministic clique pseudocode

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

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

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

Algorithm 11.6 Nondeterministic satisfiability

11.1.2 The Classes $\mathcal{NP}$-hard and $\mathcal{NP}$-complete

In measuring the complexity of an algorithm, we 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 11.3 $\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time. □

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

Is it possible that for all the problems in $\mathcal{NP}$, there exist polynomial time deterministic algorithms that have remained undiscovered? This seems unlikely, at least because of the tremendous effort that has already been expended by so many people on these problems. Nevertheless, a proof that $\mathcal{P} \neq \mathcal{NP}$ 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 question of whether $\mathcal{NP} \subseteq \mathcal{P}$ is no exception. Figure 11.1 displays the relationship between $\mathcal{P}$ and $\mathcal{NP}$ assuming that $\mathcal{P} \neq \mathcal{NP}$.

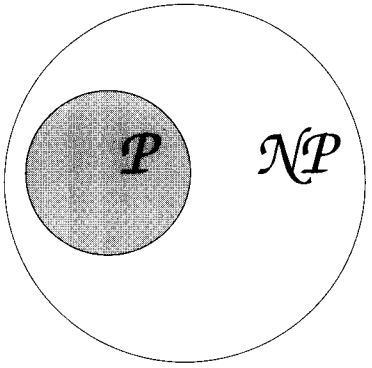


Figure 11.1 Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

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

Theorem 11.1 [Cook] Satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

Proof: See Section 11.2. □

We are now ready to define the $\mathcal{NP}$-hard and $\mathcal{NP}$-complete classes of problems. First we define the notion of reducibility. Note that this definition is similar to the one made in Section 10.3.

Definition 11.4 Let L_1 and L_2 be problems. Problem 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 can readily verify that $\propto$ is a transitive relation (that is, if $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$).

Definition 11.5 A problem L is $\mathcal{NP}$-hard if and only if satisfiability reduces to L (satisfiability $\propto L$). A problem L is $\mathcal{NP}$-complete if and only if L is $\mathcal{NP}$-hard and $L \in \mathcal{NP}$. □

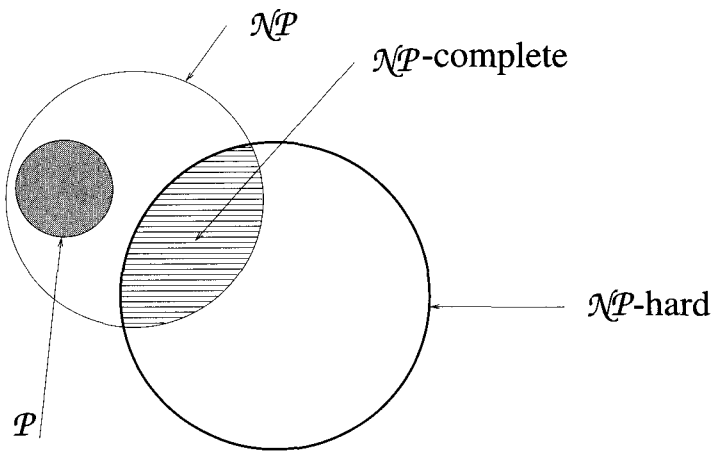


Figure 11.2 Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems

It is easy to see that there are $\mathcal{NP}$-hard problems that are not $\mathcal{NP}$-complete. Only a decision problem can be $\mathcal{NP}$-complete. However, an optimization problem may be $\mathcal{NP}$-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 can trivially show that the knapsack decision problem reduces to the knapsack optimization problem. For the clique problem one can easily show that the clique decision problem reduces to the clique optimization problem. In fact, one can also show that these optimization problems reduce to their corresponding decision problems (see the exercises). Yet, optimization problems cannot be $\mathcal{NP}$-complete whereas decision problems can. There also exist $\mathcal{NP}$-hard decision problems that are not $\mathcal{NP}$-complete. Figure 11.2 shows the relationship among these classes.

Example 11.10 As an extreme example of an $\mathcal{NP}$-hard decision problem that is not $\mathcal{NP}$-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 $\mathcal{NP}$. To show satisfiability $\propto$ the 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 whether X is satisfiable. If it is, then A stops. If it is not, then A enters an infinite loop. Hence, A halts on input X if and only

if 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 $\mathcal{NP}$-hard problem that is not in $\mathcal{NP}$. □

Definition 11.6 Two problems L_1 and L_2 are said to be *polynomially equivalent* if and only if $L_1 \propto L_2$ and $L_2 \propto L_1$. □

To show that a problem L_2 is $\mathcal{NP}$-hard, it is adequate to show $L_1 \propto L_2$, where L_1 is some problem already known to be $\mathcal{NP}$-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 that an $\mathcal{NP}$-hard decision problem is $\mathcal{NP}$-complete, we have just to exhibit a polynomial time nondeterministic algorithm for it.*

Later sections show many problems to be $\mathcal{NP}$-hard. Although we restrict ourselves to decision problems, it should be clear that the corresponding optimization problems are also $\mathcal{NP}$-hard. The $\mathcal{NP}$-completeness proofs are left as exercises (for those problems that are $\mathcal{NP}$-complete).

EXERCISES

1. Given two sets S_1 and S_2, the disjoint sets problem is to check whether the sets have a common element (see Section 10.3.2). Present an $O(1)$ time nondeterministic algorithm for this problem.

2. Given a sequence of n numbers, the distinct elements problem is to check if there are equal numbers (see Section 10.3, Exercise 5). Give an $O(1)$ time nondeterministic algorithm for this problem.

3. Obtain a nondeterministic algorithm of complexity $O(n)$ to determine whether there is a subset of n numbers a_i, $1 \leq i \leq n$, that sums to m.

4. (a) 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 q, then $\sum p_i \leq n2^q$, where n is the number of objects. Use a binary search to determine the optimal solution value.)

   (b) 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 $\sum p_i x_i = r$ and $\sum w_i x_i \leq m$ by making n applications of DK.

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

6. Let $\mathsf{Sat}(E)$ be an algorithm to determine whether 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 $\mathsf{Sat}(E)$ n times, one can determine a truth value assignment for the x_i's for which E is true.

7. Let π_2 be a problem for which there exists a deterministic algorithm that runs in time $2^{\sqrt{n}}$ (where n is the input size). Prove or disprove:

   > If π_1 is another problem such that π_1 is polynomially reducible to π_2, then π_1 can be solved in deterministic $O(2^{\sqrt{n}})$ time on any input of size n.

11.2 COOK'S THEOREM (*)

Cook's theorem (Theorem 11.1) states that satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$. We now prove this important theorem. We have already seen that satisfiability is in $\mathcal{NP}$ (Example 11.9). Hence, if $\mathcal{P} = \mathcal{NP}$, then satisfiability is in $\mathcal{P}$. It remains to be shown that if satisfiability is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$. To do this, we 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 is $O(p^3(n) \log n) = O(p^4(n))$. The time needed to construct Q is also $O(p^3(n) \log n)$. A deterministic algorithm Z to determine the outcome of A on any input I can be easily obtained. Algorithm Z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether Q is satisfiable. If $O(q(m))$ is the time needed to determine whether 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 $\mathcal{P}$, 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 $\mathcal{P}$, then for every nondeterministic algorithm A in $\mathcal{NP}$ we can obtain a deterministic Z in $\mathcal{P}$. So, the above construction shows that if satisfiability is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Before going into the construction of Q from A and I, we make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions do not in any way alter the class of decision problems in $\mathcal{NP}$ or $\mathcal{P}$. The simplifying assumptions are as follows.

1. The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction, and so on,

between numbers one word long take one unit of time. If 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.

2. 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 expression are $-B, B + C, D$ **or** E, and F. We assume that all assignments in A are in one of the following forms:

   (a) ⟨simple variable⟩ := ⟨simple expression⟩
   (b) ⟨array variable⟩ := ⟨simple variable⟩
   (c) ⟨simple variable⟩ := ⟨array variable⟩
   (d) ⟨simple variable⟩ := Choice(S), where S is a finite set $\{S_1, S_2, \ldots, S_k\}$ or 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 can be replaced by a set of statements of these types. Hence, this restriction does not alter the class $\mathcal{NP}$.

3. All variables in A are of type integer or boolean.

4. Algorithm A contains no **read** or **write** 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).

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

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

   (a) The statement **goto** k, where k is an instruction number.
   (b) The statement **if** c **then goto** a;. Variable c is a simple boolean variable (i.e., not an array) and a is an instruction number.
   (c) Success(), Failure().
   (d) Algorithm A may contain type declaration and dimension statements. 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 consecutive 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 **goto** instructions in (a) and (b) use this numbering scheme to effect a branch. It should be easy to see how to rewrite **repeat-until**, **for**, and so on, statements in terms of **goto** and **if** c **then goto** a statements. Also, note that the **goto** k statement can be replaced by the statement **if true then goto** k. So, this may also be eliminated.

7. 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 assumption of 1), A cannot change or use more than $p(n)$ words of memory. We assume that A uses some subset of the words indexed 1, 2, 3, $\ldots, p(n)$. This assumption does not restrict the class of decision problems in $\mathcal{NP}$. To see this, let $f(1), f(2), \ldots, f(k)$, $1 \leq k \leq p(n)$, be the distinct words used by A while working on input I. We can construct another polynomial time nondeterministic algorithm A' that uses $2p(n)$ words indexed $1, 2, \ldots, 2p(n)$ and solves the same decision problem as A does. A' simulates the behavior of A. However, A' maps the addresses $f(1), f(2), \ldots, f(k)$ onto the set $\{1, 2, \ldots, k\}$. The mapping function used is determined dynamically and is stored as a table in words $p(n)+1$ through $2p(n)$. If the entry at word $p(n)+i$ is j, then A' uses word i to hold the same value that A stored in word j. The simulation of A proceeds as follows: Let k be the number of distinct words referenced by A up to this time. Let j be a word referenced by A in the current step. A' searches its table to find word $p(n)+i$, $1 \leq i \leq k$, such that the contents of this word is j. If no such i exists, then A' sets $k := k+1$; $i := k$; 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 $\mathcal{P}$ and $\mathcal{NP}$.

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

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

   $B(i, j, t)$ 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 is constructed so that in any truth assignment for which Q is true, $B(i, j, t)$ is true if and only if the corresponding bit has value 1 following t steps of some successful computation of A on input I.

2. $S(j,t)$, $1 \leq j \leq \ell$, $1 \leq t \leq p(n)$

   Recall that ℓ is the number of instructions in A. $S(j,t)$ represents the instruction to be executed at time t. Q is constructed so that in any truth assignment for which Q is true, $S(j,t)$ is true if and only if the instruction executed by A at time t is instruction j.

Q is 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 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 ith step, there can be only one next instruction to execute. Hence, for any fixed i, exactly one of the $S(j,i)$, $1 \leq j \leq \ell$, can be true.

F: If $S(j,i)$ is true, then $S(j,i+1)$ is also true if instruction j is a Success or Failure statement. $S(j+1,i+1)$ is true if j is an assignment statement. If j is a **goto** k statement, then $S(k,i+1)$ is true. The last possibility for j is the **if** c **then** a; 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 if and only if there is a successful computation of A on input I. We now give the formulas C through H. While presenting these formulas, we also indicate how each may be transformed into CNF. This transformation increases the length of Q by an amount independent of n (but dependent on w and ℓ). This enables us to show that CNF-satisfiability is $\mathcal{NP}$-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. $D = S(1,1) \wedge \bar{S}(2,1) \wedge \bar{S}(3,1) \wedge \cdots \wedge \bar{S}(\ell,1)$

   Clearly, D is satisfiable only by the assignment $S(1,1) =$ true and $S(i,1) =$ false, $2 \le i \le \ell$. Using our interpretation of $S(i,1)$, this means that D is true if and only if 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 can define E_t to be

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

   One can verify that E_t is true iff exactly one of the $S(j,t)$'s, $1 \le j \le \ell$, is true. Also, note that E is in CNF.

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

   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:

   (a) If instruction i is Success or Failure, then L is $S(i,t+1)$. Hence the program cannot leave such an instruction.

   (b) If instruction i is **goto** k, then L is $S(k,t+1)$.

   (c) If instruction i is **if** X **then goto** k 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 if and only if X is true.

(d) 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 (a), (b), and (d) are in CNF. The $F_{i,t}$ in case (c) can 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. $G = \bigwedge_{\substack{1 \le i \le \ell \\ 1 \le t < p(n)}} G_{i,t}$

Each $G_{i,t}$ asserts that at time t either instruction i is not executed or 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:

(a) If instruction i is a **goto**, **if-then-goto-**, Success, or Failure statement, then M asserts that the status of the $p(n)$ words is unchanged; that is, $B(k,j,t-1) = B(k,j,t)$, $1 \le k \le p(n)$, $1 \le j \le w$.

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

In this case, $G_{i,t}$ can be written as

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

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 represents 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, $G_{i,t}$ can be transformed into CNF easily.

(b) If i is an assignment statement of type 2(a), then M depends on the operator (if any) on the right-hand side. We first describe the form of M for the case in which instruction i is of type $Y := V + Z$. Let Y, V, and Z be respectively represented in words y, v, and z. We make the simplifying assumption that all numbers are nonnegative. The exercises examine the case in which negative numbers

are allowed and 1's complement arithmetic is used. To get a formula asserting that the bits $B(y,j,t)$, $1 \leq j \leq w$, represent the sum of $B(v,j,t-1)$ and $B(z,j,t-1)$, $1 \leq j \leq w$, we have to make use of w additional bits $C(j,t)$, $1 \leq j \leq w$. $C(j,t)$ represents the carry from the addition of the bits $B(v,j,t-1), B(z,j,t-1)$, and $C(j-1,t)$, $1 < j \leq w$. $C(1,t)$ 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 bitwise 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 $B(y,1,t)$ can be transformed into CNF using this identity. For the other bits of Y, one can verify that

$$B(y,j,t) = B(v,j,t-1) \oplus (B(z,j,t-1) \oplus C(j-1,t)) \quad \text{and}$$

$$\begin{aligned} C(j,t) \quad = \quad & (B(v,j,t-1) \wedge B(z,j,t-1)) \vee (B(v,j,t-1) \\ & \wedge C(j-1,t)) \\ & \vee (B(z,j,t-1) \wedge C(j-1,t)) \end{aligned}$$

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 $C(j,t)$, $1 \leq j \leq w$. M is given by

$$\begin{aligned} M \quad = \quad & \Big(\bigwedge_{\substack{1 \leq k \leq p(n) \\ k \neq y \\ 1 \leq j \leq w}} ((B(k,j,t-1) \wedge B(k,j,t)) \\ & \wedge (\bar{B}(k,j,t-1) \wedge \bar{B}(k,j,t))) \wedge M' \end{aligned}$$

$G_{i,t}$ can be converted into CNF using the idea of 5(a). This transformation increases 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 forms $Y := V;$ and $Y := V \odot Z;$, for $\odot$ one of $-, /, *, <, >, \leq, =$, and so on.

When i is an assignment statement of type 2(b) or 2(c), then it is necessary to select the correct array element. Consider an instruction of type 2(b): $R[m] := X;$. In this case formula M can be written as

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

where u is the dimension of R. Note that because of restriction (7) on algorithm A, $u \leq p(n)$. 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 kth bit in the binary representation of j is 0 and T is $\bar{B}$ otherwise. Z is defined as

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

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 can then use as many words as needed. The number of words used depends 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 2(c), the form of M is similar to that obtained for instructions of type 2(b). Next, we describe how to construct M for the case in which i is of the form $Y :=$ 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. If S is a set, then we define

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

M_j asserts that Y is S_j. This is easily done by choosing $M_j = a_1 \wedge a_2 \wedge \cdots \wedge a_w$, where $a_\ell = B(y,\ell,t)$ if bit ℓ is 1 in S_ℓ and $a_\ell = \bar{B}(y,\ell,t)$ if bit ℓ is zero in S_ℓ. If S is of the form r, u, then M is just the formula

that asserts $r \leq Y \leq u$. This is left as an exercise. In both cases, $G_{i,t}$ can be transformed into CNF and the length of $G_{i,t}$ increased by at most a constant amount.

6. Let $i_1, i_2, \ldots, i_k$ be the statement numbers corresponding to 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 can readily verify that $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$ is satisfiable if and only if the computation of algorithm A with input I terminates successfully. Further, Q can be transformed into CNF as described above. Formula C contains $wp(n)$ literals, D contains ℓ literals, E contains $O(\ell^2 p(n))$ literals, F contains $O(\ell p(n))$ literals, G contains $O(\ell w p^3(n))$ literals, and H contains at most ℓ literals. The total number of literals appearing in Q is $O(\ell w p^3(n)) = O(p^3(n))$ as ℓw is constant. Since there are $O(wp^2(n)+\ell p(n))$ distinct literals in Q, each literal can be written using $O(\log(wp^2(n) + \ell p(n))) = O(\log n)$ 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 preceding construction shows that every problem in $\mathcal{NP}$ reduces to satisfiability and also to CNF-satisfiability. Hence, if either of these two problems is in $\mathcal{P}$, then $\mathcal{NP} \subseteq \mathcal{P}$ and so $\mathcal{P} = \mathcal{NP}$. Also, since satisfiability is in $\mathcal{NP}$, the construction of a CNF formula Q shows that satisfiability $\propto$ CNF-satisfiability. This together with the knowledge that CNF-satisfiability is in $\mathcal{NP}$ implies that CNF-satisfiability is $\mathcal{NP}$-complete. Note that satisfiability is also $\mathcal{NP}$-complete as satisfiability $\propto$ satisfiability and satisfiability is in $\mathcal{NP}$.

EXERCISES

1. 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 can 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.

   (a) $Y := Z$;
   (b) $Y := V - Z$;
   (c) $Y := V + Z$;
   (d) $Y := V * Z$;

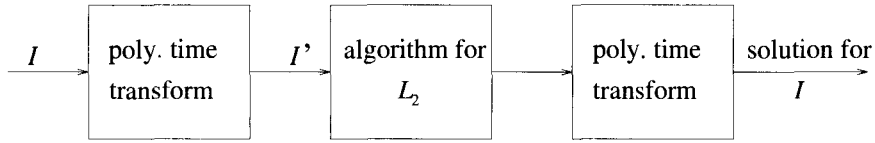


Figure 11.3 Reduction of L_1 to L_2

(e) $Y :=$ Choice$(0, 1)$;

(f) $Y :=$ Choice(r, u);, where r and u are variables

2. Show how to encode the following instructions as CNF formulas: (a) **for** and (b) **while**.

3. Prove or disprove: If there exists a polynomial time algorithm to convert a boolean formula in CNF into an equivalent formula in DNF, then $\mathcal{P} = \mathcal{NP}$.

11.3 $\mathcal{NP}$-HARD GRAPH PROBLEMS

The strategy we adopt to show that a problem L_2 is $\mathcal{NP}$-hard is:

1. Pick a problem L_1 already known to be $\mathcal{NP}$-hard.

2. 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 of L_1 (see Figure 11.3).

3. Conclude from step (2) that $L_1 \propto L_2$.

4. Conclude from steps (1) and (3) and the transitivity of $\propto$ that L_2 is $\mathcal{NP}$-hard.

For the first few proofs we go through all the above steps. Later proofs explicitly deal only with steps (1) and (2). An $\mathcal{NP}$-hard decision problem L_2 can be shown to be $\mathcal{NP}$-complete by exhibiting a polynomial time nondeterministic algorithm for L_2. All the $\mathcal{NP}$-hard decision problems we deal with here are $\mathcal{NP}$-complete. The construction of polynomial time nondeterministic algorithms for these problems is left as an exercise.

11.3.1 Clique Decision Problem (CDP)

The clique decision problem was introduced in Section 11.1. We 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 $\mathcal{NP}$-hard. Since, CDP $\in \mathcal{NP}$, CDP is also $\mathcal{NP}$-complete.

Theorem 11.2 CNF-satisfiability $\propto$ clique decision problem.

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 show how to construct from F a graph $G = (V, E)$ such that G has a clique of size at least k if and only if F is satisfiable. If the length of F is m, then G is 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: $V = \{\langle \sigma, i \rangle | \sigma$ is a literal in clause $C_i\}$ and $E = \{(\langle \sigma, i \rangle, \langle \delta, j \rangle) \mid i \neq j$ and $\sigma \neq \bar{\delta}\}$. A sample construction is given in Example 11.11.

Claim: F is satisfiable if and only if G has a clique of size $\geq k$.

Proof of Claim: 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. Between any two nodes $\langle \sigma, i \rangle$ and $\langle \delta, j \rangle$ in S there is an edge in G, since $i \neq j$ and both σ and δ have the value true. Thus, 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 of Figure 11.4. 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 (that is, $x_2 =$ false), F is satisfied. The x_3 may be set either to true or false. □

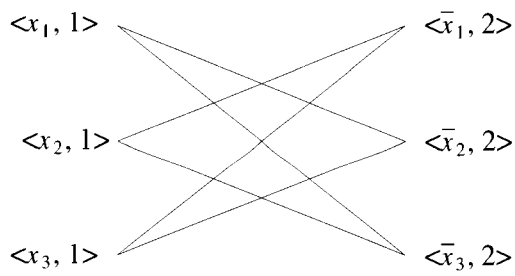


Figure 11.4 A sample graph and satisfiability

11.3.2 Node Cover Decision Problem (NCDP)

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

Example 11.12 Consider the graph of Figure 11.5. $S = \{2, 4\}$ is a node cover of size 2. $S = \{1, 3, 5\}$ is a node cover of size 3. □

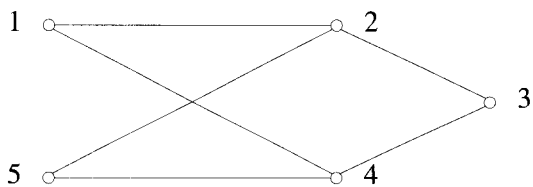


Figure 11.5 A sample graph and node cover

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

Theorem 11.3 The clique decision problem $\propto$ the node cover decision problem.

Proof: Let $G = (V, E)$ and k define an instance of CDP. Assume that $|V| = n$. We construct a graph G' such that G' has a node cover of size at

most $n-k$ if and only if G has a clique of size at least k. Graph G' is given by $G' = (V, \bar{E})$, where $\bar{E} = \{(u,v) \mid u \in V, v \in V \text{ and } (u,v) \notin E\}$. The set G' is known as the *complement* of G.

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

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

Example 11.13 Figure 11.6 shows a graph G and its complement G'. In this figure, G' has a node cover of $\{4,5\}$, since every edge of G' is incident either on the node 4 or on the node 5. Thus, G has a clique of size $5-2=3$ consisting of the nodes $1, 2$, and 3. □

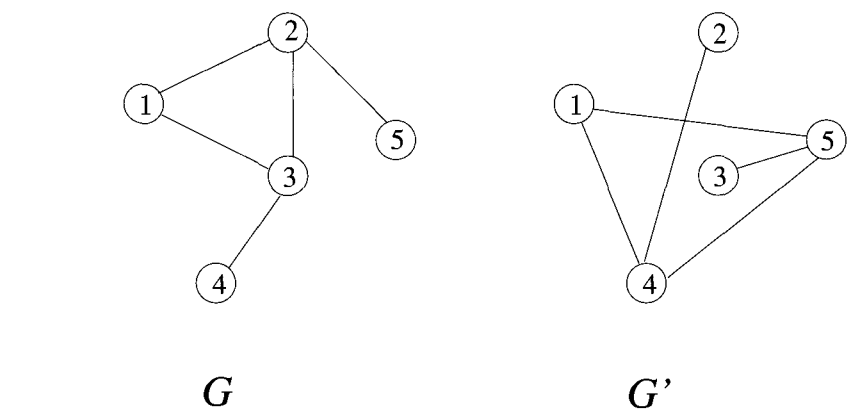


Figure 11.6 A graph and its complement

Note that since CNF-satisfiability $\propto$ CDP, CDP $\propto$ NCDP and $\propto$ is transitive, it follows that NCDP is $\mathcal{NP}$-hard. NCDP is also in $\mathcal{NP}$ because we can nondeterministically choose a subset $C \subseteq V$ of size k and verify in polynomial time that C is a cover of G. So NCDP is $\mathcal{NP}$-complete.

11.3.3 Chromatic Number Decision Problem (CNDP)

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* is to determine whether G has a coloring for a given k.

Example 11.14 A possible 2-coloring of the graph of Figure 11.5 is $f(1) = f(3) = f(5) = 1$ and $f(2) = f(4) = 2$. Clearly, this graph has no 1-coloring. □

In proving CNDP to be $\mathcal{NP}$-hard, we shall make use of the $\mathcal{NP}$-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 $\propto$ chromatic number decision problem.

Proof: Let F be a CNF formula having at most three literals per clause and having r clauses $C_1, C_2, \ldots, C_r$. Let x_i, $1 \leq i \leq n$, be the n variables in F. We can assume $n \geq 4$. If $n < 4$, then we can determine whether F is satisfiable by trying out all eight possible truth value assignments to x_1, x_2, and x_3. We construct, in polynomial time, a graph G that is $n+1$ colorable if and only if F is satisfiable. The graph $G = (V, E)$ is defined by

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

where $y_1, y_2, \ldots, y_n$ are new variables, and

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

To see that G is $n+1$ colorable if and only if 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 can 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 $n+1$ is called a *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 can be assigned the color $n+1$. Also, no C_i can 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 if and only if there is a true vertex corresponding to each C_i. So, G is $n+1$ colorable iff F is satisfiable. □

11.3.4 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 whether G has a directed Hamiltonian cycle.

Example 11.15 1, 2, 3, 4, 5, 1 is a directed Hamiltonian cycle in the graph of Figure 11.7. If the edge $\langle 5, 1\rangle$ is deleted from this graph, then it has no directed Hamiltonian cycle. □

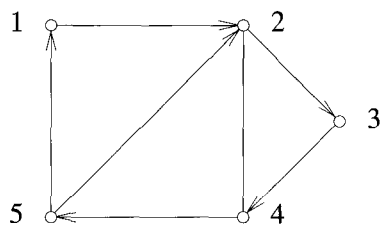


Figure 11.7 A sample graph and Hamiltonian cycle

Theorem 11.5 CNF-satisfiability $\propto$ directed Hamiltonian cycle.

Proof: Let F be a propositional formula in CNF. We show how to construct a directed graph G such that F is satisfiable if and only if 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 the construction of G is greatly facilitated by the use of an example. The example we 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 denotes clause C_i. Each variable x_i is represented by two adjacent columns, one for each of the literals x_i and $\bar{x}_i$. Figure 11.8 shows the array for the example formula. Insert a $\odot$ into column x_i and row C_j if and only if x_i is a literal in C_j. Insert a $\odot$ into column $\bar{x}_i$ and row C_j if and only if $\bar{x}_i$ 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 upward 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 11.8). Now, draw edges $\langle u_i, v_{i+1}\rangle$, $1 \leq i < n$. Introduce a box $\boxed{\text{i}}$ at the right end of each row C_i, $1 \leq i < r$. Draw the edges $\langle u_n, \boxed{1}\rangle$ and $\langle \boxed{\text{r}}, v_1\rangle$. Draw edges $\langle \boxed{\text{i}}, \boxed{\text{i+1}}\rangle$, $1 \leq i < r$ (see Figure 11.8).

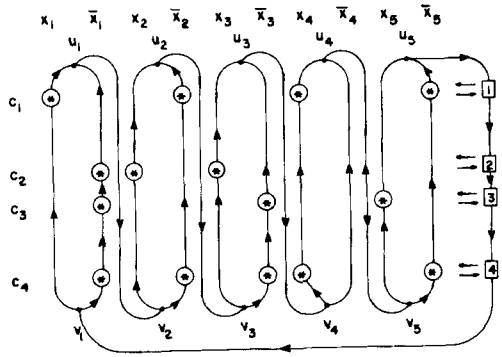


Figure 11.8 Array structure for the formula in Theorem 11.5

To complete the graph, we replace each $\odot$ and $\boxed{\text{i}}$ by a subgraph. Each $\odot$ is replaced by the subgraph of Figure 11.9(a) (of course, unique vertex labelings are needed for each copy of the subgraph). Each box $\boxed{\text{i}}$ is replaced by the subgraph of Figure 11.10. In this subgraph A_i is an entrance node and B_i an exit node. The edges $\langle \boxed{\text{i}}, \boxed{\text{i+1}}\rangle$ 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 $\langle \boxed{\text{r}}, v_1\rangle$ is $\langle B_r, v_1\rangle$. The variable j_i is the number of literals in clause C_i. In the subgraph of Figure 11.10 an edge of the type shown in Figure 11.11 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.

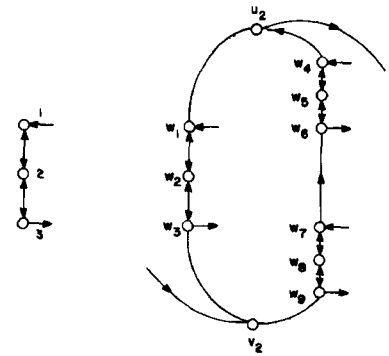


Figure 11.9 The $\odot$ subgraph and its insertion into column 2

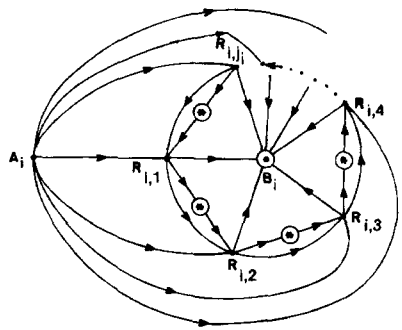


Figure 11.10 The H_i subgraph

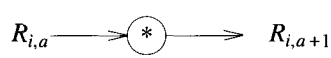


Figure 11.11 A construct in the proof of Theorem 11.5

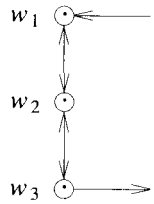


Figure 11.12 Another construct in the proof of Theorem 11.5

Thus in the $\odot$ subgraph (shown in Figure 11.12) of Figure 11.9(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 and go to u_1, then to v_2, then to u_2, then to v_3, then to $u_3, \ldots$, and then to u_n. In going from v_i to u_i, this cycle uses the column corresponding to x_i if x_i is true in S. Otherwise it goes up the column corresponding to $\bar{x}_i$. From u_n this cycle goes to A_1 and then through $R_{1,1}, R_{1,2}, R_{1,3}, \ldots, R_{1,j_1}$, and B_1 to A_2 to $\cdots$ to v_1. In going from $R_{i,a}$ to $R_{i,a+1}$ in any subgraph $\boxed{\text{i}}$, a diversion is made to a $\odot$ subgraph in row i if and only if the vertices of that $\odot$ subgraph are not already on the path from v_i to $R_{i,a}$. Note that if C_i has i_j literals, then the construction of $\boxed{\text{i}}$ 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 (because 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 can be seen by starting at vertex v_1 on any Hamiltonian cycle for G. Because of the construction of the $\odot$ and $\boxed{\text{i}}$ 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, $1 \leq i \leq n$, define a truth assignment for which F is true.

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

11.3.5 Traveling Salesperson Decision Problem (TSP)

The traveling salesperson problem was introduced in Chapter 5. The corresponding decision problem is to determine whether a complete directed

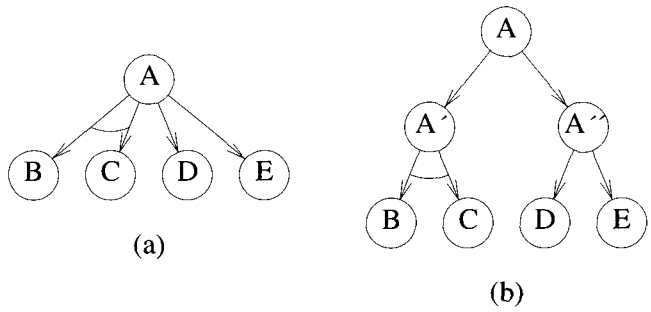


Figure 11.13 Graphs representing problems

graph $G = (V, E)$ with edge costs $c(u, v)$ has a tour of cost at most M.

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

Proof: From the directed graph $G = (V, E)$ construct the complete directed graph $G' = (V, E')$, $E' = \{\langle i, j\rangle \mid i \neq j\}$ 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. □

11.3.6 AND/OR Graph Decision Problem (AOG)

Many complex problems can be broken down into a series of subproblems such that the solution of all or some of these results in the solution of the original problem. These subproblems can be broken down further into subsubproblems, and so on, until the only problems remaining are sufficiently primitive as to be trivially solvable. This breaking down of a complex problem into several subproblems can be represented by a directed graphlike structure in which nodes represent problems and descendents of nodes represent the subproblems associated with them.

Example 11.16 The graph of Figure 11.13(a) represents a problem A that can be solved by solving either both the subproblems B and C or the single subproblem D or E. □

Groups of subproblems that must be solved in order to imply a solution to the parent node are joined together by an arc going across the respective edges (as the arc across the edges $\langle A, B\rangle$ and $\langle A, C\rangle$). By introducing dummy

nodes in Figure 11.13(b), all nodes can be made to be such that their solution requires either all descendents to be solved or only one descendent to be solved. Nodes of the first type are called AND nodes and those of the latter type OR nodes. Nodes A and A'' of Figure 11.13(b) are OR nodes whereas node A' is an AND node. The AND nodes are drawn with an arc across all edges leaving the node. Nodes with no descendents are called *terminal*. Terminal nodes represent primitive problems and are marked either solvable or not solvable. Solvable terminal nodes are represented by rectangles. An AND/OR graph need not always be a tree.

Breaking down a problem into several subproblems is known as *problem reduction*. Problem reduction has been used on such problems as theorem proving, symbolic integration, and analysis of industrial schedules. When problem reduction is used, two different problems may generate a common subproblem. In this case it may be desirable to have only one node representing the subproblem (this would imply that the subproblem is to be solved only once). Figure 11.14 shows two AND/OR graphs for cases in which this is done.

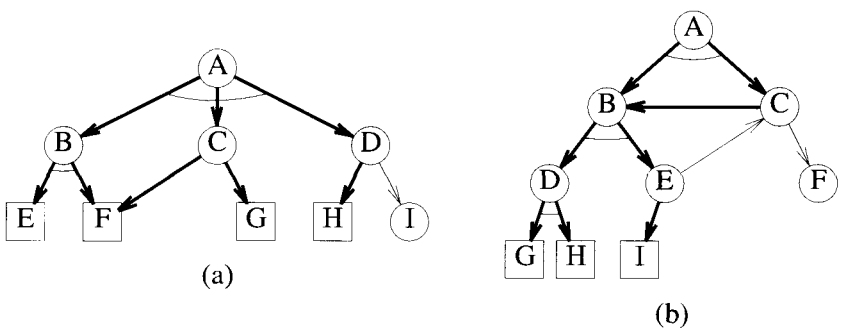


Figure 11.14 Two AND/OR graphs that are not trees

Note that the graph is no longer a tree. Furthermore, such graphs may have directed cycles as in Figure 11.14(b). The presence of a directed cycle does not in itself imply the unsolvability of the problem. In fact, problem A of Figure 11.14(b) can be solved by solving the primitive problems G, H, and I. This leads to the solution of D and E and hence of B and C. A *solution graph* is a subgraph of solvable nodes that shows that the problem is solved. Possible solution graphs for the graphs of Figure 11.14 are shown by heavy edges.

Let us assume that there is a cost associated with each edge in the AND/OR 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 whether G has a solution graph of cost at most k, for k a given input.

Example 11.17 Consider the directed graph of Figure 11.15. The problem to be solved is P_1. To do this, one can solve node 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 can solve either P_5 or P_6. The minimum cost to do this is 1. Node P_7 is free. In this example, then, the optimal way to solve P_1 is to solve P_6 first, then P_3, and finally P_1. The total cost for this solution is 3. □

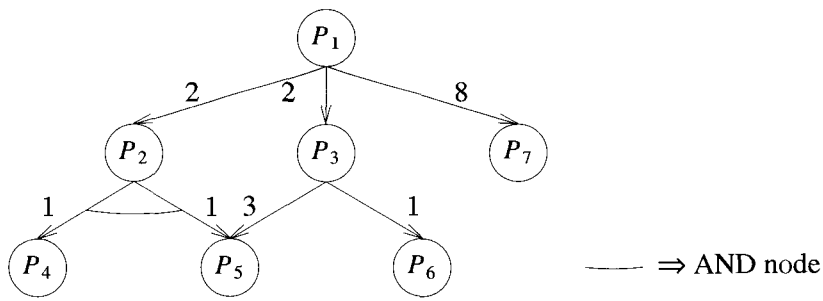


Figure 11.15 AND/OR graph

Theorem 11.7 CNF-satisfiability $\propto$ the 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 if and only if P is satisfiable. Let

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

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. The node 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 that is terminal (i.e., has no edges leaving it). These terminal nodes are 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 clause C_i. This is done for all variables x_j appearing in the clause C_i. Clause 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 of 1.

3. All other edges have a cost of 0.

In order to solve S, each of the nodes $P, x_1, x_2, \ldots, x_n$ must be solved. Solving nodes $x_1, x_2, \cdots, x_n$ 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. That is, 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 of true. From this it follows that the entire graph (that is, 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 if and only if P is satisfiable. Otherwise the cost is more than n. The construction clearly takes only polynomial time. This completes the proof. □

Example 11.18 Consider the formula

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

Figure 11.16 shows the AND/OR graph obtained by applying the construction of Theorem 11.7.

The nodes Tx_1, Tx_2, and Tx_3 can be solved at a total cost of 3. The node P 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 of true to the variables of P results in P's being true. □

EXERCISES

1. 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 have to introduce some new variables. Any assignment that satisfies the original clause must satisfy all the new clauses created.

2. Let SAT3 be similar to SATY (Exercise 1) except that each clause has exactly three literals. Show that SATY $\propto$ SAT3.

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

4. 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 3).

5. [Feedback node set]

   (a) Let $G = (V, E)$ be a directed graph. Let $S \subseteq V$ be a subset of vertices such that the 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 whether G has a feedback node set of size at most k. Show that the node cover decision problem $\propto$ FNS.

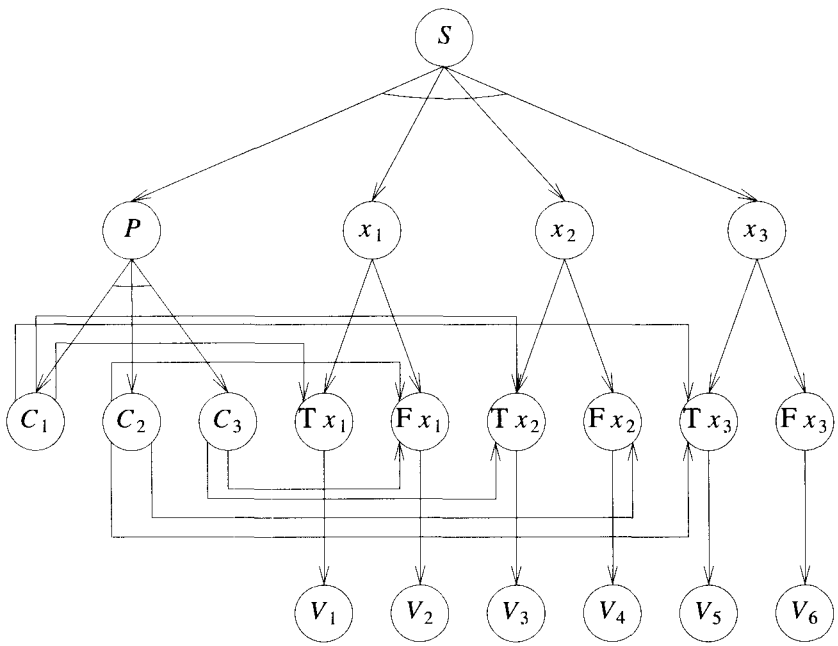


Figure 11.16 AND/OR graph for Example 11.18

(b) Write a polynomial time nondeterministic algorithm for FNS.

6. [Feedback arc set] Let $G = (V, E)$ be a directed graph. $S \subseteq E$ is a feed back arc set of G if and only if every directed cycle in G contains an edge in S. The feedback arc set decision problem (FAS) is to determine whether G has a feedback arc set of size at most k.

   (a) Show that the node cover decision problem $\propto$ FAS.

   (b) Write a polynomial time nondeterministic algorithm for FAS.

7. The feedback node set optimization problem is to find a minimum feedback node set (see Exercise 5). Show that this problem reduces to FNS.

8. Show that the feedback arc set minimization problem reduces to FAS (Exercise 6).

9. [Hamiltonian cycle] Let UHC be the problem of determining whether 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).

10. Show UHC $\propto$ CNF-satisfiability.

11. Show DHC $\propto$ CNF-satisfiability.

12. [Hamiltonian path] An i to j Hamiltonian path in 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 whether G has an i to j Hamiltonian path.

13. [Minimum equivalent graph] A directed graph $G = (V, E)$ is an equivalent graph of the directed graph $G' = (V, E')$ if and only if $E \subseteq E'$ and the transitive closures of G and G' are the same. G is a minimum equivalent graph if and only if $|E|$ is minimum among all equivalent graphs of G'. The minimum equivalent graph decision problem (MEG) is to determine whether 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.

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

15. [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 whether F has a cover T containing no more than k sets. Show that the node cover decision problem is reducible to this problem.

16. [Exact cover] Let $F = \{S_j\}$ be as in Exercise 15. $T \subseteq F$ 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 whether F has an exact cover.

17. Show that SAT3 $\propto$ EXACT COVER (see Exercise 16).

18. [Hitting set] Let F be as in Exercise 16. The hitting set problem is to determine whether 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.

19. [Tautology] A propositional formula is a tautology if and only if it is true for all possible truth assignments to its variables. The tautology problem is to determine whether 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 if and only if F is not a tautology.

20. [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 if and only if G is true. Show that deciding whether F has an equivalent formula of length no more than k is $\mathcal{NP}$-hard. (*Hint*: Show DNF tautology reduces to this problem.)

11.4 $\mathcal{NP}$-HARD SCHEDULING PROBLEMS

To prove the results of this section, we need to use the $\mathcal{NP}$-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 $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$. We can show this problem is $\mathcal{NP}$-hard by first showing the sum of subsets problem (Chapter 7) to be $\mathcal{NP}$-hard. Recall that in the sum of subsets problem we have to determine whether $A = \{a_1, a_2, \ldots, a_n\}$ 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 $\mathcal{NP}$-hard in Section 11.3, Exercise 16. In this problem we are given a family of sets $F = \{S_1, S_2, \ldots, S_k\}$ and are required to determine whether 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 subsets problem $A = \{a_1, \ldots, a_k\}$ with $a_j = \sum_{1 \le i \le n} \epsilon_{ji}(k+1)^{i-1}$, where $\epsilon_{ji} = 1$ if $u_i \in S_j$ and $\epsilon_{ji} = 0$ otherwise, and $M = \sum_{0 \le i < n}(k+1)^i = ((k+1)^n - 1)/k$. Clearly, F has an exact cover if and only if $A = \{a_1, \ldots, a_k\}$ has a subset with sum M. Since A and M can 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 $b_{n+2} = (\sum_{1 \le i \le n} a_i) + 1 - M$. B has a partition if and only if A has a subset with sum M. Since B can be obtained from A and M in polynomial time, sum of subsets $\propto$ partition. □

One can easily show partition $\propto$ 0/1-knapsack and partition $\propto$ job sequencing with deadlines. Hence, these problems are also $\mathcal{NP}$-hard.

11.4.1 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* (FT) of S is

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

Schedule S is a *nonpreemptive schedule* if and only if 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 4.6 and nonpreemptive 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 4.6. Obtaining minimum weighted mean finish time and minimum finish time nonpreemptive schedules is $\mathcal{NP}$-hard.

Theorem 11.10 Partition $\propto$ minimum finish time nonpreemptive schedule.

Proof: We 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 nonpreemptive schedule for this set of jobs on two processors with finish time at most $\sum t_i/2$ iff there is a partition of the a_i's. □

Example 11.19 Consider the following input to the partition problem: $a_1 = 2, a_2 = 5, a_3 = 6, a_4 = 7$, and $a_5 = 10$. The corresponding minimum finish time nonpreemptive schedule problem has the input $t_1 = 2, t_2 = 5, t_3 = 6, t_4 = 7$, and $t_5 = 10$. There is a nonpreemptive schedule for this set of jobs with finish time 15: P_1 takes the jobs t_2 and t_5; P_2 takes the jobs t_1, t_3, and t_4. This solution yields a solution for the partition problem also: $\{a_2, a_5\}, \{a_1, a_3, a_4\}$. □

Theorem 11.11 Partition $\propto$ minimum WMFT nonpreemptive schedule.

Proof: Once again we prove this for $m = 2$ only. The extension to $m > 2$ is trivial. Let a_i, $1 \le i \le n$, define an instance of the partition problem. Construct a two-processor scheduling problem with n jobs and $w_i = t_i = a_i$, $1 \le i \le n$. For this set of jobs there is a nonpreemptive schedule S with weighted mean flow time at most $1/2 \sum a_i^2 + 1/4(\sum a_i)^2$ if and only if 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

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

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

Example 11.20 Consider again the partition problem $a_1 = 2, a_2 = 5, a_3 = 6, a_4 = 7$, and $a_5 = 10$. Here, $\frac{1}{2}\sum a_i^2 = \frac{1}{2}(2^2 + 5^2 + 6^2 + 7^2 + 10^2) = 107$, $\sum a_i = 30$, and $\frac{1}{4}(\sum a_i)^2 = 225$. Thus, $1/2\sum a_i^2 + 1/4(\sum a_i)^2 = 107 + 225 = 332$. The corresponding minimum WMFT nonpreemptive schedule problem has the input $w_i = t_i = a_i$ for $1 \leq i \leq 5$. If we assign the jobs t_2 and t_5 to P_1 and the remaining jobs to P_2,

$$n * \text{WFMT}(S) = 5 * 5 + 10(5 + 10) + 2 * 2 + 6(2 + 6) + 7(2 + 6 + 7) = 332$$

The same also yields a solution to the partition problem. □

11.4.2 Flow Shop Scheduling

We shall use the flow shop terminology developed in Section 5.10. 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 nonpreemptive) is $\mathcal{NP}$-hard. For the case of nonpreemptive schedules this is easy to see (Exercise 2). We prove the result for preemptive schedules. The proof we give is also valid for the nonpreemptive case. However, a much simpler proof exists for the nonpreemptive case.

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

Proof: We 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:

$$\begin{aligned} &t_{1,i} = a_i; \quad t_{2,i} = 0; \quad t_{3,i} = a_i, 1 \leq i \leq n \\ &t_{1,n+1} = T/2; \quad t_{2,n+1} = T; \quad t_{3,n+1} = 0 \\ &t_{1,n+2} = 0; \quad t_{2,n+2} = T; \quad t_{3,n+2} = T/2 \end{aligned}$$

$$\text{where } T = \sum_{1}^{n} a_i$$

We now show that the preceding flow shop instance has a preemptive schedule with finish time at most $2T$ if and only if A has a partition.

1. If A has a partition u, then there is a nonpreemptive schedule with finish time $2T$. One such schedule is shown in Figure 11.17.

2. 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 preemptive schedule for FS with finish time at most $2T$. We make the following observations regarding this schedule:

   (a) 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.

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

Observation (a) 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 (b) 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 $2T$. □

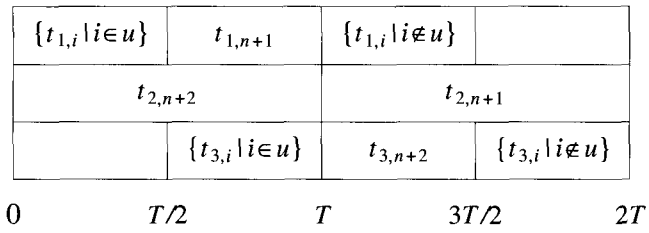


Figure 11.17 A possible schedule

11.4.3 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 $1, 2, 3, \ldots$, and so on. 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 nonpreemptive 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 schedule or a minimum finish time nonpreemptive schedule is $\mathcal{NP}$-hard even when $m = 2$. The proof for the nonpreemptive case is very simple (use partition). We present the proof for the preemptive case. This proof will also be valid for the nonpreemptive 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 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.

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

$$\text{where } T = \sum_{1}^{n} a_i$$

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

1. If A has a partition, u then there is a schedule with finish time $2T$ (see Figure 11.18).

2. 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 11.18. 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, $\sum_{j \in R'} t_{i,j,1} < T/2$. Consequently, $\sum_{j \in R'} t_{2,j,2} < T/2$. Since only the second tasks of jobs in R' can 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 $2T$. □

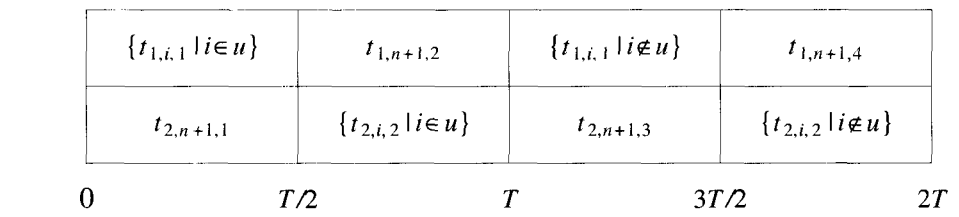


Figure 11.18 Another schedule

EXERCISES

1. [Job sequencing] Show that the job sequencing with deadlines problem (Section 8.1.4) is $\mathcal{NP}$-hard.

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

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

4. 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 $\mathcal{NP}$-hard. (*Hint*: Use partition.)

5. Let J_i, $1 \leq i \leq n$, be n jobs as in Exercise 4. 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 $\sum w_i T_i$. Show that finding a schedule minimizing $\sum w_i T_i$ is $\mathcal{NP}$-hard. (*Hint*: Use partition).

6. 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 $\sum w_i f_i$ is $\mathcal{NP}$-hard.

7. Show that the problem of obtaining optimal finish time preemptive schedules for a two-processor flow shop is $\mathcal{NP}$-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.

11.5 $\mathcal{NP}$-HARD CODE GENERATION PROBLEMS

The function of a compiler is to translate programs written in some source language into an equivalent assembly language or machine language program. Thus, the C++ compiler on the Sparc 10 translates C++ programs into the machine language of this machine. We look at the problem of translating arithmetic expressions in a language such as C++ into assembly language code. The translation clearly depends on the particular assembly language (and hence machine) being used. To begin, we assume a very simple machine model. We call this model machine A. This machine has only one register called the *accumulator*. All arithmetic has to be performed in this register. If $\odot$ represents a binary operator such as $+, -, *$, and $/$, then the left operand of $\odot$ must be in the accumulator. For simplicity, we restrict ourselves to these four operators. The discussion easily generalizes to other operators. The relevant assembly language instructions are:

LOAD X load accumulator with contents of memory location X.
STORE X store contents of accumulator into memory location X.
OP X OP may be ADD, SUB, MPY, or DIV.

The instruction OP X computes the operator OP using the contents of the accumulator as the left operand and that of memory location X as the right operand. As an example, consider the arithmetic expression $(a+b)/(c+d)$. Two possible assembly language versions of this expression are given in Figure 11.19. $T1$ and $T2$ are temporary storage areas in memory. In both

cases the result is left in the accumulator. Code (a) is two instructions longer than code (b). If each instruction takes the same amount of time, then code (b) will take 25% less time than code (a). For the expressions $(a+b)/(c+d)$ and the given machine A, it is easy to see that code (b) is optimal.

LOAD	a	LOAD	c
ADD	b	ADD	d
STORE	$T1$	STORE	$T1$
LOAD	c	LOAD	a
ADD	d	ADD	b
STORE	$T2$	DIV	$T1$
LOAD	$T1$		
DIV	$T2$		
(a)		(b)	

Figure 11.19 Two possible codes for $(a + b)/(c + d)$

Definition 11.7 A *translation* of an expression E into the machine or assembly language of a given machine is *optimal* if and only if it has a minimum number of instructions. □

Definition 11.8 A binary operator $\odot$ is *commutative* in the domain D iff $a \odot b = b \odot a$ for all a and b in D. □

Machine A can be generalized to another machine B. Machine B has $N \geq 1$ registers in which arithmetic can be performed. There are four types of machine instructions for B:

1. LOAD M, R
2. STORE M, R
3. OP $R1, M, R2$
4. OP $R1, R2, R3$

These four instruction types perform the following functions:

1. LOAD M, R places the contents of memory location M into register $R, 1 \leq R \leq N$.
2. STORE M, R stores the contents of register $R, 1 \leq R \leq N$, into memory location M.

3. OP $R1$, M, $R2$ computes contents($R1$) OP contents(M) and places the result in register $R2$. OP is any binary operator (for example, +, −, ∗, or /); $R1$ and $R2$ are registers; $R1$ may equal $R2$; M is a memory location.

4. OP $R1, R2, R3$ is similar to instruction type (3). Here $R1, R2$, and $R3$ are registers. Some or all of these registers may be the same.

In comparing the two machine models A and B, we note that when $N = 1$, instructions of types (1), (2) and (3) for model B are the same as the corresponding instructions for model A. Instructions of type (4) only allow trivial operations like $a+a, a-a, a*a$, and a/a to be performed without an additional memory access. This does not change the number of instructions in the optimal codes for A and B when $N = 1$. Hence, model A is in a sense identical to model B when $N = 1$. For model B, we see that the optimal code for a given expression E may be different for different values of N. Figure 11.20 shows the optimal code for the expression $(a + b)/(c * d)$. Two cases are considered, $N = 1$ and $N = 2$. Note that when $N = 1$, one store has to be made whereas when $N = 2$, no stores are needed. The registers are labeled $R1$ and $R2$. Register $T1$ is a temporary storage location in memory.

LOAD	$c, R1$	LOAD	$c, R1$
MPY	$R1, d, R1$	MPY	$R1, d, R1$
STORE	$R1, T1$	LOAD	$a, R2$
LOAD	$a, R1$	ADD	$R2, b, R2$
ADD	$R1, b, R1$	DIV	$R2, R1, R1$
DIV	$R1, T1, R1$		
(a) $N = 1$		(b) $N = 2$	

Figure 11.20 Optimal codes for $N = 1$ and $N = 2$

Given an expression E, the first question we ask is: can E be evaluated without any STOREs? A closely related question is: what is the minimum number of registers needed to evaluate E without any stores? We show that this problem is $\mathcal{NP}$-hard.

11.5.1 Code Generation with Common Subexpressions

When arithmetic expressions have common subexpressions, they can 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 are 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. Node P is the parent of its children. Figure 11.21 shows some expressions and their dag representations.

Definition 11.9 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. □

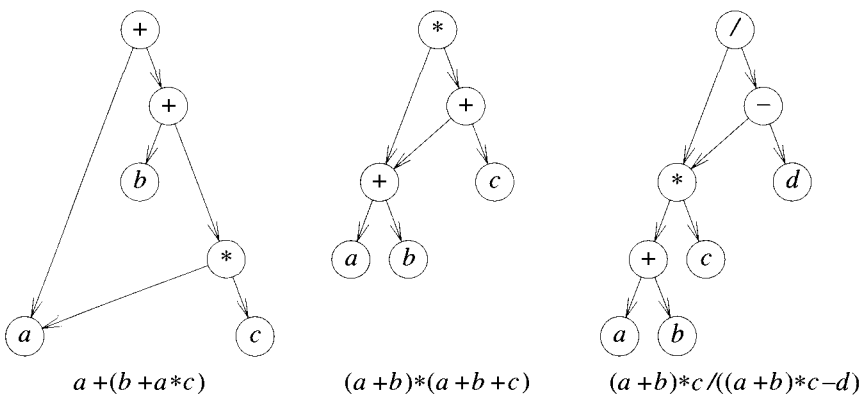


Figure 11.21 Expressions and their dags

Example 11.21 The dag of Figure 11.21(a) is a leaf dag. Figure 11.21(b) is a level-one dag. Figure 11.21(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 $\odot$ b, where a and b are simple variables or constants and $\odot$ is an operator.

The problem of generating optimal code for level-one dags is $\mathcal{NP}$-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 $\mathcal{NP}$-hard.

Example 11.22 The optimal codes for the dag of Figure 11.21(b) for one- and two-register machines is given in Figure 11.22.

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

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 11.22 Optimal codes for one- and two-register machines

To prove the above statements, we use the feedback node set (FNS) problem that is shown to be $\mathcal{NP}$-hard in Exercise 5 (Section 11.3).

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

We explicitly prove only that generating optimal code is $\mathcal{NP}$-hard. Using the construction of this proof, we can also show that determining the minimum number of registers needed to evaluate a dag with no STOREs is $\mathcal{NP}$-hard as well. The proof assumes that expressions can 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 R. Sethi, the proof is easily extended to the case in which only noncommutative operations are allowed (see Exercise 1).

Theorem 11.14 FNS $\propto$ the 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 construct a dag A with the property that the optimal code for the expression corresponding to A has at most $n + k$ LOADs if and only if 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 (for example, +). Leaf nodes represent distinct variables. We 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 11.23 and Figure 11.23). Vertex 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 tails 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 11.23(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 $\odot$ 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 can 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} can 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} can 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 can 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 can 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. If we had used noncommutative operators for chain nodes and made each successor on a chain the left child of its parent, then the p LOADs would 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 11.23 Figure 11.23(b) shows the dag A corresponding to the graph G of Figure 11.23(a). The set $\{r, s\}$ is a minimum feedback node set for G. The operator in each chain and tree node can be assumed to be $+$. Each code for A has a load corresponding to one of $(p_L, p_R), (q_L, q_R), \ldots,$ 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, and a. Note that a LOAD is needed to compute s_1 and also to compute r_3. □

11.5.2 Implementing Parallel Assignment Instructions

A *parallel assignment instruction* has the format $(v_1, v_2, \ldots, v_n) := (e_1, e_2;, \ldots, e_n)$ where the v_i's are distinct variable names and the e_i's 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, $1 \leq i \leq n$. 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.

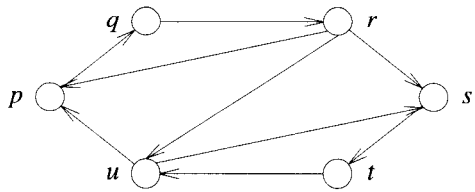
Example 11.24

1. $(A, B) := (B, C)$; is equivalent to $A := B$; $B := C$;.
2. $(A, B) := (B, A)$; is equivalent to $T := A$; $A := B$; $B := T$;.
3. $(A, B) := (A + B, A - B)$; is equivalent to $T1 := A$; $T2 := B$; $A := T1+T2$; $B := T1-T2$; and also to $T1 := A$; $A := A+B$; $B := T1-B$;.

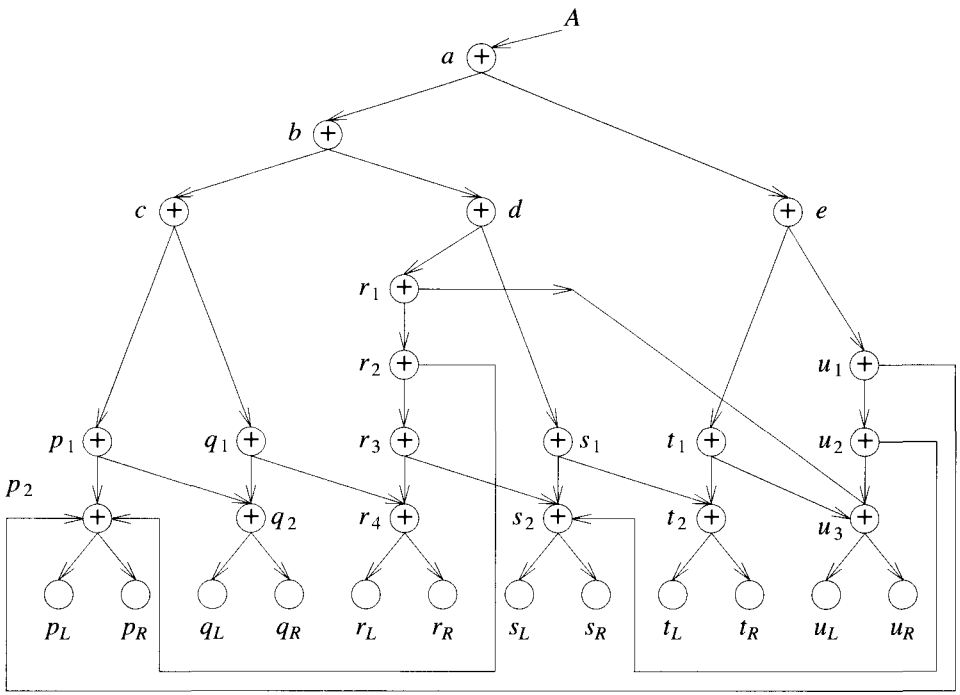
□

As the above example indicates, it may be necessary to store some of the v_i's in temporary locations when executing a parallel assignment. These stores are needed only when some of the v_i's appear in the expressions e_j, $1 \leq j \leq n$. A variable v_i is *referenced* by expression e_i if and only if v_i appears in e_j. It should be clear that only referenced variables need to be copied into temporary locations. Further, parts (2) and (3) of Example 11.24 show that not all referenced variables need to copied.

An implementation of a parallel assignment statement is a sequence of instructions of types $T_j = v_i$ and $v_i = e_i'$, where e_i' is obtained from e_i by replacing all occurrences of a v_i that have already been updated with references to the temporary locations in which the old values of v_i has been



a) Graph G



b) Corresponding dag A

Figure 11.23 A graph and its corresponding dag

saved. Let $R = (\tau(1), \ldots, \tau(n))$ be a permutation of $(1, 2, \ldots, n)$. Then R is a *realization* of an assignment statement. It specifies the order in which statements of type $v_i = e'_i$ appear in an implementation of a parallel assignment statement. The order is $v_{\tau(1)} = e'_{\tau(1)}, v_{\tau(2)} = e'_{\tau(2)}$, and so on. The implementation also has statements of type $T_j = v_i$ interspersed. Without loss of generality we can assume that the statement $T_j = v_i$ (if it appears in the implementation) immediately precedes the statement $v_i = e'_i$. Hence, a realization completely characterizes an implementation. The minimum number of instructions of type $T_j = v_i$ 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 $v_j = e'_j$ that appears after the instruction $v_i = e'_i$.

Example 11.25 Consider the statement $(A, B, C) := (D, A + B, A - B)$;. The $3! = 6$ different realizations and their costs are given in Figure 11.24. The realization 3, 2, 1 corresponding to the implementation $C = A - B; B = A + B; A = D;$ needs no temporary stores ($C(R) = 0$). □

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

Figure 11.24 Realization for Example 11.25

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 $\mathcal{NP}$-Hard. We prove this statement using the feedback node set problem.

Theorem 11.15 FNS $\propto$ the minimum-cost realization.

Proof: Let $G = (V, E)$ be any n-vertex directed graph. Construct the parallel assignment statement $P : (v_1, v_2, \ldots, v_n) := (e_1, e_2, \ldots, e_n)$, where the v_i's correspond to the n vertices in V and e_i is the expression $v_{i_1} + v_{i_2} + \cdots + v_{i_j}$. The set $\{v_{i_1}, v_{i_2}, \ldots, v_{i_j}\}$ is the set of vertices adjacent from v_i

(that is, $\langle v_i, v_{i_j} \rangle \in E(G)$, $1 \leq l \leq j$). This construction requires at most $O(n^2)$ time.

Let U be any feedback node set for G. Let $G' = (V', E') = (V - U, E - \{\langle x, y \rangle | x \in U \text{ or } y \in U\})$ 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 $V - U$ can 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 and j, $1 \leq i < j \leq m$. Hence, an implementation of P in which variables corresponding to vertices in U are first stored in temporary locations followed by the instructions $v_i = e'_i$ 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. (Note that 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 if and only if 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. □

EXERCISES

1. (a) How should the proof of Theorem 11.14 be modified to permit recomputation of shared nodes?

   (b) [R. Sethi] Modify the proof of Theorem 11.14 so that it holds for level-one dags representing expressions in which all operators are noncommutative. (*Hint*: Designate the successor vertex on a chain to be the left child of its predecessor vertex and use the $n+1$ node binary tree of Figure 11.25 to connect together the tail nodes of the n chains.)

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

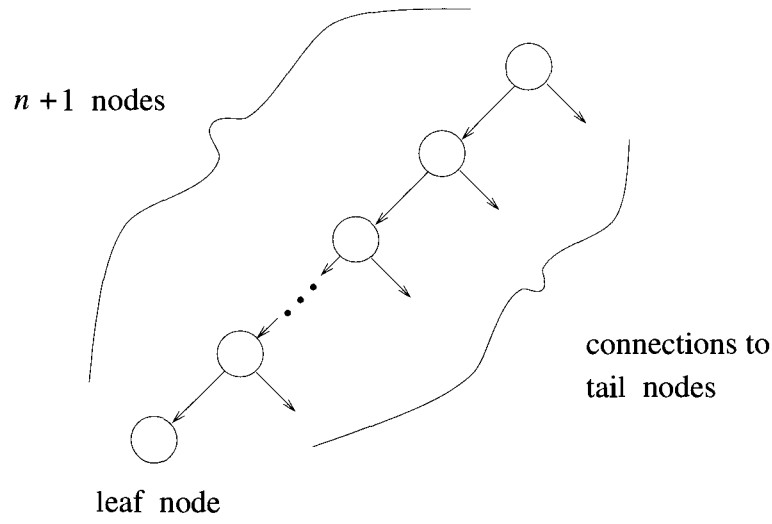


Figure 11.25 Figure for Exercise 1

11.6 SOME SIMPLIFIED $\mathcal{NP}$-HARD PROBLEMS

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

1. CNF-satisfiability with at most three literals per clause is $\mathcal{NP}$-hard. If each clause is restricted to have at most two literals, then CNF-satisfiability is polynomially solvable.

2. Generating optimal code for a parallel assignment statement is $\mathcal{NP}$-hard. However, if the expressions e_i are restricted to be simple variables, then optimal code can be generated in polynomial time.

3. Generating optimal code for level-one dags is $\mathcal{NP}$-hard, but optimal code for trees can be generated in polynomial time.

4. Determining whether a planar graph is three colorable is $\mathcal{NP}$-hard. To determine whether it is two colorable, we only have to see whether it is bipartite.

Since it is very unlikely that $\mathcal{NP}$-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 $\mathcal{NP}$-hard or $\mathcal{NP}$-complete.

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

Theorem 11.16 The following decision problems are $\mathcal{NP}$-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 three 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, $1 \leq i \leq m$, n, and B such that

$$\sum_{1 \leq i \leq m} a_i = nB, \quad \frac{B}{4} < a_i < \frac{B}{2}, \quad 1 \leq i \leq m, \quad m = 3n$$

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

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

8. **Unary Flow Show**
Input: Task times in unary notation and an integer T.
Decision: Is there a two-processor nonpreemptive 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 $|\{(u, v) | u \in V_1, v \in V_2 \text{ and } (u, v) \in E\}| \leq W$?

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 $f : V \to \{1, 2, \ldots, n\}$ such that

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

11.7 REFERENCES AND READINGS

A comprehensive treatment of $\mathcal{NP}$-hard and $\mathcal{NP}$-complete problems can be found in *Computers and intractability: A Guide to the Theory of NP-Completeness*, by M. Garey and D. Johnson, W. H. Freeman, 1979.

Our proof satisfiability $\propto$ directed Hamiltonian cycle is due to P. Hermann. The proof satisfiability $\propto$ AND/OR Graphs and the proof used in the text for Theorem 11.11 were given by S. Sahni. Theorem 11.11 is due to J. Bruno, E. G. Coffman, Jr., and R. Sethi.

Theorems 11.12 and 11.13 are due to T. Gonzalez and S. Sahni. The proof of Theorem 11.13 is due to D. Nassimi. The proof of Theorem 11.14 is due to A. Aho, S. Johnson, and J. Ullman.

The fact that the code generation problem for one-register machines is $\mathcal{NP}$-hard was first proved by J. Bruno and R. Sethi. The result in their 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 results stated in Section 11.6 were presented by D. Johnson and L. Stockmeyer.

For additional material on complexity theory see *Complexity Theory*, by C. H. Papadimitriou, Addison-Wesley, 1994.

11.8 ADDITIONAL EXERCISES

1. [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 whether $f(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)$, where F is a propositional formula, is $\mathcal{NP}$-hard.

2. Show that determining whether C is a minimum circuit (i.e., has a minimum number of gates, see Exercise 1) realizing a formula F is $\mathcal{NP}$-hard.

3. [0/1 knapsack] Show that Partition $\propto$ the 0/1 knapsack decision problem.

4. [Quadratic programming] Show that finding the maximum of a function $f(x_1, \ldots, x_n)$ subject to the linear constraints $\sum_{1leqj\leq n} a_{ij}x_j \leq b_i, 1 \leq i \leq n$, and $x_i \geq 0,\ 1 \leq i \leq n$ is $\mathcal{NP}$-hard. The function f is restricted to be of the form $\sum c_i x_i^2 + \sum d_i x_i$.

5. 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 $\mathcal{NP}$-hard.

6. [Plant location] Let S_i, $1 \leq i \leq n$, be n possible sites at which plants can 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 has 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 can be supplied from many plants. Define $y_i = 0$ if no plant is located at 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

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

   All x_{ij} are nonnegative integers. We assume that $\sum C_{ij} \geq \sum d_i$. Show that finding y_i and x_{ij} so that the total cost is minimized is $\mathcal{NP}$-hard.

7. [Concentrator location] This problem is very similar to the plant location problem (Exercise 6). The only difference is that each destination may be supplied by only one 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 they supply. Show that the concentrator location problem is $\mathcal{NP}$-hard under each of the following conditions:

   (a) $n = 2, C_1 = C_2$, and $F_1 = F_2$. (*Hint*: Use Partition.)

   (b) $F_i/C_i = F_{i+1}/C_{i+1}, 1 \leq i < n$, and $d_i = 1$. (*Hint*: Use exact cover.)

8. [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 $\mathcal{NP}$-hard.

9. Assume that P is a parallel assignment statement $(v_1, \ldots, v_n) := (e_1, \ldots, e_n)$;, 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 $e_{i_j} = v_{i_j}$. Then write an $O(n)$ time algorithm to find an optimal realization for P.

10. 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 and S_j be two sets in T. Also S_i and S_j are disjoint if and only if $S_i \cap S_j = \phi$. T is a disjoint subset of F if and only if every two sets in T are disjoint. The set packing problem is to determine a disjoint subfamily T of maximum size. Show that clique $\propto$ set packing.

11. Show that the following decision problem is $\mathcal{NP}$-complete.
Input: Positive integer $n; w_i, 1 \leq i \leq n$, and M.
Decision: Do there exist nonnegative integers $x_i \geq 0, 1 \leq i \leq n$; such that
$$\sum_{1 \leq i \leq n} w_i x_i = M$$

12. An *independent set* in an undirected graph $G(V, E)$ is a set of vertices no two of which are connected. Given a graph G and an integer k, the problem is to determine whether G has an independent set of size k. Show that this problem is $\mathcal{NP}$-complete.

13. Given an undirected graph $G(V, E)$ and an integer k, the goal is to determine whether G has a clique of size k **and** an independent set of size k. Show that this problem is $\mathcal{NP}$-complete.

14. Is the following problem in $\mathcal{P}$? If yes, give a polynomial time algorithm; if not, show it is $\mathcal{NP}$-complete.

    Input are an undirected graph $G = (V, E)$ of degree 1000 and an integer $k(\leq |V|)$. Decide whether G has a clique of size k.

15. Given an integer $m \times n$ matrix A and an integer $m \times 1$ vector b, the *0-1 integer programming problem* asks whether there is an integer $n \times 1$ vector x with elements in the the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is $\mathcal{NP}$-complete.

16. Input are finite sets $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$. The *set intersection problem* is to decide whether there is a set T such that $|T \cap A_i| \geq 1$ for $i = 1, 2, \ldots, m$, and $|T \cap B_j| \leq 1$ for $j = 1, 2, \ldots, n$. Show that the set intersection problem is $\mathcal{NP}$-complete.

17. We say an undirected graph $G(V, E)$ is k colorable if each node of G can be labeled with an integer in the range $[1, k]$, such that no two nodes connected by an edge have the same label. Is the following problem in $\mathcal{P}$? If yes, present a polynomial time algorithm for its solution. If not, show that it is $\mathcal{NP}$-complete.

    Given an undirected acyclic graph $G(V, E)$ and an integer k, decide whether G is k colorable.

18. Is the following problem in $\mathcal{P}$? If yes, present a polynomial time algorithm; if not, show that it is $\mathcal{NP}$-complete.

    Input are an undirected graph $G(V, E)$ and an integer $1 \leq k \leq |V|$. Also, assume that the degree of each node in G is $|V| - O(1)$. The problem is to check whether G has a vertex cover of size k.

19. Assume that there is a polynomial time algorithm CLQ to solve the CLIQUE decision problem.

    (a) Show how to use CLQ to determine the maximum clique size of a given graph in polynomial time.

    (b) Show how to use CLQ to find a maximum clique in polynomial time.