Unit -IV

BACKTRACKING

1.The General Method:

It is one of the most general algorithm design technique.

- ✓ Many problems which deal with searching for a set of solutions or for an optimal solution satisfying some constraints can be solved using the backtracking formulation.
- ✓ To apply backtracking method, the desired solution must be expressible as an n- tuple $(X_1 \ldots X_n)$ where X_i is chosen from some finite set S_i.
- ✓ The problem is to find a vector, which maximizes or minimizes or satisfies a criterion function $P(X_1 \ldots X_n)$.
- ✓ Its basic idea is to build up the solution vector, one component at a time and to test whether the vector being formed has any chance of success.
- ✓ Advantage of this method is, once we know that a partial vector $(X_1 \ldots X_i)$ will not lead to an optimal solution, then $(m_{i+1} \ldots m_n)$ possible test vectors may be ignored entirely.

✓ Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

These constraints are classified as:

i) Explicit constraints.
ii) Implicit constraints.

1) Explicit constraints:

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

Some examples are,

Xi >=0	or	Si = {all non-negative real nos.}
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 I of the problem being solved
All tuples that satisfy the explicit constraint define a possible solution space for I

2) Implicit constraints:

The implicit constraints determine which of the tuples in the solution space I can actually satisfy the criterion functions.

Solution is represented by using state space tree
Each node in the tree is called a **problem state.**
All paths from the root to other nodes define the **state space** of the problem.

Solution states: These are the problem states S for which the path from root to S define a tuple in the solution space.

Answer states: These are the leaf nodes which correspond to an element in the set of solutions, i.e. these are the states which satisfy the implicit constraints.

The tree organization of the solution space is referred to as the **state space tree**.

- ✓ A node which has been generated and all of whose children have not yet been generated is called **live node**.
- ✓ The live node whose children are currently being generated is called an **E-node**.
- ✓ A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

➢ There are two methods of generating state space tree

1. Backtracking**:**

In this method, as soon as a new child C of current E-node N is generated, this child will be the new E-node. The N will become the E-Node again when the sub tree C has been fully explored.

2. Branch and Bound:

E-node will remain as E-node until it is dead.

Back Tracking Process:

All answer nodes are to be found (i.e all possible solutions)
Let $(x_1, x_2 \ldots x_i)$ be a path from the root to a node in a state space tree.

Let $T(x_1 \ldots x_i)$ be the set of all possible values for x_{i+1}. S.T. $(x_1,x_2,..x_{i+1})$ is also a path to a problem state.

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

Bounding function $B_{i+1}(x_1,x_2,..x_{i+1})$ is false, if path $(x_1,x_2,..x_{i+1})$ cannot be extended to reach an answer node.

Backtracking starts with node 1.

- Recursive Backtracking Algorithm:

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k − 1 values
// x[1], x[2], . . . , x[k − 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], . . . , x[k − 1]) do
    {
        if (Bk(x[1], x[2], . . . , x[k]) ≠ 0) then
        {
            if (x[1], x[2], . . . , x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

- Iterative Backtracking Algorithm:

```
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 ≠ 0) do
    {
        if (there remains an untried x[k] ∈ T(x[1], x[2], . . .,
            x[k − 1]) and Bk(x[1], . . . , x[k]) is true) then
        {
                if (x[1], . . . , 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.
    }
}
```

2.8-Queens Problem

This 8 queens problem is generalized case of n- queens problem.

N-queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

The solution vector $X(X_1 \ldots X_n)$ represents a solution in which Xi is the column of the I^{th} row where I^{th} queen is placed.

1. Check no two queens are in same row.
2. Check no two queens are in same column.

   The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I^{th} queen, where I represents the row and X (j) represents the column position.
3. Check no two queens are in the same diagonal.

   Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same (row - column) value.
4. Every element on the same diagonal that runs from upper right to lower left has the same (row + column) value.

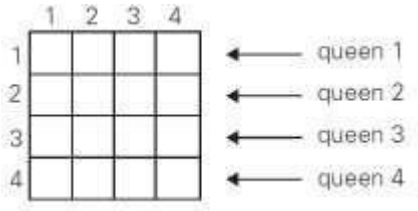
   Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|i-k|.

To Find the Solution:

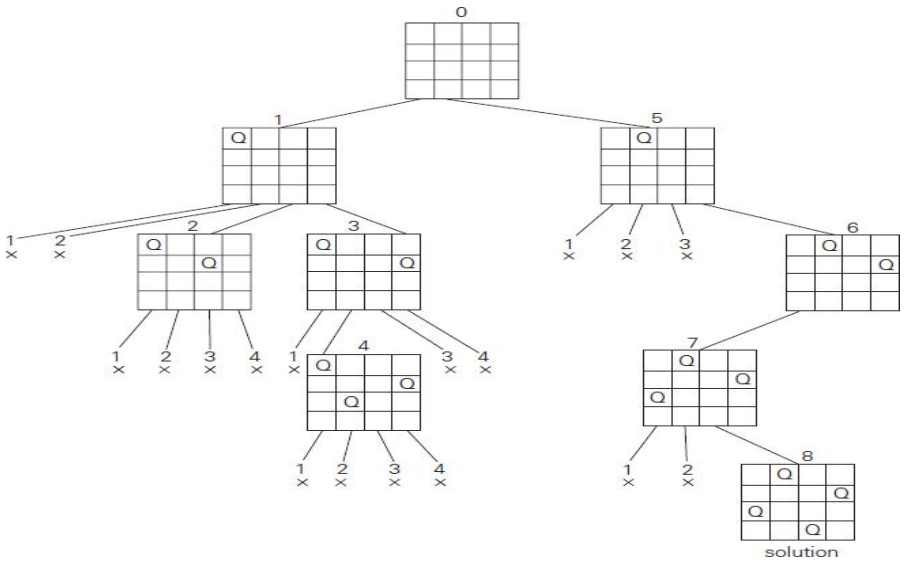
Initialize x array to zero and start by placing the first queen in k=1 in the first row.
To find the column position start from value 1 to n, where 'n' is the no. of columns or no. of queens.
If k=1 then x(k)=1.so (k,x(k)) will give the position of the k^{th} queen. Here we have to check whether there is any queen in the same column or diagonal.
For this considers the previous position, which had already, been found out.
Check whether
X(i)=X(k) for column |X(i)-X(k)|=(i-k) for the same diagonal.
If any one of the conditions is true then return false indicating that kth queen can't be placed in position X(k).
For not possible condition increment X(k) value by one and precede until the position is found.
If the position X(k) n and k=n then the solution is generated completely.
If k<n, then increment the 'k' value and find position of the next queen.
If the position X(k)>n then k^{th} queen cannot be placed as the size of the matrix is 'N*N'.
So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

4 Queens Problem:

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.

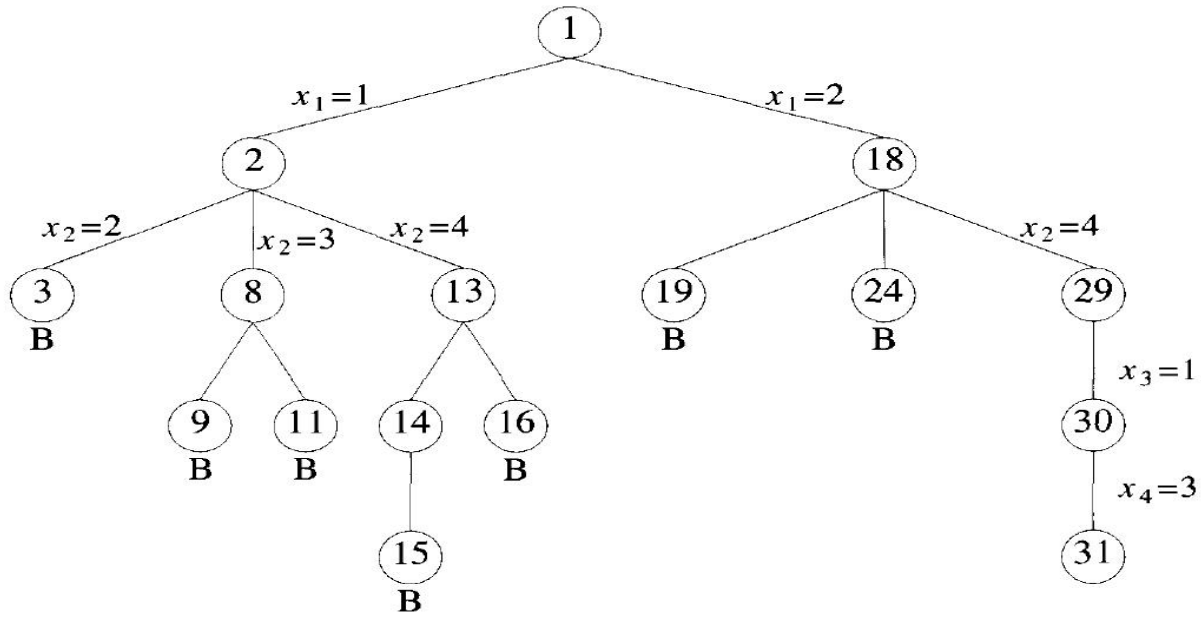


We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves 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 all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to(3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in figure.



If other solutions need to be found, the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, we can use the board's symmetry for this purpose.

Finally, it should be pointed out that a single solution to the n-queens problem for any n ≥ 4 can be found in **linear time**.



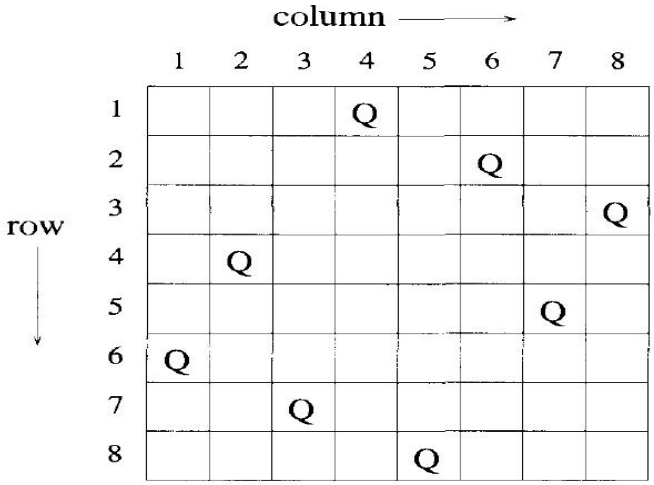
Algorithm to Solve N-queens Problem

```
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 place will determine whether placement is safe or not.

```
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;
}
```

One possible solution to the 8 queens problem



3.Sum of subsets problem

- We are given 'n' positive numbers called weights and we have to find all
- combinations of these numbers whose sum is M. This is called sum of subsets problem.
- If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
- For the state space tree of the solution, for a node at level i, the left child corresponds to X(i)=1 and right to X(i)=0.

For example, for W = {1, 2, 5, 6, 8} and m = 9, there are two solutions: {1, 2, 6} and {1, 8}.
Constraints are..

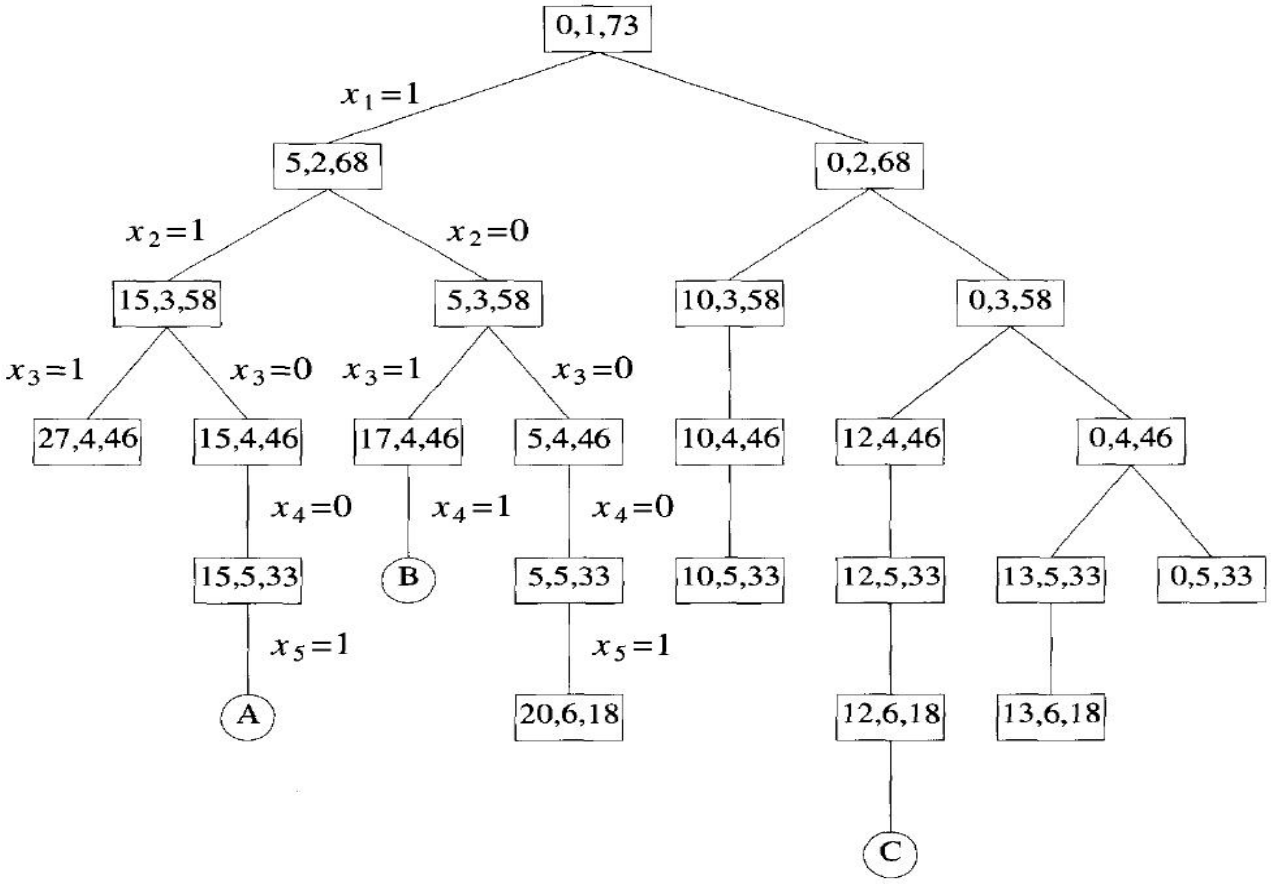
$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

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

Algorithm:

```
Algorithm SumOfSub(s, k, r)
// Find all subsets of w[1 : n] that sum to m. The values of x[j],
// 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
    // Generate left child. Note: s + w[k] ≤ 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 ≤ j ≤ n.
    else if (s + w[k] + w[k + 1] ≤ m)
        then SumOfSub(s + w[k], k + 1, r - w[k]);
    // Generate right child and evaluate B_k.
    if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```

Example:
Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).
We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.
In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets, 'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.
The state space tree for the given problem is,

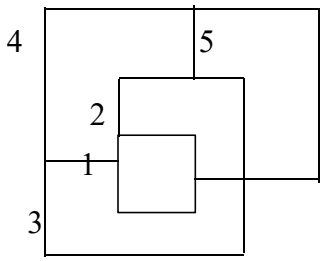


3. Graph Coloring Problem

- ✓ Let 'G' be a graph and 'm' be a given positive integer. 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. So it's called M-color ability decision problem.
- ✓ The graph G can be colored using the smallest integer 'm'. This integer is referred to as chromatic number of the graph.
- ✓ Graph coloring found its applications in coloring various regions in a map.
- ✓ Before finding minimum number of colors needed map is represented in planar graph representation.
- ✓ A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.

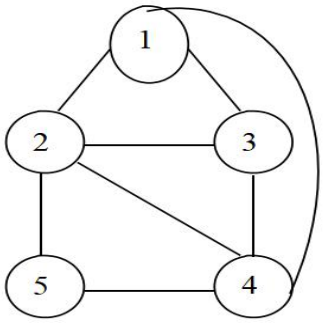
Suppose we are given a map we have to convert it into planar.
Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.
2 is adjacent to 1, 3, 4, 5
3 is adjacent to 1, 2, 4
4 is adjacent to 1, 2, 3, 5
5 is adjacent to 2, 4

Planar graph representation of map is



To color the Graph

- ✓ First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.
- ✓ The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),……..,X(n) ,X(index) is the color, index is the node.
- ✓ The formula which is used to set the color is, X(k) = (X(k)+1) % (m+1)
- ✓ First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.
- ✓ Repeat the procedure until all possible combinations of colors are found.
- ✓ The function which is used to check the adjacent nodes and same color is, If((Graph (k,j) == 1) and X(k) = X(j)).

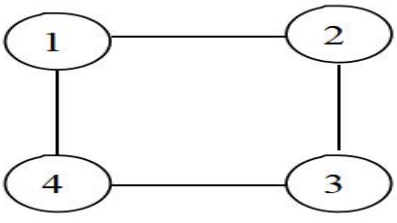
- Algorithm:

```
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, ..., 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 NextValue determine the next safe color

```
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.
}
```

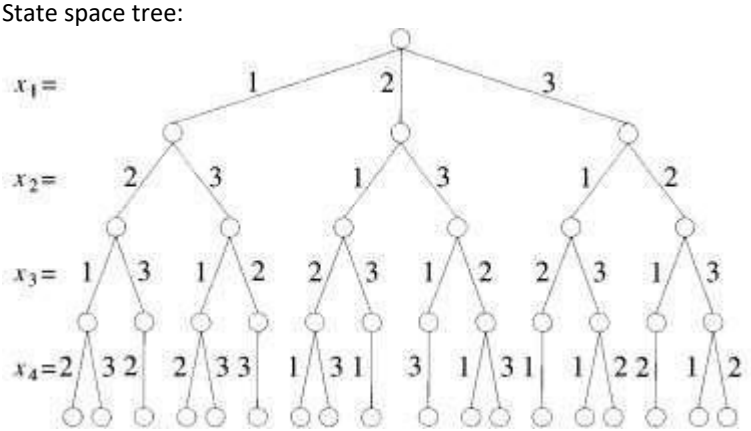
Example: n=4 m=3



To generate solution algorithm requires graph to be represented in boolean valued adjacency matrix representation.

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

State space tree:

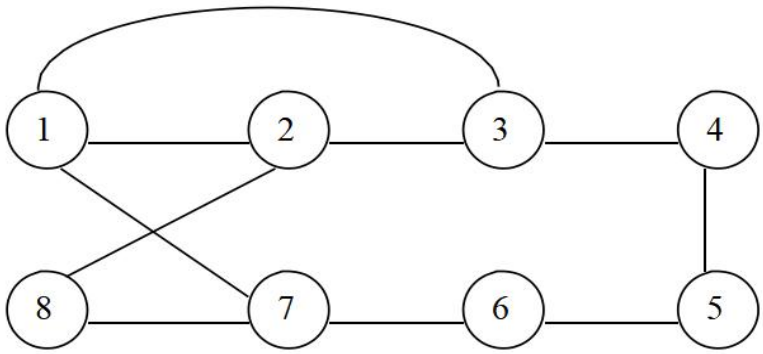


Time complexity of Nextvalue to determine the children is O(mn)
Total time is = $O(nm^n)$.

5.Hamiltonian Cycles

Let G=(V,E) be a connected graph with 'n' vertices.
Hamiltonian cycle is a round trip path along 'n' edges of G that visits every vertex once and returns to its starting position.

If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertices of G are visited in the order of V1,V2…….Vn+1,then the edges (Vi,Vi+1) are in E,1<=i<=n, and the Vi are distinct except for V1 and Vn+1 which are equal.
Consider an example graph G1.



Hamiltonian cycles are

- 1,3,4,5,6,7,8,2,1
- 1,2,8,7,6,5,4,3,1.

The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

- ✓ To find Hamiltonian Cycles in a graph G
  - Initialize solution vector X(X1……..Xn) where Xi represents the ith visited vertex of the proposed cycle.
  - The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.
  - Create a cost adjacency matrix for the given graph.
  - Now we have to find the second vertex to be visited in the cycle.
  - The vertex from 1 to n are included in the cycle one by one by checking 2 conditions

1. There should be a path from previous visited vertex to current vertex.
2. The current vertex must be distinct and should not have been visited earlier.

- When these two conditions are satisfied the current vertex is included in the
- Cycle; else the next vertex is tried.
- When the nth vertex is visited we have to check, is there any path from nth vertex to first vertex. If no path, the go back one step and after the previous visited node.
- Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:

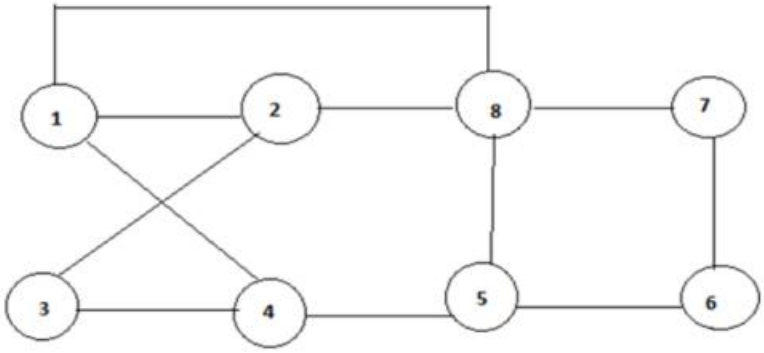
```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k);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

- ✓ This algorithm uses the recursive formulated of backtracking to find all the Hamiltonian cycles of a graph.
- ✓ The graph is stored as an adjacency matrix g[1: n, 1: n].
- ✓ In the next value k, x [1: k-1] is a path with k-1 distinct vertices.
- ✓ if x[k] == 0 then no vertex has to be yet been assign to x[k].
- ✓ After execution x[k] is assigned to the next highest numbered vertex which does not already appear in the path x [1: k-1] and is connected by an edge to x [k – 1] otherwise x[k] == 0.

✓ If k == n, (here n is the number of vertices) then in addition x[n] is connected to x [1].

Example:



Hamiltonian cycle 1 8 7 6 5 4 3 2 1