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

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

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). □

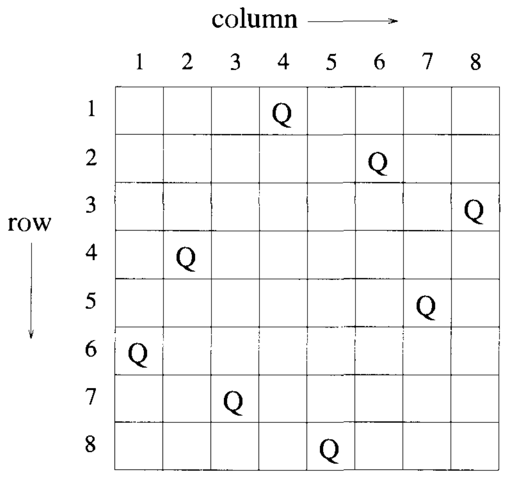


Figure 7.1 One solution to the 8-queens problem

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. □

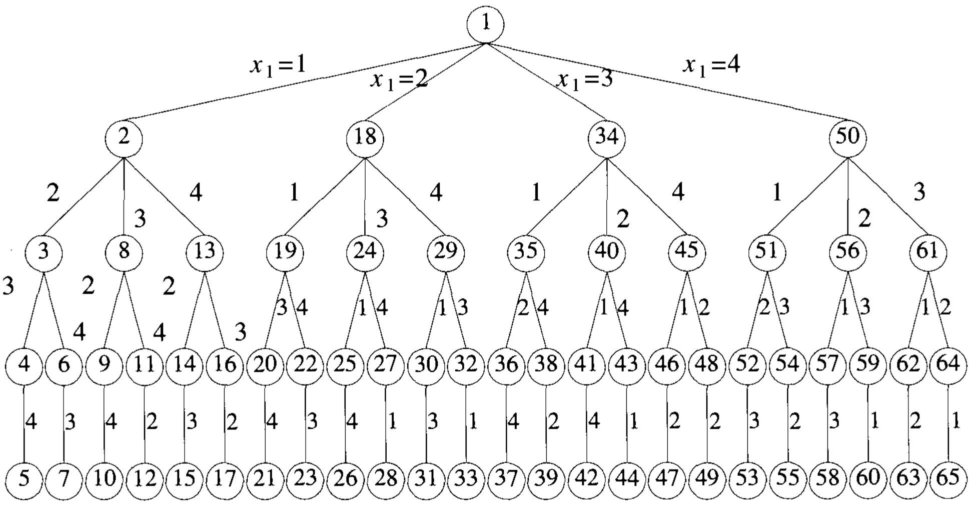


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

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. □

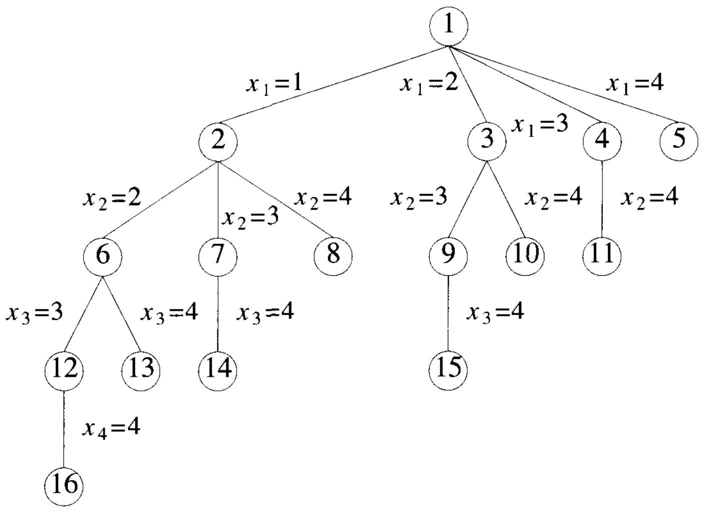


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

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.*

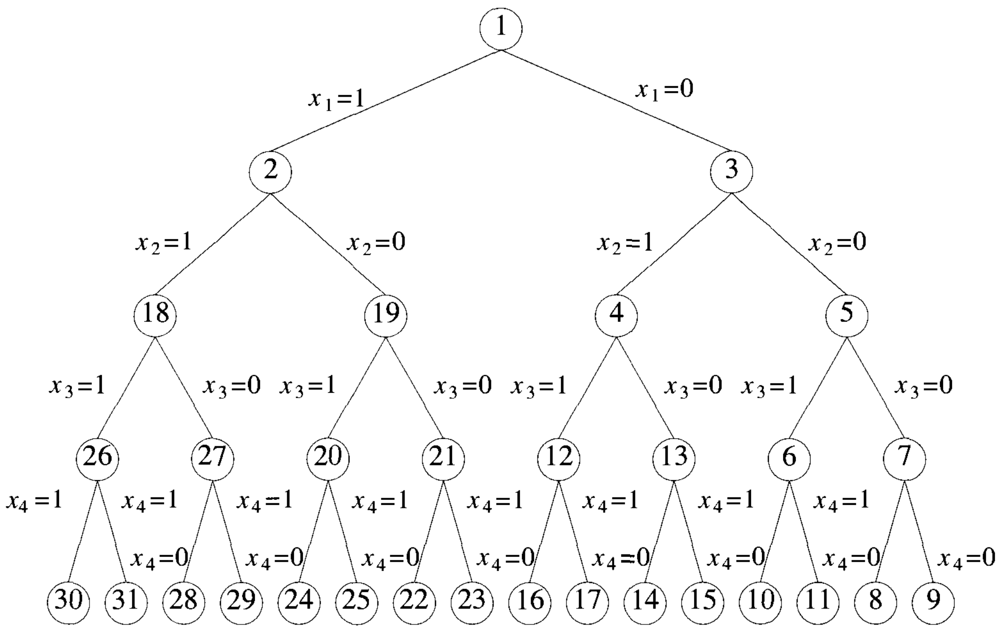


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

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. $T(x_1, x_2, \ldots, x_n) = \emptyset$.

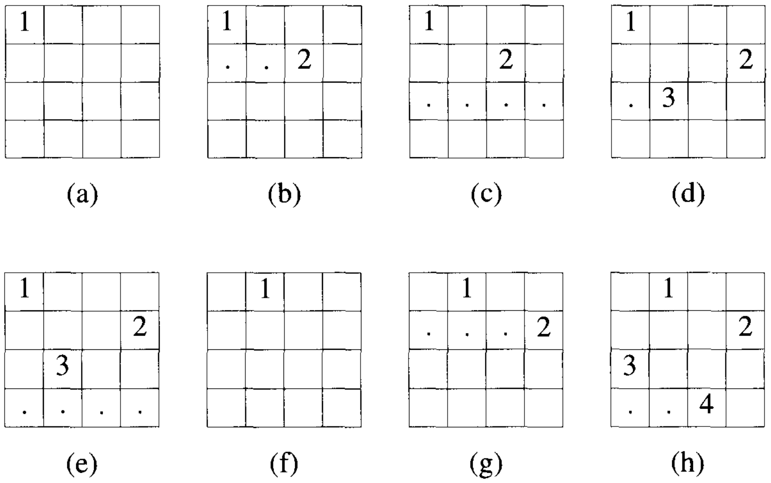


Figure 7.5 Example of a backtrack solution to the 4-queens problem

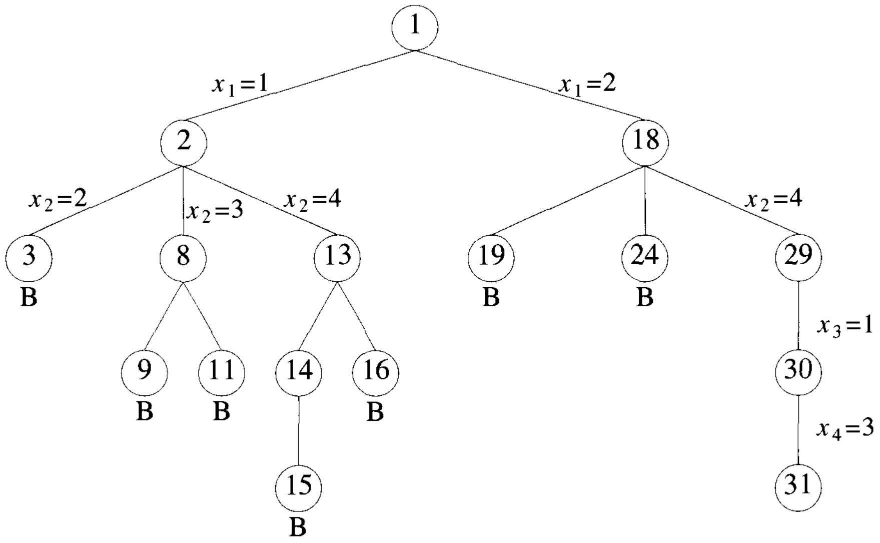


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

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.

```
Algorithm IBacktrack($n$)
// This schema describes the backtracking process.
// All solutions are generated in $x[1:n]$ and printed
// as soon as they are determined.
{
    $k := 1$;
    while ($k \neq 0$) do
    {
        if (there remains an untried $x[k] \in T(x[1], x[2], \ldots,$
            $x[k-1])$ and $B_k(x[1], \ldots, x[k])$ is true) then
        {
                if ($x[1], \ldots, x[k]$ is a path to an answer node)
                    then write ($x[1:k]$);
                $k := k + 1$; // Consider the next set.
        }
        else $k := k - 1$; // Backtrack to the previous set.
    }
}
```

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-

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

Algorithm 7.4 Can a new queen be placed?

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

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

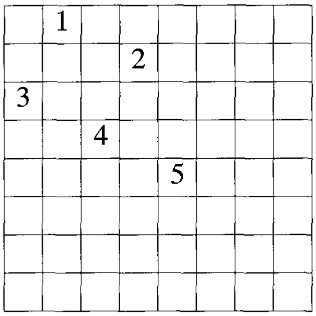
1. Algorithm NQueens can be made more efficient by redefining the function Place(k, i) so that it either returns the next legitimate column on which to place the kth queen or an illegal value. Rewrite both functions (Algorithms 7.4 and 7.5) so they implement this alternate strategy.

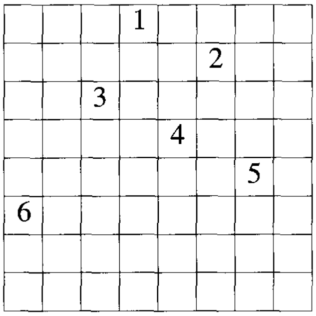
2. For the n-queens problem we observe that some solutions are simply reflections or rotations of others. For example, when $n = 4$, the two solutions given in Figure 7.9 are equivalent under reflection.

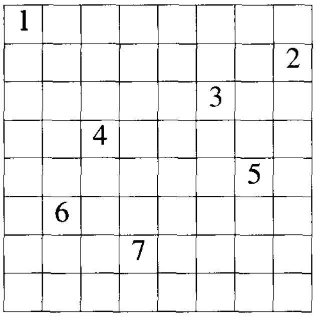
   Observe that for finding inequivalent solutions the algorithm need only set $x[1] = 2, 3, \ldots, \lceil n/2 \rceil$.

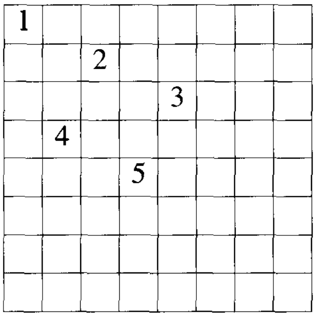
   (a) Modify NQueens so that only inequivalent solutions are computed.

   (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

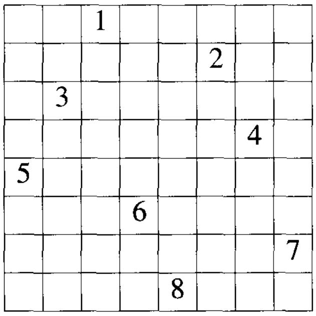

(8,5,3,2,2,1,1,1) = 2329

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

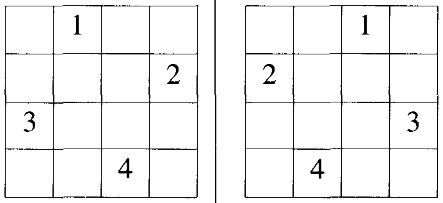


Figure 7.9 Equivalent solutions to the 4-queens problem

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). □

```
Algorithm SumOfSub($s, k, r$)
// Find all subsets of $w[1:n]$ that sum to $m$. The values of $x[j]$,
// $1 \le j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] \ge m$.
{
    // Generate left child. Note: $s + w[k] \le m$ since $B_{k-1}$ is true.
    $x[k] := 1$;
    if ($s + w[k] = m$) then write ($x[1:k]$); // Subset found
        // There is no recursive call here as $w[j] > 0,\ 1 \le j \le n$.
    else if ($s + w[k] + w[k+1] \le m$)
        then SumOfSub($s + w[k], k+1, r - w[k]$);
    // Generate right child and evaluate $B_k$.
    if (($s + r - w[k] \ge m$) and ($s + w[k+1] \le m$)) then
    {
        $x[k] := 0$;
        SumOfSub($s, k+1, r - w[k]$);
    }
}
```

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.

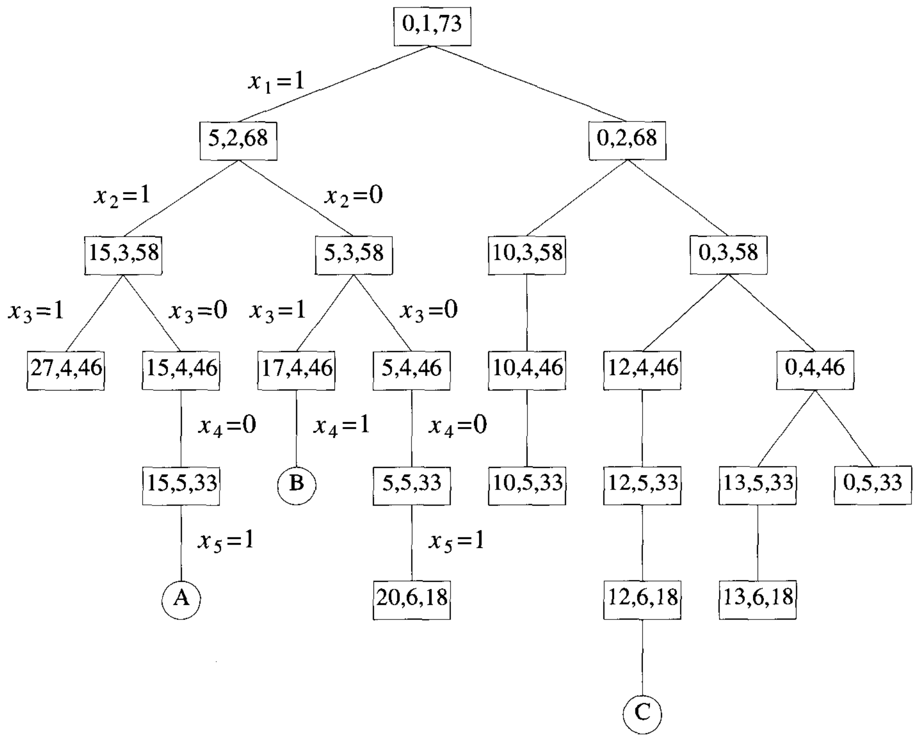


Figure 7.10 Portion of state space tree generated by SumOfSub

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.

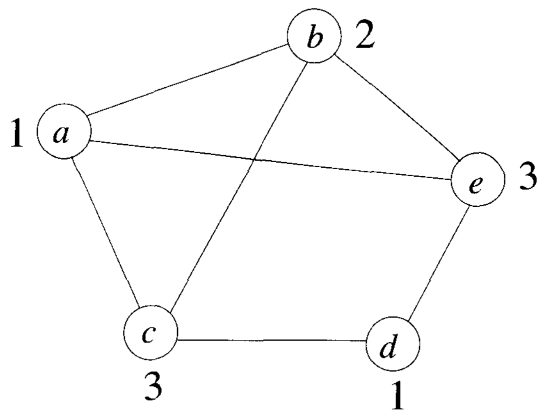


Figure 7.11 An example graph and its coloring

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, $1 \le i \le n$. Nodes at

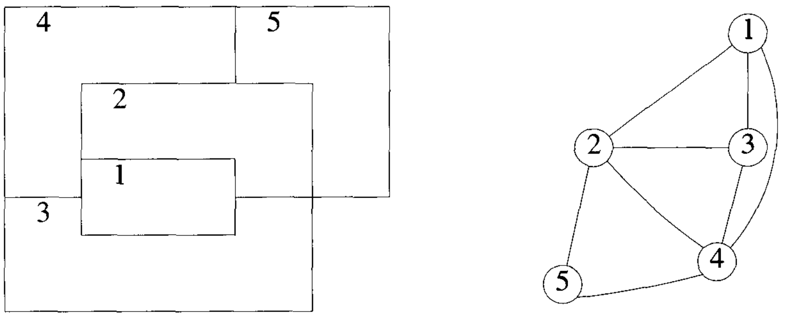


Figure 7.12 A map and its planar graph representation

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 $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

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

Algorithm 7.7 Finding all m-colorings of a graph

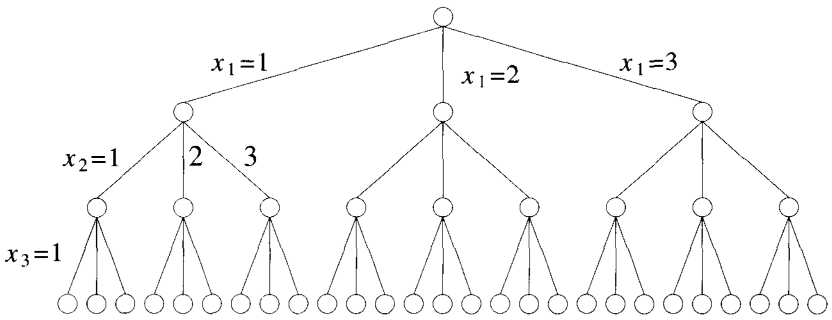


Figure 7.13 State space tree for mColoring when n =3 and m = 3

```
Algorithm NextValue(k)
// x[1], ..., x[k-1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then  break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

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

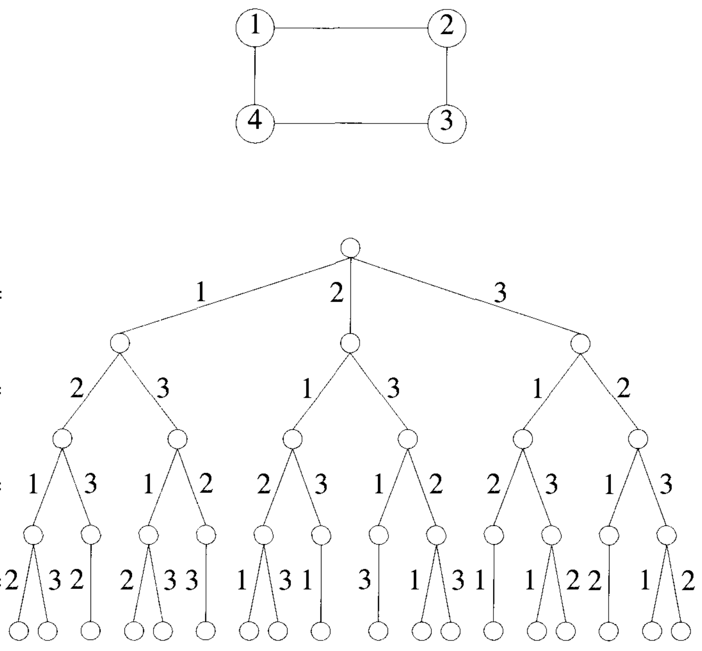


Figure 7.14 A 4-node graph and all possible 3-colorings

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, $1 \le i \le n$, 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

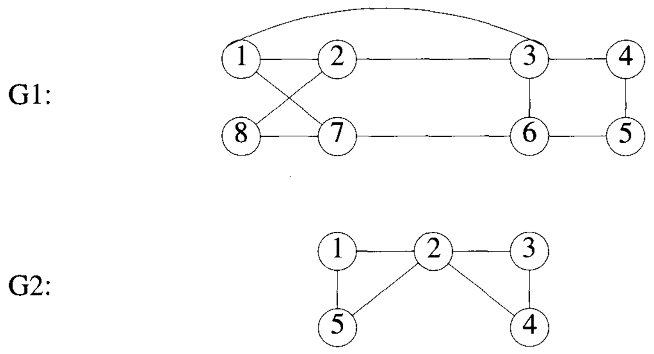


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

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.

```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k]=0$. If $k=n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(n+1)$; // Next vertex.
        if $(x[k]=0)$ then return;
        if $(G[x[k-1], x[k]] \neq 0)$ then
        { // Is there an edge?
            for $j := 1$ to $k-1$ do if $(x[j]=x[k])$ then break;
                               // Check for distinctness.
            if $(j=k)$ then // If true, then the vertex is distinct.
                if $((k<n)$ or $((k=n)$ and $G[x[n], x[1]] \neq 0))$
                    then return;
        }
    } until (false);
}
```

Algorithm 7.9 Generating a next vertex

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

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

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

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 $p[i]/w[i] \geq p[i+1]/w[i+1]$, $1 \leq i < n$.

```
Algorithm Bound($cp, cw, k$)
// $cp$ is the current profit total, $cw$ is the current
// weight total; $k$ is the index of the last removed
// item; and $m$ is the knapsack size.
{
    $b := cp$; $c := cw$;
    for $i := k+1$ to $n$ do
    {
        $c := c + w[i]$;
        if $(c < m)$ then $b := b + p[i]$;
        else return $b + (1 - (c-m)/w[i]) * p[i]$;
    }
    return $b$;
}
```

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

```
Algorithm BKnap(k, cp, cw)
// m is the size of the knapsack; n is the number of weights
// and profits. w[ ] and p[ ] are the weights and profits.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
// knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
// is not in the knapsack; else x[k] = 1.
{
    // Generate left child.
    if (cw + w[k] ≤ m) then
    {
        y[k] := 1;
        if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
        if ((cp + p[k] > fp) and (k = n)) then
        {
            fp := cp + p[k]; fw := cw + w[k];
            for j := 1 to k do x[j] := y[j];
        }
    }
    // Generate right child.
    if (Bound(cp, cw, k) ≥ fp) then
    {
        y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
        if ((cp > fp) and (k = n)) then
        {
            fp := cp; fw := cw;
            for j := 1 to k do x[j] := y[j];
        }
    }
}
```

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

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

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

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).

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

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).

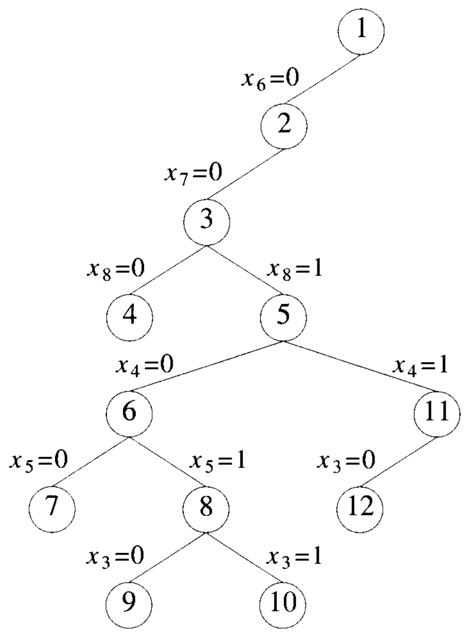


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

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: $p =$