Design and Analysis of Algorithms

Unit - IV

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

Periyar Govt. Arts College
Cuddalore

Backtracking

Syllabus

UNIT–IV

Backtracking: General Method – 8 Queens problem – sum of subsets – graph coloring – Hamiltonian problem – knapsack problem.

Text Book:

Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms C++, Second Edition, Universities Press, 2007. (For Units II to V)

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 $P(x_1, \ldots, x_n)$.
- Suppose m_i is the size of set S_i. Then there are $m = m_1, m_2, \ldots., m_n$ 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 $m_{i+1}, \ldots, m_n$ 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

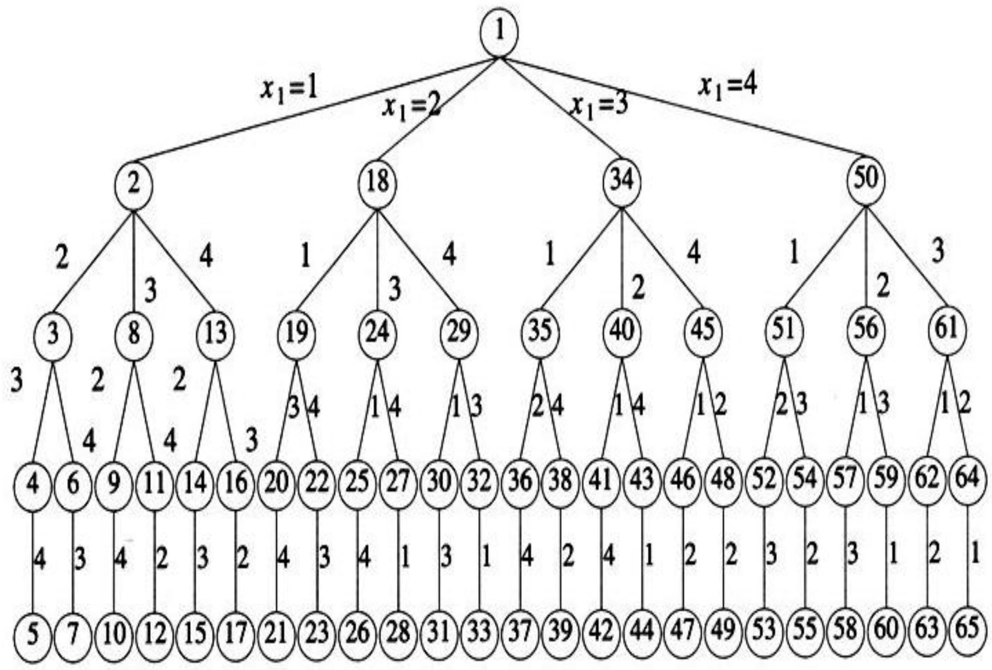
Explicit constraints are rules that restrict each x_i to take on values only from a given set.

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
      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;
  }
   else
       k=k-1;
 }
}
```

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

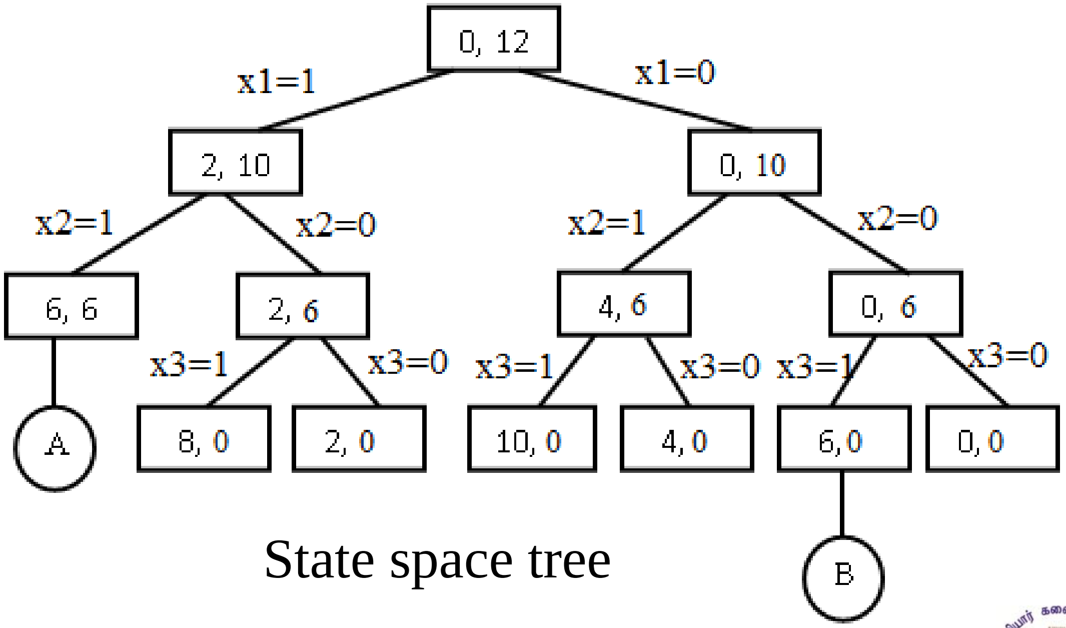
$$B_k(x_1, \dots, x_k) = \; true\; iff \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

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

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).



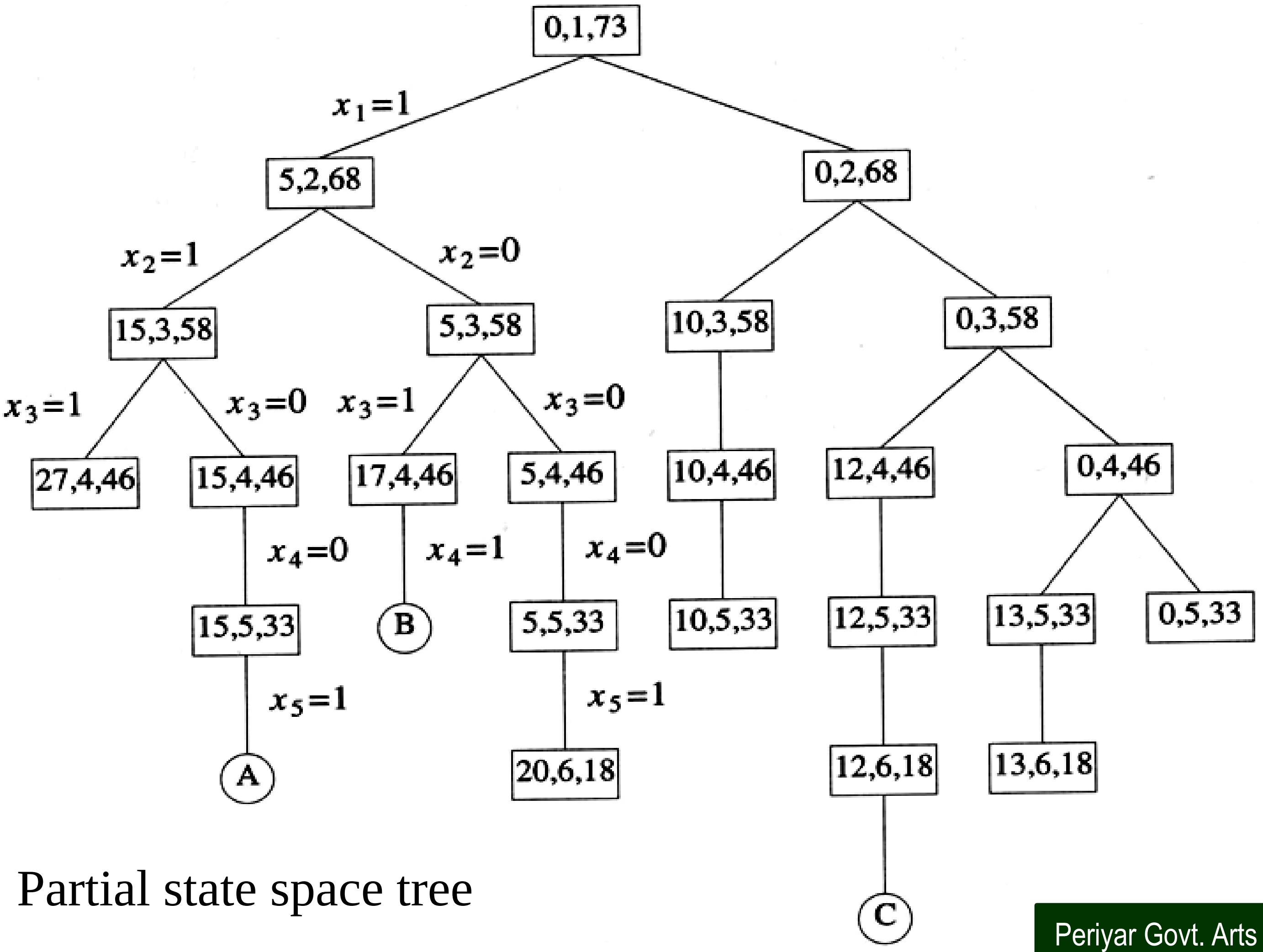
State space tree

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



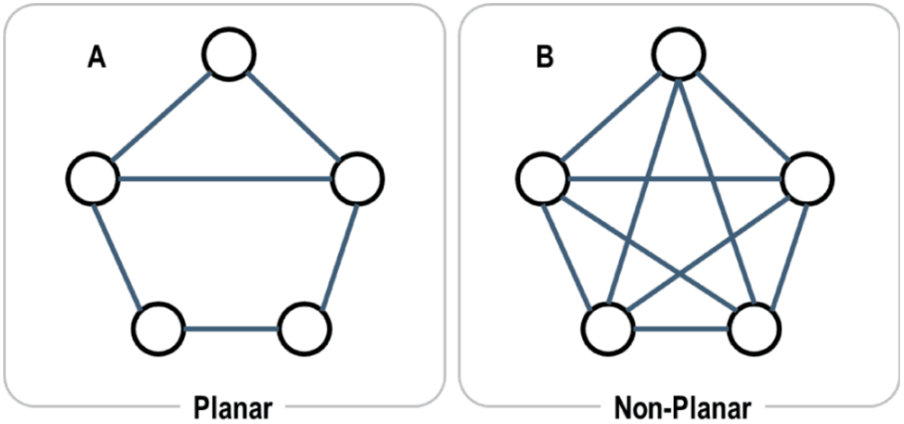
Partial state space tree

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.

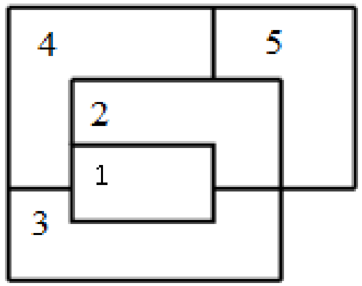
Graph Coloring

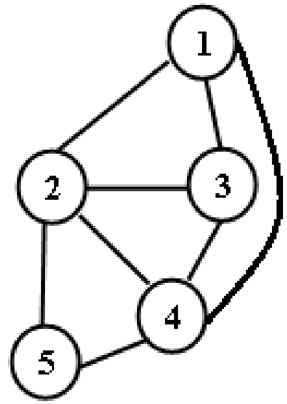
- A graph is said to be planar if and only if it can be drawn in a plane in such a way no two edges cross each other.



- 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

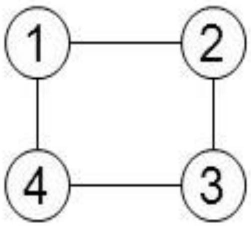


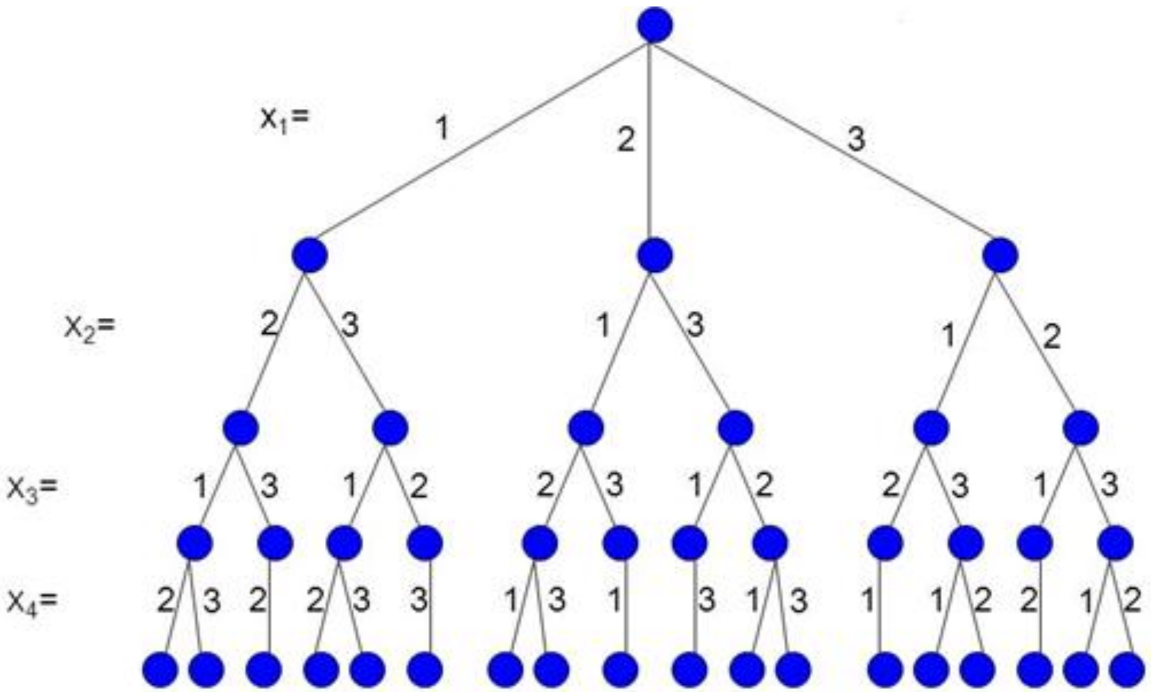


- For solving the graph coloring problem, we represent the graph by its adjacency matrix G[1:n, 1:n], where, G[i, j]= 1 if (i, j) is an edge of G, and G[i, j] = 0 otherwise.
- The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple (x_1, x_2, x_3, ..., x_n),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

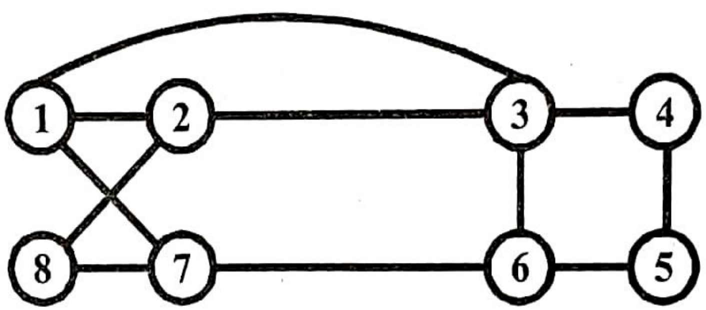
Graph Coloring

```
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;
}
```

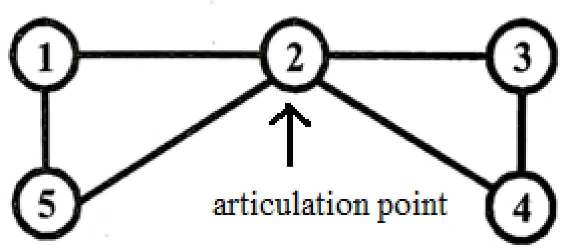
```
Algorithm NextValue(k)
{
  repeat
  {
    x[k] = (x[k]+1)mod(m+1);
    if(x[k]=0) then return;
    for j =1 to n do
    {
     if((G[k,j]≠0) and (x[k] = x[j])) then
           break;
    }
   if(j=n+1) then return;
}until(false);
}
```

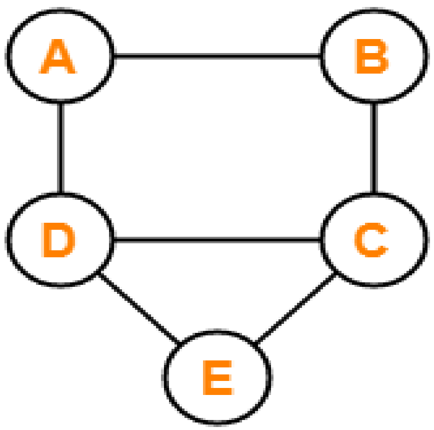
Hamiltonian Cycles

- 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.

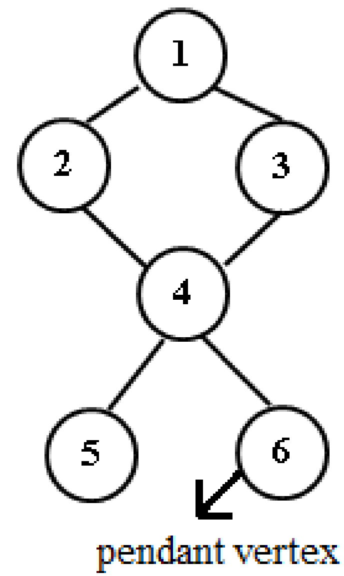


1, 2, 8, 7,6, 5, 4, 3, 1



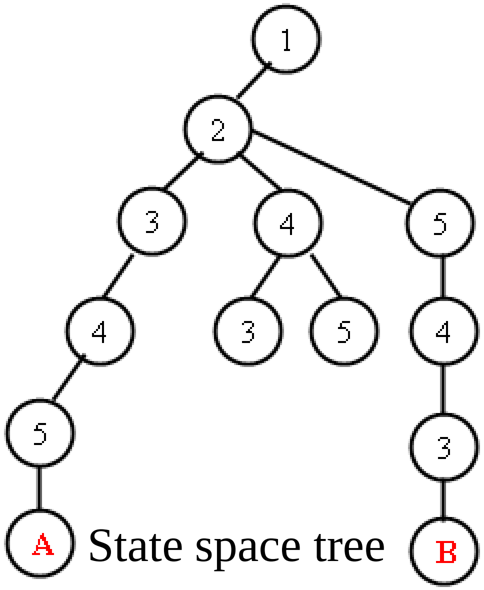


A, B, C, E, D, A
A, D, E, C, B, A



Hamiltonian Cycles

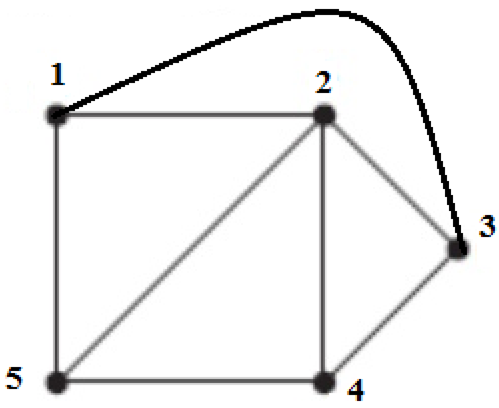
- The input for the Hamiltonian graph problem can be the directed or undirected graph. The Hamiltonian problem involves checking if the Hamiltonian cycle is present in a graph **G** or not.
- While generating the state space tree following bounding functions are to be considered, which are as follows: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

Knapsack

- Given *n* positive weights w_i, n positive profits p_i , and a positive number M which is the knapsack capacity, the 0/1 knapsack problem calls for choosing a subset of the weights such that

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m$$

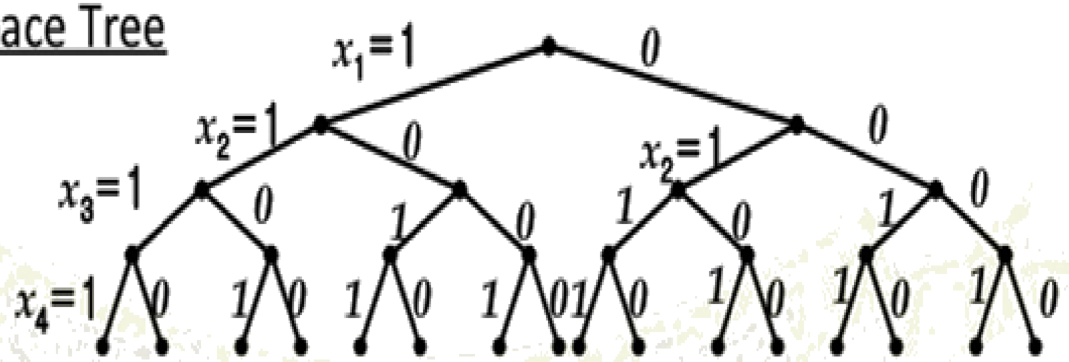
$$x_i = 0 \text{ or } 1, 1 \leq i \leq n$$

- The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's.
- Bounding function is needed to help kill some live nodes without actually expanding them.
- A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants.

Knapsack

- If this upper bound is not higher than the value of the best solution determined so far then that live node may be killed.
- If at node Z the values of x_i , $1 \leq i \leq k$ have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$ and use the greedy method to solve the relaxed problem.
- Procedure **Bound(p,w,k,M)** determines an upper bound on the best solution obtainable by expanding any node Z at level k+1 of the state space tree.

State-Space Tree



Knapsack

```
Algorithm Bknap(k, cp, cw)
// m is the size of the knapsack; n is the
//number of weights and profits.  w[] and p[]
//are the weights and profits p[i]/w[i] ≥
//p[i+1]/w[i+1]. fw is the final weight of
//knapsack; fp is the final maximum profit.
//x[k] = 0 if w[k] is not in the knapsack; else
//x[k] = 1.
{
 if(cw+w[k] ≤ m) then
 {
   y[k]= 1;
   if(k < n) then
     Bknap(k+1, cp+p[k], cw+w[k]);
   if((cp + p[k] > fp) and (k = n)) then
   {
     fp = cp + p[k]; fw = cw + w[k];
     for j =1 to k do x[j] = y[j];
   }
 }
 if (Bound(cp,cw,k) ≥ fp) then
 {
   y[k] = 0;
   if(k < n) then
       Bknap(k+1, cp, cw);
   if((cp > fp) and (k = n)) then
   {
      fp = cp; fw = cw;
      for j = 1 to k do x[j] = y[j];
   }
 }
}
```

Knapsack

```
Algorithm Bound(cp, cw, k)
//cp is the current profit total, cw is
//the current weight total; k is the
//index of the last removed item; and
//m is the knapsack size.
{
  b = cp; c = cw;
  for i = k+1 to n do
  {
     c = c + w[i];
     if(c < m) then b = b + p[i];
     else return b+(1-(c-m)/w[i])*p[i];
  }
  return b;
}
```