Design and Analysis of Algorithms

Unit – V

Dr. R. Bhuvaneswari
Assistant Professor
Department of Computer Science
Periyar Govt. Arts College, Cuddalore.

Periyar Govt. Arts College
Cuddalore

Syllabus

UNIT - V: TRAVERSAL, SEARCHING & BACKTRACKING

Techniques for Binary Trees- Techniques for Graphs - The General Method - The 8-Queens Problem – Sum of Subsets- Graph Colouring- Hamiltonian Cycles.

TEXT BOOK

Fundamentals of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Galgotia Publications, 2015.

Techniques of Binary Trees

- A tree whose nodes have at most 2 children is called a binary tree.
- A traversal is a process that visits all the nodes in the tree.
- Since a tree is a nonlinear data structure, there is no unique traversal.
- We consider several traversal algorithms which we group in the following two kinds
  - depth-first traversal
  - breadth-first traversal
- There are three different types of depth-first traversals:
  - **In-order Traversal**
  - **Pre-order Traversal**
  - **Post-order Traversal**
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

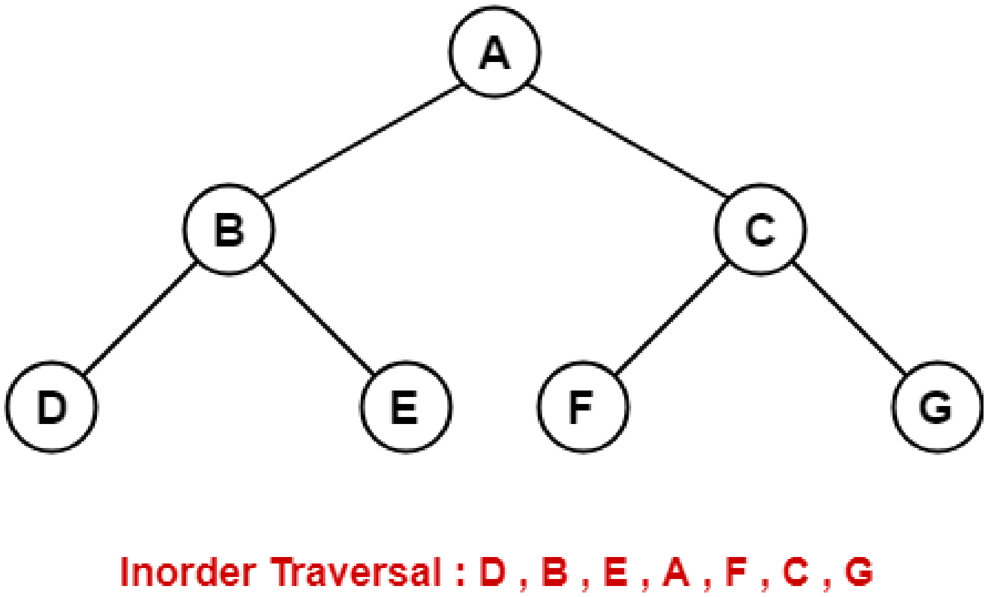
In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.
- To traverse a non-empty binary tree in in-order, we perform the following three operations:
  1. traverse the left sub-tree in in-order.
  2. visit the root.
  3. traverse the right sub-tree in in-order.
- A node of a binary tree is defined as

```
struct node
{
    char data;
    struct node *lchild, *rchild;
};
```

In-order Traversal

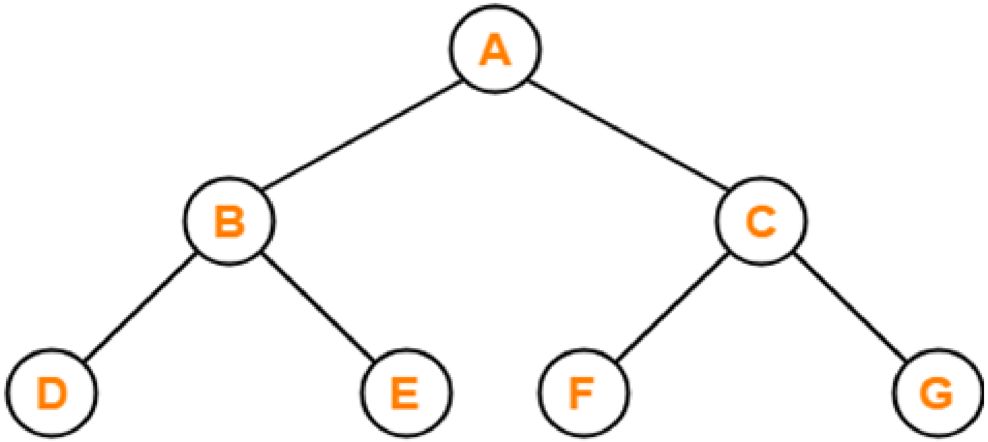
Algorithm Inorder(t)
{
  if t ≠ 0 then
  {
    Inorder(t→lchild);
    visit(t);
    Inorder(t→rchild);
  }
}



Pre-order Traversal

- To traverse a non-empty binary tree in pre-order, we perform the following three operations:
  1. visit the root.
  2. traverse the left sub-tree in pre-order.
  3. traverse the right sub-tree in pre-order.

Algorithm Preorder(t)
{
  if t $\neq$ 0 then
  {
    visit(t);
    Preorder(t$\rightarrow$lchild);
    Preorder(t$\rightarrow$rchild);
  }
}



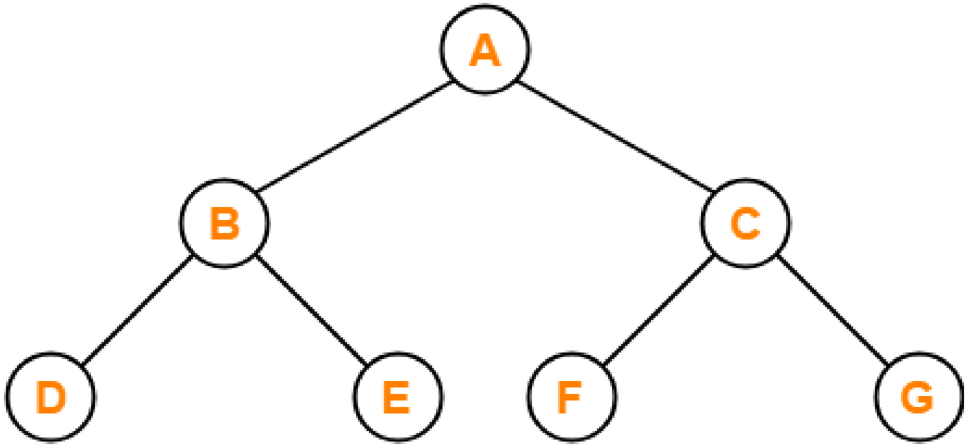
Preorder Traversal : A, B, D, E, C, F, G

Post-order Traversal

- To traverse a non-empty binary tree in post-order, we perform the following three operations:
    1. traverse the left sub-tree in post-order.
    2. traverse the right sub-tree in post-order.
    3. visit the root.

Algorithm Postorder(t)

```
{
  if t ≠ 0 then
  {
    Postorder(t→lchild);
    Postorder(t→rchild);
    visit(t);
  }
}
```



Postorder Traversal : D , E , B , F , G , C , A

Techniques for Graphs

- The graph is a non-linear data structure. It consists of some nodes and their connected edges. The edges may be directed or undirected.
- A Graph G is a pair (V, E), where V is a finite set of elements called vertices or nodes and E is a set of pairs of elements of V called edges or arcs.
- A graph in which every edge is directed is called **directed graph** or **digraph**.
- A graph in which edges are undirected are called **undirected graph**.
- A graph which contains parallel edges is called **multi-graph**.
- A graph which does not contain parallel edges are called **simple graph**.
- Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way.
- The graph has two types of traversal algorithms.
    - ➢ **Depth First Search or Traversal**
    - ➢ **Breadth First Search or Traversal**

Depth First Search (DFS)

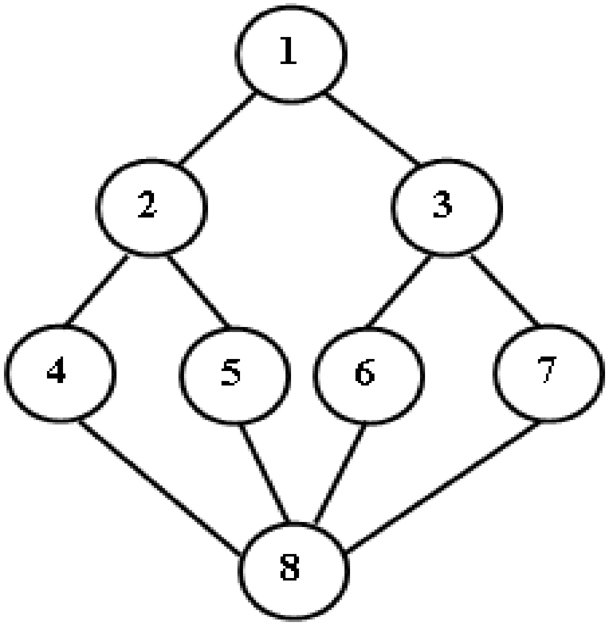
DFS follows the following rules:

1. Select an unvisited node s, visit it, and treat it as the current node.
2. Find an unvisited neighbor of the current node, visit it, and make it the current new node.
3. If the current node has no unvisited neighbors, backtrack to its parent and make that the new current node.

   Repeat the steps 2 and 3 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from step 1.

Depth First Search (DFS)

```
DFS(v)
{
    visited[v] = 1;
    for each vertex w adjacent from v do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```



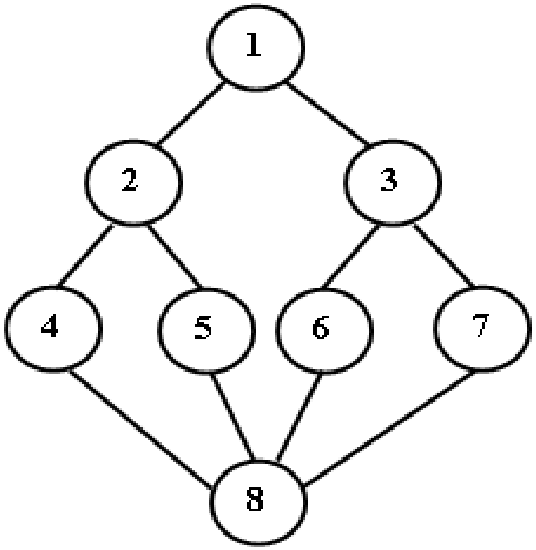
1, 2, 4, 8, 5, 6, 3, 7

Breadth First Search

- The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.
- In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
- After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.
- This method can be implemented using a queue.
- A Boolean array is used to ensure that a vertex is visited only once.
  - ✓ Add the starting vertex to the queue.
  - ✓ Repeat the following until the queue is empty.
    - ✓ Remove the vertex at the front of the queue, call it v.
    - ✓ Visit v.
    - ✓ Add the vertices adjust to v to the queue, that were never visited.

Breadth First Search

```
BFT(G, n)
{
    for i = 1 to n do
        visited[i] = 0;
    for i = 1 to n do
       if(visited[i] = 0) then BFS(i)
}
```



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

```
BFS(v)
// q is a queue of unexplored vertices
{
    u = v;
    visited[v] = 1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if(visited[w] = 0) then
            {
                add w to q;
                visited[w] = 1;
            }
        }
        if q is empty then return;
        delete u from q;
    } until(false);
}
```

Backtracking

- Backtracking is technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- The desired solution is expressed as an n-tuple (x_1, …, x_n), where x_i are chosen from some finite set S_i.
- The problem to be solved finds a vector that maximizes (or minimizes) a criterion function P(x_1, …, x_n).
- Suppose m_i is the size of set S_i. Then there are m = m_1, m_2, …., m_n n-tuples are possible candidates for satisfying the function P.
- If it is realized that the partial vector (x_1, x_2, …, x_n) can in no way lead to an optimal solution, then m_{i+1}, …, m_n possible test vectors can be ignored entirely.
- Problems solved through backtracking requires that all the solutions satisfy a complex set of constraints.
- Constraints are divided into two categories:
    - Implicit constraints
    - Explicit constrains

Backtracking

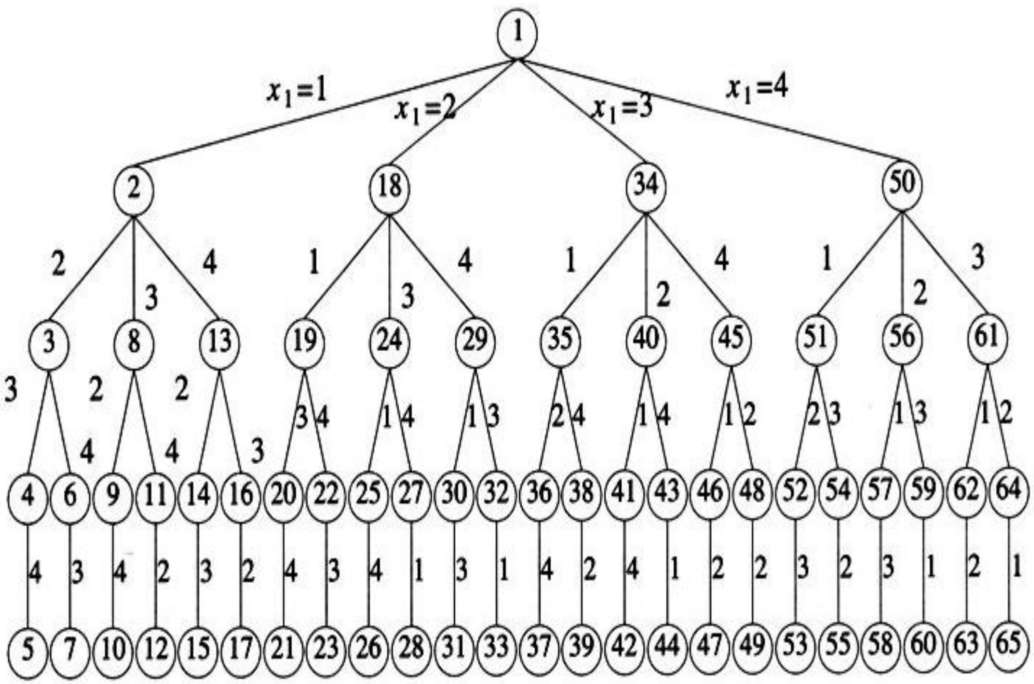
Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.

Eg. 4-queens problem

Explicit constraints – each queen on different row.

Implicit constraints – all queens must be on different columns and no two queens can be on the same diagonal.



Tree organization of 4-queens solution space

Backtracking

- Tuples that satisfy the explicit constraints define a **solution space.**
- The solution space can be organized into a tree.
- All paths from the root to other nodes define the **state-space** of the problem.
- **Live node** is a node which has been generated and all of whose children are not yet been generated .
- **E-Node** (Node being expanded) is the live node whose children are currently being generated .
- **Dead node** is a node that is either not to be expanded further, or for which all of its children have been generated.
- **Bounding function** will be used to kill live nodes without generating all their children.

Backtracking

```
Algorithm IBacktrack(n)
{
 k=1;
 while(k≠0) do
 {
  if(there remains an untired
      x[k] ∈ T(x[1], x[2], …, x[k-1]) and
      B_k(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;
   }
   else
        k=k-1;
 }
}
```

```
Algorithm Backtrack(k)
{
 for(each x[k] ∈ T(x[1], x[2], …,
    x[k-1]) do
 {
  if (B_k(x[1], …, 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);
  }
 }
}
```

8-Queens Problem

- n – queens are placed on a n x n chess board, which means that the chessboard has n rows and n columns and the n queens are placed on n x n chessboard such that no two queens are placed in the same row or in the same column or in same diagonal.
- All solutions to the **n – queen's problem** can be represented as n–tuples (x_1, x_2... x_n) where x_i is the column of the i^{th} row where **i^{th}** queen is placed.
- x_i's will all be distinct since no two queens can be placed in the same column.
- Consider queen at [4,2]. Diagonal to this queen are a[3,1]
- 2 queens are placed at positions (i, j) and (k, l).
- They are on the same diagonal only if

  i-j = k-l   e.g. 1-1 = 2-2

  or i+j = k+l   e.g 1+4= 2+3

  $\Rightarrow$ i-k = j-l
- Therefore 2 queens lie on the same diagonal if and only if **|j-l| = |i-k|**

1, 1	1, 2	1, 3	1, 4
2, 1	2, 2	2, 3	2, 4
3, 1	3, 2	3, 3	3, 4
4, 1	4, 2	4, 3	4, 4

8-Queens Problem

Q			
		Q	

Q			
			Q
	Q		

	Q		
			Q
Q			
		Q	

		Q	
Q			
			Q
	Q		

8-Queens Problem

```
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 n.
{
 for j = 1 to k-1 do
 {
  if((x[j] = i)) or (abs(x[j]-i) = abs(j-k))) then
            return false;
 }
 return true;
}
```

```
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);
  }
 }
}
```

Sum of Subsets

- Sum of Subsets problem is to find subset of elements that are selected from a given set whose sum adds up to a given number **m**.
- We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).
- Here backtracking approach is used for trying to select a valid subset.
- When an item is not valid, backtracking is done to get the previous subset and add another element to get the solution.

Finding all subsets of w_i, whose sum is m.

Ex. 1:

n = 4, $(w_1, w_2, w_3, w_4) = (11,13,24,7)$, m = 31

Possible subsets are {11, 13, 7} and {24, 7}

Ex. 2:

n = 7, w = {5, 10, 12, 13, 15, 18}, m = 30

Possible subsets are {5, 10, 15}, {5, 12, 13} and {12, 18}

Sum of Subsets

The bounding functions used are

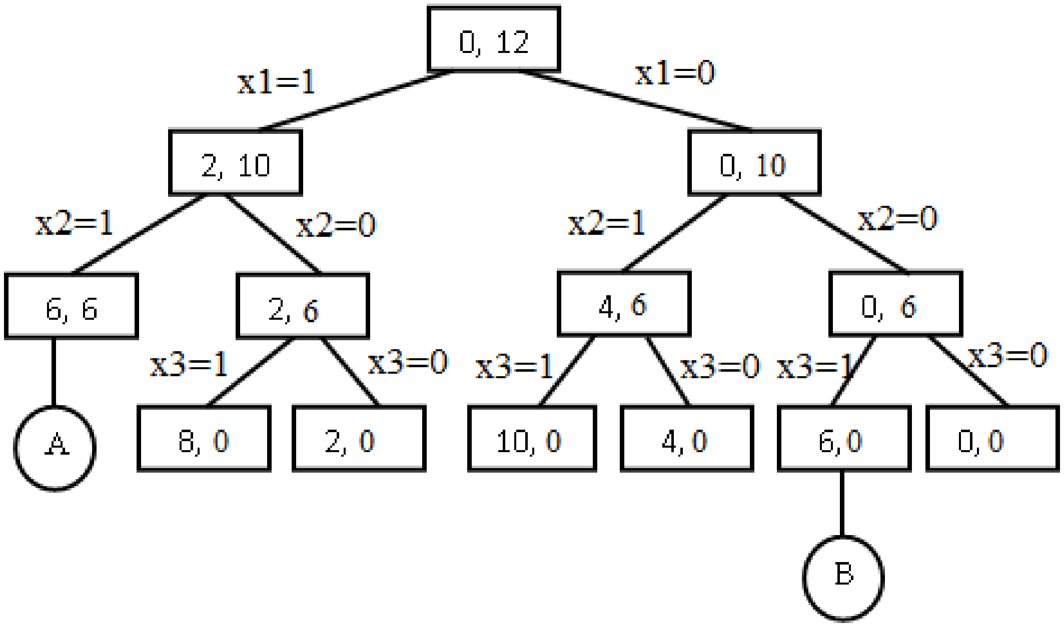
$$B_k(x_1, \dots, x_k) = \; true \; iff \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

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

Example:

n = 3, m = 6, w = {2, 4, 6}

The full space tree for n = 3 contains $2^3 - 1 = 7$ nodes from which call could be made (this excludes the leaf nodes).

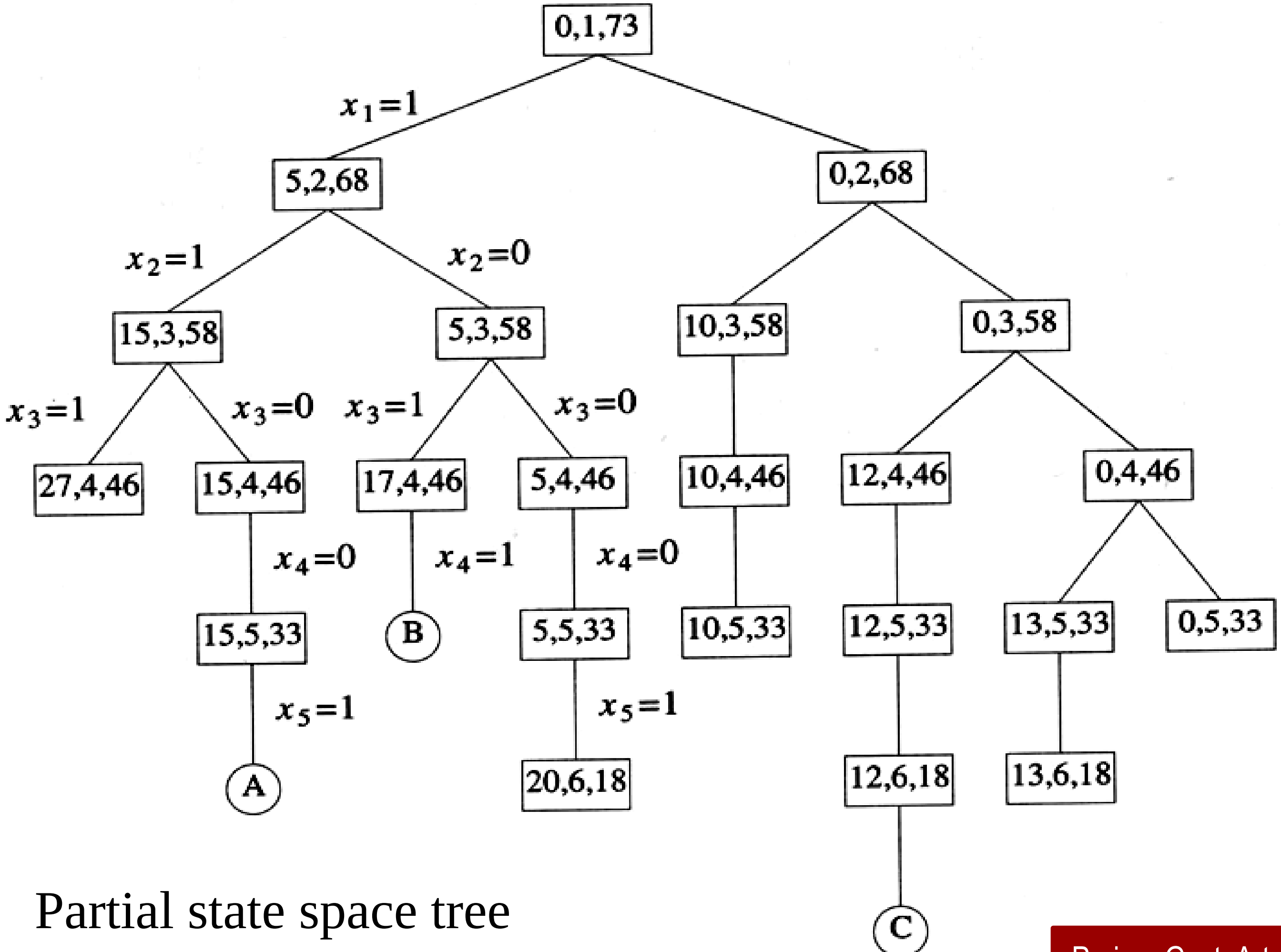


Sum of Subsets

```
Algorithm SumOfSubsets(s, k, r)
// S = Σw[j]*x[j] and r = Σw[j]. w[j]'s are in non decreasing order. It is
//assumed that w[1] ≤ m and Σw[j] ≥ m.
{
  x[k] = 1;
  if(s+w[k] = m) then write(x[1:k]);
  else if(s+w[k]+w[k+1] ≤ m) then
          SumOfSubset(s+w[k],k+1,r-w[k]);
  if((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m) then
  {
     x[k]=0;
     SumOfSubset(s,k+1,r-w[k]);
  }
}
```

Sum of Subsets

Example: n = 6, w[1:6] = {5,10,12,13,15,18}, m = 30



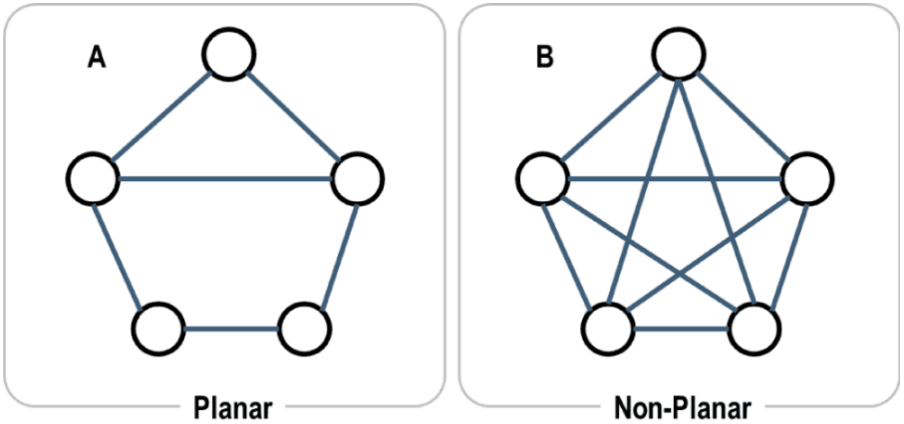
Partial state space tree

Graph Coloring

- Let **G** be a graph and **m** be a given positive integer.
- The graph coloring problem is to discover whether the nodes of the graph G can be colored 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 smallest number of colors required to color a graph G is referred to as the **chromatic number** of that graph.
- As the objective is to minimize the number of colors the graph coloring problem is also known as **m-colorability optimization** problem.
- Graph coloring problem is a **NP Complete** problem.
- If **d** is the degree of the given graph, then it can be colored with **d+1** colors.

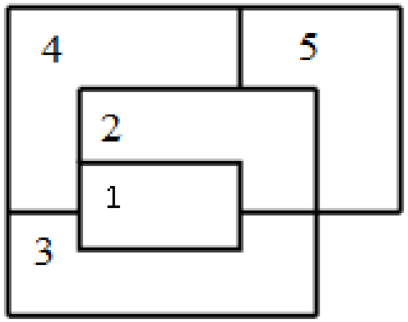
Graph Coloring

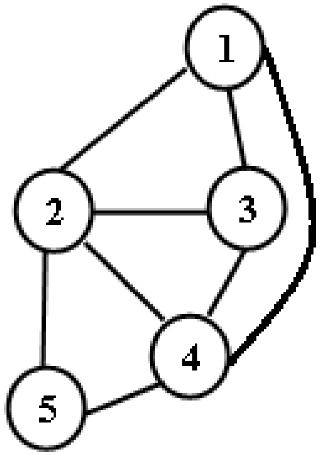
- A graph is said to be planar if and only if it can be drawn in a plane in such a way no two edges cross each other.



- 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.
- 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

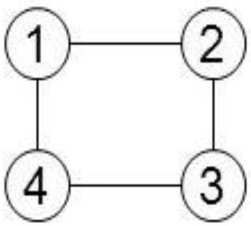




- For solving the graph coloring problem, we represent the graph by its adjacency matrix G[1:n, 1:n], where, G[i, j]= 1 if (i, j) is an edge of G, and G[i, j] = 0 otherwise.
- The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple (x_1, x_2, x_3, ..., x_n), where x_i is the color of node i.
- The total computing time of mcoloring is $O(nm^n)$.

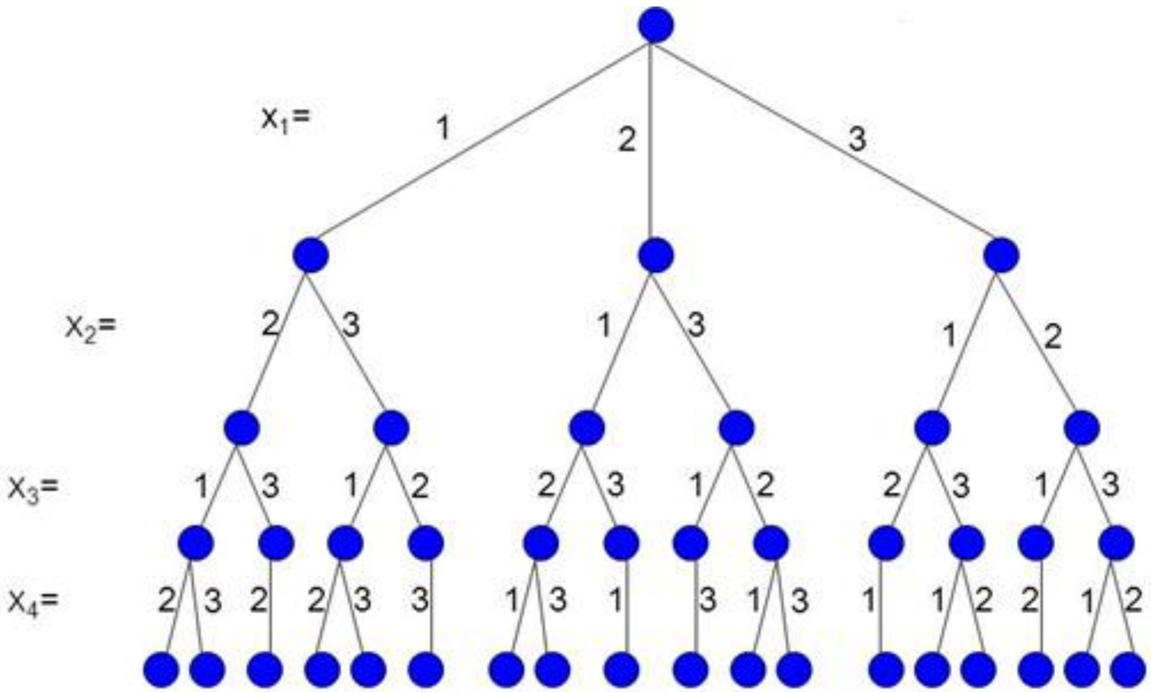
Graph Coloring

A 4-node graph and all possible 3-colorings



	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Adjacency matrix



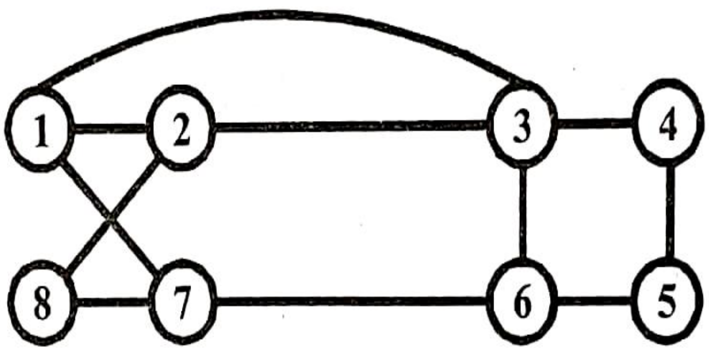
Graph Coloring

```
Algorithm mColoring(k)
//The graph is represented by its
//boolean adjacency matrix G[1:n,1:n].
{
 repeat
 {
   NextValue(k);
   if(x[k]=0) then return;
   if(k=n) then
      write(x[1:n]);
   else mColoring(k+1);
 }until false;
}
```

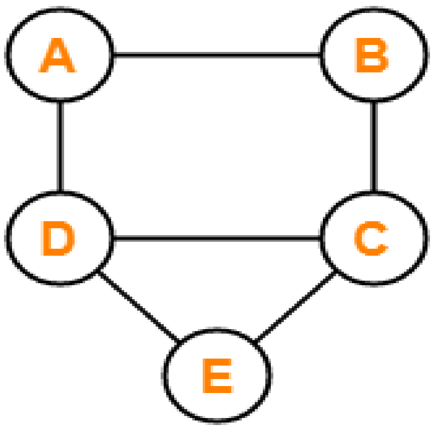
```
Algorithm NextValue(k)
{
 repeat
 {
   x[k] = (x[k]+1)mod(m+1);
   if(x[k]=0) then return;
   for j =1 to n do
   {
    if((G[k,j]≠0) and (x[k] = x[j])) then
          break;
   }
   if(j=n+1) then return;
}until(false);
}
```

Hamiltonian cycles

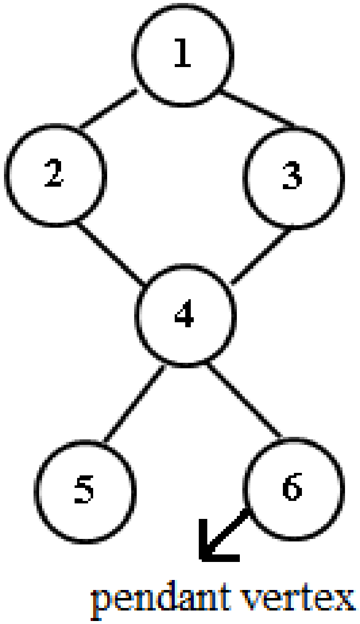
- Let G = (V, E) be a connected graph with n vertices.
- A Hamiltonian cycle is a round trip path along n edges of G that visits every vertex once and returns to its starting position.
- A graph that contains a Hamiltonian cycle is called a Hamiltonian graph.

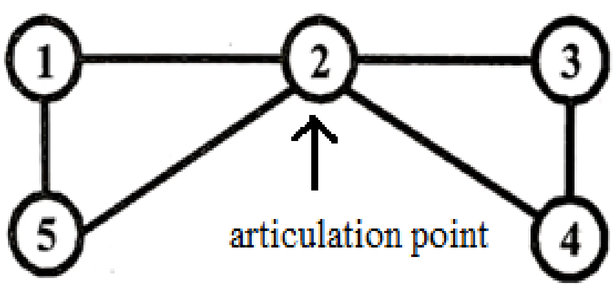


1, 2, 8, 7,6, 5, 4, 3, 1



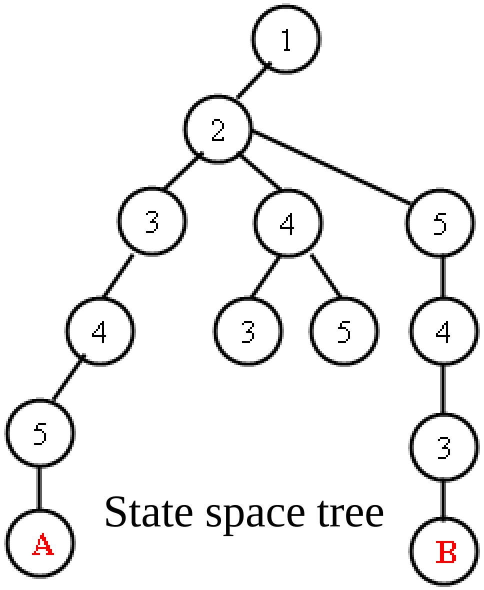
A, B, C, E, D, A
A, D, E, C, B, A





Hamiltonian cycles

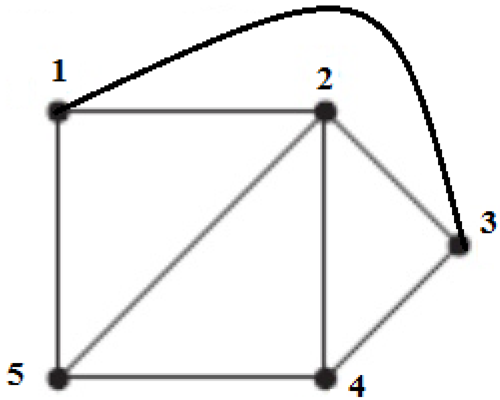
- The input for the Hamiltonian graph problem can be the directed or undirected graph. The Hamiltonian problem involves checking if the Hamiltonian cycle is present in a graph **G** or not.
- While generating the state space tree following bounding functions are to be considered, which are as follows:
  - The **i**$^{\mathbf{th}}$ vertex in the path must be adjacent to the **(i-1)**$^{\mathbf{th}}$ vertex in any path.
  - The starting vertex and the **(n-1)**$^{\mathbf{th}}$ vertex should be adjacent.
  - The ith vertex cannot be one of the first **(i-1)**$^{\mathbf{th}}$ vertex in the path.



State space tree

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Boolean adjacency matrix



Hamiltonian cycles

```
Algorithm Hamiltonian(k)
//The graph is stored as an
//adjacency matrix G[1:n,1:n].
{
  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);
}
```