Introduction:

In back tracking technique, we will solve problems in an efficient way, when compared to other methods like greedy method and dynamic programming. The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P(x1, …. xn). Form a solution

at any point seems not promising, ignore it. All possible solutions require a set of constraints divided into two categories:

1. **Explicit Constraint:** Explicit constraints are rules that restrict each xi to take on values only from a given set. Ex: xn= 0 or 1.

2. **Implicit Constraint:** Implicit Constraints are rules that determine which of the tuples in the solutions space of I satisfy the criterion function.

Implicit constraints for this problem are that no two queens can be on the same diagonal.

Back tracking is a modified depth first search tree. Backtracking is a procedure whereby, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child. State space tree exists implicitly in the algorithm because it is not actually constructed.

Terminologies which is used in this method:

1. **Solution Space:** All tuples that satisfy the explicit constraints define a possible solution space for a particular instance T of the problem.

Example:

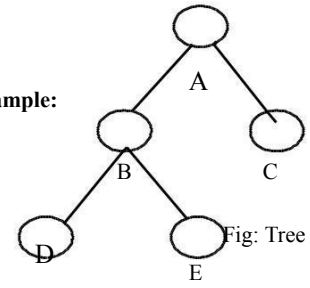


Fig: Tree organization of a solution space

2. **Problem State:** A problem state is the state that is defined by all the nodes within the tree organization.

Example:

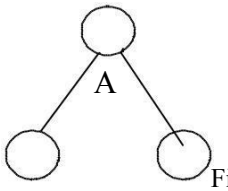


Fig: Problem State

3. **Solution States:** These are the problem states S for which the path form the root to S defines a tuple in the solution space.

Here, square nodes ( ) indicate solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e., (ghk-,B,D),(A,C,F) and (A,C,G) are the solution states.

Example:

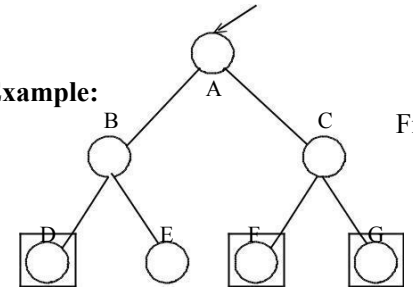


Fig: Solution State

4. **State Space Tree:** Is the set of paths from root node to other nodes. State space tree is the tree organization of the solution of the solution space.

Example: State space tree of a 4-queen problem.

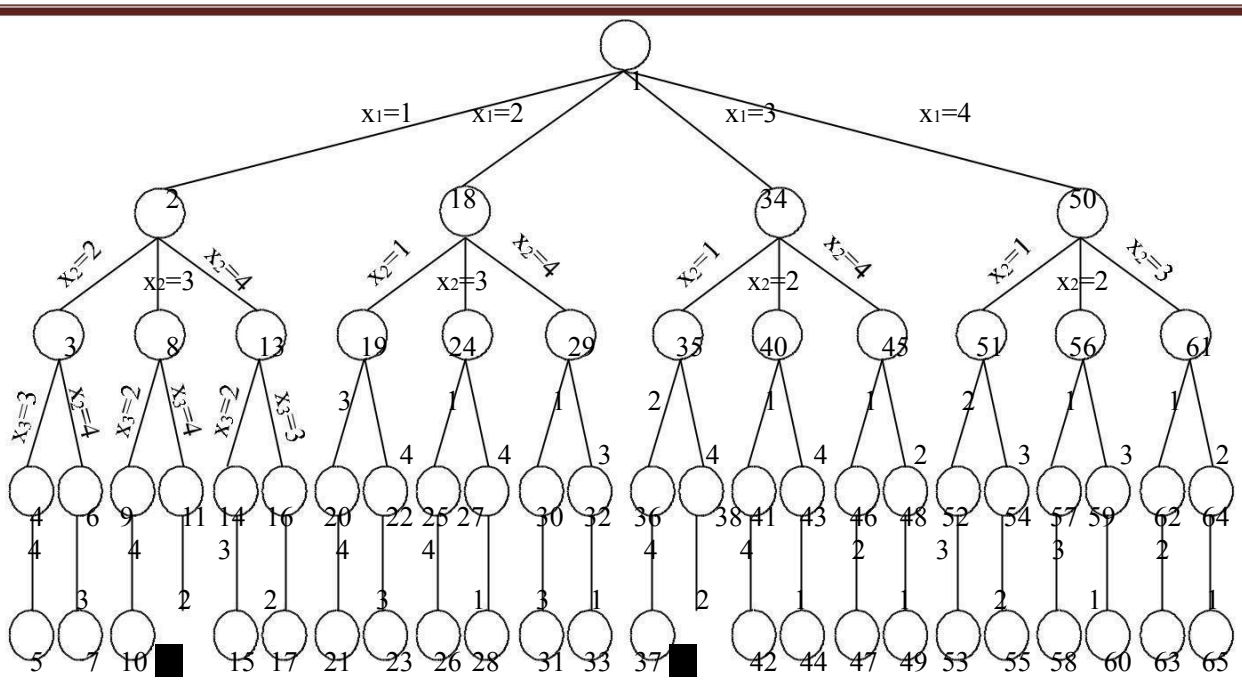


Fig: Tree organization of the 4-queens solution space

In the above figure nodes are numbered as in depth first search. Initially, (x1=1 or 2 or 3 or 4, it means we can place first queen in either first, second, third or fourth column. If x1=1 then x2 can be placed in either 2nd, 3rd or 4th columns. If (x2=2) then, x2 can be placed either in 3rd, or 4th column. If x3=3, then x4=4. So nodes 1-2-3-4-5 is one solution in solution space. It may not be a feasible solution.

5. **Answer States:** These solution states S, for which the path from the root node to S defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem.

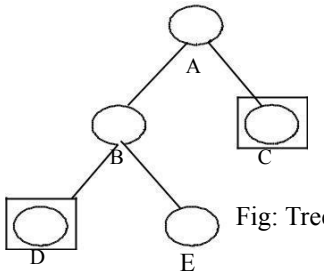


Fig: Tree (Answer States)

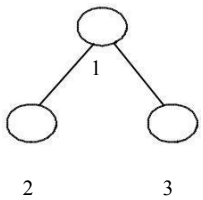
Here are C, D are answer states. (A, C) and (A, C, D) are solution states.

6. **Live Node:** A node which has been generated but whose children have not yet been generated is live node.

Example 1:

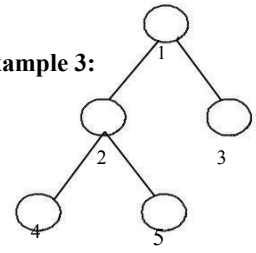
This node 1 is called as live node since the children of node 1 have not been generated.

Example 2:



In this, node 1 is not a live node but node 2, node 3 are live nodes.

Example 3:



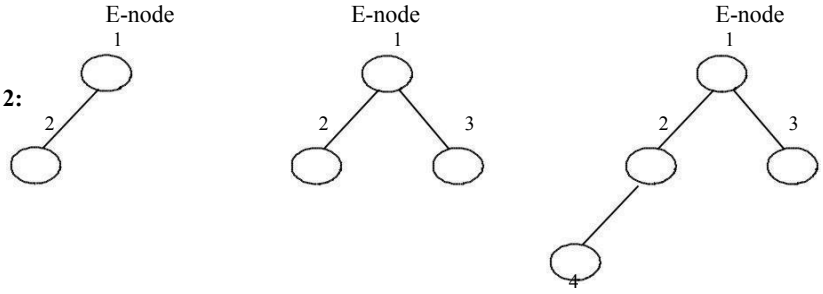
Here, 4, 5, 3 are live nodes because the children of these nodes not yet been generated.

7. **E-Node:** The live nodes whose children are currently being generated is called the E-node (node being expanded).

Example 1:

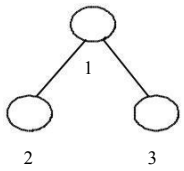
This node 1 is live node and its children are currently being generated (expanded).

Example 2:



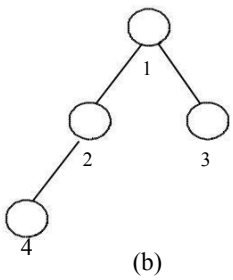
Here, node 2 is E-node.

8. **Dead Node:** It is generated node, that is either not to be expanded further or one for which all of its children has been generated.



(a)

Nodes 1, 2, 3, are dead nodes. Since node 1's children generated and node 2, 3 are not expanded. Assumed that node 2 generated one more node, So, 1, 3, 4 are dead nodes.



(b)

Fig: Dead nodes

General Method:

The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.

The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space (i.,e set of all feasible solutions) for the given problem.

Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.

```
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 (Bk(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: Recursive backtracking

Applications of Backtracking

Backtracking is an algorithm design technique that can effectively solve the larger instances of combinational problems. It follows a systematic approach for obtaining solution to a problem. The applications of backtracking include,

1) **N-Queens Problem:** This is generalization problem. If we take n=8 then the problem is called as 8 queens problem. If we take n=4then the problem is called 4 queens problem. A classic combinational problem is to place n queens on a n*n chess board so that no two attack, i.,e no two queens are on the same row, column or diagonal.

Algorithm of n-queens problem is given below:

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

Algorithm: All solutions to the n-queens problem

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

Algorithm: Can a new queen be placed?

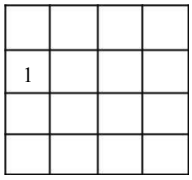
4-Queens problem:

Consider a 4*4 chessboard. Let there are 4 queens. The objective is place there 4 queens on 4*4 chessboard in such a way that no two queens should be placed in the same row, same column or diagonal position.

The explicit constraints are 4 queens are to be placed on 4*4 chessboards in 44 ways.

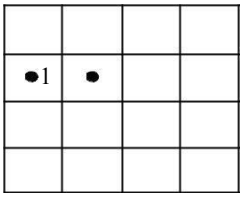
The implicit constraints are no two queens are in the same row column or diagonal.
Let{x1, x2, x3, x4} be the solution vector where x1 column on which the queen i is placed.

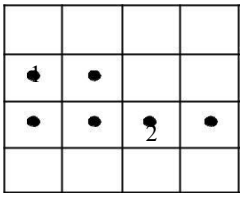
First queen is placed in first row and first column.



(a)

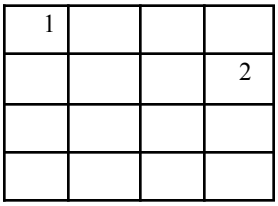
The second queen should not be in first row and second column. It should be placed in second row and in second, third or fourth column. It we place in second column, both will be in same diagonal, so place it in third column.



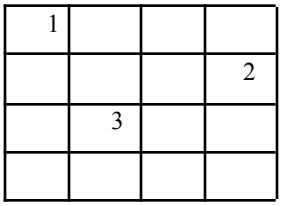


(b) (c)

We are unable to place queen 3 in third row, so go back to queen 2 and place it somewhere else.



(d)



(e)

Now the fourth queen should be placed in 4th row and 3rd column but there will be a diagonal attack from queen 3. So go back, remove queen 3 and place it in the next column. But it is not possible, so move back to queen 2 and remove it to next column but it is not possible. So go back to queen 1 and move it to next column.

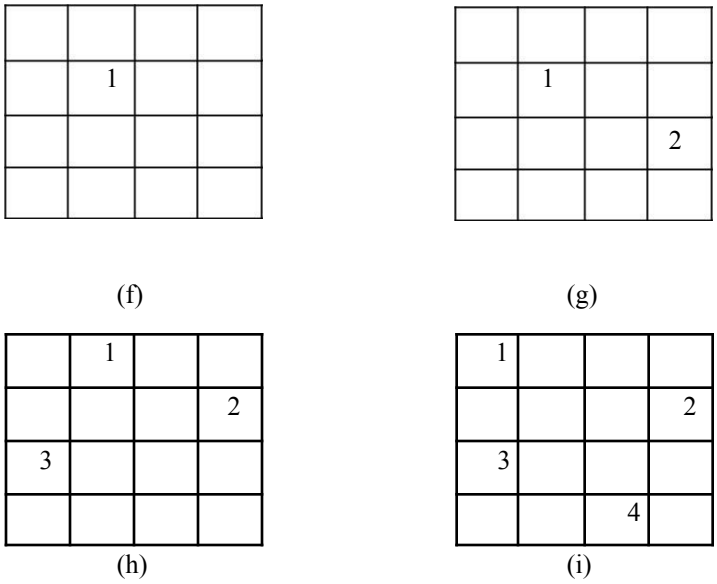


Fig: Example of Backtrack solution to the 4-queens problem

Hence the solution of to 4-queens's problem is x1=2, x2=4, x3=1, x4=3, i.,e first queen is placed in 2nd column, second queen is placed in 4th column and third queen is placed in first column and fourth queen is placed in third column.

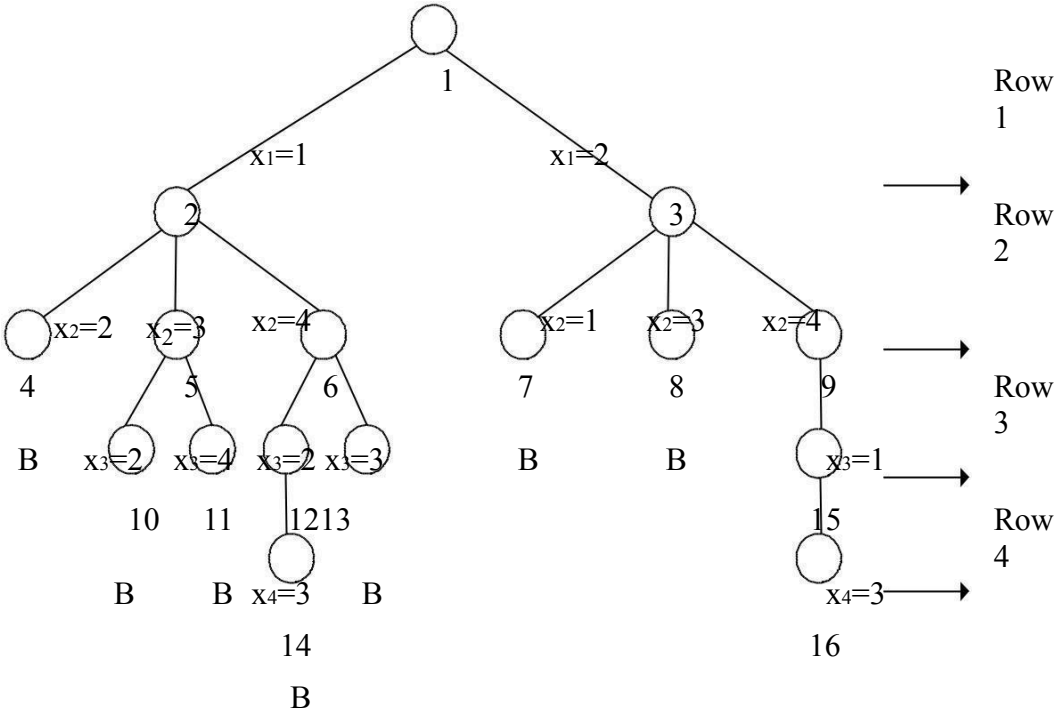


Fig: Portion of the tree that is generated during Backtracking

8-queens problem

A classic combinatorial problem is to place 8 queens on a 8*8 chess board so that no two attack, i.,e no two queens are to the same row, column or diagonal.

Now, we will solve 8 queens problem by using similar procedure adapted for 4 queens problem. The algorithm of 8 queens problem can be obtained by placing n=8, in N queens algorithm. We observe that, for every element on the same diagonal which runs from the upper left to the lower right, each element has the same “row-column” value. Also every element on the same diagonal which goes from upper right to lower left has the same “row+column” value.

If two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if

i-j=k-l …….(1) or
i+j=k+l …….(2).

From (1) and (2) implies
j-l=i-k and

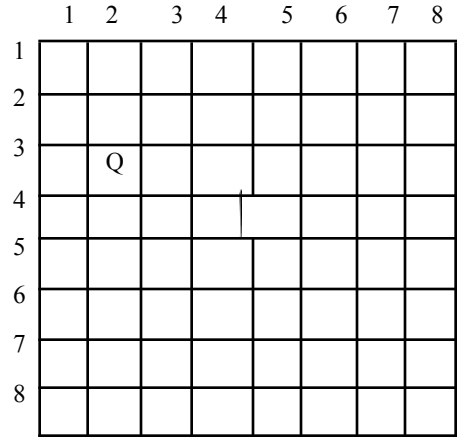
j-l=k-i
Two queens lie on the same diagonal iff

|j-l|=|i-k|

But how can we determine whether more than one queen is lying on the same diagonal? To answer this question, a technique is deviced. Assume that the chess board is divided into rows

And columns say A: $\left[\underbrace{1....8}_{\text{rows}}, \underbrace{1....8}_{\text{columns}}\right]$

This can be diagrammatically represented as follows



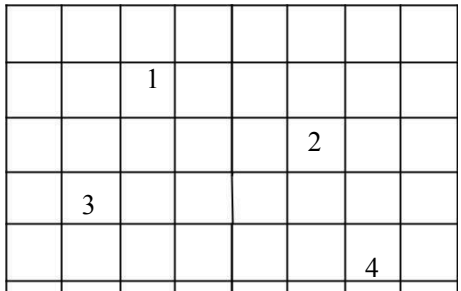
Now, assume that, we had placed a queen at position (3,2).
Now, its diagonal cells includes (2,1)(4,3)(5,4)….(if we traverse from upper left to lower

right). If we subtract values in these cells say 2-1=1,4-3=1,5-4=1, we get same values, also if we traverse from upper right to lower left say (2,3) (1,4)(4,1)….we get common values when we add the

bits of these cells i.,e 2+3=5, 1+4=5, 4+1=5. Hence, we say that, on traversing from upper left to lower right, if (m,n)(a,b) are the diagonal elements(of a cell) than m-n=a-b or on traversing from upper right to lower left if(m,n)(a,b) are the diagonal elements(of a cell) then m+n=a+b.

The solution of 8 queens problem can be obtained similar to the solution of 4 queens. problem.X1=3, X2=6, X3=2, X4=7, X5=1, X6=4, X7=8, X8=5,

The solution can be shown as



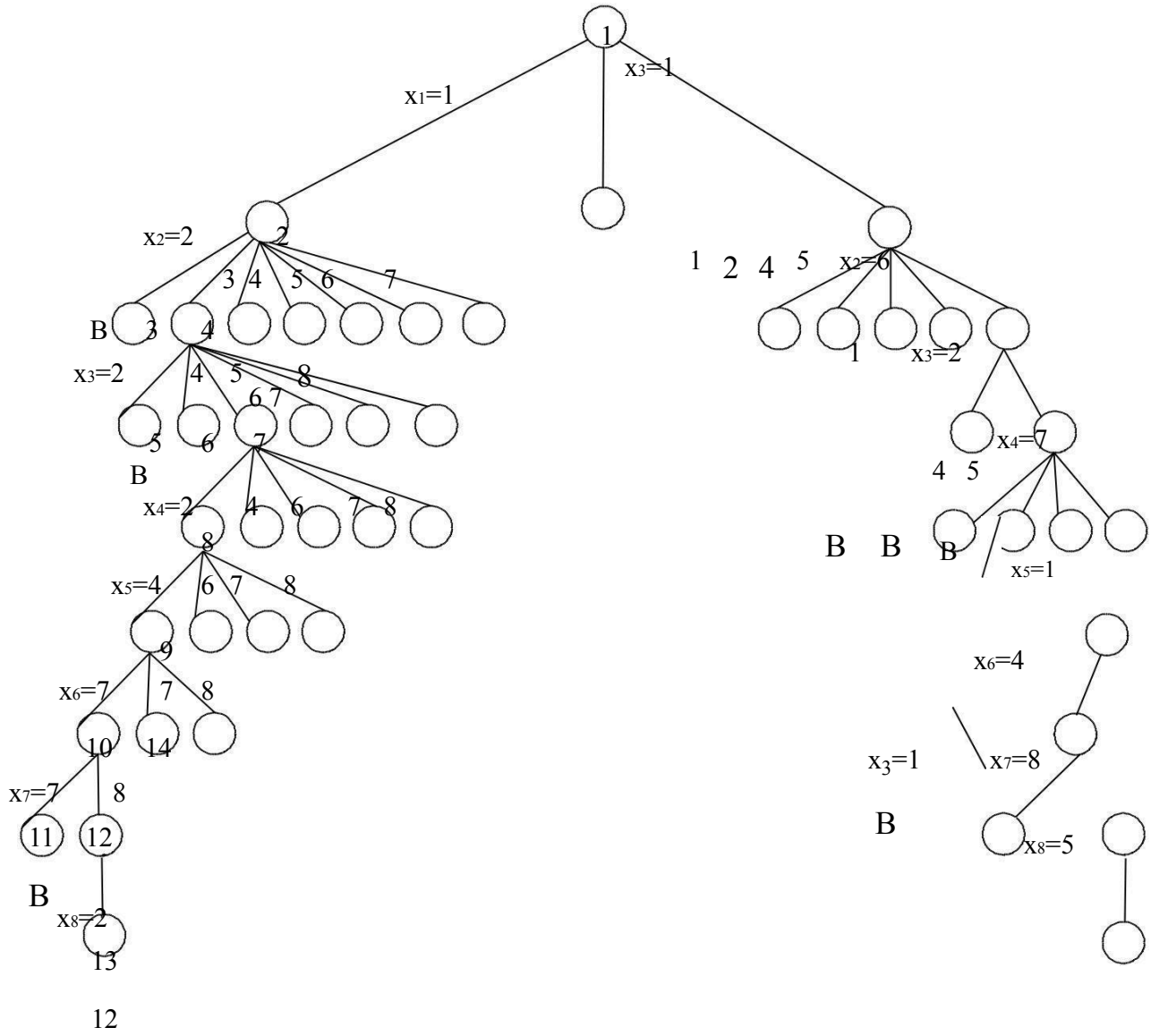
5

6

7

8

Time complexity: The solution space tree of 8-queens problem contains 8^8 tuples. After imposing implicit constraints, the size of solution space is reduced to 8! tuples.
The state space tree for the above solution is given



2) Sum of Subsets Problem
Given a set of n objects with weights (w1,w2,…..w3) and a positive integer M. We have to find a subset S' of the given set S, such that

S'⊆ S

Sum of the elements of subset S' is equal to M.

For example, if a given set S=(1,2,3,4) and M=5, then there exists sets S'(3,2) and S'=(1,4) whose sum is equal to M.
It can also be noted that some instance of the problem does not have any solution.

For example, if a given set S=(1,3,5) and M=7, then no subset occurs for which the sum is equal to M=7.

The sum of subsets problem can be solved by using the back tracking approach. In this implicit tree is created, which is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that, the elements of the given set are arranged increasing order.

The left child of the root node indicates that, we have to include the first element and right child of the root node indicates that, we have to exclude the first element and so on for other nodes. Each node stores the sum of the partial solution element. If at any stage, the number equals to 'M' then the search is successful. At this time search will terminate or continues if all the possible solutions need to be obtain. The dead end in the tree occurs only when either of the two inequalities exists.

The sum of S' is too large.

The sum of S' is too small.
Thus we take back one step and continue the search.

ALGORITHM:

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

Algorithm: Recursive backtracking algorithm for sum of subsets

Example:

Let m=31 and w= {7, 11, 13, 24} draw a portions of state space tree.

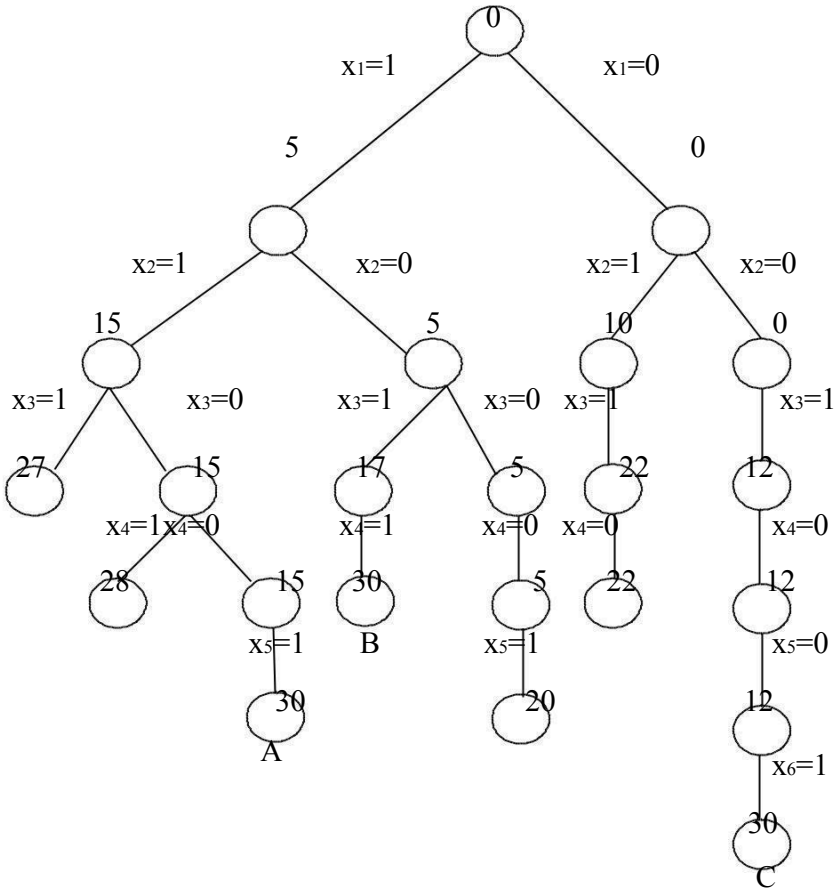
Solution: Initially we will pass some subset (0, 1, 55). The sum of all the weights from w is 55, i.e., 7+11+13+24=55. Hence the portion of state –space tree can be
Here Solution A={1, 1, 1, 0} i.e., subset {7, 11, 13}

And Solution B={1, 0, 0,1} i.e. subset {7, 24}
Satisfy given condition=31;

Example: Consider a set S={5, 10, 12, 13, 15, 18} and N=30.

Subset {Empty}	Sum=0	Initially subset is 0
5	5	
5, 10	15	
5, 10, 12	27	
5, 10, 12, 13	40	Sum exceeds N=30,Hence Backtrack
5, 10, 12, 15		Not Feasible
5, 10, 12, 18		Not feasible
5, 10		List ends. Backrack
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible. Backtrack
5, 10	15	
5, 10, 15	30	Solution obtained

We can represent various solutions to sum of subset by a state space tree as,



3) Graph Coloring

Let G be a graph and m be a given positive integer. The graph coloring problem is to find 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. This is termed the m-colorability decision problem. 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.

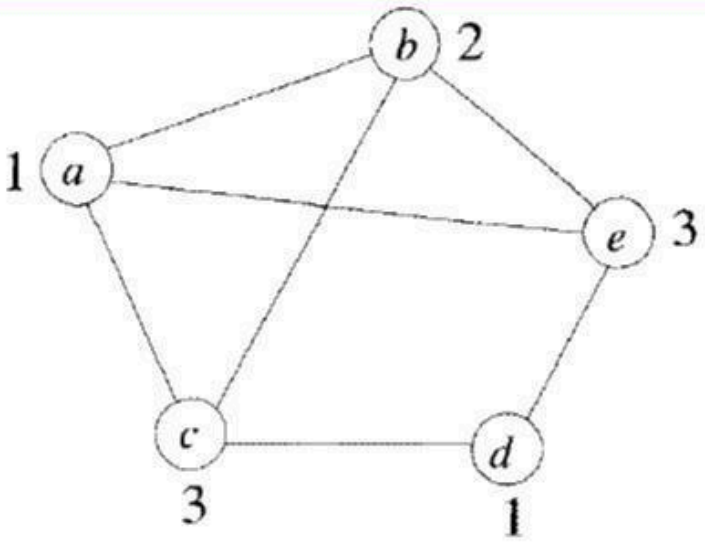


Figure. An Ex. Of graph coloring

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

Algorithm: Finding all m-colorings of graph

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

Algorithm: Finding next color

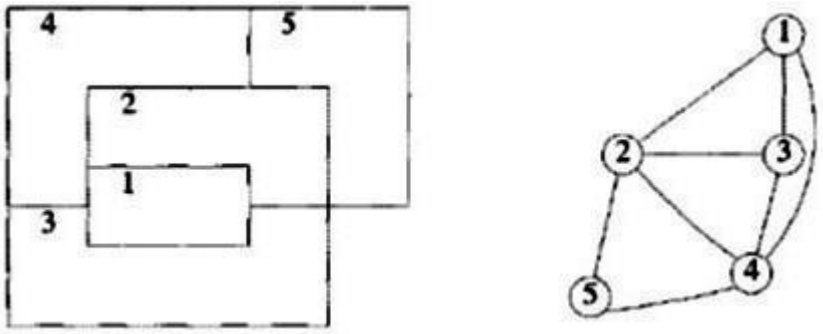


Fig. A map and it's planar graph representation

To color the above graph chromatic number is 4. And the order of coloring is X1=1, X2=2, X3=3, X4=4, X5=1

Time Complexity: At each internal node O(mn) time is spent by Nextcolor to determine the children corresponding to legal coloring. Hen the total time is bounded by,

$$\sum_{i=1} m^i n = n(m+m^2+.....+m^n$$

$$=n.m^n$$

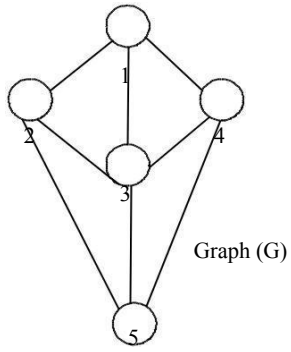
$$=O(n.m^n)$$

4) Hamiltonian Cycle

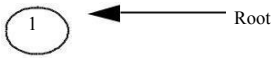
Let G = (V', E) be a connected graph with n vertices. A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and ret urns to its starting position. In other words if a Hamiltonian cycle begins at some vertex Vi E G and the vertices of G are visited in the order '01,112,"" Vn+l, then the edges (Vi, Vi+1) are in E, 1 <=i<=n, and the 'Vi are distinct except for V1 and Vn+l, which are equal.

Given a graph G=(V,E)we have to find the Hamiltonian circuit using backtracking approach, we start out search from any arbitrary vertex, say x. This vertex ‘x’ becomes the root of our implicit tree. The next adjacent vertex is selected on the basis of alphabetical / or numerical order. If at any stage an arbitrary vertex, say ‘y’ makes a cycle with any vertex other than vertex ‘o’ then we say that dead end is reached. In this case we backtrack one step and again the search begins by selecting another vertex. It should be noted that, after backtracking the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

Example: Consider a graph G=(V,E), we have to find the Hamiltonian circuit using backtracking method.

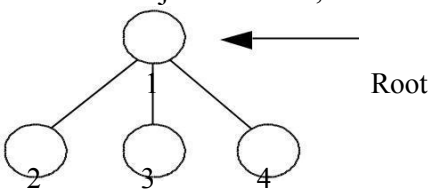


Solution: Initially we start out search with vertex ‘1’ the vertex ‘1’ becomes the root of our implicit tree.



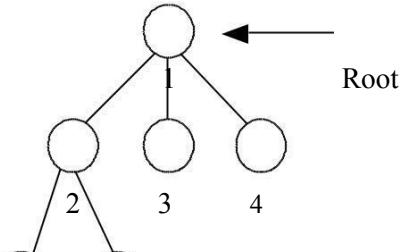
(a)

Next we choose vertex ‘2’ adjacent to ‘1’, as it comes first in numerical order (2, 3, 4).



(b)

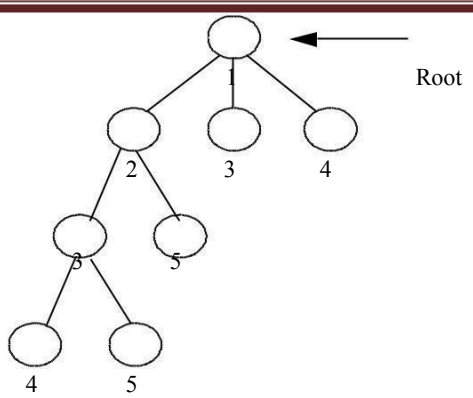
Next vertex ‘3’ is selected which is adjacent to ‘2’ and which comes first in numerical order (3,5).



3 5

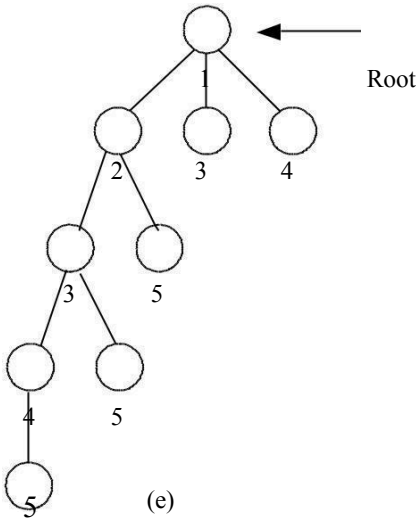
(c)

Next we select vertex ‘4’ adjacent to ‘3’ which comes first in numerical order (4, 5).



(d)

Next vertex ‘5’ is selected. If we choose vertex ‘1’ then we do not get the Hamiltonian cycle.



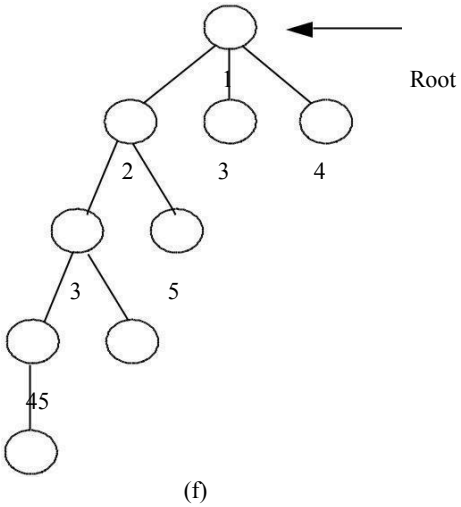
(e)

Dead end

The vertex adjacent to 5 is 2, 3, 4 but they are already visited. Thus, we get the dead end. So, we backtrack one step and remove the vertex ‘5’ from our partial solution.

The vertex adjacent to ‘4’ are 5,3,1 from which vertex ‘5’ has already been checked and we are left with vertex ‘1’ but by choosing vertex ‘1’ we do not get the Hamiltonian cycle. So, we again backtrack one step.

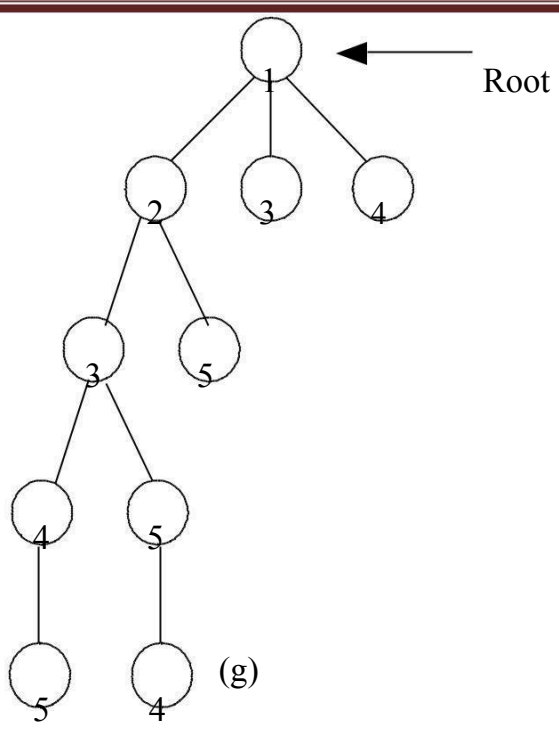
Hence we select the vertex ‘5’ adjacent to ‘3’.



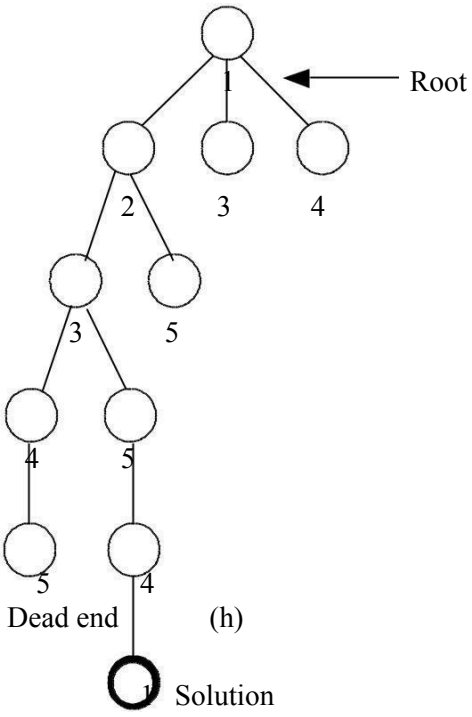
(f)

Dead end

The vertex adjacent to ‘5’ are (2,3,4) so vertex 4 is selected.



The vertex adjacent to '4' are (1, 3, 5) so vertex '1' is selected. Hence we get the Hamiltonian cycle as all the vertex other than the start vertex '1' is visited only once, 1- 2- 3- 5- 4- 1.



The final implicit tree for the Hamiltonian circuit is shown below. The number above each node indicates the order in which these nodes are visited.

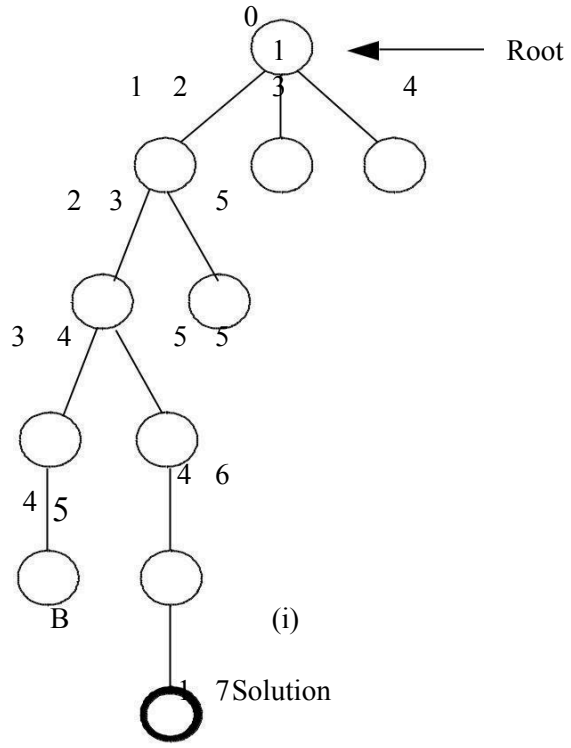


Fig Construction of Hamilton Cycle using Backtracking

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

Algorithm: Finding all Hamiltonian cycles

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

Algorithm: Generating a next vertex