UNIT - IV

BACKTRACKING

We studied three algorithm strategies till now. Backtracking is a more intelligent variation of these approaches. The principle idea is to construct solutions one component at a time and evaluate such partially constructed solutions as follows :

- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

In brief, the task is to determine algorithm to find solutions to specific problem not by following a fixed rule of computation, but by trial and error. The common pattern is to decompose the trial and error process into partial tasks. Often these tasks are most naturally expressed in recursive terms and consists of the exploration of a finite number of subtasks.

It is convenient to implement this kind of processing by constructing a tree of choices made, called the **state-space-tree**. Its root represents an initial state before the search for solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component and so on. A node in a state space-tree is said to be **promising** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise it is called **non-promising**. Leaves represent either non-promising dead ends or complete solutions found by the algorithm. In the majority of cases, a state space tree for backtracking algorithm is constructed as depth first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component, if there is no such option, it backtracks one more level up and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or backtracks to continue searching for other possible solutions.

1 THE GENERAL METHOD

In the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2, ..., x_n)$, 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 $P(x_1, x_2, ..., x_n)$. Sometimes it seeks all vectors that satisfy P.

ne Control Abstraction

We assume that all answer nodes are to be found and not just one. Let $(x_1, x_2, ..., x_i)$ be a path from the root of state space tree to a node. Let $T(x_1, x_2, ... x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, ..., x_{i+1})$ is also a path to a problem state.

$$T(x_1, x_2, x_3, ..., x_n) = \theta.$$

Recursive Backtrack Algorithm

```
Algorithm Backtrack (k)
    x[1], x[2], ....., x[k - 1] solution vectors
    begin
        for (each x[k] ∈ T(x[1], .....x[k - 1])) do
```

```
        begin
            if (B_k(x[1],x[2], ....., x[k]) ≠ 0) then
                begin
                    if (x[1],x[2], .....x[k] is a path to an answer node)
                        then print (x[1 : k]);
                    if (k<n) then Backtrack (k + 1);
                end
        end
    end
```

2. Iterative Backtrack Method

```
Algorithm IBacktrack (n)
    begin
        k = 1
        while (k≠0) do
            begin
                if (there remains an untried x[k] ∈ T (x[1], x[2], ..... , x[k - 1]) and
                B_k (x[1], ..... , x[k]) is a path to an answer node)
            then print (x[1 : k]);
                k = k + 1;
                else k = k - 1;
            end
        end
```

4.2 PECULIAR CHARACTERISTICS AND USE

- Backtracking is an algorithm design technique for solving problems in which the number of choices grows atleast exponentially with their instance size. This technique constructs a solution one component at a time, trying to terminate the process as soon as one can ascertain that no solution can be obtained as a result of the choices already made.
- Backtracking employs a state space tree as its principle mechanism, which is a rooted tree whose nodes represent partially constructed solutions.
- The backtrack algorithm has as its virtue the ability to yield the same answer for 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, ..., x_i) and 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, ..., x_i) can in no way lead to an optimal solution, then x_1, ..., x_n possible test vectors can be ignored entirely.

4.2.1 State Space Tree

- In all the problems, we are going to find solution for given n weights (w_1, w_2, w_3, ..., w_n). There are two ways to state the solution: for example, first way is to form a solution array as (w_1, w_2, w_4) or form an array of indices (1, 2, 4). Here solution will be of size k, ($1 \le k \le n$). In general, all solutions are k-tuples (x_1, x_2, ..., x_k). Different solutions may have different size tuples.
- Second way to state the solution is fixed-size n-tuple (x_1, x_2, ..., x_n) where $x_i = 0$ /1, $1 \le i \le n$. If object i belongs to solution, then $x_i = 1$, otherwise $x_i = 0$.
- In backtracking approach and branch-and-bound approach the problem solutions are systematically searched using a **tree organization** 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. All the states s for which the path from the root to s defines a tuple in the solution space is called **solution states**.

- Solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem are called **answer states**. The tree organization of the solution space is called as the **state space tree**.
- The tree which is independent of the problem instance being solved is called **static tree**. The tree which depends on the problem instance being solved is called **dynamic tree**.
- For given problem, when a state space tree is generated it is easy to determine problem states, then determine which problem states are solution states, then determine which solution states are answer states. Generation of problem states can be done in two ways : backtracking and branch-and-bound.
- Both backtracking and branch-and-bound approaches start construction of nodes from the root of tree. A node which has been constructed, but whose children are not yet constructed is called a live node. A **live node** whose children are being constructed is called an **E-node**. A node which is not to be expanded or all of whose children are constructed is called a **dead node**. As the construction of tree proceeds, E-node goes on changing. To kill live nodes without generating their children, bounding functions are used.

In all the topics ahead, our constraints are that in solutions x_i is always an integer, $1 \le i \le n$. No two x_i's are same. Multiple instances represent the same subset for example, (2, 3) is same as (3, 2).

3 N-QUEENS PROBLEM

The n-queens problem is to place n queens on an n × n chessboard, so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2 and n = 3. So let us consider the 4-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in figure below :

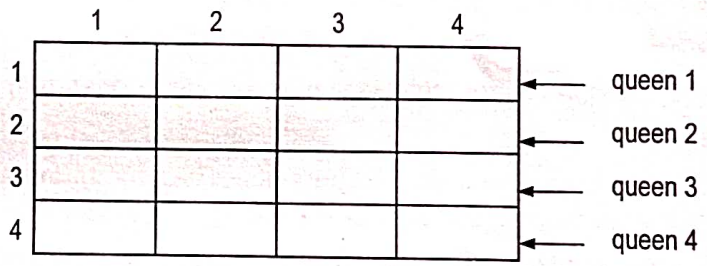


Fig. 4.1

We start with the empty board and then place queen 1 in the first possible position that is Board [1] [1].

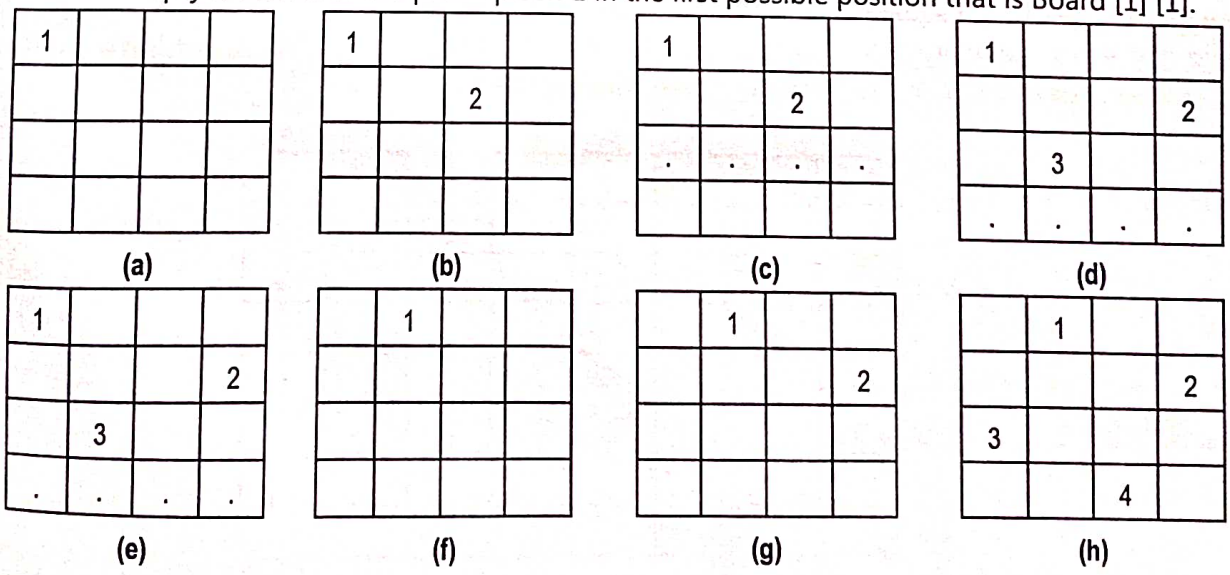


Fig. 4.2 : Backtrack solution for 4-queens problem

Fig. 4.2 shows backtrack solution to the 4-queens problem.

We started with empty board and one queen has been placed. Now we place queen 2, after trying unsuccessfully columns 1 and 2 in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts

queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks and puts queen 1 at position (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3), which is a solution to the problem. The state space tree to this search is given below :

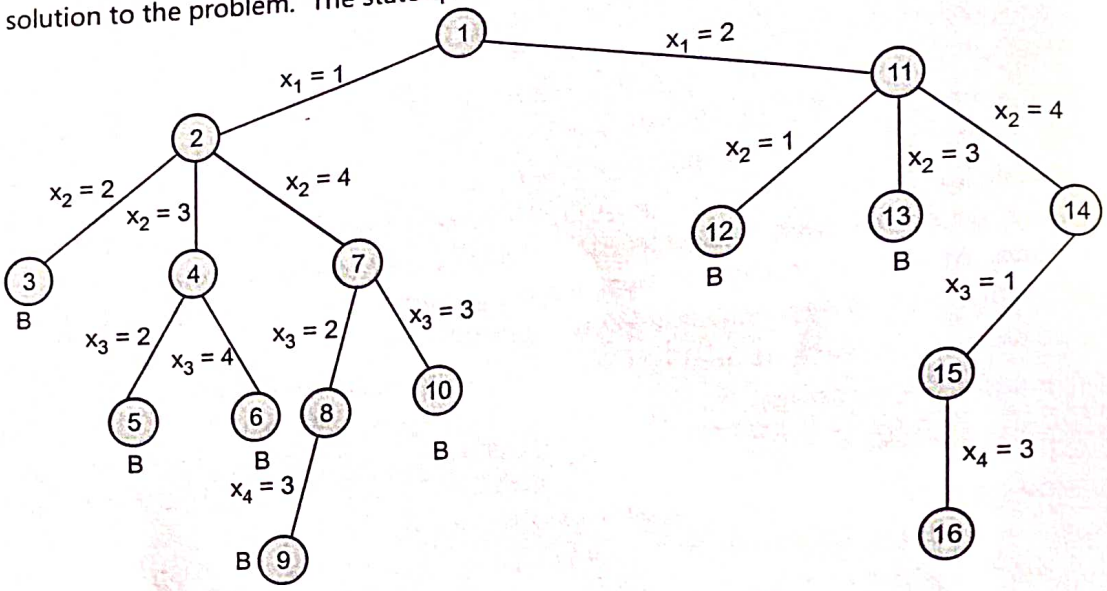


Fig. 4.3 : State space tree for 4-queens problems

- Nodes are numbered level-wise. Node at which backtracking occurs is represented with B. Because rows in which queens are to be placed are already known, we just have to find out column. x_i in the above state space tree denotes column number i.
- If other solutions need to be found, the algorithm can simply resume its operation at which it stopped.

Algorithm: N-Queens Problem

- Let us generalize the problem and consider an n × n chess board and try to find all the ways to place n non-attacking queens. We observed from 4-queens problem that we can let $(x_1, ..., x_n)$ represent a solution in which x_i is the column of the i^{th} row where the i^{th} queen is placed. The x_i's will all be distinct since no two queens can be placed in the same column. Now how do we test whether two queens are on the same diagonal?
- If we imagine the chessboard squares being numbered as the indices of the two dimensional array a[1 : n, 1 : n], then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value. For example in the following figure consider the queen at a[4, 2].

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5								
6	Q							
7			Q					
8					Q			

Fig. 4.4 : One possible solution to the 8-queens problem

- The squares that are diagonal to this queen (running from the upper left to the lower right) are a[3, 1], a[5, 3], a[6, 4], a[7, 5] and a[8, 6]. All these squares have a difference of 2 between row and column values i.e. for all, (row – column) is same. Also, every element on the same diagonal that goes from the upper right to the lower left has the same (row + column) value. Suppose two queens are placed at position (i, j) and (k, l). Then by the above rules, they are on the same diagonal only if (i – j) = (k – l) or (i + j) = (k + l).

- The first equation implies :

$$j - l = i - k$$

The second implies :

$$j - l = k - i$$

- Therefore, two queens are on the same diagonal if and only if $|j - l| = |i - k|$.
- PlaceQueen(k, i) algorithm returns a boolean value that is true if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values x[1], ..., x[k – 1] and whether there is no other queen on the same diagonal. Its computing time is (k – 1).
- Using algorithm PlaceQueen we can define the general backtracking method as given by algorithm discussed in control abstraction and give a precise solution to the n-queens problem. The array x[ ] is global. The algorithm PlaceQueen is invoked by NQueens algorithm.

Algorithm: PlaceQueen (k ,i)

```
begin
    for j = 1 to k – 1
    begin
        if (x[j] = i) // Two queens are on the same column
        or (abs(x[j] – i ) = abs (x [j] – k ) ) // same diagonal
        then
            return false;
    end for
    return true
end
```

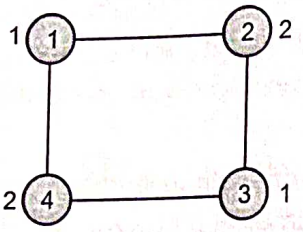
Let us see an algorithm which gives all possible solutions to place n queens on an n × n chessboard in non-attacking positions. The algorithm is called for the first time as : NQueens (1, n). Here is the algorithm.

Algorithm: NQueens (k, n)

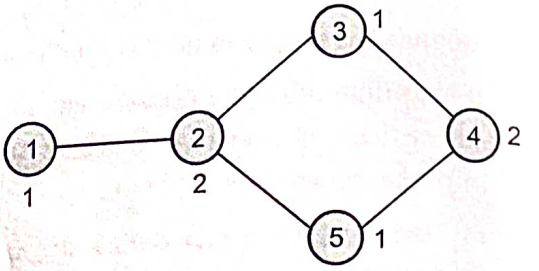
```
begin
    for i = 1 to n
    begin
        if PlaceQueen (k, i) then
        begin
            x [k] = i
            if (k = n) then
                print solution x [ 1 ... n ]
            else
                NQueens (k + 1, n)
            end if
        end if
    end for
end
```

4.4 GRAPH COLORING

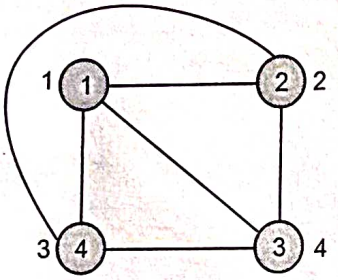
- Given a graph, it is always possible to colour the nodes of a graph such that no two adjacent nodes have same colour. It is always desired that the number of colours required are minimum. This is a graph coloring problem. For example, consider the graphs shown below :



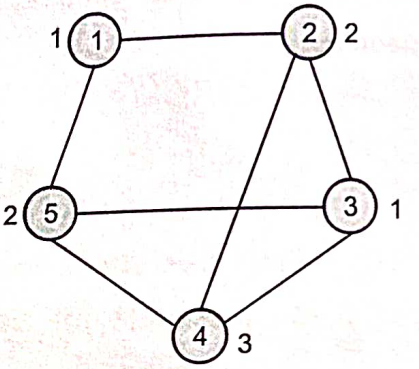
Graph 1



Graph 2



Graph 3



Graph 4

Graph 1 has 4 nodes and requires only 2 colours.

Graph 2 has 5 nodes and requires only 2 colours.

Graph 3 has 4 nodes and requires only 4 colours.

Graph 4 has 5 nodes and requires only 3 colours.

- If a graph G can be coloured with minimum m nodes, then m is called as the chromatic number of the graph. Findi the minimum value of m is called the m-colourability optimization problem.
- Application of graph coloring is to colour all the regions of a map such that no two adjacent regions are of sa colour. Each region can be considered as a node and an edge represents two adjacent regions.
- Let us see the algorithm to find all possible ways to colour a graph G using m colours. Let a graph is represented as adjacency matrix G[n, n]. G[p ,q] = 1 denotes that there is an edge <p, q> in graph. G[p, q] = 0 denotes that there is edge <p, q>. Let n denotes the number of vertices in G. Let there are m colours denoted as 1, 2, ..., m. The gra coloring solution is given by an array x[1 : n] where x_i denotes the colour of node i in solution.
- Initially solution x[1 : n] is set to 0 for all elements. The algorithm starts from vertex 1.

Algorithm: mColoring (k)

```
begin
    repeat
        {
        // First find the next possible colour for node k,
        // Otherwise set its colour to 0,
        // FindNextColour(k);
        // If colour of node k is 0, then stop because
        // New colour is not available
        if (x[k] = 0)
            then return
```

```
    //If all the nodes are coloured, then display solution.
    if (k = n)
        display solution x [1 .. n]
    //Otherwise, continue process for next node k+1
    else
        mColoring (K+1)
    endif
  } until (false)
end
```

This algorithm will print all possible coloring solutions x[1..n].

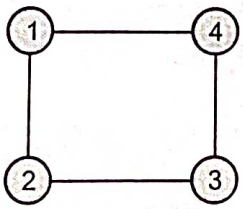
Algorithm FindNextColour will assign the next available colour to node k, else assigns 0. The algorithm is given below :

Algorithm: FindNextColour (k)

```
begin
  repeat
  {
  //Get next highest colour in x[k]
  x[k] = (x[k] + 1) mod(m+1)
  //if no colour available, return
  if (x[k] = 0)
      return
  //Check whether the new colour is not same as colour of adjacent nodes
  for p = 1 to n
  begin
      if G[k,p] ≠ 0 and x[k] = x[p]
          break;
  end for
  if (p = n+1)
      return //  .. new colour found
  //Otherwise continue to find next colour
  } until (false)
end
```

Computing time of this algorithm is O(mn) for all legal colours.

A graph with 4-nodes and all possible 2 coloring is shown in Fig. 4.5 and all possible 3 coloring is shown in Fig. 4.6.



Graph G

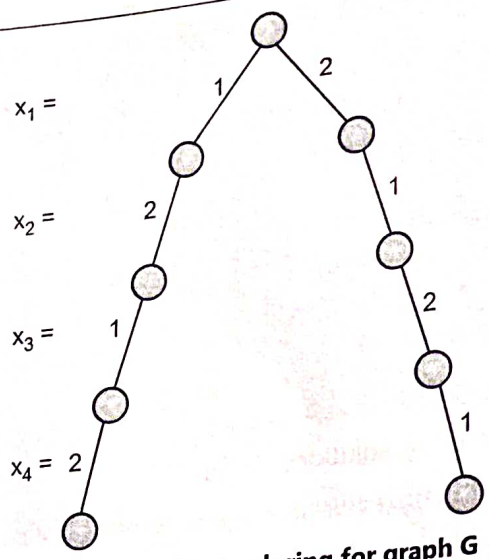


Fig. 4.5 : Possible 2-coloring for graph G

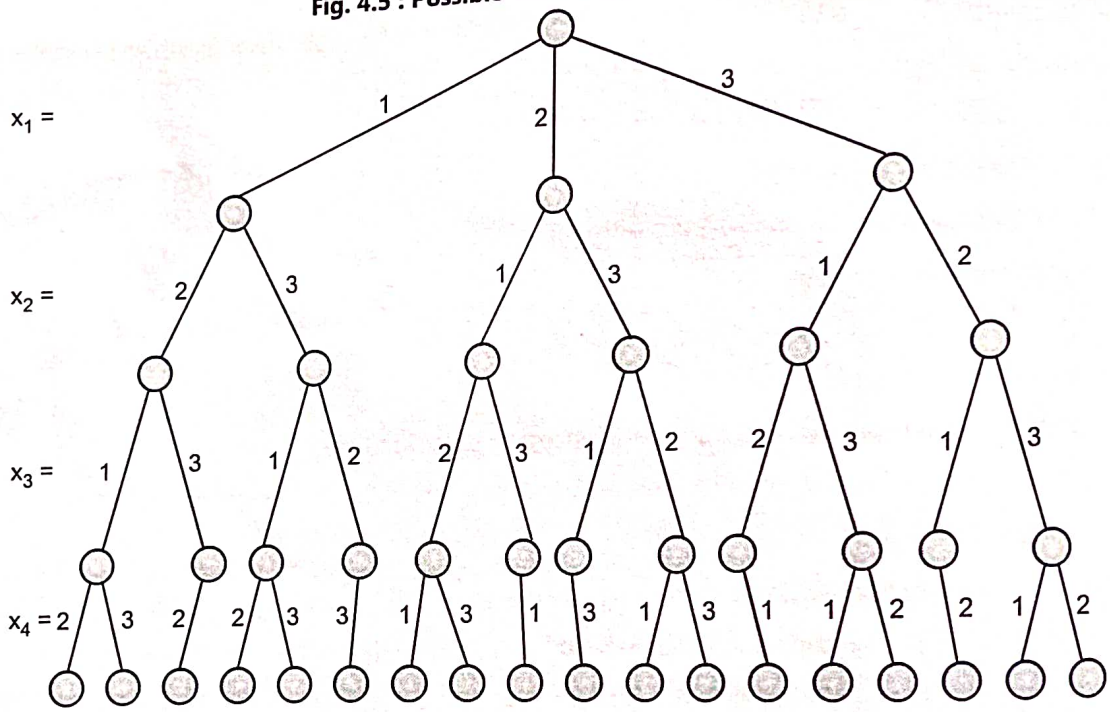


Fig. 4.6 : Possible 3-coloring for graph G

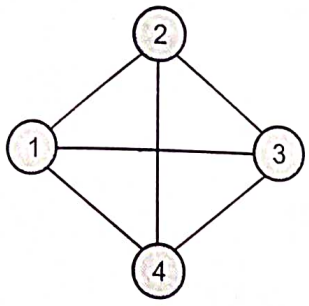
- So using 2 colours, there are only 2 possible ways and using 3 colours, there are 18 possible coloring ways for given graph.
- The number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$, which is upper bound on the computing time mColoring algorithm. For each internal node, FindNextColour algorithm is called which requires O(mn) time. Hence total time is bounded by

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

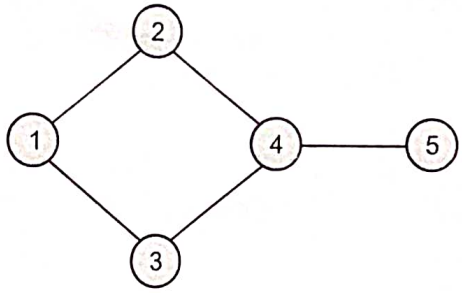
4.5 HAMILTONIAN CYCLES

- In a connected graph, a path which starts at a vertex, visits every vertex of graph once and returns to its starting position is called as a Hamiltonian cycle. It is suggested by Sir William Hamilton. Let G = (V, E) be a connected graph of n vertices. Let $v_1, v_2, ..., v_n$ be its vertices. Then $v_1, v_2, ..., v_{n+1}$ is called as a Hamiltonian cycle if it begins at v_1, visits other

vertices in the order $v_1, v_2, \ldots, v_{n+1}$ once such that E contains an edge $<v_i, v_{i+1}>$ for $1 \le i \le n$ where each v_i is distinct, except v_1 and v_{n+1} which represent the same vertex. For example, consider the graph shown on next page :



Graph 1



Graph 2

Graph 1 contains a Hamiltonian cycle 1, 2, 3, 4, 1. Graphs 2 has no Hamiltonian cycle.

Let us see an algorithm which gives all possible Hamiltonian cycles in a graph. Let G[n, n] is an adjacency matrix for graph.

G[i, j] = 1 if edge <i, j > ∈ E.

G[i, j] = 0 if edge <i, j > ∉ E.

Let an array x[1 ... n] denotes a Hamiltonian cycle where x[i] denotes the i^{th} vertex visited in the path. Algorithm miltonianCycle is called with vertex 1 initially.

orithm: HamiltonianCycle (k)

```
begin
   repeat
   {
       //find next vertex to be visited and store at position x[k].
        FindNextVertex (k);
       //if no new vertex is available, return
        if (x[k]=0)
           return
   //if all the vertices visited, then print Hamiltonian cycle.
    displayHCycle x[1..n]
   //otherwise continue to find next vertex in the path
   else
       HamiltonianCycle (k+1)
   } until (false);
end
```

Algorithm FindNextVertex(k) searches a vertex which is connected to vertex k – 1 and which is unvisited (means does not appear in x[1 .. k–1].

If no such vertex is found, then x[k] = 0. If k = n, then it is checked whether x[n] and x[1] are connected or not. Initially array x[ ] is set to 0.

orithm: FindNextVertex (k)

```
begin
   repeat
   {
   //Find next vertex
       x[k]= (x[k] +1) mod (n+1)
   //If no new vertex available, return
```

```
        if (x[k]=0)
            return
    //If new vertex available, check whether it is connected to previous vertex
    if(G(x[k - 1], x[k]) ≠ 0)
    begin
    //check whether vertex k is already visited
        for p = 1 to k - 1
            if (x[p] = x[k])
                break;
        end for
        /* if vertex k is unvisited, and k = n, then check whether it is connected to first vertex */
         if (p = k) // means vertex k is unvisited
         if ((k = n) and (G[x[k], x[1]] ≠ 0))
            return
    } until (false)
end
```

We have already seen travelling salesperson problem which is a Hamiltonian cycle. If all the edges have cost m and there are n edges in Hamiltonian cycle, then its cost is mn.

4.6 0/1 KNAPSACK PROBLEM

- We have already solved a 0 / 1 knapsack problem. Given weights w_i and profits p_i for n objects and if M is the knapsack capacity, then the knapsack problem is to fill the knapsack such that :

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

$$\sum_{1 \le i \le n} p_i x_i \text{ is maximized and}$$

$$x_i = 0 / 1, 1 \le i \le n$$

- Until now we have assumed that there are n objects available. Now we will assume that n types of objects are available and sufficient numbers of objects are available for each type. All the weights, profits and capacity M are positive values.
- We can solve the knapsack problem using backtracking and represent it as an **implicit graph**. Nodes of this graph can be built as the search progresses. Hence computing time is saved. Storage space is also saved because unwanted nodes can be discarded.
- Backtracking resembles a depth-first search in an implicit tree, by building partial solutions. Each move along an edge of the implicit tree adds a new element to a partial solution. If a complete solution is found, then process can be continued to get better solution. If it is not possible to get complete solution, backtracking is done, and nodes are removed in each step of backtracking. Let us solve one example.

SOLVED EXAMPLES

Example 4.1 : *We have $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5,)$ and profits $(p_1, p_2, p_3 p_4) = (3, 5, 6,10)$. M = 8. Find out an optimal solution to fill knapsack using backtracking.*

Solution :

- Initially the knapsack is empty, hence the partial solution is also empty, given by : 0. Then we can put one object from weights 2, 3, 4, 5 in knapsack getting profit 3, 5, 6, 10 respectively. So there are four partial solutions possible. (2 : 3), (3 : 5), (4 : 6), (5 : 10). We can continue with partial solution 2 : 3, and get next partial solutions putting one of 2, 3, 4, 5 objects giving (2, 2 : 6), (2, 3 : 8), (2, 4 : 9), (2, 5 : 13). All the partial solutions which exceed maximum capacity M = 8 will be discarded. The process continues to get an optimal solution. This can be represented as an implicit tree as shown below :

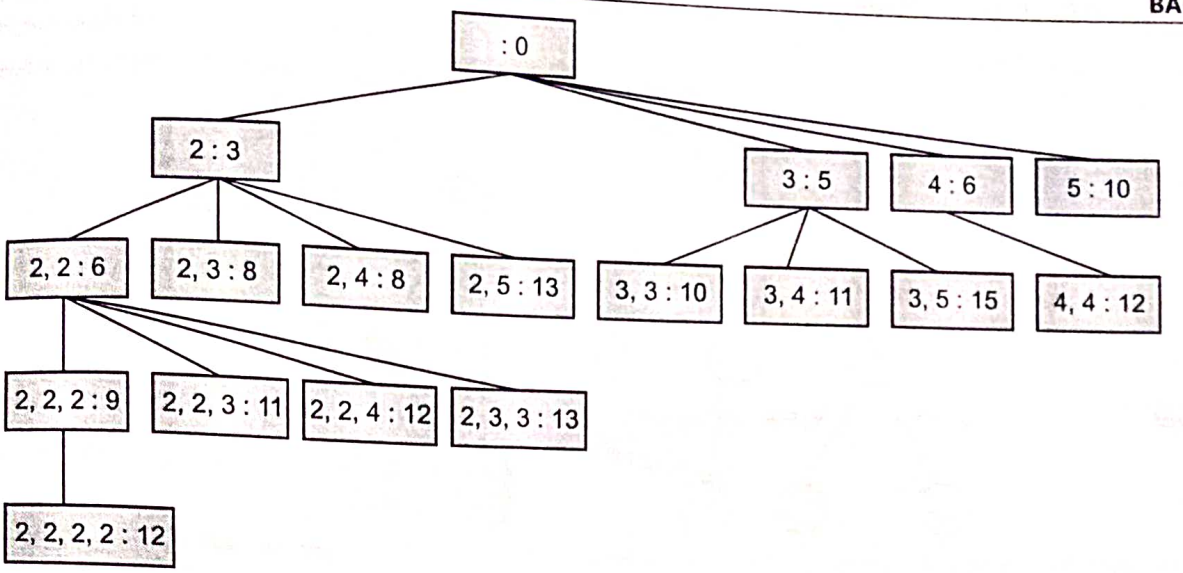


Fig. 4.7 : An implicit tree

First we get (2, 2, 2, 2 : 12) as a solution, so remember it. Proceed in the depth-first-search manner, then we get (2, 3, 3 : 13) as a next better solution so remember it. Then we get (3, 5 : 15) as more better solution which gives maximum profit, hence it is optimal.

Note that partial solution (4 : 6) leads to four partial solutions (4, 2 : 9), (4, 3 : 11), (4, 4 : 12), (4, 5 : 16). But (4, 2 : 9) is same as (2, 4 : 9) and hence we have not considered such partial solutions again. Also (4, 5 : 16) violates maximum capacity, hence discarded.

Let there are n objects, an array w[1 ... n] stores their weights, an array p[1..n] stores profits and M is maximum capacity of Knapsack. The algorithm BKnapsack is as shown below :

gorithm: BKnapsack (p, x)

```
gin
 val = 0
 for k = p to n
 begin
     if (w[k] ≤ x)
         val = max (val, p[k] + BKnapsack(k, x - w[k]))

     end if
 end for
 return val;
d
```

Initially the algorithm should be called as BKnapsack(1, M). It will find an optimal solution to fill the knapsack.

.7 SUM OF SUBSETS PROBLEM

Given n distinct weights (w_1, w_2, ..., w_n), finding all combinations of these weights whose sums are M, is called the sum of subsets problem. We will consider fixed-size solution (x_1, x_2, ..., x_n) where x_i = 1 if object i is included in solution, otherwise x_i = 0. Consider the following example.

Example 4.2 : *If (w_1, w_2, w_3, w_4) = (10, 5, 7, 8) are weights and we want to find all combinations of these weights whose sums are M = 15.*

Solution : The variable-sized solutions are (10, 5, 0, 0) and (0, 0, 7, 8). The solution space can be stated using BFS to show all possible solutions using tree organization, as given below :

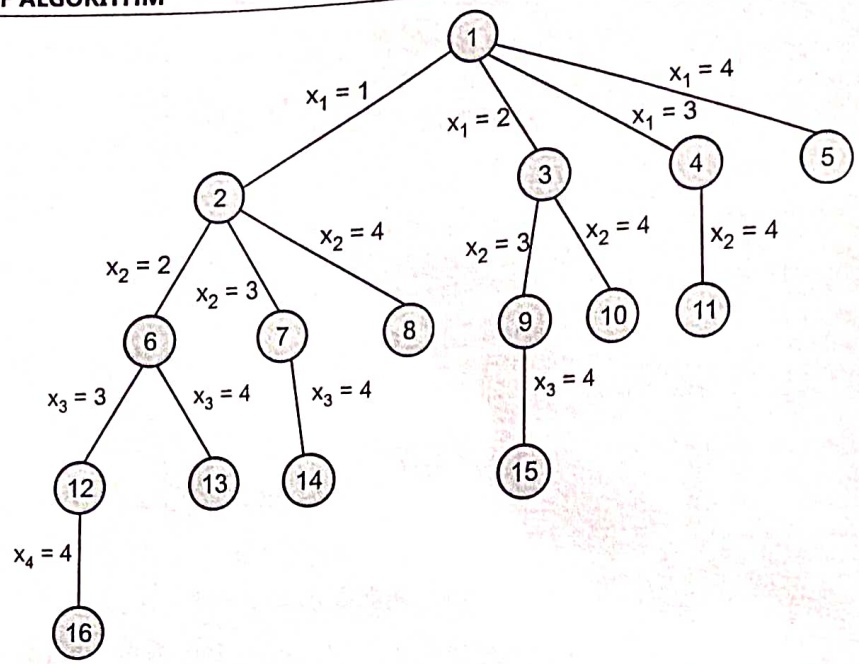


Fig. 4.8 : Solution space

- Here the nodes are numbered as in breadth-first-search. In the tree, root has n children. From the left, first child leads to the subtree which defines all subsets containing weights (w_1, ..., w_n). Second child of root leads to the subtree which defines all subsets containing weights (w_2, ..., w_n), but not w_1. Third child leads to the subtree which defines all subsets containing weights (w_3, ..., w_n), but not w_1 and w_2, and so on.
- If fixed-size solution tuples are used, then solutions are (1, 1, 0, 0) and (0, 0, 1, 1). In the first solution, $w_1 + w_2 = 10 + 5 = 15$, hence (w_1, w_2) are included in the solution. In the second solution, $w_3 + w_4 = 7 + 8 = 15$, hence (w_3, w_4) are included in the solution. As n = 4, there are total $2^4 = 16$ possible solution tuples. All paths from the root to a leaf node define the solution space. The tree organization is shown below in Fig. 4.9.
- Here each node at level k leads to two edges: left edge labelled $x_i = 1$ which denotes inclusion of w_i, and right edge labelled $x_i = 0$ which denotes that w_i is not included in solution.

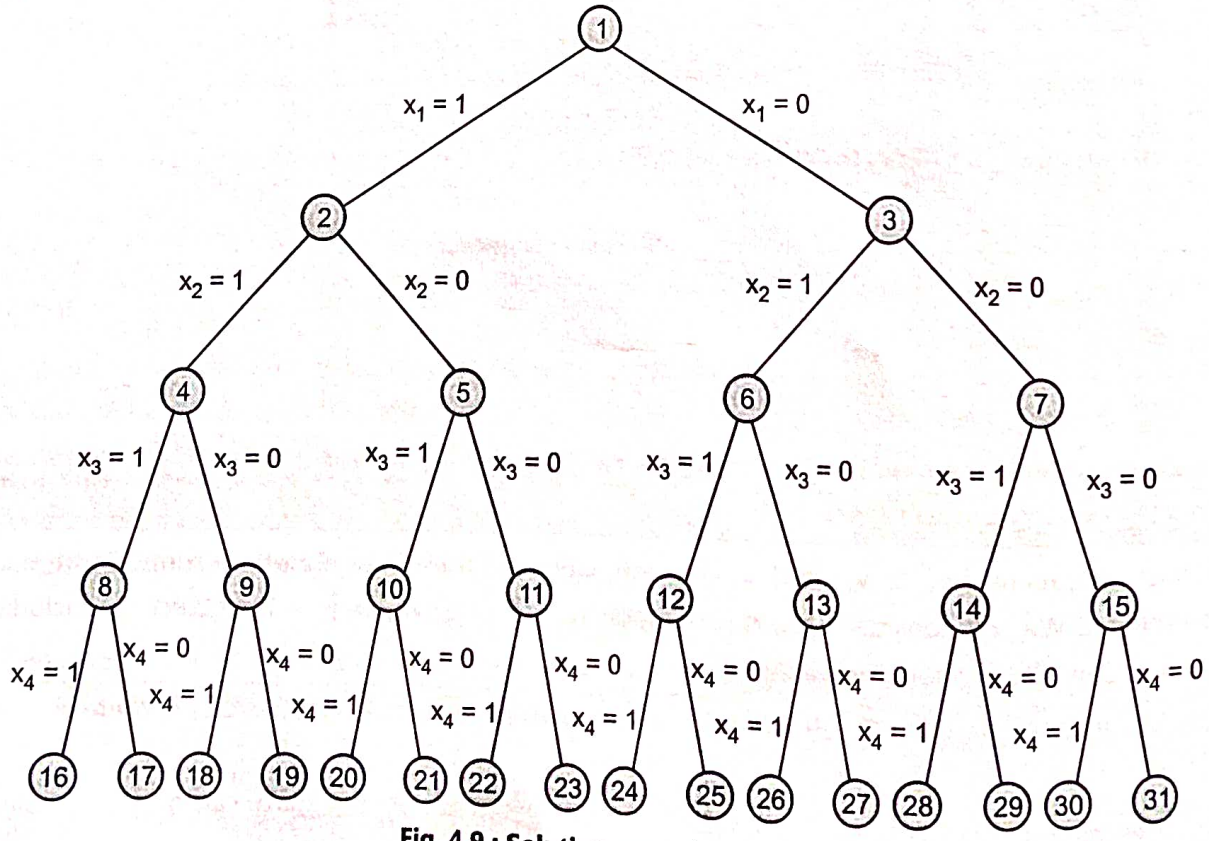


Fig. 4.9 : Solution space for n = 4

In backtracking algorithm, we need a bounding function to kill live nodes which do not lead to answer nodes. For the sum of subsets problem. The bounding function can be defined as

$$B_k(x_1, \ldots, x_k) = \text{true if}$$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M \text{ and}$$

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq M.$$

In simple words, if $x_k = 1$, then $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > M$ ... (1)

We will use this result in the algorithm. The algorithm SumOfSubsets(a, k, b) has three inputs: Value of 'a' denotes $\sum_{i=1}^{k-1} w_i x_i$, object k to be considered next, value of 'b' denotes $\sum_{i=k}^{n} w_i$.

When algorithm is called with SubOfSubsets(a, k, b), then $[x_1, \ldots, x_{k-1}]$ have already been determined. Assume that weights w[1, ..., n] are in increasing order. Also the algorithm assumes that $w_1 \leq M$ and $\sum_{i=1}^{n} w_i \geq M$.

gorithm: SumOfSubsets (a, k, b)

```
gin
 //Generate left child of state space tree considering xk = 1
 x[k] = 1
 if (a + w[k] = M)
 then //subset is found so display it. Obviously x[k+1, ---, n] contains zeroes.
     Display x[1...k]
 else if (a + w[k] + w[k+1] ≤ M)
     then    //include k^th object and check for (k+1)^th object
             SumOfSubsets (a + w[k], k+1, b - w[k]);
     end if
 end if

 //Generate right child of state space tree considering xk = 0.
 if(a+b - w[k] ≥ M) and (a+w[k +1] < M)
 then
     x[k] = 0
     SumOfSubsets (a, k+1, b - w[k]);
 end if
nd
```

Initially the algorithm is called as SumOfSubsets $(0, 1, \sum_{i=1}^{n} w_i)$.

Let us solve few examples for sum of subsets problem.

Example 4.3 : *Given n = 4 weights* $(w_1, w_2, w_3, w_4) = (7, 11, 13, 24)$. *Find all possible subsets whose sums are M = 31 using sum of subsets algorithm.*

Solution : The state space tree is shown below in Fig. 4.10. Each node shows a, k, b values.

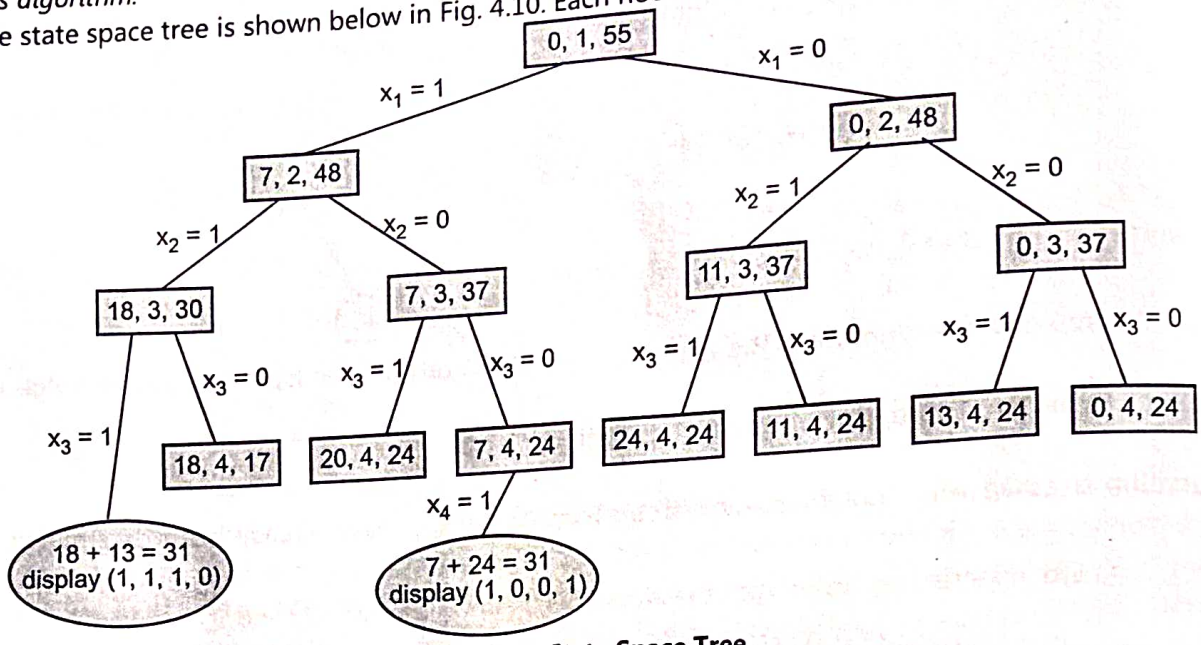


Fig. 4.10 : State Space Tree

So there are two possible subsets for which sum is M : (1, 1, 1, 0) and (1, 0, 0, 1).

Example 4.4 : *Given n = 6 weights*

$$w = \{5, 10, 12, 13, 15, 18\} \text{ and}$$
$$M = 30.$$

Find all possible subsets for which sum = M using SumOfSubsets algorithm. Draw the generated partial state space tree.

Solution : The state space tree is shown below in Fig. 4.11.

So there are total 3 solutions :

P denotes (1, 1, 0, 0, 1, 0).

Q denotes (1, 0, 1, 1, 0, 0).

R denotes (0, 0, 1, 0, 0, 1).

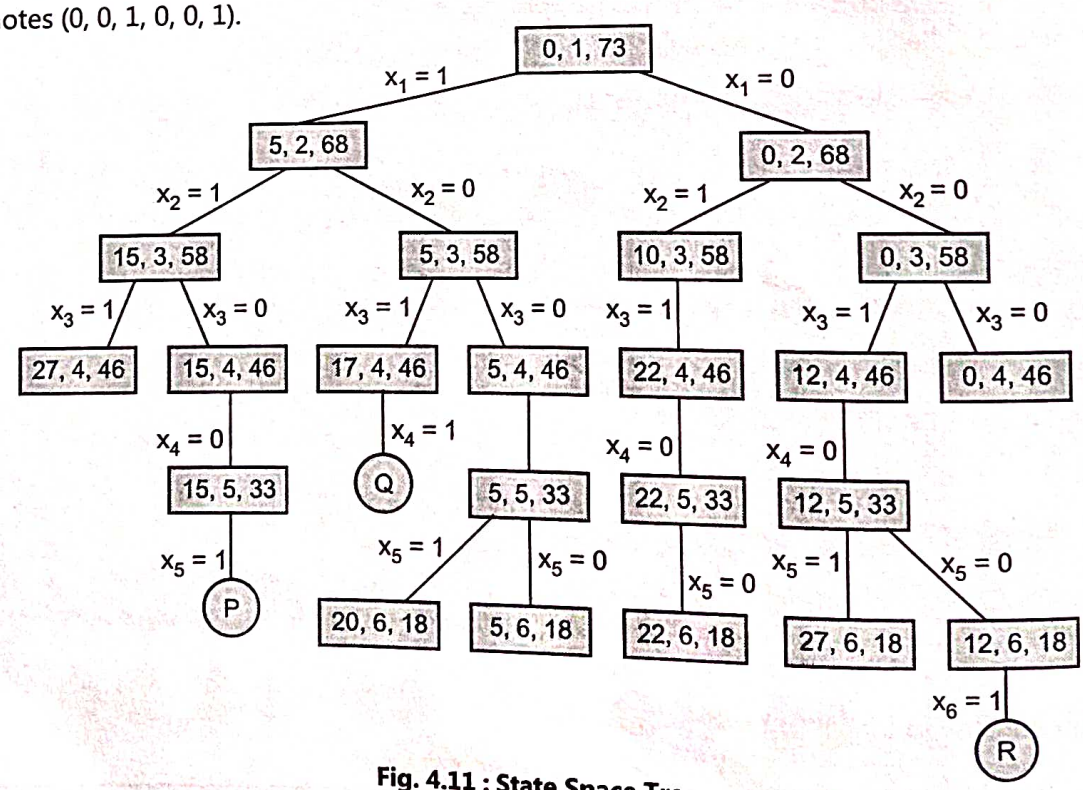


Fig. 4.11 : State Space Tree

Example 4.5 : *Given n = 7 weights w = {5, 7, 10, 12, 15, 18, 20} and M = 35. Find all possible subsets for which sum is M using SumOfSubsets algorithm. Draw the generated state space tree partially.*

Solution : The state space tree is shown in Fig. 4.12.

Hence there are total 3 solutions :

P denotes (1, 0, 1, 0, 0, 0, 1)

Q denotes (1, 0, 0, 1, 0, 1, 0)

R denotes (0, 0, 0, 0, 1, 0, 1)

In general, a full state space tree for n weights contains $2^n - 1$ internal nodes from which calls could be made. The tree may have 2^n leaf nodes which do not generate calls.

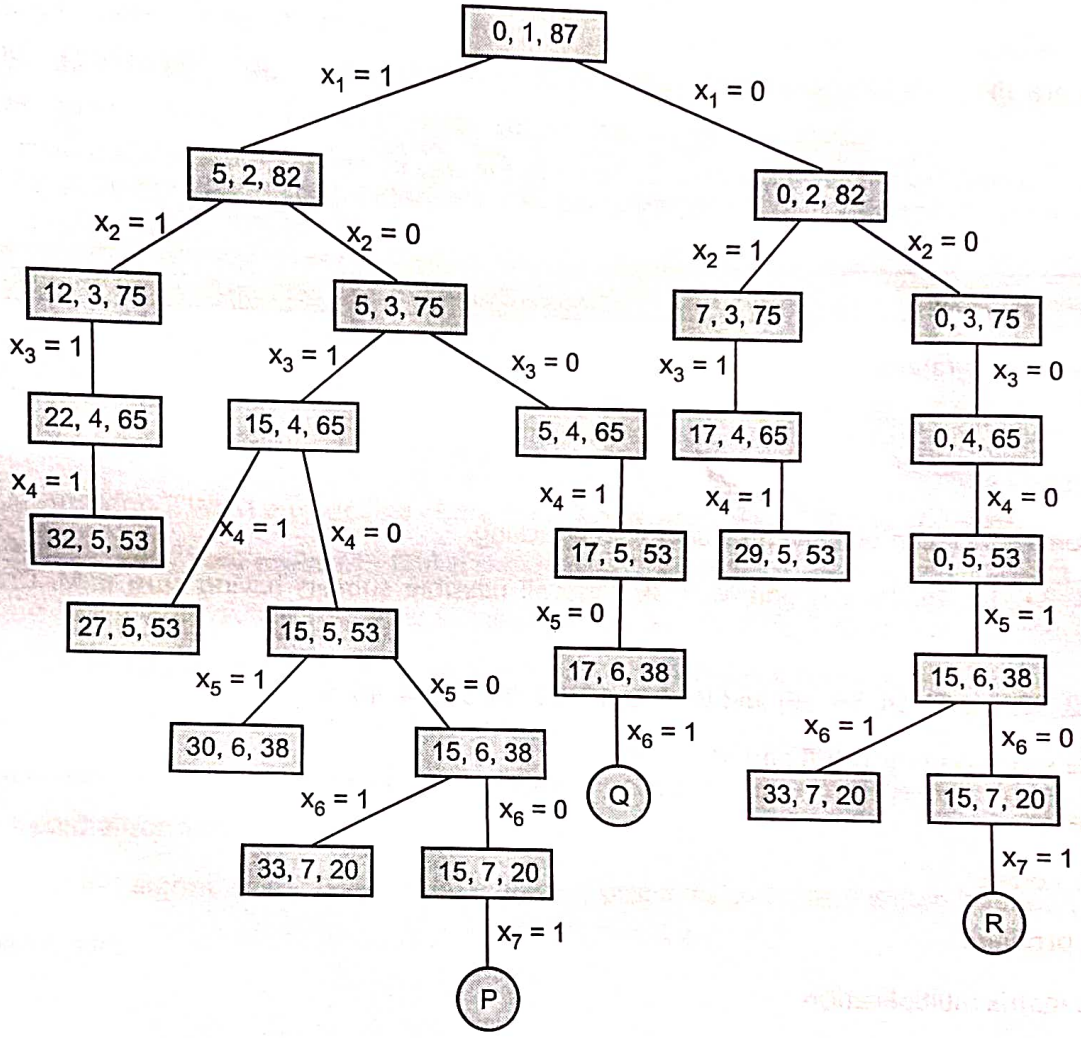


Fig. 4.12 : State space tree

SOLVED EXERCISE

1. **Match the pairs :**

(a) Triangulization	1. Greedy
(b) Quick sort	2. Back track
(c) N-queens	3. Divide a conquer
(d) Job scheduling	4. Dynamic

Ans. : (a) 4, (b) 3, (c) 2, (d) 1

II. Long Question Answers

1. Define the terms: Explicit constraints and implicit constraints. Explain these terms for 8-queens problem.

Solution : 8-queens problem is to place 8 queens on a 8 × 8 chessboard such that no two queens can attack, that is, no two queens are on the same row or same column or same diagonal.

We can keep 8 queens $(q_1, q_2, ..., q_8)$ on the 8 rows (1, 2, ..., 8) respectively. Let $(x_1, x_2, ... x_8)$ gives their column number respectively.

Explicit constraints are rules that restrict each x_i to take values from a given set. It means that each x_i can have one value from the set {1, 2, ..., 8} only. Thus explicit constraints specify the solution space for a particular instance of the problem being solved.

Implicit constraints are the rules to check which tuples in the solution space satisfy the criteria. For 8-queens problem implicit constraint is that no two queens can be placed on the same diagonal or same column, assuming that no two queens are on the same row. Thus implicit constraints describe the way in which x_i must relate to each other.

EXERCISE

1. Explain backtracking strategy.
2. Write a short note on :

   State space tree.
3. List the problems which can be solved by using backtracking.
4. Let w = {6, 8, 11, 13, 16, 19, 21} and M = 36. Find all possible subsets having sum = M. Draw state space tree generated.
5. Let p = {12, 22, 32, 34, 44, 54, 56, 66} and W = {2, 12, 22, 24, 34, 44, 46, 56}.

   Draw the state space tree for n = 8 and M = 110.
6. Discuss in brief :
   (i) Huffman's codes.
   (ii) 8-queens problem.
   (iii) Strassen's matrix multiplication

UNIVERSITY QUESTION BANK

1. (a) Write backtracking algorithm (non-recursive) for the following problem and also mention bounding function, if any :
   (i) Hamiltonian cycle.
   (ii) 8 queen's problem.
   (iii) Graph colouring problem.

   (b) Write a procedure which finds the mode and frequency of an unsorted array. Analyze its computing time. Is your method better than sorting.

2. Write short notes on :
   (a) 0/1 knapsack by backtracking,
   (b) Backtracking and state space tree.
3. Write backtracking algorithm for graph coloring problem.
4. Write backtracking based recursive algorithm for :
   (i) 8-queens problem
   (ii) Graph-coloring problem
5. (a) Write non-recursive backtrack algorithm for n-queens problem.
   (b) Write algorithm using backtrack for subsets-sum for a given set $s = \{s_1, s_2, ..., s_n\}$ of positive integers whose sum is equal to a given positive integer d.

Consider the following problem of assigning jobs to various old and new machines. There are n jobs and n machines. Let t (i, j) be the time taken by machine 'i' to complete job 'j'. Each machine must be assigned exactly one job and each job is to be assigned to exactly one machine. Job assignments are to be done to minimize the net time to complete all jobs. Design a backtracking algorithm to solve this problem. Comment on the time complexity of this algorithm.

Write non-recursive back-track algorithm for n-queens problem. For n=8, compute total number of possible solutions.

(a) What is backtracking ? What are peculiar characteristic and applications of this approach ?

(b) Consider the subset = sum problem : find a subset of a given set $S = \{s_1, s_2, ..., s_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example for S = {1, 2, 5, 6, 8} and d = 9, there are two solutions : {1, 2, 6} and {1, 8}. Of course some instances of this problem may have no solutions.

Write recursive backtrack algorithm for the same.

(c) Can greedy approach give solution for the above problem. If yes, write the algorithm. If no, give dynamic approach based algorithm.

Write the backtracking algorithm for TWO of the following problems and also mention bounding function if any :

(a) Hamiltonian Cycle,

(b) Knight's Tour,

(c) Sum of subset

Represent a graph by its adjacency matrix GRAPH (1 : n, 1 : n), where GRAPH (i, j) = False. Let integers 1.....m represent various colors. Determine all the different ways in which a given graph may be coloured using at most m colours. Find the solution in terms of an n-tuple (X(1),.....X(n)) where X(i) is the color of node i. Use recursive backtracking formulation and write a suitable algorithm for m coloring.

Suggest suitable algorithm strategy and write algorithm for each (any two)

(a) Strassen's matrix multiplication

(b) 8 queen's problem

(c) Sum-of-subsets.

12. (a) Let G be graph and m be given positive integer. We want to discover if 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, where m is the smallest integer for which the graph G can be colored. Devise suitable algorithm for the same.

    (b) Write dynamic algorithm to compute minimum cost path from source to destination in multistage graph.

13. State and explain the algorithmic strategies applied for the following problems :

    (a) Tower of Hanoi

    (b) Graph coloring problem

    (c) 4-Queen's problem.

✠ ✠ ✠