Chapter 7

BACKTRACKING

7.1 THE GENERALMETHOD

* 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.Lehmerin 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 (x1,.,.xn), where the xi are chosen from some finite set Sj.
* Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a criterion function P(x1,.,.xn).
* Sometimes it seeks all vectors that satisfy P. For example, sorting the array of integers in a[l: n] is a problem whose solution is expressible by an n tuple, where xi is the index in a of the ith smallest element.
* The criterion function P is the inequality a[xi] < a[xi +1] for 1<i <n.
* The set Sj is finite and includes the integers 1through 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.
* Suppose mi is the size of set Si. Then there are m = m1, m2,..mn) 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 Pi{x1,.,.xn) (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(x1,x2, … xi) can in no way lead to an optimal solution, then mi+1, …, mn) 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.

Definition7.1

Explicit constraints are rules that restrict each xi to take on values only from a given set.

Common examples of explicit constraints are

X >= 0 or Si = {all nonnegative real numbers}

xi = 0 or 1 or Si = {0,1}

li <= xi <= ui or Si = {a: li <= a <= ui}

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

Definition7.2

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 Xi must relate to each other.

Example 7.1

[8-queens] classic combinatorial problem is to place eight queens on an 8 x 8 chess board 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 chess board 1 through 8 (Figure7.1). 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 (x1,x2,… x8), where X{ is the column on which queen i is placed.
* The explicit constraints using this formulation are Si = {1,2,3,54,6,7, 8}, 1< i < 8. Therefore the solution space consists of 88 8-tuples.
* The implicit constraints for this problem are that no two xi’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 88 tuples to 8! tuples. We see later how to formulate the second constraint in terms of the xi.
* Expressed as an 8-tuple, the solution in Figure7.1 is (4, 6,8,2,7, 1,3,5).



Figure7.1 One solution to the 8-queens problem.

Example7.2

[Sum of subsets] Given positive numbers wi, 1< i < n, and m, this problem calls for finding all subsets of the wi whose sums are m.

For example, if n = 4, (w1,  w2, w3,  w4 )= (11,13,24, 7), and m = 31, then the desired subsets are (11,13,7) and (24, 7).

* Rather than representing the solution vector by the wi which sum to m, we could represent the solution vector by giving the indices of these wj. Now the two solutions are described by the vectors (1,2, 4) and (3, 4).
* In general, all solutions are k-tuples(x1,x2, … xk), 1< k < n, and different solutions may have different-sized tuples. The explicit constraints require xi ℇ {j | j is an integer and 1< j < n}.
* The implicit constraints require that no two be the same and that the sum of the corresponding problem.
* 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 xi < xi+i, 1<i <k.
* In another formulation of the sum of subsets problem, each solution subset is represented by an n-tuple (x1,x2, … xn) Such that xi  ℇ {0,1} 1<i <n.
* Then xi = 0 if Wi is not chosen and xi = 1 if Wj 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 2n 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.

Example7.3

[n-queens]

* The n-queens problem is a generalization of the 8-Queens problem of Example7.1.
* Now n queens are to be placed on an n x n chess board so that no two attack; that is, no two queens are on the same row, column, or diagonal.
* Generalizing our earlier discussion, n the solution space consists of all n! permutations of the n-tuple(1,2,,n.)...
* Figure7.2 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 xi . Edges from level 1 to level 2 nodes specify the values for xi. Thus, the leftmost subtree contains all solutions with xi = 1; its leftmost subtree contains all solutions with xi = 1and x<i = 2, and so on.
* Edges from level i to level i +1 are labeled with the values of xi. 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 Figure7.2.



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 Example7.2 we gave two possible formulations of the solution space for the sum of subsets problem. Figures7.3 and 7.4 show a possible tree organization for each of these formulations for the case n = 4. The tree of Figure7.3 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 xi.
* 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,) and (2,3) and so on.
* Thus, the leftmost subtree defines all subsets containing w\\, the next subtree defines all subsets containing W2 but not wi, and so on.
* The tree of Figure7.4 corresponds to the fixed tuple size formulation.



* Edges from level i nodes to level i + 1 nodes are labeled with the value of Xi, 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 w2, the right subtree defines all subsets not containing w1, and soon. Now there are 2 leaf nodes which represent 16 possible tuples.
* Figure7.3 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

Example7.5

[4-queens] Let us see how backtracking works on the 4-queens problem of Example7.3.

* As a bounding function, we use the obvious criteria that if {x1,x2,…xi) is the path to the current E-node, then all children nodes with parent-child labelings Xi+1 are such that (x1, xi+i) represents a chess board 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 Figure7.2 is generated and the path is now (1).
* This corresponds to placing queen 1 on column 1.
* Node 2 becomes the E-node.
* Node3 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 node2 and generate another child, node 13.
* The path is now (1,4). Figure7.5 shows the board configurations as backtracking proceeds.

Figure7.5 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 Figure7.5(b) the second queen is placed on columns 1 and 2 and finally settles on column 3.
* In Figure7.5(c) the algorithm tries all four columns and is unable to place the next queen on a square.
* Backtracking now takes place. In Figure7.5(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 Figure7.5(e), (f), (g),and (h) show the remaining steps that the algorithm goes through until a solution is found.
* Figure7.6 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 Figure7.2 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.

Backtrack(l);

1 Algorithm Backtrack(/c)

2 // This schema describes the backtracking 3//process using

4 // recursion. On entering, the first k-1 5//values x[1], x[2], …x[k-1] of the solution 6//vector x[1:n] have been assigned.x[] and 7 //n are global.

8 {

9 for (each x[k] ℇ T( x[2], …x[k-1] ) do

10{

11if (Bk{x[l],x[2][,…x[k]) != 0) then

12 {

13 if (x[l],x[2][,…x[k]) is a path to an 14//answer node)

15 then write (x[l :k]);

16 if {k <n) then Backtrack(k+ 1);

17 }

18}

Algorithm7.1 Recursive backtracking algorithm