4. Backtracking

(*) General Method: Backtracking represents one of the most general searching technique. 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 backtracking.

In many applications of the backtrack method, the desired solution is expressed as an n-tuple $(x_1, x_2 \cdots x_n)$, where the x_i are chosen from some finite set S_i. The solution maximizes or minimizes or satisfies a criterion function $p(x_1, x_2 \cdots x_n)$ is the required solution

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. If the partial vector generated does not lead to an optimal solution, it can be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space tree for the given problem. Backtracking is a depth first search with some bounding function. All solutions using back-tracking are required to satisfy a complex set of constraints. the constraints may be explicit or implicit.

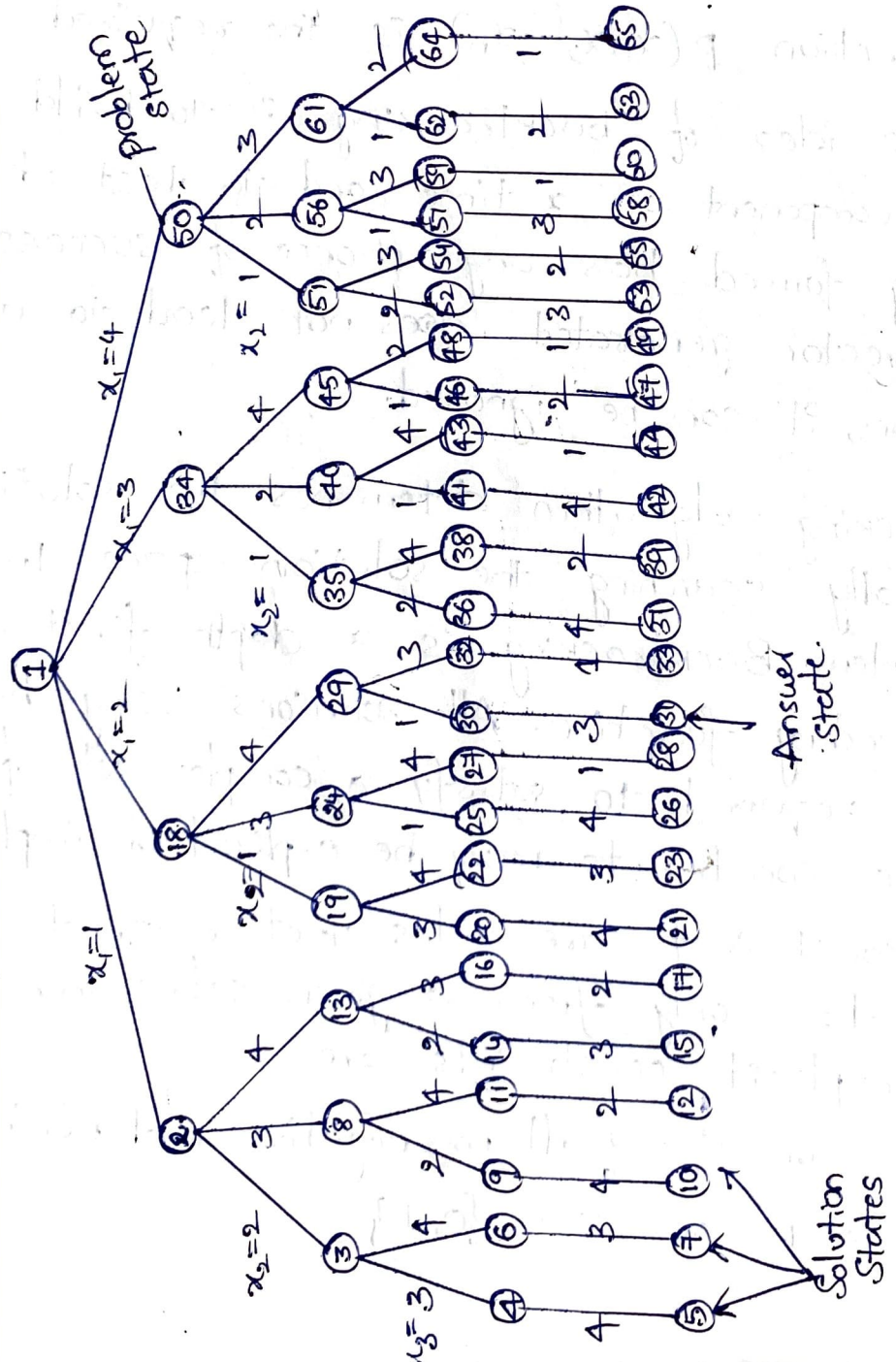
Explicit constrainst are rules that restrict each x_i to take values only from a given set. Common examples of explicit constraints are

$x_i \geq 0$ or $S_i = \{$all non negative real numbers$\}$

$x_i = 0$ or 1 or $S_i = \{0, 1\}$

The implicit constraints are rules that determine which of the tuples in the solution space tree satisfy the criterion function.

For example, Consider a 4-Queen's problem. It could be stated as 'there are 4-Queen's to be placed on 4×4 chessboard such that no two Queens can attack each other. Solution space tree for this problem is drawn as below.



All paths from root to other nodes define the state space of the problem.

Each node in the state space tree is called problem state.

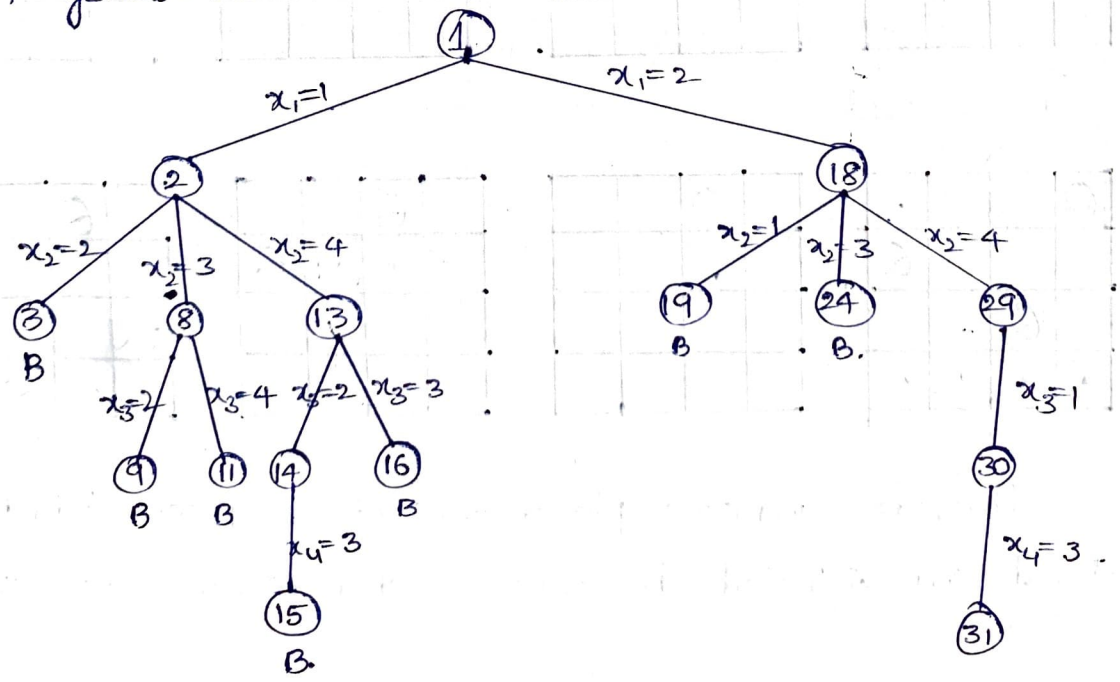
The solution states are the problem states S for which the path from root to S defines a tuple in the solution space.

The leaf nodes which correspond to an element in the set of solutions which satisfy the implicit constraints are called answer states.

A node which is generated and whose children have not yet been generated is called Live node.

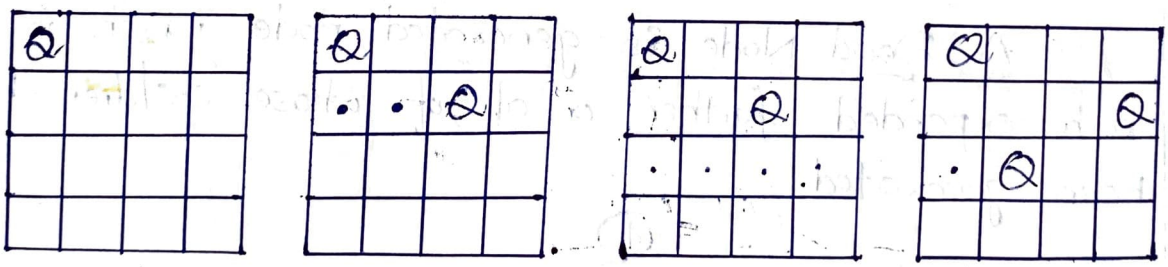
The Live Node whose children are currently being expanded is called E-node.

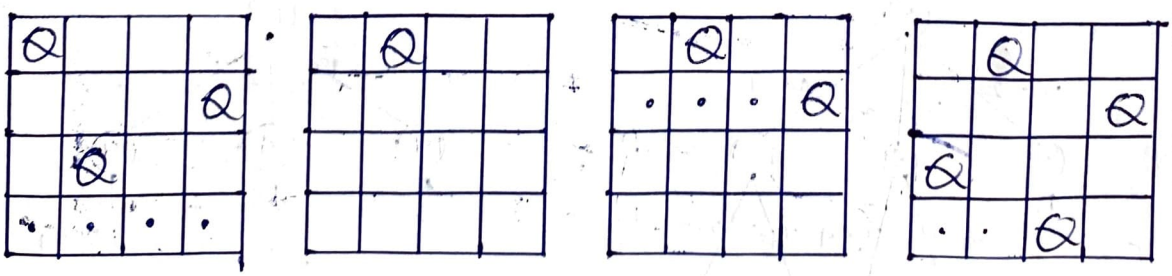
A Dead Node is generated node which is not to be expanded further or all of whose children have been generated.



the above tree represents portion of the state space that is generated during the backtracking.

Initially start with the root node as the only live node. This becomes the E-node and its one of the children (2) is generated. Node 2 becomes E-node. Node 3 is generated and immediately killed by applying bounding function. the next node generated is 8 and the path becomes (1,3). Node 8 becomes the E-node. However, it gets killed as its children cannot lead to an answer node. Now backtrack to node 2 and generate another child, node 13. The path is now (1,4) and so on. the following diagrams shows the board configurations as backtracking proceeds.





Here, dots indicate placements of a queen which were tried and rejected because another Queen was attacking.

(*) Recursive Backtracking Algorithm:

```
Algorithm Backtrack(k)
{
  for (each x[k] ∈ T(x[1]... x[k-1])) do
  {
    if (B_k(x[1], x[2]... x[k]) ≠ 0) then
    {
      if (x[1], x[2]... x[k] is a path to answer node)
          then write (x[1:k]);
      if (k<n) then Backtrack(k+1);
    }
  }
}
```

Iterative Backtracking Algorithm:

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

(*) Applications of Backtracking:

(1) The n-Queen's Problem: [8-Queen's Problem]

Consider a $n \times n$ chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. Let $(x_1, x_2 \cdots x_n)$ represent a solution in which x_i is the column of the ith row where ith queen is placed. The x_is will all be distinct since no two ~~columns~~ queens can be placed in the same column.

Imagine that the chessboard squares being numbered as the indices of two-dimensional array $a[1:n, 1:n]$, then 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, consider the queen at $a[4,2]$. the squares that are diagnal to this queen are $a[3,1]$, $a[5,3]$, $a[6,4]$, $a[7,5]$ and $a[8,6]$. All these squares have a row-column value of 2. 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 positions (i,j) and (k,l). then they are on the same diagonal only if

$$i-j = k-l \quad \text{or} \quad i+j = k+l$$
$$\Rightarrow i-k = j-l \quad \Rightarrow i-k = l-j$$

$\therefore$ Two queens lie on the same diagonal if

$$|j-l| = |i-k|.$$

Algorithm for N-Queen's Problem:

```
Algorithm NQueens(k,n)
{
  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(k,i)
// Returns true if a queen can be placed in kth row
// and ith column. Otherwise it returns false.
// Abs(r) returns the absolute value of r.
{
  for j:=1 to k-1 do
    if (x[j]:=i) or (Abs(x[j]-i) = Abs(j-k))
        then return false;
  return true;
}
```

② Sum of Subsets Problem: Let $S = \{S_1, S_2 \cdots S_n\}$ be n distinct positive numbers with $S_1 \leq S_2 \leq \cdots \leq S_n$. Then we have to find all combinations of these numbers whose sums are m. this is called the sum of subsets problem. In this case the element x_i of the solution vector is either one or zero depending on whether the weight $w_i(S_i)$ is included or not. the children of any node are easily generated. for a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

Let S be a set of elements and m is the expected sum of subsets. then.

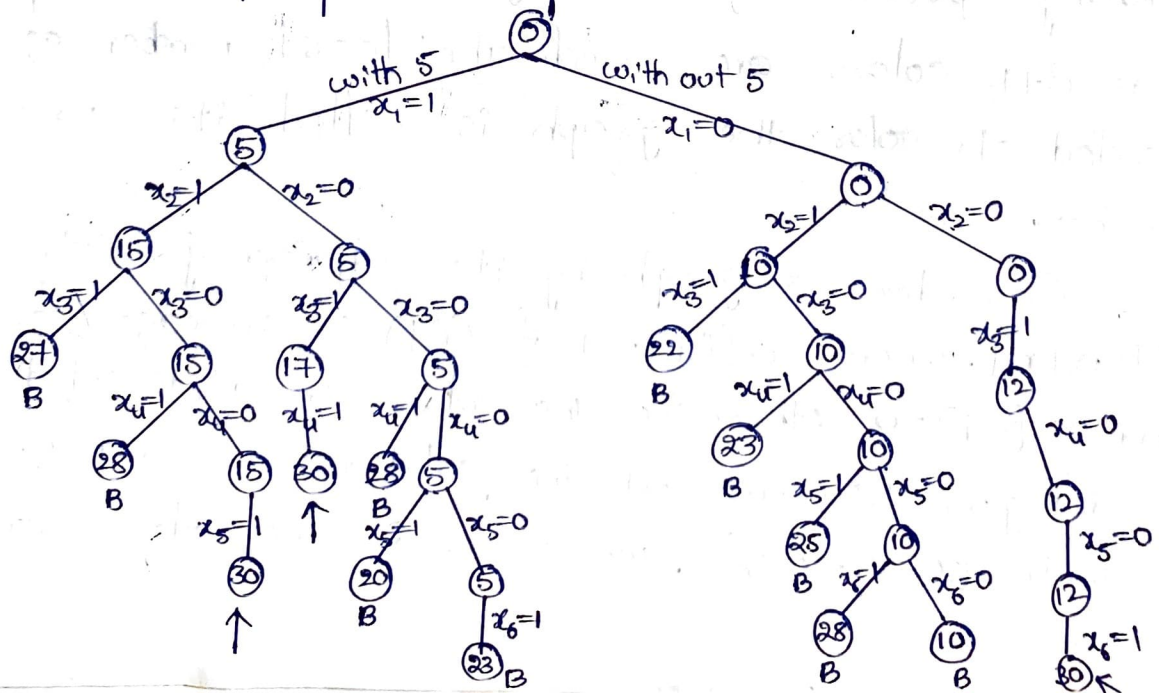
1. Start with an empty set.
2. Add next element from the list to the subset.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible, or if end of the set is reached then backtrack through the subset until finding the optimal solution.
5. If the subset is feasible then repeat step 2.
6. If all the elements are visited without finding a solution and if no backtracking is possible then stop without solution.

(*) Ex: Let $w = \{5, 10, 12, 13, 15, 18\}$ and $m = 30$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

Sol:

Initially Subset = { }	Sum = 0	Action
5	5	Add next element
5,10	15 ∵ 15<30	Add next element
5,10,12	27 ∵ 27<30	Add next element
5,10,12,13	40	Sum exceeds 30, hence backtrack.
5,10,12,15	42	Sum exceeds 30, hence backtrack
5,10,12,18	45	Sum exceeds 30, hence backtrack
5,10,13	28	∵ 28<30, add next element
5,10,13,15	43	Sum exceeds 30, hence backtrack.
5,10,13,18	46	Sum exceeds 30, hence backtrack
5,10,15	30	∵ sum = 30, Solution is obtained.

State space tree can be drawn as follows.



Algorithm for Sum of Subsets:

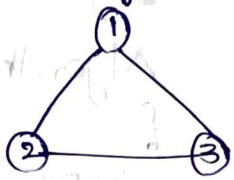
$r = \sum_{j=1}^{n} w_j$, $s=0$, $k=1$

```
Algorithm SumOfSub(s,k,r)
{
   x[k] := 1;
   if (s+w[k]=m) then write (x[1:k]);
   else if (s+w[k]+w[k+1] ≤ m)
      then SumOfSub(s+w[k], k+1, r-w[k]);
   if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
   {
      x[k] := 0;
      SumOfSub(s, k+1, r-w[k]);
   }
}//
```

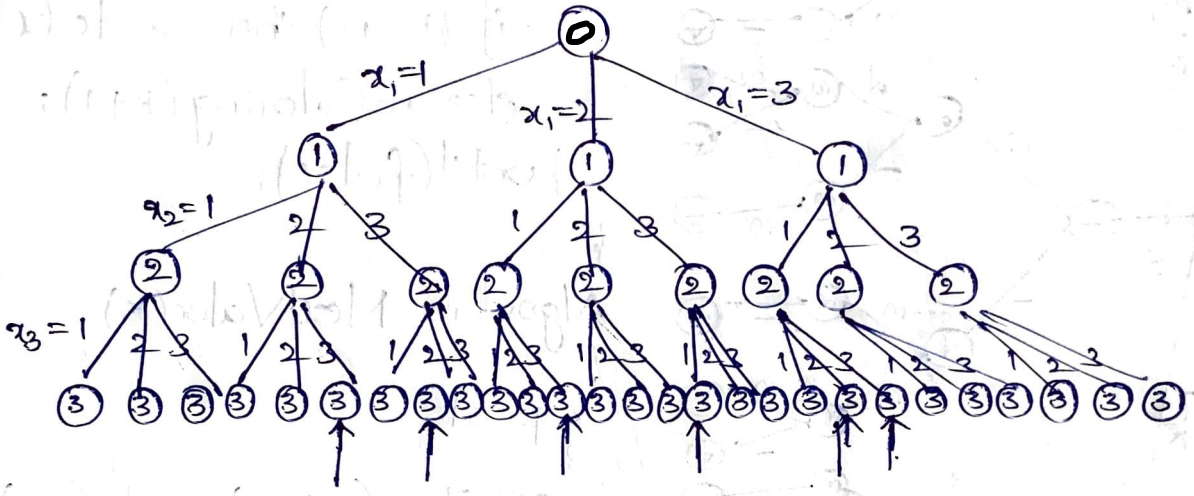
③ Graph Coloring Problem: Graph Coloring is a problem of coloring each vertex in graph G in such away that no two adjacent vertices have same color and yet m-colors are used. This problem is called m-coloring problem. If the degree of given graph is d then d+1 colors are used. the least number of colors needed to color the graph is called its chromatic number.

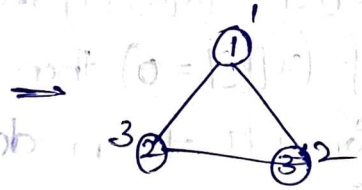
Consider a graph by its adjacency matrix G[1:n, 1:n], where G[i,j] = 1 if (i,j) is an edge of G and G[i,j] = 0 otherwise. the colors are represented by the integers 1, 2 ... m and the solutions are given by the n-tuple $(x_1, x_2 \cdots x_n)$, where x_i is the color of node i.

Example: Color the following graph using m-Coloring problem

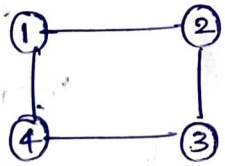


Sol: Since degree of the graph is 2, Hence 3 colors are required. State space tree for m-Coloring when n=3 and m=3 is as shown below.

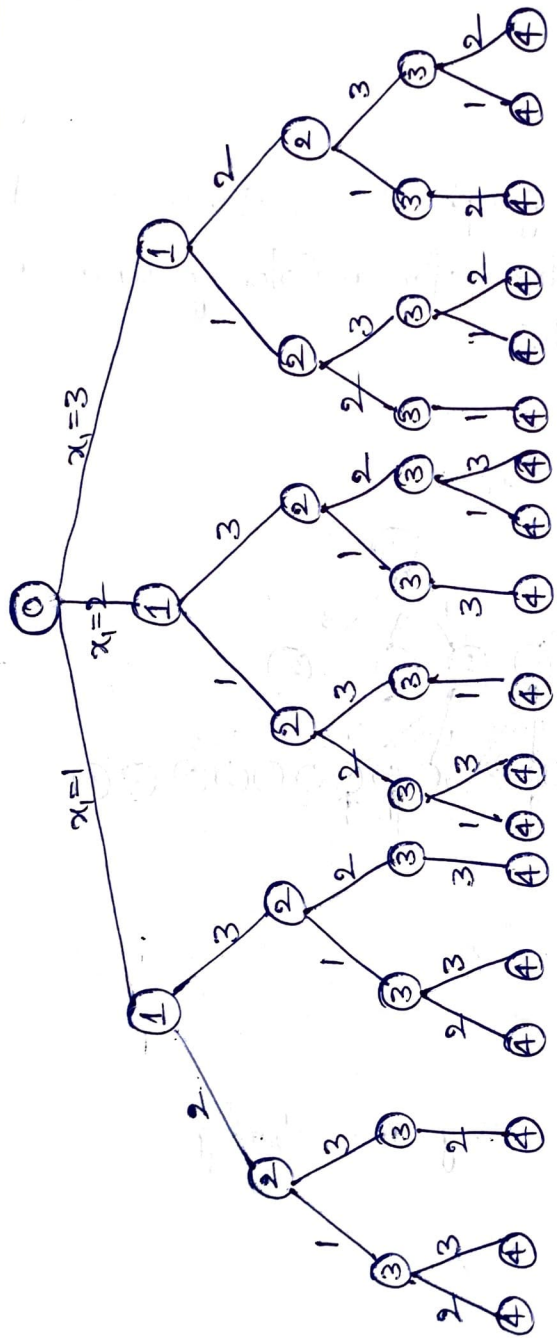




Ex: Color the following graph using m-Coloring



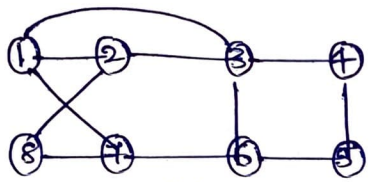
Sol: Since degree of the graph is 2. Hence 3 colors are required. State space tree for a 4-node graph and all possible 3-Colorings.



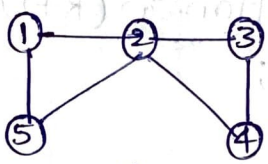
Algorithm for Graph Coloring

```
Algorithm mColoring(k)
{
  repeat
  {
    NextValue(k);
    if (x[k]=0) then return;
    if (k=n) then write (x[1:n]);
    else mColoring(k+1);
  } until(false);
}
Algorithm NextValue(k)
{
  repeat
  {
    x[k]:= (x[k]+1) mod (m+1);
    if (x[k]=0) then return;
    for j=1 to n do
    {
      if (G[k,j]≠0 and
          x[k] = x[j])
        then break;
    }
    if (j=n+1) then return;
  } until(false);
}
```

4. **Hamiltonian Cycles:** 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 returns to its starting position. For example, the following graph G1 contains the Hamiltonian Cycle 1,2,8,7,6,5,4,3,1. The graph G2 contains no Hamiltonian cycle.

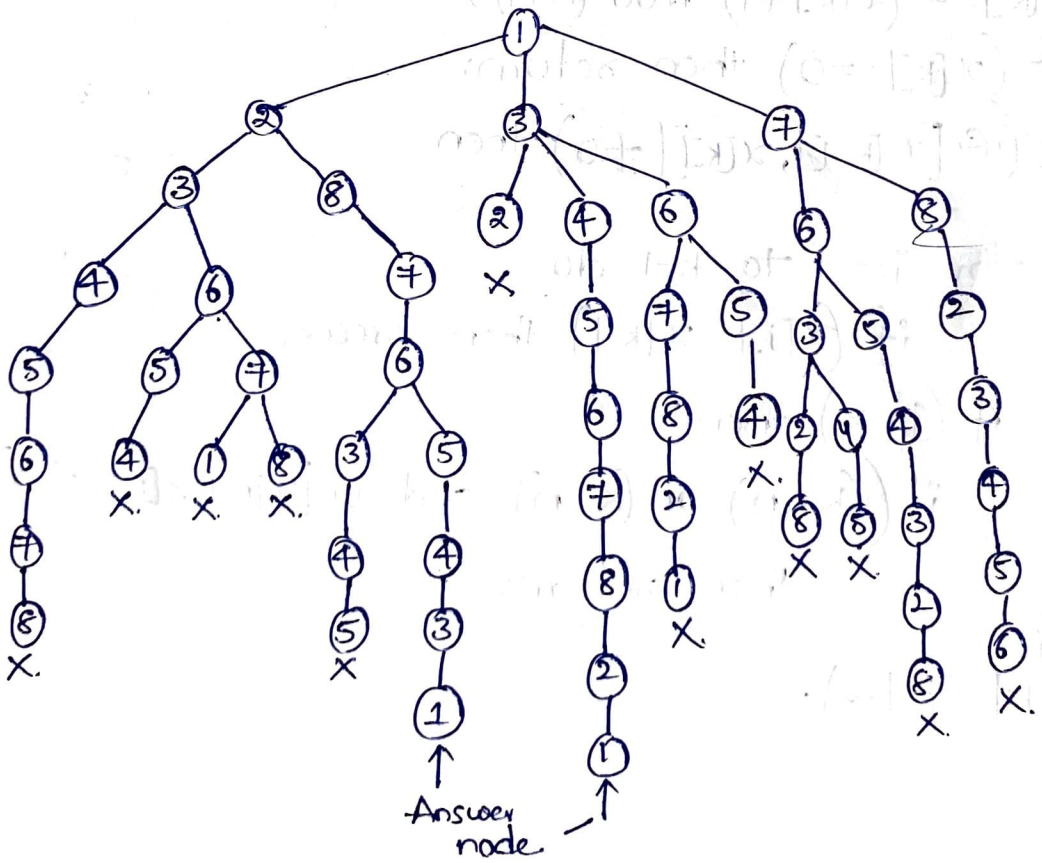


G1



G2

the graph may be directed or undirected, Only distinct cycles are output. State Space tree for G1 is ashown below:

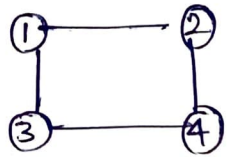


(*) Algorithm for Hamiltonian Cycle:

```
Algorithm Hamiltonian(k)
{
  repeat
  {
    NextValue(k);
    if (x[k]:=0) then return;
    if (k=n) then write (x[1:n]);
    else Hamiltonian(k+1);
  } until(false);
}
Algorithm NextValue(k)
{
  repeat
  {
    x[k]:= (x[k]+1) mod (n+1);
    if (x[k]:=0) then return;
    if (G[x[k-1], x[k]] ≠0) then
    {
      for j:= 1 to k-1 do
          if (x[j]= x[k]) then break;
      if (j=k) then
        if((k<n) or (k=n) and G[x[n], x[1]]≠0)
              then return;
    }
  } until (false);
}//
```

Frequently Asked Questions.

(1) Explain in detail about Backtracking?
(2) Write the control abstraction of Backtracking?
(3) Explain the general Backtracking process using recursion?
(4) Define the following terms: problem state, solution state, state space tree, answer states.
(5) Suggest a solution for 8-Queen's problem?
(6) Describe the 4-Queen's problem using Backtracking?
(7) Draw and explain the portion of the tree for 4-Queens problem that is generated during Backtracking?
(8) Describe the 8-Queen's problem?
(9) Write Backtracking algorithm for 8-Queens?
(10) Explain about n-Queens problem?
(11) Write an algorithm for n-Queens problem?
(12) Write a recursive Backtracking algorithm for Sum of Subsets?
(13) Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Do this using Sum of Subset Draw the portion of the state space tree that is generated.
(14) Give an example of Sum of Subsets?
(15) Explain about the sum of Subsets problem?
(16) Explain about Graph Coloring problem and chromatic Number?
(17) Write an algorithm for Graph Coloring problem? (or)
(18) Device a backtracking algm for m-coloring problem?
(19) for the below graph draw the portion of state space tree generated by MCOLORING.



(20) Give the state space tree for 3-coloring problem?

(21) Explain how the Hamiltonian circuit problem is solved by using the backtracking concept.

(22) Write an algorithm for generating all Hamiltonian Cycles.

(23) Find the Hamiltonian circuit in the following graph by using Backtracking.