MODULE 5

Basic Search and Traversal Techniques

Definition 1

: Traversal of a binary tree involves examining every node in the tree.

Definition 2

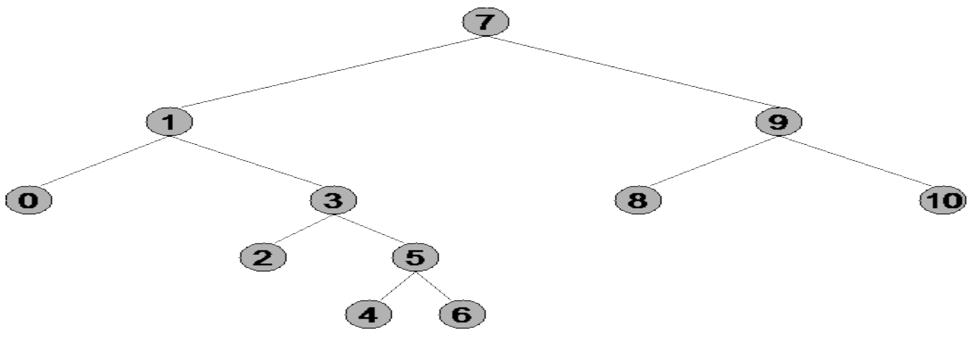
Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes.

•Different nodes of a graph may be visited, possibly more than once, during traversal or search

•If search results into a visit to all the vertices, it is called traversal

Binary tree traversal: Preorder, Inorder, and Postorder

in order to illustrate few of the binary tree traversals, let us consider the below binary tree:



Preorder traversal

: To traverse a binary tree in Preorder, following operations are carried out

(i) Visit the root, (ii) Traverse the left subtree, and (iii) Traverse the right subtree.

Therefore, the Preorder traversal of the above tree will outputs:

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

```
Algorithm
Algorithm preorder(t)
//t is a binary tree. Each node of t has
//three fields: lchild, data, and rchild.
{if t!=0 then
{Vist(t);
preorder(t->lchild);
preorder(t->rchild);
}}
```

Inorder traversal

: To traverse a binary tree in Inorder, following operations are carriedout

(i) Traverse the left most subtree starting at the left external node,

(ii) Visit the root, and

(iii) Traverse the right subtree starting at the left external node.

Therefore, the Inorder traversal of the above tree will outputs:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```
Algorithm:
Algorithm inorder(t)
//t is a binary tree. Each node of tHas
//three fields: lchild, data, and rchild.
{
if t!=0 then
{postorder(t->lchild);
Visit(t);
postorder(t->rchild);
}}
```

Postorder traversal

: To traverse a binary tree in Postorder, following operations arecarriedout

(i) Traverse all the left external nodes starting with the left most subtree which is then followed

by bubbleup all the internal nodes,

(ii) Traverse the right subtree starting at the left external node which is then followed by bubble up allthe internal nodes, and

(iii) Visit the root.

Therefore, the Postorder traversal of the above tree will outputs:

0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

```
1.  Algorithm:
2.  Algorithm postorder(t)
3.  //t is a binary tree. Each node of t has
4.  //three fields: lchild, data,
5.  and rchild.
6.  {if t!=0 then
7.  {postorder(t->lchild);
8.  postorder(t->rchild);
9.  Visit(t);
10. }
11. }
```

Graph Traversal

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

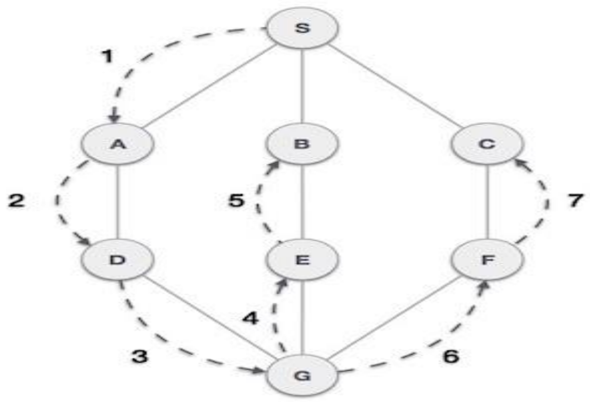
During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

1.DepthFirst Traversal

- Depth first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph.
- One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.
- Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children.
- Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.
-

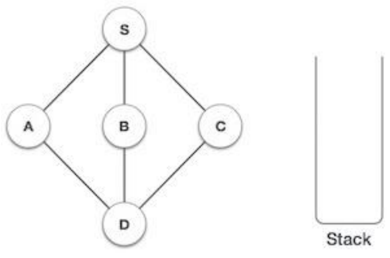
```
1. Algorithm DFS(v)
2. //Given an undirected graph G=(V,E) with n vertices and an array
   visited[] initially set to zero,this algorithm visits all vertices
   reachable from v.G and visite [] are global
3. {
4. Visited[v]=1
5. For each vertex w adjacent from v do
6. {
7. If(visited[w]=0) then DFS (w);
8. }}
```

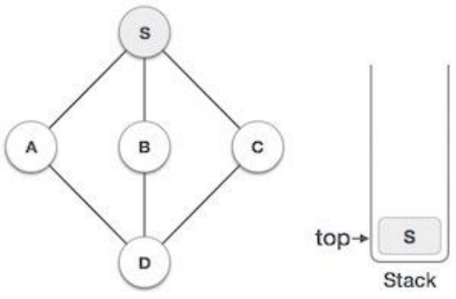
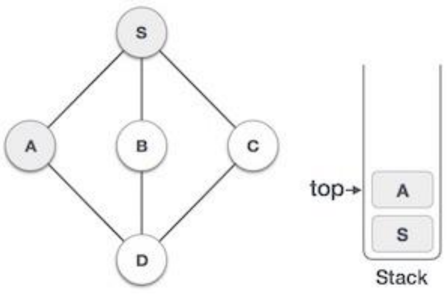
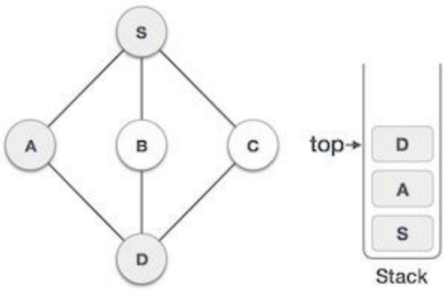
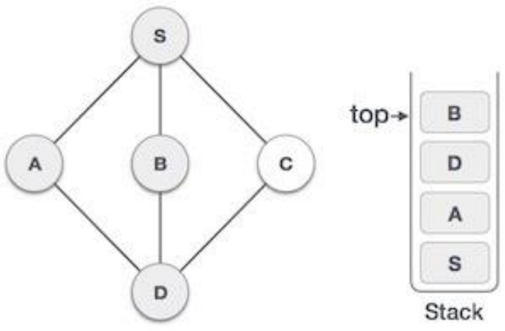
Example:

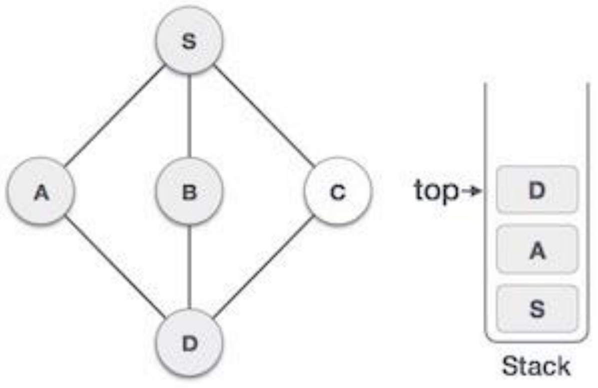
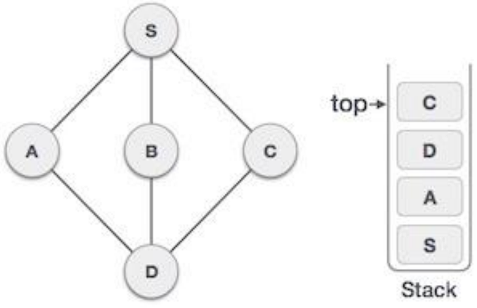


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack.

6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

The time and space analysis of DFS differs according to its application area. In theoreticalcomputer science, DFS is typically used to traverse an entire graph, and takes time O(|V|+|E|)linear in the size of the graph

2.Breadth first search and traversal

- BFS is an uninformed search method that aims to expand an d examine all nodes of a Graph or combination of sequences by systematically searching through every solution.
- In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.
- From the standpoint of the algorithm , all child nodes obtained by expanding a node are added to a FIFO(i.e., First In, First Out) queue.
- In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed"

Space complexity:

Since all of the nodes of a level must be saved until their child nodes in the next level have been

generated, the space complexity is proportional to the number of nodes at the deepest level. Given a branching factorband graph depth dthe asymptotic space complexity is the number of nodes at the deepest level, O(b^d). When the number of vertices in the graph is known ahead

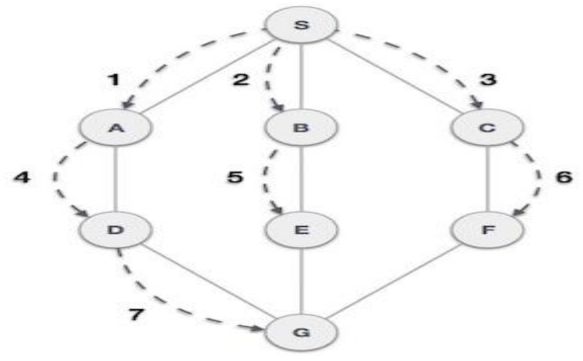
of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can also be expressed as O(| V|)where | V|is the cardinalityof the set of vertices. In the worst case the graph has a depth of 1 and all vertices

must be stored.

Time complexity Since in the worst case breadth-first search hasto consider all paths to all possible nodes the time complexity of breadth-first search is

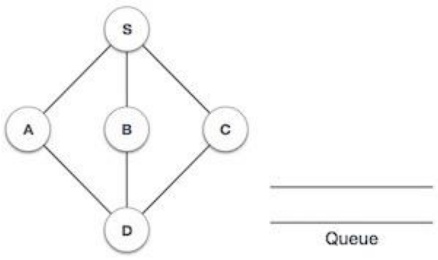
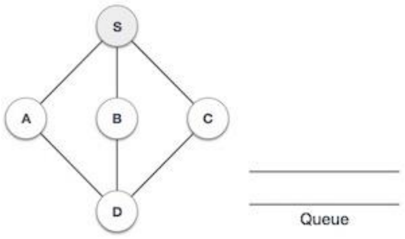
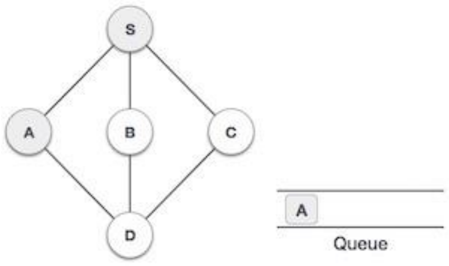
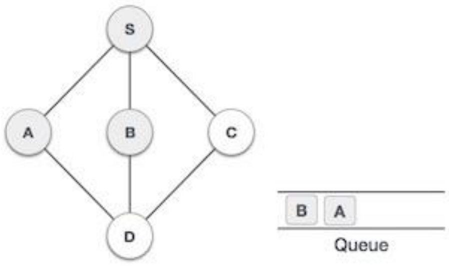
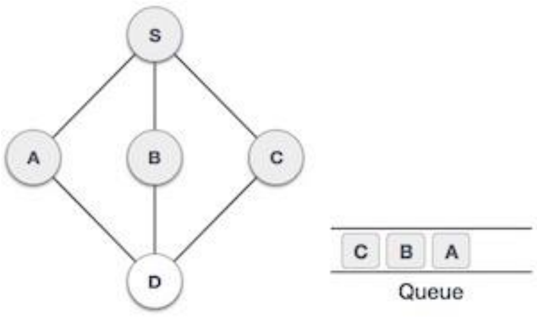
which is O(b^d). The time complexity can also beexpressed as O(| E| + | V|)since every vertex and every edge will be explored in the worst case.

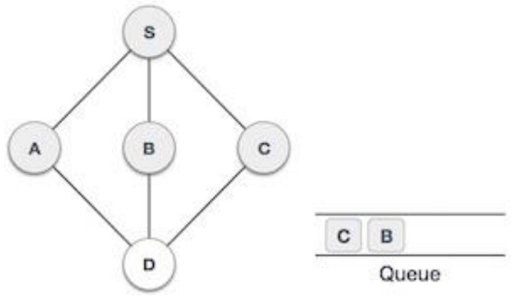
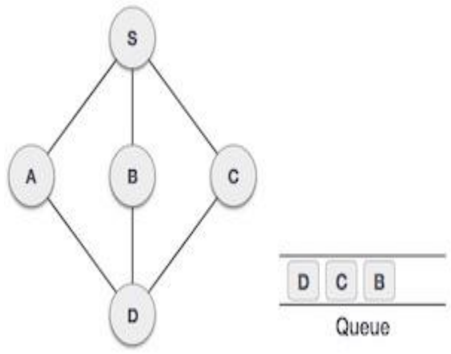
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.
-

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S**(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

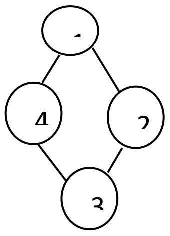
```
Algorithm BFS(v)
2.// Abredth first search of G is carried out beginning at vertex v for any node i,vistited[i]=1 if i has already been visited.The graph G and array visited[] are global; visited [] is initialized to zero
{ u=v;// q is a queue of unexplored vertices
  Visited [v]=1;
  Repeat
  {
    For all vertices w adjacent from u do
    { if (visited [w])=0 then
      {
        Add w to q;// w is unexplored
        Visited[w]=1;
    }}
    If q is empty then return;// No unexplored vertex
    Delete the next element u from q;
    //get first unexplored vertex
  }until(false);
}
```

BI-CONNECTED COMPONENTS AND DFS

- A vertex v in a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

A graph is said to be Biconnected if:

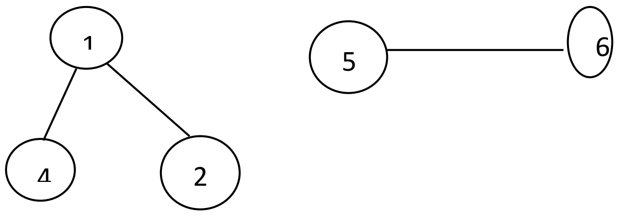
- It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.
- For example, consider the graph in the following figure
-



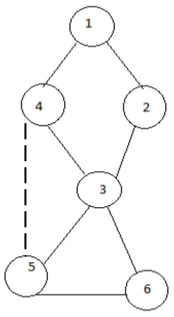
AtriculationPoint:In a graph if there is any vertex whose removal divide the graph into multiple components,then that vertex is called articulation point.

In the above diagram vertex3 is thearticulation point.

- Removal of vertex 3 make the graph

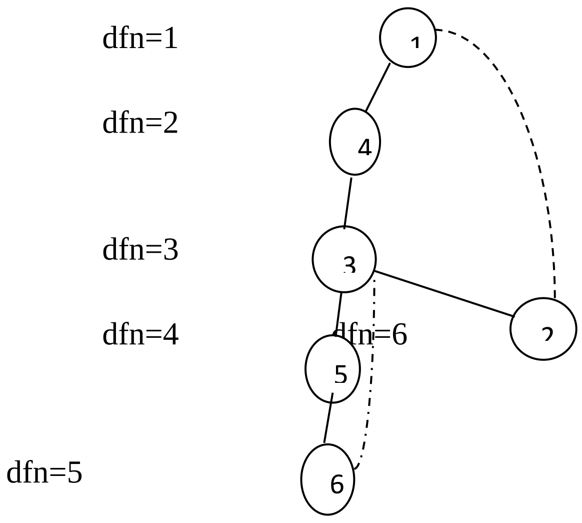


- A graph G is Biconnected if and only if it contains no articulation point
- How to remove attriculation point?
  -----Connect the components with an edje



*Procedure for finding articulation point

- Step1:Prepare a DFS spanning tree for the graph



- Step 2:Discover Depth first search number of each vertex in the order they are visited

  It is represented as dfn(discovery time)

Vertex	1	2	3	4	5	6
Dfn	1	6	3	2	4	5

- Step 3: find out the lowest discovery number from any of the vertex by one back edje
- Findout the value of L(a path that is going back to the parent)

Vertex	1	2	3	4	5	6
L	1	1	1	1	3	3

- L value of each vertex
- From vertex 1:1-4-3-2-1 L=1(source vertex 1 and destination vertex 1 and dfn of 1 ie parent is 1)
- From vertex2, path :2-1 L=1(source vertex 2 and destination vertex 1 and dfn of 1 ie parent is 1)
- Vertex 3 3-2-1 L=1 (source vertex 3 and destination vertex 1 and dfn of 1 ie parent is 1)
- Vertex 4 4-3-2-1 (source vertex 4 and destination vertex 1 and dfn of 1 ie parent is 1)
- Vertex 5 5-6-3 L=3 (source vertex 5 and destination vertex 3 and dfn of 3 ie parent is 3)
- Vertex 6,6-3 L=3 (source vertex 6 and destination vertex 3 and dfn of 3 ie parent is 3)
- The value of L is the dfn number of parent vertex.

Step 4:find articulation point

Consider 2vertices u---parent,v—child

L[v]>=d[u] then u will be the articulation point

- This will not work for root

U	V	L[v]>=dfn[u]
Vertex 3	Vertex 4	L[4]>=dfn[3]
		1>=3 not true
Vertex 5	Vertex 3	L[5]>=dfn[3]
		3>=3 true so vertex 3 is the attriculation point

Pseudocode Art carries out a depth first search of G

During this search each newly visited vertex gets assigned its depth first number.At the same time ,L[i] is computed for each vertex.

```
1. Algorithm  Art(u,v)
2. \\u is the start vertex for depth first search,v is its parent if any in
   the depth first spanning tree.It is assumed athat the global array
   dfn is initialized to zero and that the global variable num is
   initialized to 1.n is the number of vertices in G
3. { dfn[u]=num;L[u]=num;num=num+1;
4. For each vertex w adjacent from u do
5. { if (dfn[w]=0) then
6. { Art(w,u);\\ wis unvisited
7. L[u]=min(L[u],L[w]);
8. }
9. Else if(w!=v) then L[u]=min(L[u],dfn[w])}}
```

```
1.  Algorithm Bicomp(u,v)
2.  \\u is a start vertex for dept first search.visits parent if any in the depth
    firstspanningtree.It is assumed that the global arraydfn is initially zero and that
    the global variable num is initialized to 1.n is the number of vertices in G.
3.  { dfn [u]=num,L[u]=num;num=num+1;
4.  For each vertex w adjacent from u do
5.  {if((v!=w) and (dfn[w]<dfn[u]))then
6.  Add(u,w) to the top of a stack s;
7.  If(dfn[w]=0)then
8.  {if(L[w]>=dfn[u])then
9.  {write("new bicomponent")
10. Repeat
11. {deleteanedje from the top of stack s.let this edge be(x,y);
12. Write(x,y);
13. }until((x,y)=(u,w))or ((x,y)=(w,u));}
14. Bicomp(w,u);//w is unvisited.
15. L[u]=min(L[u],L[w]);
16. }
17. Else if (w!=v) then L[u]=min(L[u],dfn[w]);
18. }}
```

Backtracking Definition

Backtracking is a process where steps are taken towards the final solution and the details are recorded. If these steps do not lead to a solution some or all of them may have to be retraced and the relevant details discarded. In theses circumstances it is often necessary to search through a large number of possible situations in search of feasible solutions.

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem P and data D corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by C.

A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set S. **S = {}**

1. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree s in the search tree of the algorithm.
2. Check if S+s satisfies each of the constraints in C.
   - If Yes, then the sub-tree s is “eligible” to add more “children”.
   - Else, the entire sub-tree s is useless, so recurs back to step 1 using argument S.
3. In the event of “eligibility” of the newly formed sub-tree s, recurs back to step 1, using argument S+s.
4. If the check for S+s returns that it is a solution for the entire data D. Output and terminate the program.
   If not, then return that no solution is possible with the current s and hence discard it.

Applications of Backtracking:
 N Queens Problem
 Sum of subsets problem
 Graph coloring
 Hamiltonian cycles.

1. General method

•Useful technique for optimizing search under some constraints
• Express the desired solution as an n-tuple (x1, . . . ,xn) where each xi 2 Si, Si being a finite set
• The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P(x1, . . . , xn)
• **Sorting an array** a[n] – Find an n-tuple where the element xi is the index of ith smallest element in a – Criterion function is given by a[xi] _ a[xi+1] for 1 _ i< n – Set Si is a finite set of integers in the range [1,n]
• **Brute force approach** – Let the size of set Si be mi – There are m = m1m2 · · ·mn n-tuples that satisfy the criterion function P – In brute force algorithm, you have to form all the m n-tuples to determine the optimal solutions
• **Backtrack approach** – Requires less than m trials to determine the solution – Form a solution (partial vector) and check at every step if this has any chance of success – If the

solution at any point seems not-promising, ignore it – If the partial vector (x1, x2, . . . , xi) does not yield an optimal solution, ignore mi+1 · ··mn possible test vectors even without looking at them

2. Solution space and tree organization

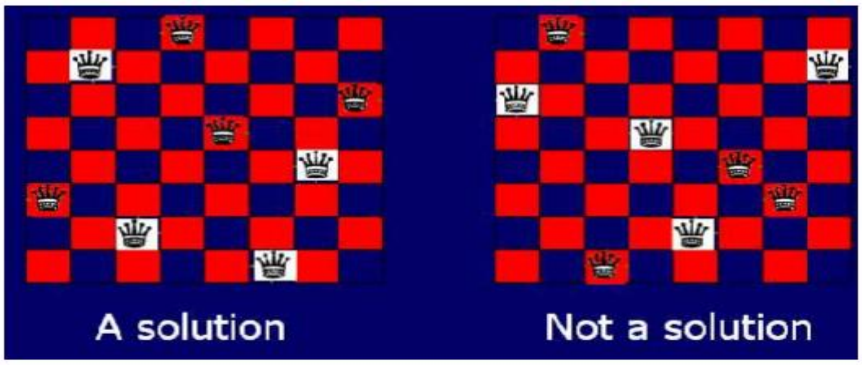
State-space search methods in problem solving have often been illustrated using tree diagrams. We explore a set of issues related to coordination in collaborative problem solving and design
and we present a variety of interactive features for state-space search trees intended to facilitate such activity.
A node in the state space tree is promising if it corresponds to the partials constructed solution that may lead to the complete solution otherwise the nodes are called non-promising. Leaves of the tree represent either the non- promising dead end or complete solution found by the algorithm.
— The tree organization of the solution space is state space tree
— Each node in the state space tree defines problem state
— Solution state are those problem states s for which the path from the root to s defines a tuple in the solution space — Answer state are those solution states s for which the path from root to s defines a tuple that is a member of solutions (it satisfies the implicit constraints)
— Solution space is partitioned into disjoint sub-solution space at each internal node
— Static vs. dynamic tree Static trees are independent of the problem instance Dynamic trees are dependent of the problem instance
— A node which has been generated and all of whose children have not yet been generated is called a live node
— The live node whose children are currently being generated is called E-node
— A dead node is a generated node which is not to be expanded further or all of whose children have been generate

Recursive backtracking algorithm

```
1.  Algorithm Backtrack(k)
2.  //This schema describes the backtracking process using recursion.Onentering,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
3.  { for(each x[k] €T(x[1]...x[k-1])) do
4.  {
5.  If (Bk(x[1],x[2]..x[k])!=0) then
6.  {
7.  If (x[1],x[2]...x[k] is a path to an answer node)
8.  Then write (x[1:k]);
9.  If (k<n) then Backtracking(k+1);
10. }}}
```

<u>The Eight Queens problem</u>

Place 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal.



Formulation :

> States: any arrangement of 0 to 8 queens on the board

> Initial state: 0 queens on the board

> Successor function: add a queen in any square

> Goal test: 8 queens on the board, none attacked Ideaofsolution:

• Each recursive call attempts to place a queen in a specific column – A loop is used, since there are 8 squares in the column

• For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)

• Current step backtracking: If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column

• Previous step backtracking: When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)

• Pruning: If a queen cannot be placed into column i, do not even try to place one onto column i+1 – rather, backtrack to column i-1 and move the queen that had been placed there

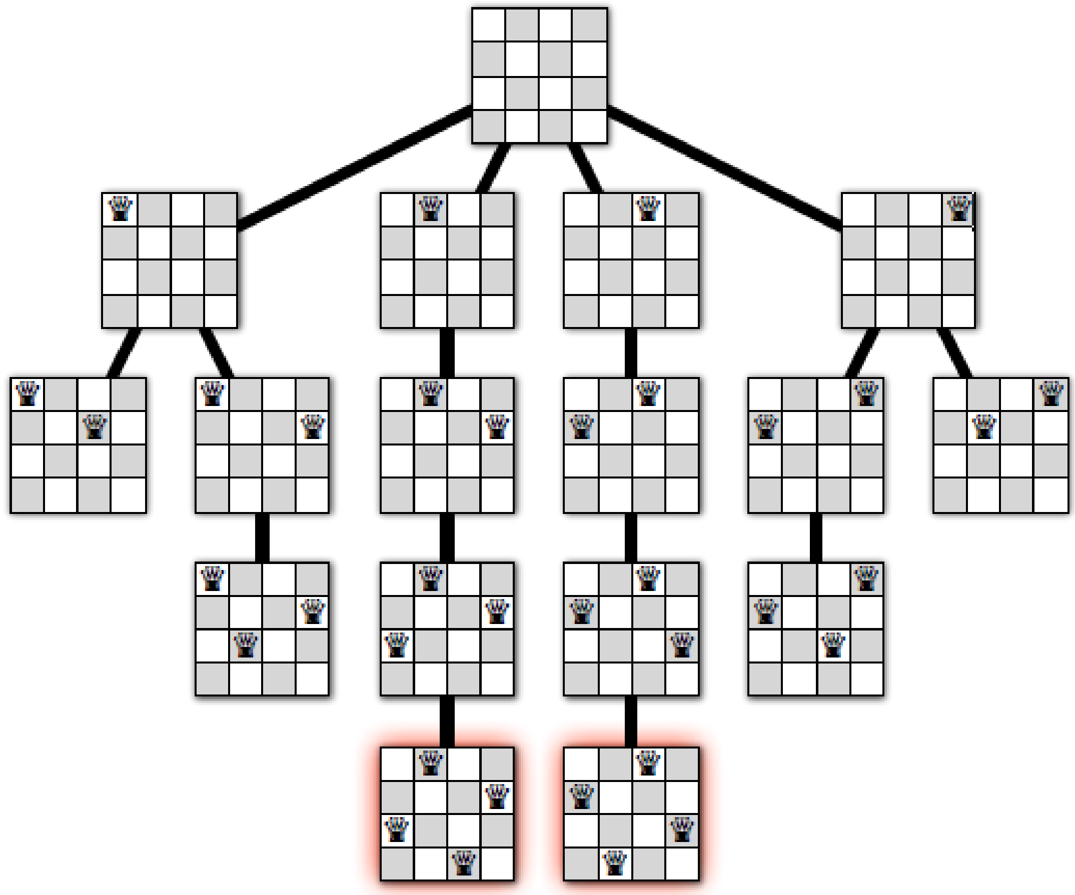
• Using this approach we can reduce the number of potential solutions even more.

All solutions to the n·queens problem

```
Algorithm void NQueens(int k, int n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an nXn
// 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 for new queen be placed

```
Algorithm Place(int k, inti)
// 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);
}
```


The complete recursion tree for our algorithm for the 4 queens problem.

SUM OF SUBSETS

- Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K.
- We are considering the set contains non-negative values.
- It is assumed that the input set is unique (no duplicates are presented).
- In general all solution are **k-tuples (x1, x2, x3---xk) 1 ≤ k ≤ n**, different solutions may have different sized tuples.
- Explicit **constraints(conditions) requires xi ∈ {j / j is an integer 1 ≤ j ≤ n }** Implicit constraints requires:
  No two be the same & that the sum of the corresponding wi's be m
  i.e., (1, 2, 4) & (1, 4, 2) represents the same.
  Another **constraint is xi < xi+1, 1 ≤ i ≤ k**
  - Wi--------- weight of item i
  - M---- Capacity of bag (subset)
  - Xi--------- the element of the solution vector is either one or zero.
  - Xi value depending on whether the weight wi is included or not.
  - If Xi=1 then wi is chosen.

- If Xi=0 then wi is not chosen
- $$\underbrace{\sum_{i=1}^{k} W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^{n} W(i) \geq M}_{\text{Still there}}$$
- The above equation specify that x1, x2, x3, --- xk cannot lead to an answer node if this condition is not satisfied.
- $$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$
- The equation cannot lead to solution.

$$B_k(X(1), \ldots, X(k)) = true \text{ iff } \left(\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M \text{ and } \sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M \right)$$

$$s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

EXAMPLE

Let, S = {S1 …. Sn} be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.It is always convenient to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤…. ≤ Sn

Execution of Algorithm:

Let, S is a set of elements and m is the expected sum of subsets. Then:

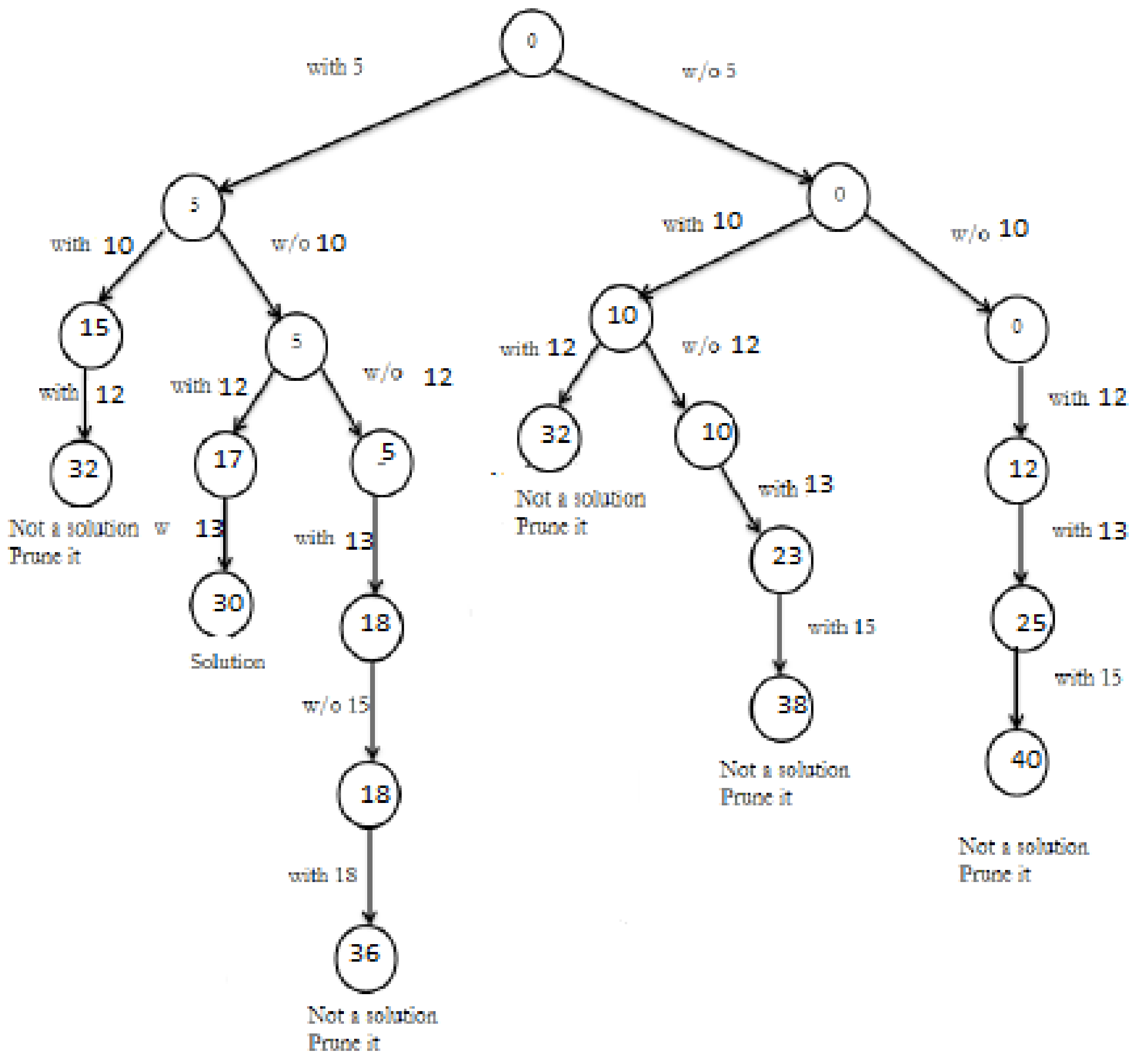
1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.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

Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18}

Solution:

Initially subset = {}	**Sum = 0**	**Description**
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

The state space tree is shown as below in figure. {5, 10, 12, 13, 15, 18}



Recursive backtracking algorithm for sum of subsets problem

```
1.  Algorithm SumOfSub(s, k, r)
2.  // Find all subsets of w[1:n] that sum to
    m.Thevaues of x[j], 1<-j<k,have already been
    determined.
    $// s = \sum_{j=1}^{k-1} W(j)X(j).$  and  $r = \sum_{j=k}^{n} W(j)$
3.  //The w[j]'s are in nondecresing order.it is assumed
    that w[1]<=m and  εw[i]>=m
4.  //Generate left child.Notes+w[k]<=m since Bk-1 is
    true.
5.  X[k]=1
6.  If(S+w[k]=m) then write(x[1: k]); // subset found.
7.  //There is no recursive call here as w[j]>0,1<=j<=n
8.  Else if (S+w[k] + w{k+1] ≤ M)
9.  Then SumOfSub(S+w[k], k+1, r-w[k]);
10. //Generate right child and evaluate Bk.
11. if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M) ) then
12. {
13. X[k]=0;
14. SumOfSub(S, k+1, r-w[k]);
15. }
16. }
```

Time complexity of sum of subsets problem using backtracking

If any **sum** of the numbers can be specified with at most P bits, then solving the**problem** approximately with c = 2^{-P} is equivalent to solving it exactly. Then, the polynomial **time** algorithm for approximate **subset sum** becomes an exact algorithm with running **time** polynomial in N and 2^{P} (i.e., exponential in P).

<u>GRAPH COLORING</u>

- ❖ The graph coloring problem is to discover whether the nodes of the graph G can be covered in such a way, that no two adjacent nodes have the same color yet only m colors are used.
- ❖ This graph coloring problem is also known as M-colorability decision problem.
- ❖ The M – colorability optimization problem deals with the smallest integer m for which the graph G can be colored.
- ❖ The integer is known as a chromatic number of the graph.
- ❖ Here, it can also be noticed that if d is the degree of the given graph, then it can be colored with d+ 1 color.
- ❖ A graph is also known to be planar if and only if it can be drawn in a planar in such a way that no two edges cross each other.

A special case is the 4 - colors problem for planar graphs.

- The problem is to color the region in a map in such a way that no two adjacent regions have the same color. Yet only four colors are needed.
- This is a problem for which graphs are very useful because a map can be easily transformed into a graph. Each region of the map becomes the node, and if two regions are adjacent, they are joined by an edge.
- Graph coloring problem can also be solved using a state space tree, whereby applying a backtracking method required results are obtained.

For solving the graph coloring problem,

we suppose that the graph is represented 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] = 0otherwise.

The colors are represented by the integers 1, 2, ..., m and the solutions are given by the

n-tuple (x1, x2, x3, ..., xn), where x1 is the color of node i.

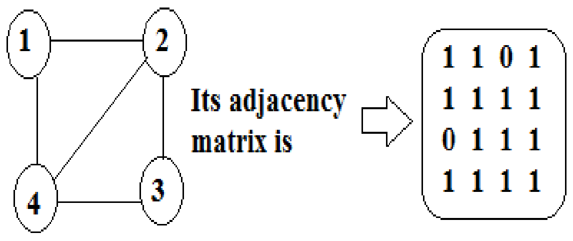
Finding all m-coloring of a graph

```
Algorithm mColoring(k){
// g(1:n, 1:n)□ boolean adjacency matrix.
// k□index (node) of the next vertex to color.
repeat{
    nextvalue(k); // assign to x[k] a legal color.
    if(x[k]=0) then return; // no new color possible
    if(k=n) then write(x[1: n];
    else mcoloring(k+1);
}
until(false)
}
```

Getting next color

```
Algorithm NextValue(k){
//x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m]
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
    {
        if ((g[k,j]≠0) and (x[k]=x[j]))
        then break;
    }
    if(j=n+1) then return; //new color found
} until(false)
}
```

Example



Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.

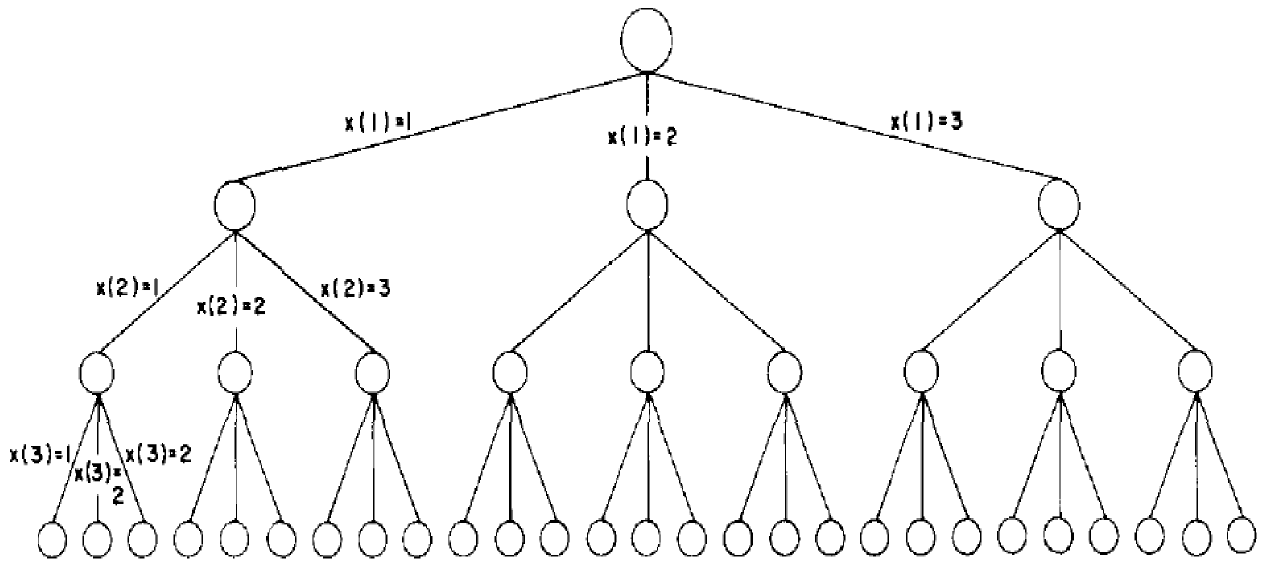
Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)

xi->Color of node i.

State Space Tree for

n=3->nodes

m=3->colors



State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways

2nd node coloured in 3-ways

3rd node coloured in 3-ways

So we can colour in the graph in 27 possibilities of colouring.

The total time required by the above algorithm is O (nm^n).

HAMILTONIAN CYCLES:

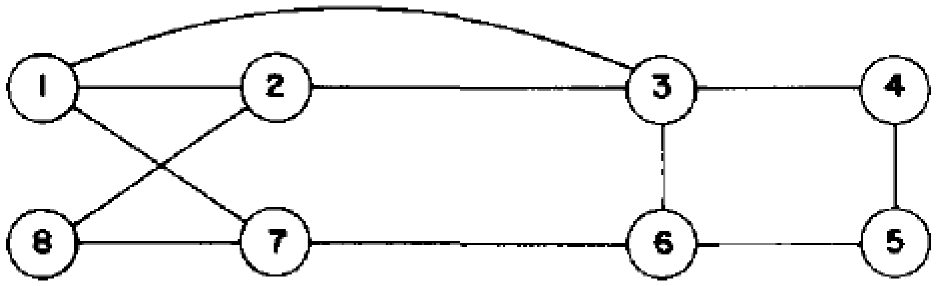
 Def: Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.

 It is also called the Hamiltonian circuit.

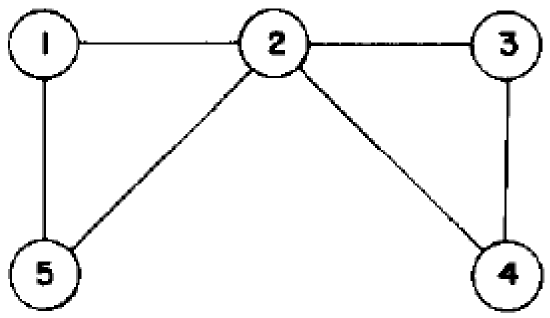
 Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.

 A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.

 In graph G, Hamiltonian cycle begins at some vertiex v1 ∈ G and the vertices of G are visited in the order v1,v2,---vn+1, then the edges (vi, vi+1) are in E, 1 ≤ i ≤ n.



The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

 There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.

 By using backtracking method, it can be possible

 Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.

 The graph may be directed or undirected. Only distinct cycles are output.

 From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}

 The backtracking solution vector (x1, x2, --- xn) xi->ith visited vertex of proposed cycle.

- By using backtracking we need to determine how to compute the set of possible vertices for xk if x1,x2,x3---xk-1 have already been chosen.
- If k=1 then x1 can be any of the n-vertices.
- By using “NextValue” algorithm the recursive backtracking scheme to find all Hamiltoman cycles.
- This algorithm is started by 1st initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Generating Next Vertex

```
Algorithm NextValue(k)
{
// x[1: k-1]□ is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
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
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
If(j=k) then //if true , then vertex is distinct
If((k<n) or (k=n) and G[x[n], x[1]]≠0))
Then return ;
}
}
Until (false);
}
```

Finding all Hamiltonian Cycles

```
Algorithm Hamiltonian(k)
{
Repeat{
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)
}
```

Time complexity of the above algorithm is O($2^n n^2$)