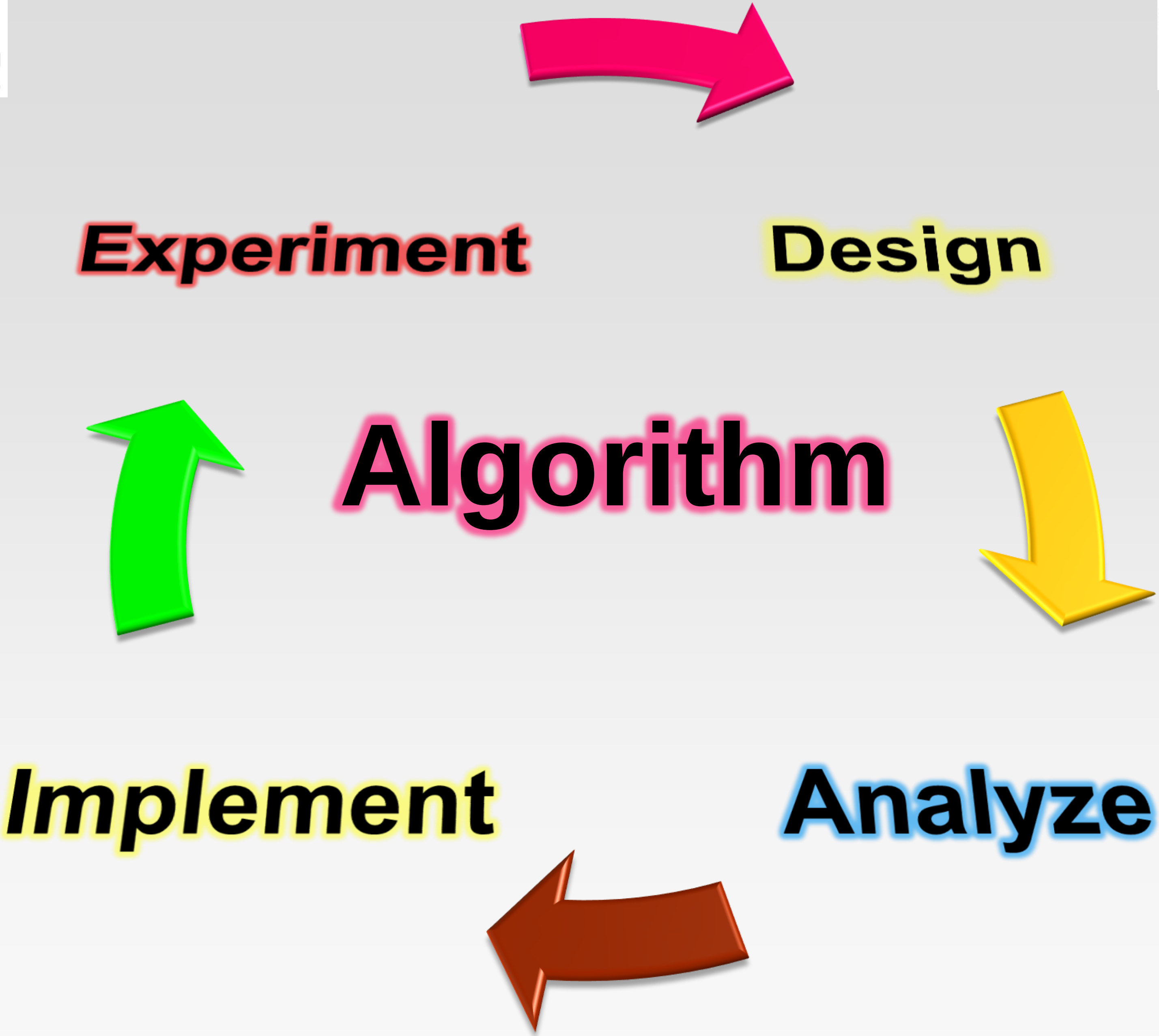
Design and Analysis of Algorithms (DAA) (A34EC)

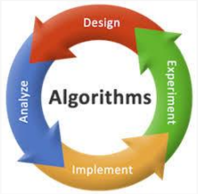
By :-

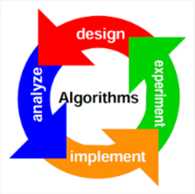
Vijaykumar Mantri,

Associate Professor.

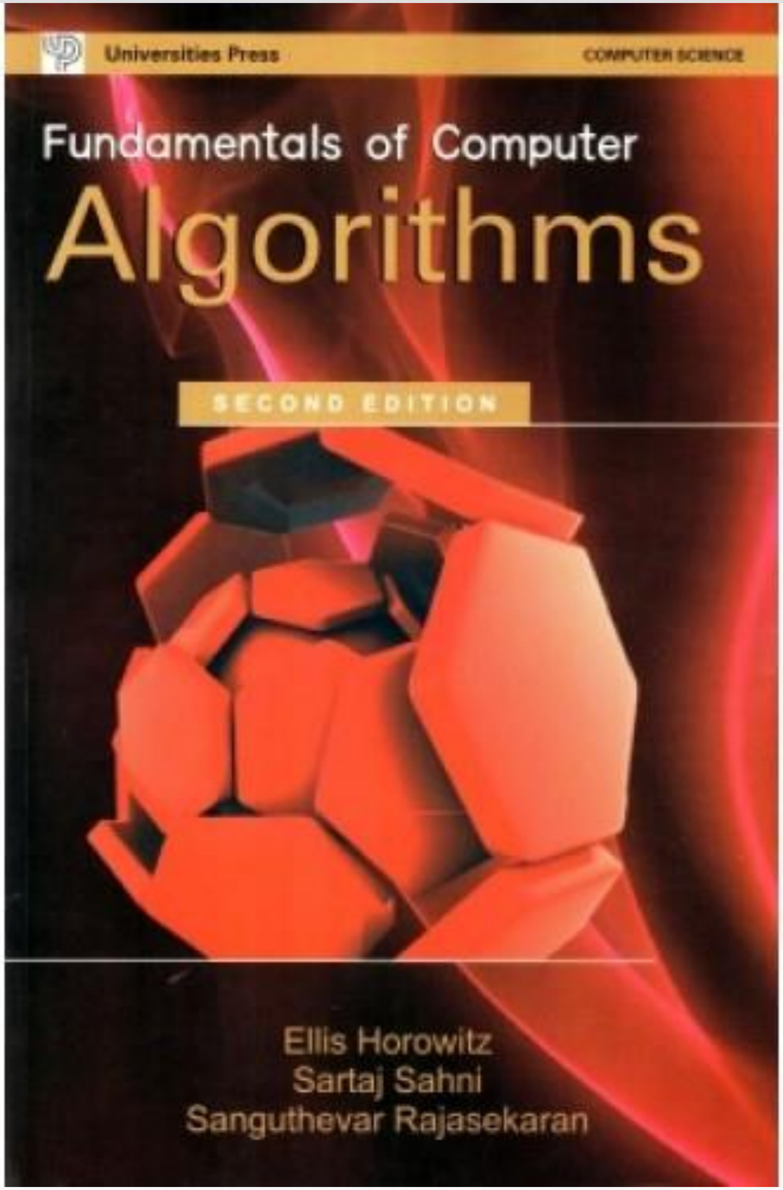
vijay_mantri.it@bvrit.ac.in


Experiment
Design
Algorithm
Implement
Analyze


Design
Experiment
Algorithms
Analyze
Implement


design
experiment
Algorithms
analyze
implement

Textbook

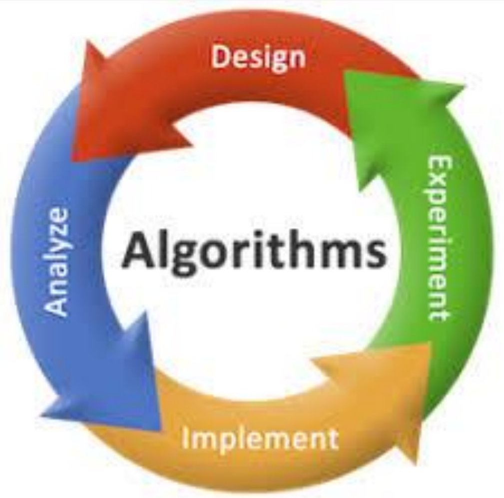


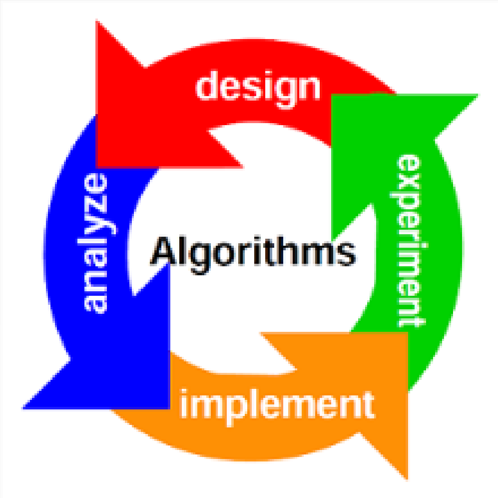
Universities Press
COMPUTER SCIENCE
Fundamentals of Computer
Algorithms
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran

DAA Unit V

Backtracking

Branch-and-Bound





Unit V Syllabus

Backtracking :

- General Method
- Applications-
  - N-Queens Problem
  - Sum of Subsets Problem
  - Graph Coloring
  - Hamiltonian Cycles.

Branch-and-Bound:

- General Method
- Applications
  - Travelling Sales Person Problem
  - 0/1 Knapsack Problem
    - LC Branch-and-Bound Solution
    - FIFO Branch-and-Bound Solution.

Backtracking – General Method

- Backtracking is one of the most general techniques for algorithm design.
- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, the desired solution must be expressible as an $\boldsymbol{n}$ –tuple $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\ldots \boldsymbol{x_n})$ where $\boldsymbol{x_i}$ is chosen from some finite set $\boldsymbol{S_i}$.
- The problem is to find a vector, which maximizes (or minimizes or satisfies) a criterion function $\mathbf{P}(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\ldots \boldsymbol{x_n})$
- Ex : N-Queens Problem, Sum of Subsets Problem, Graph Coloring, Hamiltonian Cycles.

Backtracking – General Method

- Let $\boldsymbol{m_i}$ is the size of set $\boldsymbol{S_i}$.
- Then there are $\boldsymbol{m} = (\boldsymbol{m_1}, \boldsymbol{m_2}, \boldsymbol{m_3}, \ldots\ldots \boldsymbol{m_n})$, $\boldsymbol{n}$ –tuples that are possible candidate for satisfying the function $\boldsymbol{P}$.
- The brute force approach tries all possible $\boldsymbol{n}$ –tuples for getting Optimal Solution.
- But the backtracking approach yields optimal solution with far fewer than $\boldsymbol{m}$ trials.
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $\mathbf{P}(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\ldots \boldsymbol{x_n})$ - Sometimes called bounding function.
- The major advantage of this method is, if it is realized that the partial vector $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\ldots \boldsymbol{x_n})$ can no way lead to Optimal solution, then $\boldsymbol{m_{i+1}}, \boldsymbol{m_{i+2}}, \ldots\ldots \boldsymbol{m_n}$ possible test vectors can be ignored.

- Backtracking solves the problems having two types of constraints as:

1. Explicit constraints.
2. Implicit constraints.

1) Explicit constraints : Explicit constraints are rules that restrict each $\boldsymbol{x_i}$ to take on values only from a given set.

- Some examples are,
- $\boldsymbol{x_i \geq 0 \quad or \quad S_i = \{all\ non - negative\ real\ number\}}$
- $\boldsymbol{x_i = 0\ or\ 1 \quad or \quad S_i = \{0, 1\}}$.
- $\boldsymbol{l_i \leq x_i \leq u_i \quad or \quad S_i = \{a : l_i \leq a \leq u_i\}}$
- The explicit constraint depend on the particular instance $\boldsymbol{I}$ of the problem being solved.
- All tuples that satisfy the explicit constraint define a possible solution space for $\boldsymbol{I}$.

2) Implicit constraints : The implicit constraint are the rules that determine which of the tuples in the solution space I satisfy the criterion functions.

- Thus implicit constraints describe the way in which the x_i must relate to each other.

- **Example :** $n - Queens\ Problem$
- A classic combinatorial problem is to place n queens on $n \times n$ chessboard so that no two queens attack; that is no two queens are on the same row or column or diagonal.
- Consider in particular $8 - Queens\ Problem$

- $8 - \boldsymbol{Queens\ Problem}$
- Let us number the rows and columns of the chessboard 1 through 8 & queens can also be numbered 1 through 8.
- Since each queen must be on a different row, we can assume queen $\boldsymbol{i}$ is to be placed on row $\boldsymbol{i}$.
- Therefore solution is represented in 8-tuple $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots\ldots \boldsymbol{x_n})$ where $\boldsymbol{x_i}$ is the column on which queen $\boldsymbol{i}$ is placed.
- The Explicit constraints using this formation are $\boldsymbol{S_i} = \{\boldsymbol{1, 2, 3, 4, 5, 6, 7, 8}\}, \boldsymbol{1} \leq \boldsymbol{i} \leq \boldsymbol{8}$.
- So the solution space consists of $\boldsymbol{8^8}$.
- The implicit constraints for this problem are that no two $\boldsymbol{x_i's}$ can be on same column and no two queens can be on the same diagonal.

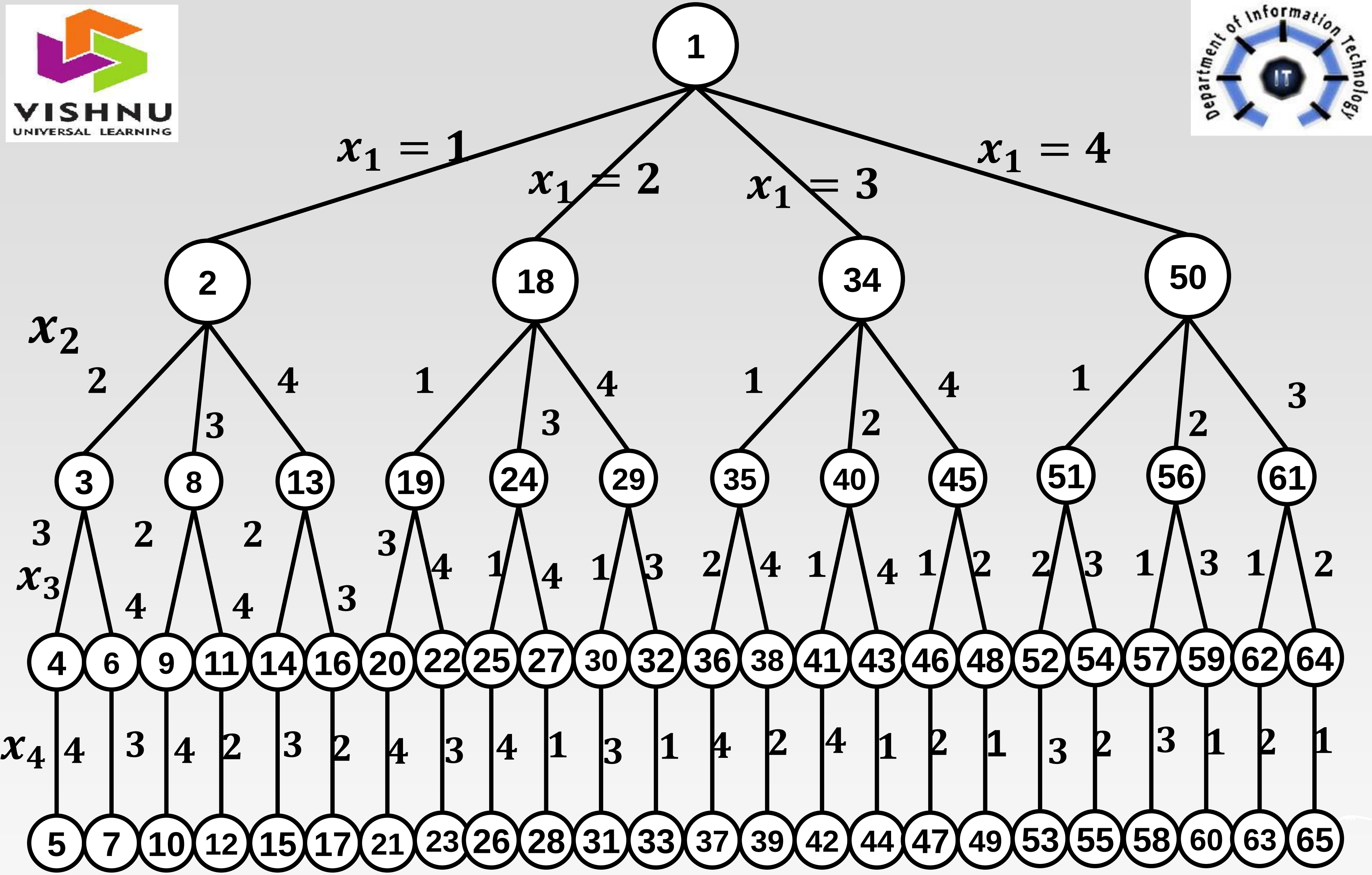
- The first constraints implies that all solutions are permutations of the 8-tuple $\{ \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8} \}$.
- So solution space reduced from $\mathbf{8^8}$ tuples to $\mathbf{8}!$ tuples.
- One of the Solution for 8-Queens problem $(\mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{2}, \mathbf{7}, \mathbf{1}, \mathbf{3}, \mathbf{5})$

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

One solution to the $8 - Queens$ problem

Solution Space for 4-Queens Problem

- The solution space forms **Permutation Tree** which defines all paths from root node to leaf node.
- The edges are labeled by all possible values of x_i.
- Edges from **Level-1 to Level-2** nodes specify values of x_1
- Thus the leftmost subtree contains all solutions with $x_1 = 1$; and its leftmost subtree contains all solutions with $x_1 = 1 \; and \; x_1 = 2$ and so on….
- Let us see a space tree organization of n –Queens problem (for $n = 4$) solution i.e. **Permutation tree** for $n = 4$.
- As value of n is four, there are $4! = 24$ leaf nodes in the tree.



Solution Space for 4-Queens Problem. Nodes are numbered as in Depth First Search (DFS)

```
Algorithm Backtrack (k)
// This schema describes the backtracking procedure
// 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[2], … , x[k − 1]) do
   {
      if (B_k(x[1], x[2], … . . x[k]) is true ) then
      {
        if (x[1], x[2], … . . x[k]) is the path to an answer node)
             then write(x[1: k]);
      if (k < n) then Backtrack(k + 1)
      }
   }
}
```

This recursive version is initially invoked by

***Backtrack*(1);**

Backtracking Solution

- We know how to prepare Solution space tree for any problem.
- These problems can be solved using Backtracking by generating problem states, checking which of these solution states are answer states.
- A solution begin with the root node and generate other nodes.
- 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 generated.
- As soon as a new child **C** of the current **E-node** R is generated, this child will become the new **E-node**.

4-Queens Problem - Backtracking Solution

- Let us see how backtracking works on 4-Queens problem.
- As a bounding function, we use criteria on a chessboard configuration in which no two queens are attacking.
- We start with the root node as the only **Live node**.
- This becomes **E-node** with **path ( )**.
- We generate one child (in ascending order) node number and **path is (1)**. This corresponding to placing queen 1 on column 1.
- Now node 2 becomes the E-node, node 3 is generated and immediately killed.
- The next node is generated is node 8 and path becomes (1, 3).
- Node 8 becomes E-node.
- However, it gets killed as all its children represent board configuration that cannot lead to an answer node.
- We backtrack to node and generate another child, node 13.
- The path is now (1, 4).

Backtrack Solution to 4-Queens Problem

1			

(a)

1			
.	.	2	

(b)

1			
		2	
.	.	.	.

(c)

1			
			2
.	3		

(d)

1			
			2
	3		
.	.	.	.

(e)

	1		

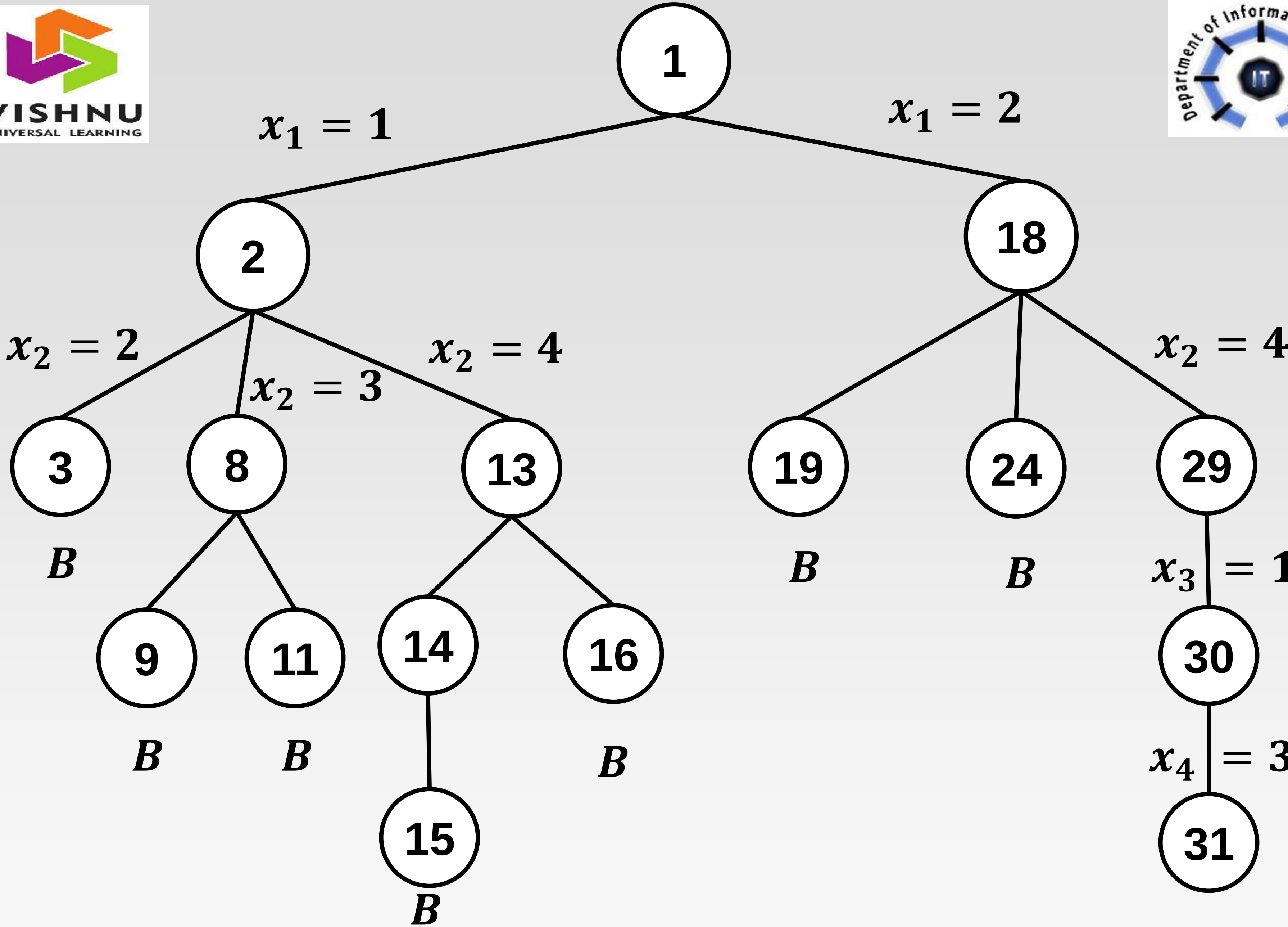
(f)

	1		
.	.	.	2

(g)

	1		
			2
3			
.	.	4	

(h)



Portion of the tree that is generated during Backtracking

n-Queens Problem - Backtracking Algorithm

- Now, let us see how solve 8-queens (& similar way n-queens) problem using backtracking.
- Consider an $\boldsymbol{n} \times \boldsymbol{n}$ chessboard (being numbered as indices of the two dimensional array $\boldsymbol{a}[\,\mathbf{1}:\boldsymbol{n}, \mathbf{1}:\boldsymbol{n}]$) and try to find all ways to place $\boldsymbol{n}$ non-attacking queens.
- We observed from 4-Queens problem that we can let $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots \boldsymbol{x_n})$ represent solution set in which $\boldsymbol{x_i}$ is the column of the $\boldsymbol{i^{th}}$ row where $\boldsymbol{i^{th}}$ queen placed.
- The $\boldsymbol{x_i}$ will all be distinct since no two queens can be placed in same column.
- Now we need to test & verify how to avoid two queens on the same diagonal.

8 – Queens Problem

- Let us consider a solution sample for 8-Queens and consider Queen at $a[4, 2]$.

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

$8 - \textit{Queens Problem}$

- The squares that are diagonals to this Queen (running from **upper left to lower right**) are $\boldsymbol{a[3,1]}$**,** $\boldsymbol{a[5,2]}$**,** $\boldsymbol{a[6,4]}$**,** $\boldsymbol{a[7,5]}$ **and** $\boldsymbol{a[8,6]}$**.**
- All these squares have $(\boldsymbol{row\ - column})\ \boldsymbol{value\ of\ 2}$ i.e**.** $(\boldsymbol{row\ - column})$ is same.
- Similarly, other squares that are diagonals to this Queen (running from **upper right to lower left**) are $\boldsymbol{a[1,5]}$**,** $\boldsymbol{a[2,4]}$**,** $\boldsymbol{a[3,3]}$**,** and $\boldsymbol{a[5,1]}$**.**
- Every element on the same diagonal to this Queen that goes from have $(\boldsymbol{row + column})\ \boldsymbol{value\ of\ 6}$ i.e. same value of $(\boldsymbol{row + column})$.

n-Queens Problem - Backtracking Algorithm

- Suppose two queens are placed at positions $(\boldsymbol{i}, \boldsymbol{j})$ and $(\boldsymbol{k}, \boldsymbol{l})$.
- Then they are on the same diagonal only if

$$\boldsymbol{i} - \boldsymbol{j} = \boldsymbol{k} - \boldsymbol{l} \qquad \text{(\textbf{upper left to lower right})}$$

$$\boldsymbol{or} \;\; \boldsymbol{i} + \boldsymbol{j} = \boldsymbol{k} + \boldsymbol{l} \qquad \text{(\textbf{upper right to lower left})}$$

- The first equation implies

$$\boldsymbol{i} - \boldsymbol{k} = \boldsymbol{j} - \boldsymbol{l}$$

- The second equation implies

$$\boldsymbol{k} - \boldsymbol{i} = \boldsymbol{j} - \boldsymbol{l}$$

- Therefore two queens lie on the same diagonal if and only if

$$|\boldsymbol{i} - \boldsymbol{k}| = |\boldsymbol{j} - \boldsymbol{l}|$$

- Place $(\boldsymbol{k}, \boldsymbol{i})$ returns a Boolean value that is true if $\boldsymbol{k}^{\boldsymbol{th}}$ queen can be placed in column $\boldsymbol{i}$.

```
Algorithm Place (k, i)
// Returns True if a queen can be placed in k^th row
// and i^th 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)) ) then
                          // or in the same diagonal
                 return false;
        return true;
  }
```

```
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 );
        }
    }
}
```

This algorithm is initially invoked by

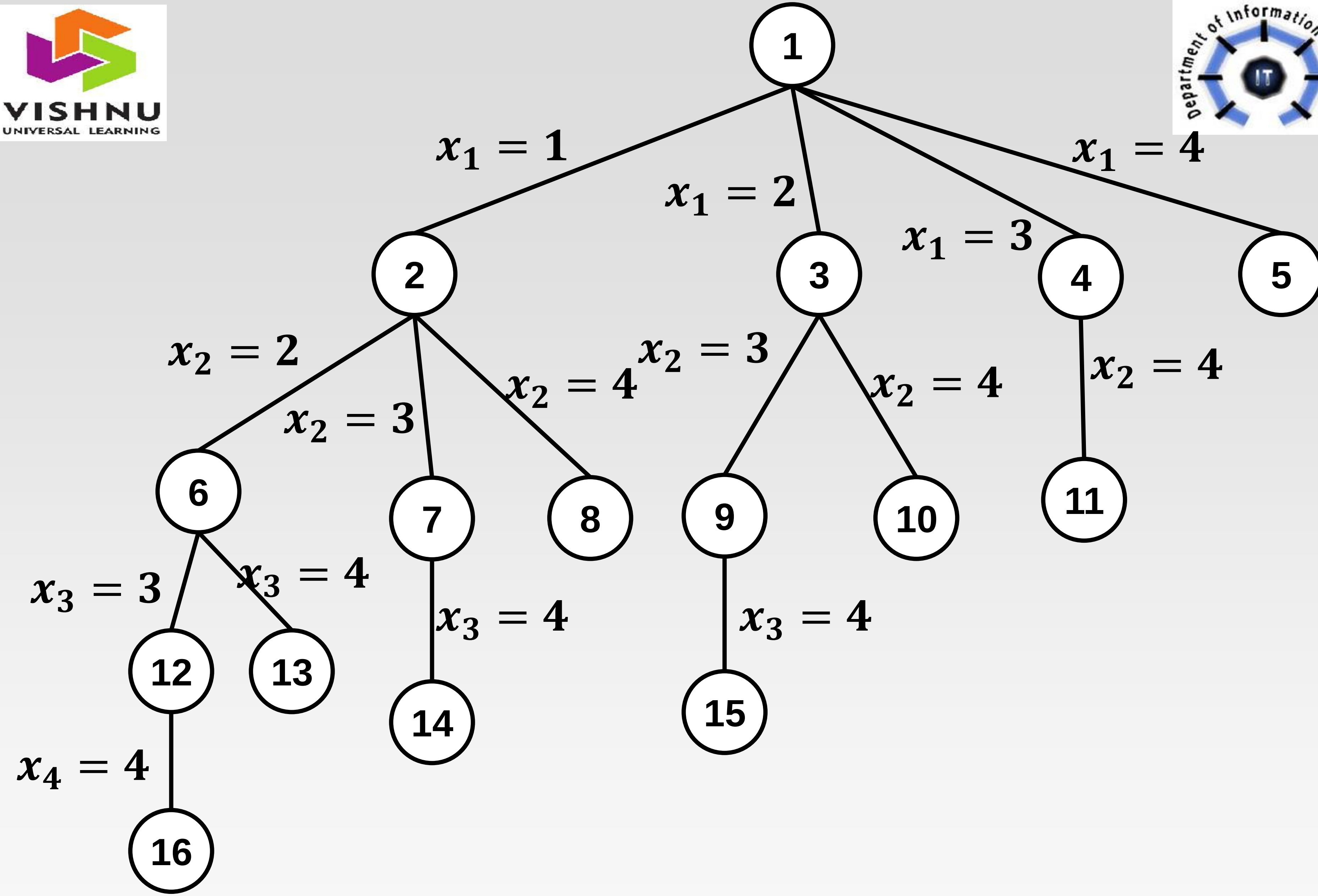
$NQueens\ (1, n);$

Sum of Subsets

- We are given n positive numbers $w_i, 1 \leq i \leq n$ and m
- We have to find all subset combinations of these numbers whose sum is m. This is called Sum of subsets problem.
- For example, if $n = 4, (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, **and** $m = 31$
- For this problem desired subsets are $(11, 13, 7)$ and $(24, 7)$**.**
- Instead of representing solution vector by the w_i which sum to m, we could represent the solution vector by giving the indices of these w_i.
- So these two solutions are described by the vectors $(1, 2, 4)$ and $(3, 4)$

Sum of Subsets

- In general, all solutions are $\boldsymbol{k}$ – tuples $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\ldots \boldsymbol{x_k})$, $\mathbf{1} \leq \mathbf{k} \leq \mathbf{n}$, and different solutions may have different-sized tuples.
- The explicit constraints require
  $$\boldsymbol{x_i} \in \{\, \boldsymbol{j} \mid \boldsymbol{j}\ \boldsymbol{is\ an\ integer\ and}\ \mathbf{1} \leq \boldsymbol{j} \leq \boldsymbol{n}\}$$
- The implicit constraints require that no two subset be the same and that the sum of the corresponding $\boldsymbol{w_i's\ be\ m}$.
- Since we wish to avoid generating multiple instances of the same subset (Eg : $(\mathbf{1}, \mathbf{2}, \mathbf{4})$ and $(\mathbf{1},\ \mathbf{4}, \mathbf{2})$ represent same subset)
- So, Another implicit constraint that is imposed is that
  $$\boldsymbol{x_i} < \boldsymbol{x_{i+1}}, \mathbf{1} \leq \boldsymbol{i} < \boldsymbol{k}$$



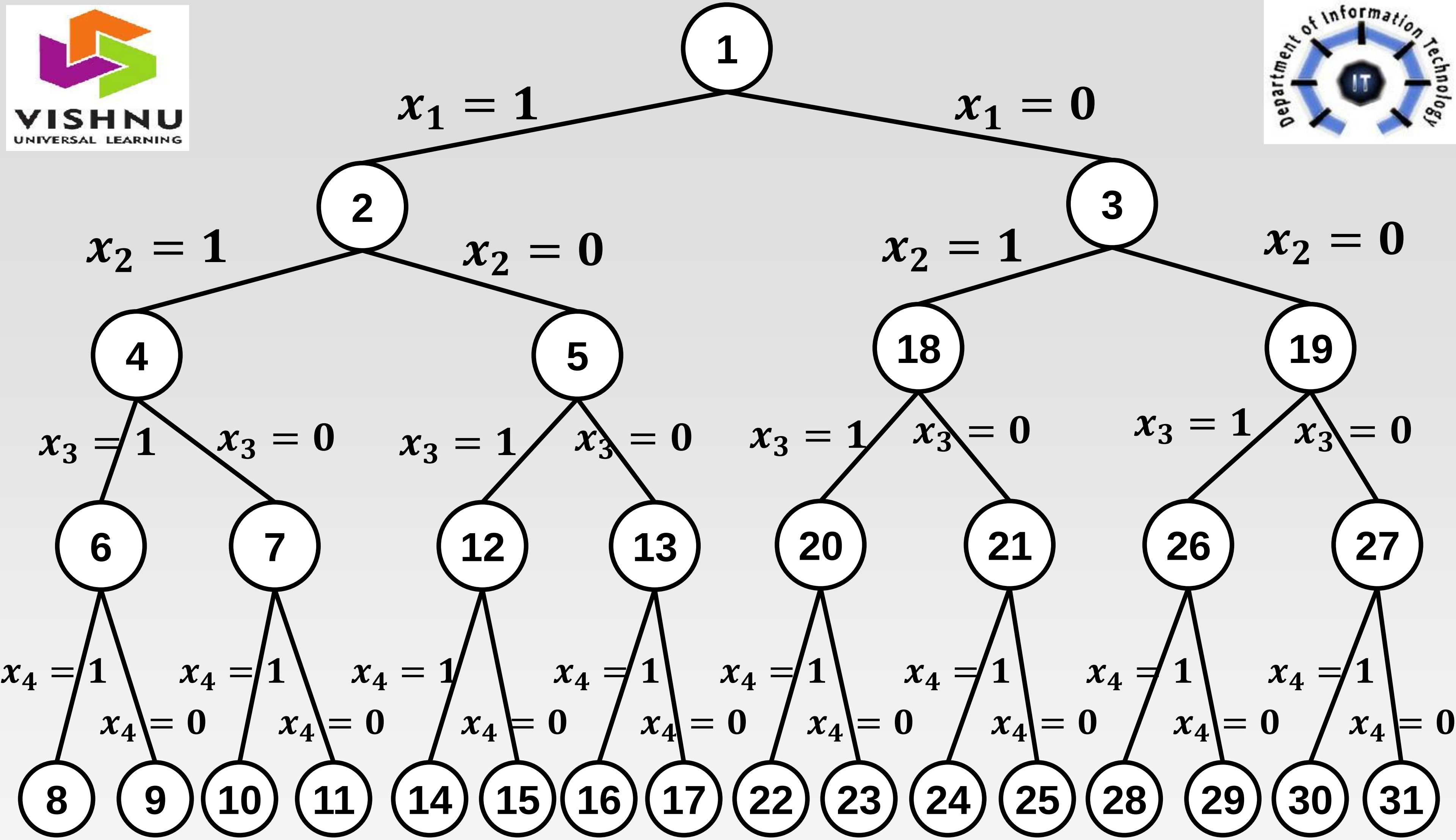
Solution Space for Sum of Subsets Problem. Nodes are numbered as in Breadth First Search (BFS)

Sum of Subsets

- In another formation of the solution set, it is represented by fixed n – tuples $(x_1, x_2, x_3, \dots\dots x_n)$ such that

$$x_i \in \{\, 0 \; 1 \,\}, 1 \le i \le n.$$

- Here $x_i = 0 \; if \; w_i$ not chosen and $x_i = 1 \; if \; w_i$ chosen.
- The solutions to previous example are $(1, 1, 0, 1)$ and $(0, 0, 1, 1)$.



Another possible Solution Space for Sum of Subsets Problem. Nodes are numbered as in Depth First Search (DFS)

Sum of Subsets

- The bounding functions we use for Sum of Subsets are

$$\sum_{i=1}^{k} w_i\, x_i + \sum_{i=k+1}^{n} w_i \;\geq\; m$$

- **And**

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

- The algorithm SumOfSub avoids computing $\sum_{i=1}^{k} w_i\, x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variable s & r
- The algorithm assumes $w_1 \leq m \; and \; \sum_{i=1}^{n} w_i \geq m$
- The initial call is **SumOfSub(0, 1,** $\sum_{i=1}^{n} w_i$)

Algorithm $SumOfSub(s, k, r)$

// *s is Sum of previous elements,* k^{th} *element we are*

// *adding &* r *is sum of remaining elements including* k

// *i.e* $s = \sum_{j=1}^{k-1} w[j] * x[j]$ *and* $r = \sum_{j=k}^{n} w[j]$.

// *Find all subsets of* $w[1:n]$ *that sum to* m. *The values*

// *of* $x[j], 1 \leq j < k$, *have already been determined.*

// *The* $w[j]$ *are in increasing order.*

// *It is assumed that* $w[1] \leq m$ *and* $\sum_{i=1}^{n} w[i] \geq m$

{

    // Generate left child

    $x[k] := 1$;

if $(s + w[k] = m)$ *then* $write(x[1:k])$; // Subset found

    else if $(s + w[k] + w[k+1] \leq m)$ *then*

        $SumOfSub(s + w[k], k + 1, r - w[k])$;

//i.e if adding $\mathrm{k^{th}}$ & assuming adding next $\mathbf{k+1^{th}}$ is less than m

```
    // Generate right child.
    if((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m))then
    // i.e if not adding k^th element then Previous Sum plus
    // remaining sum of elements (Except k^th element) should be
    // greater than or equal to m
    // & assuming adding next k + 1^th to Previous Sum
    // is less than or equal to m.
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```

Sum of Subsets

- Lets trace the algorithm by drawing a State Space tree generated by **SumOfSub** Algorithm.
- Consider the example that we have seen previously

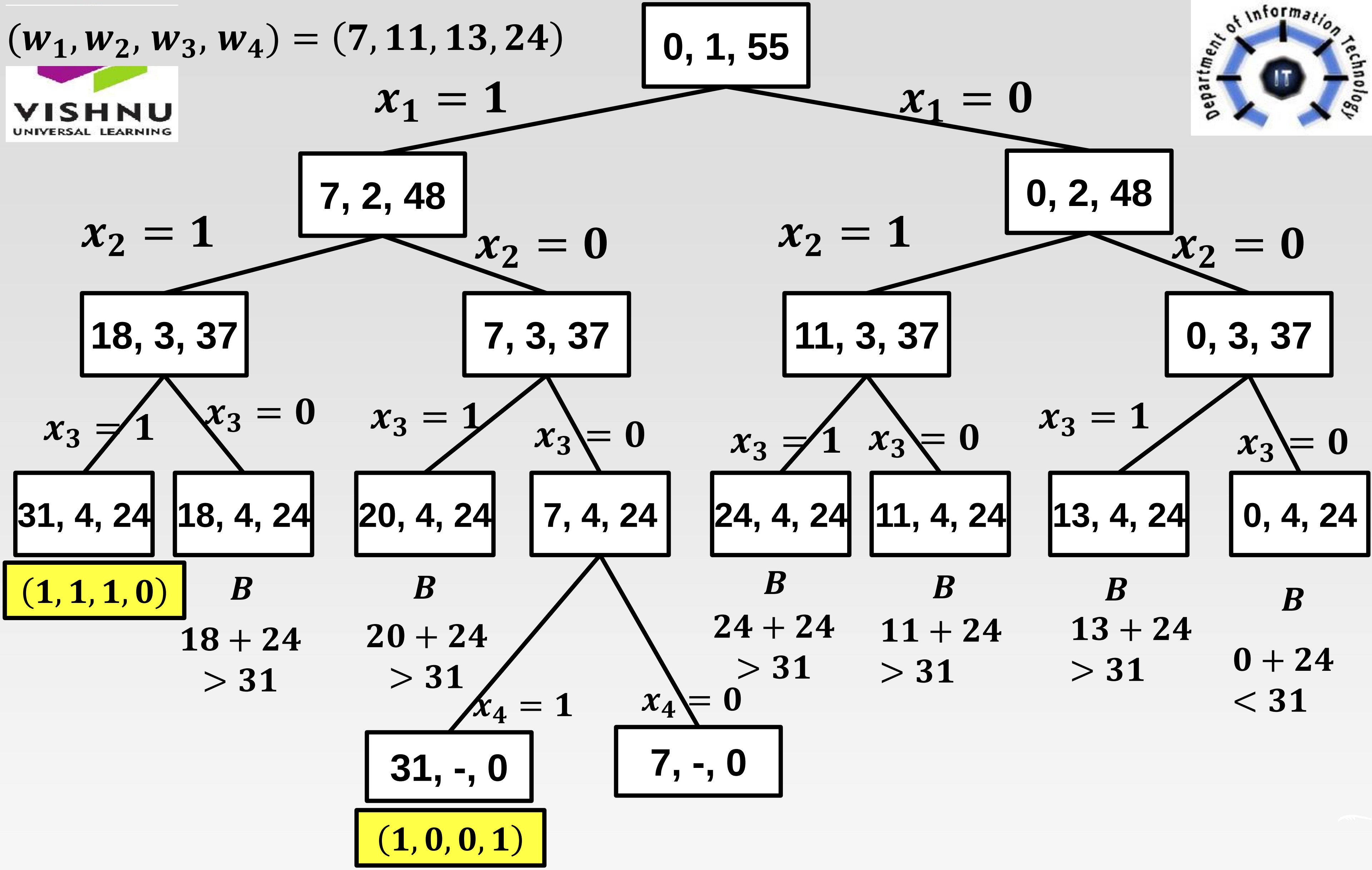
$$n = 4, (w_1, w_2, w_3, w_4) = (11, 13, 24, 7), \text{ and } m = 31$$

- As the $w_i's$ are not in increasing order, we need to sort them first. The sorted $w_i's$ are

$$(w_1, w_2, w_3, w_4) = (7, 11, 13, 24)$$

- Let's draw solution space tree for this Example.

$(w_1, w_2, w_3, w_4) = (7, 11, 13, 24)$



Solution Space tree generated by SumOfSub Algorithm

Sum of Subsets Exercise

- For the following examples, find all possible subsets of $\boldsymbol{w}$ that sum to $\boldsymbol{m}$.
- Do this using SumOfSub algorithm.
- Draw the portion of the state space tree that is generated.

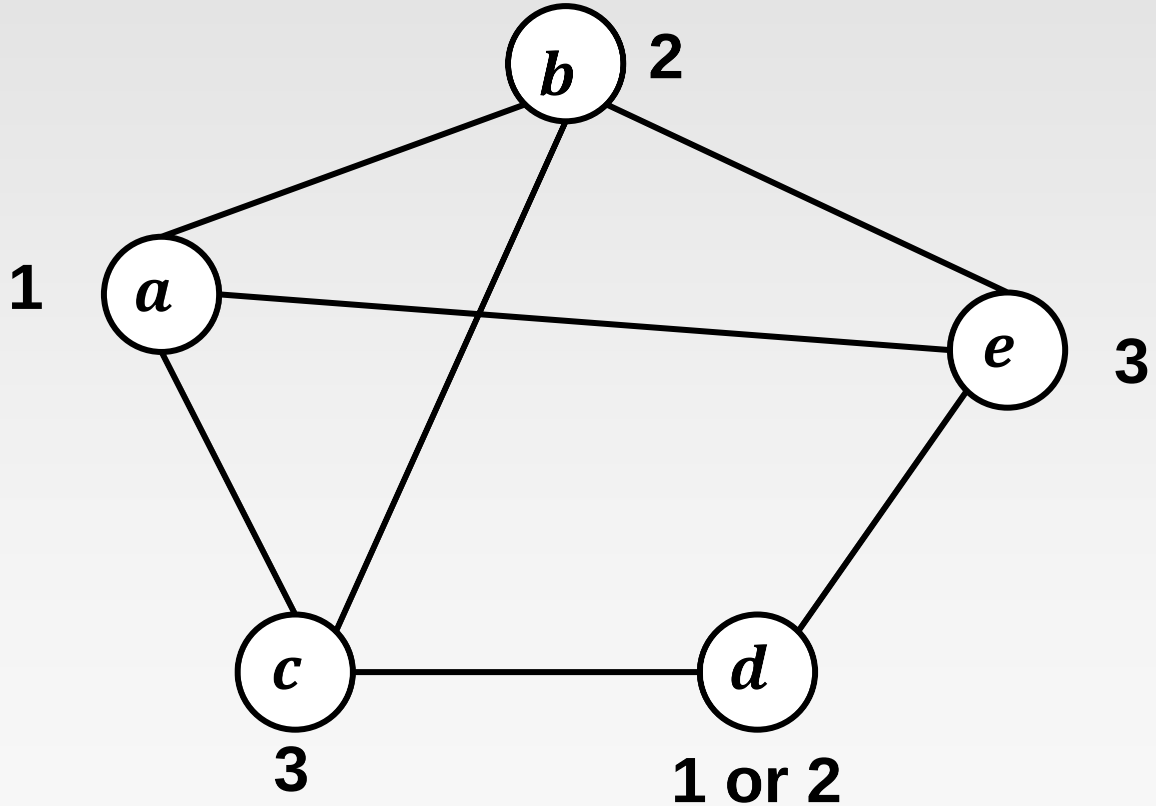
1. $\boldsymbol{n = 5\ (w_1, w_2, w_3, w_4, w_5) = (7, 3, 2, 5, 8)}$, **and** $\boldsymbol{m = 14}$
2. $\boldsymbol{n = 6\ (w_1, w_2, w_3, w_4, w_5,\ w_6) = (5, 10, 12, 13, 15, 18)}$, **and** $\boldsymbol{m = 30}$
3. $\boldsymbol{n = 7\ (w_1, w_2, w_3, w_4, w_5,\ w_6, w_7) = (5, 7, 10, 12, 15, 18, 20)}$, **and** $\boldsymbol{m = 35}$.

Graph Coloring

- Assigning colors to the nodes of a graph such that, no two adjacent nodes have same color is called Graph Coloring.
- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have same color yet only m colors are used.
- This is known as **m-colorability decision problem**.
- The **m-colorability optimization problem** is for the smallest integer m for which the graph can be colored.
- The minimum number of colors required to color the graph is called as **Chromatic Number**.

Graph Coloring

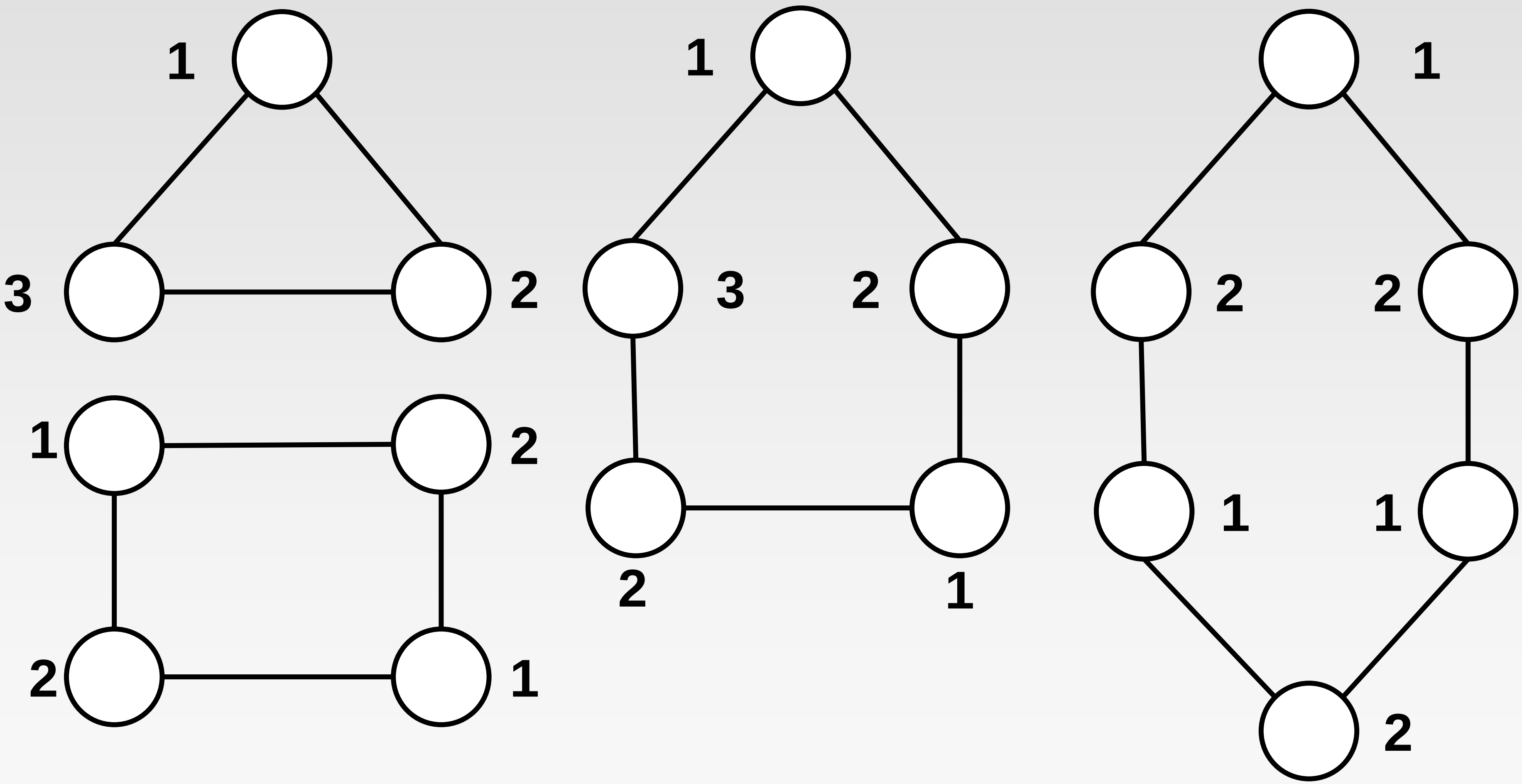
- For example, consider the given graph.
- The graph can be colored with three colors 1, 2, and 3.
- Hence chromatic number for this graph is 3.



Graph Coloring

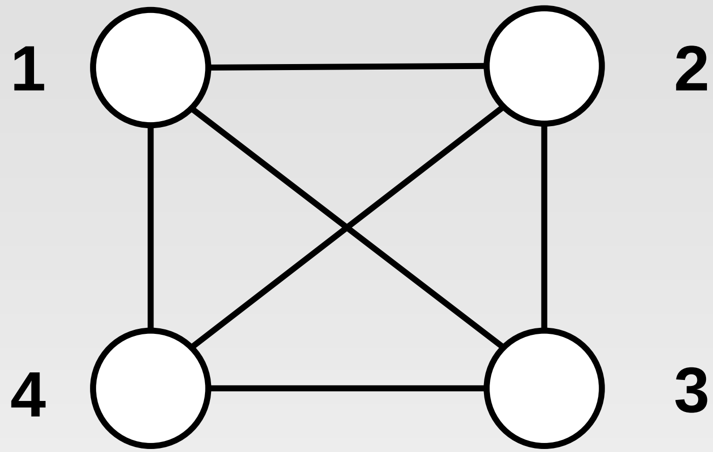
- For any Cyclic Graph C_n

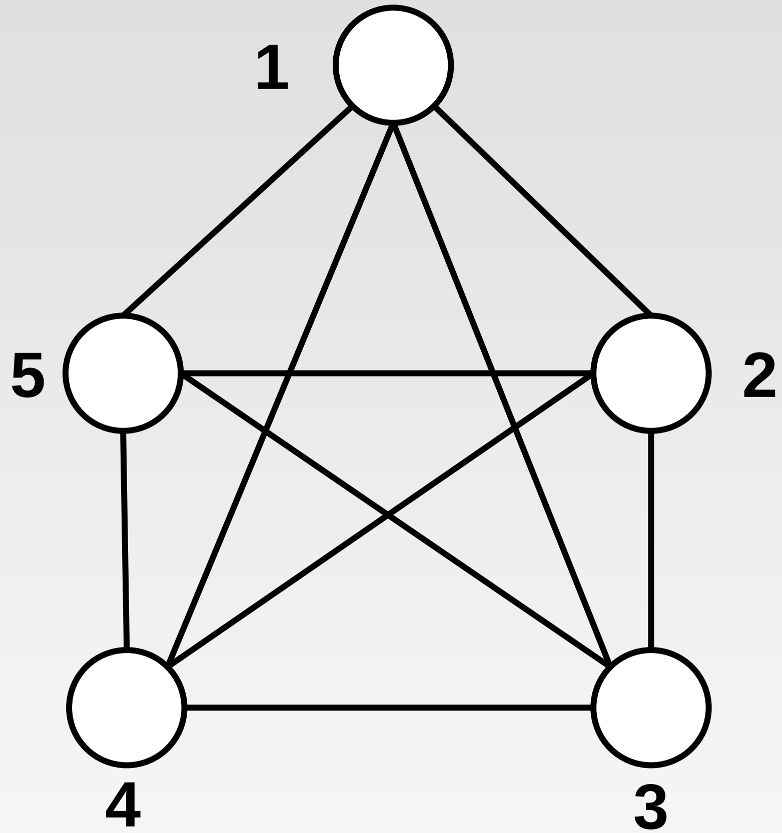
  Chromatic number $= \boldsymbol{2 \; if \; n \; is \; Even} \;\&\; \boldsymbol{3 \; if \; n \; is \; Odd}$



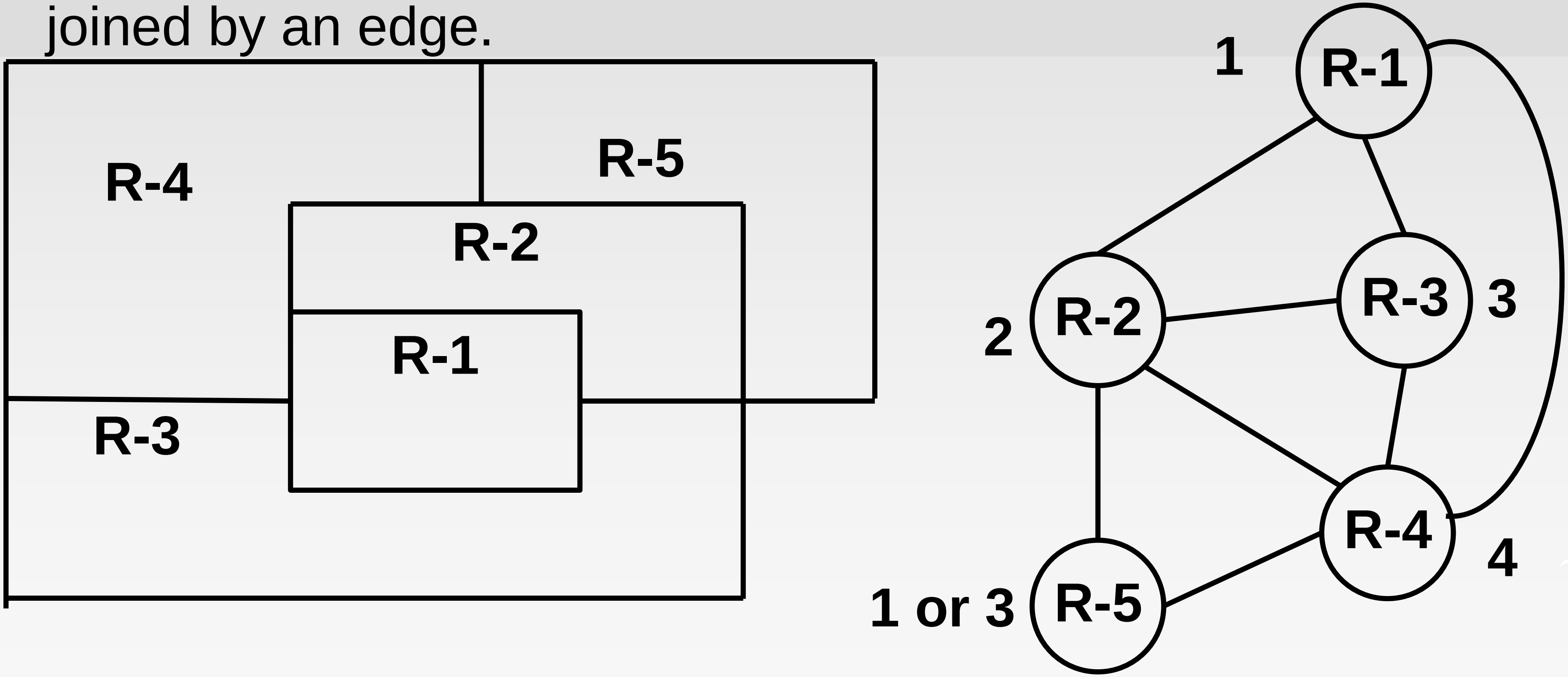
Graph Coloring

- For Complete Graph K_n chromatic number $= n$.





- 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 then, 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.



A Map and its planner graph representation

Graph Coloring

- We are interested in determining all the different ways in which a given graph can be colored using at most $\boldsymbol{m}$ colors.
- We will represent a graph by its adjacency matrix $\boldsymbol{G[1:n, 1:n]}$ where $\boldsymbol{G[i,j] = 1\ if (i,j) is\ an\ edge\ of\ G, and\ G[i,j] = 0\ if\ there\ is\ no\ Edge}$.
- The colors are represented by the integers $\boldsymbol{1, 2, \ldots., m}$.
- The solutions are given by $\boldsymbol{n - tuple\ (x_1, x_2,, \ldots . x_n), where\ x\ is\ color\ of\ node\ i}$.
- Using recursive backtracking mColoring algorithm we can solve Graph Coloring Problem.
- Function mColoring is initially invoked by mColoring(1).
- Initially array $\boldsymbol{x[\quad]\ set\ to\ Zero}$

```
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 Graph
// such that adjancent vertices are assigned distinct
//integers are printed. k is index of next vertex to color
  {
    while(true)
    {         // Generate all legal assignments for x[k].
       nextValue(k);
     if(x[k] = 0) then return;    // No new color possible
     if (k = n) then write(x[1: n]) // At most m colors used
      else mColoring(k + 1)
    }
  }
```

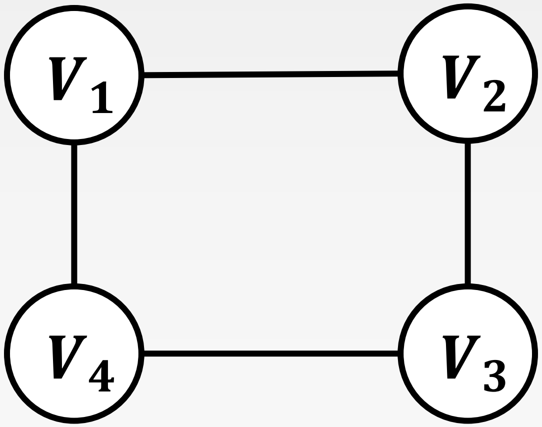
```
Algorithm NextValue(k)
//x[1], ..... x[k − 1] have been assigned integer values in
//the range [1, m] such that adjacent vertices have
//distinct integers. A value for x[k] is determined in
//the range [0, m]. x[k] is assigned for the next highest
//numbered color while maintaining distinctness from
//the adjacent vertices of vertex k.
//if no such color exists, then x[k] is 0.
  {
    while(true)
    {
      x[k] = (x[k] + 1) mod (m + 1); //Next highest color
     if (x[k] = 0) then return;  //All color have been used
```

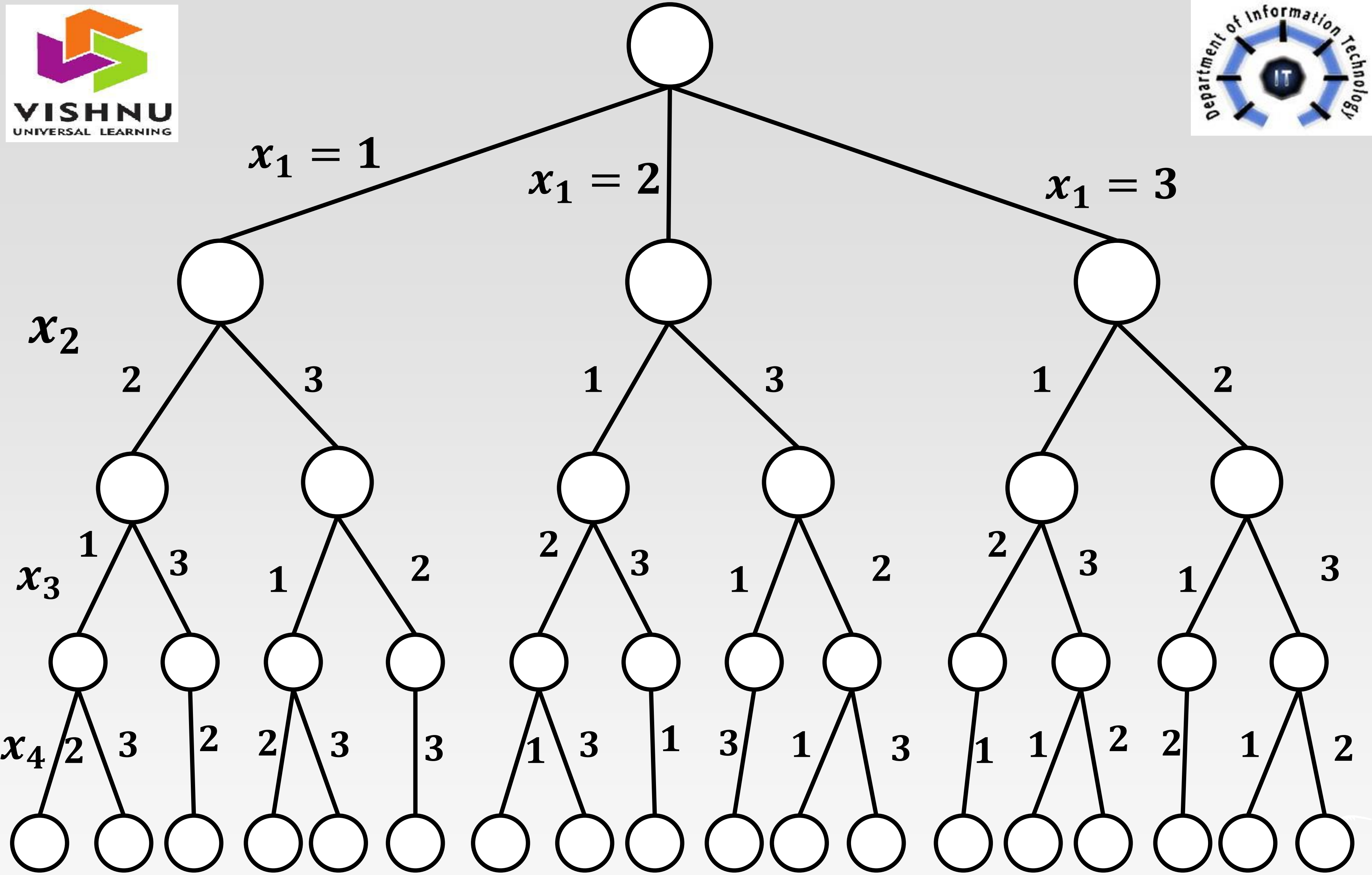
Algorithm : NextValue

```
    for j := 1 to n do
    { // Check if this color is distinct from adjacent colors
        if((G[k, j] = 1) and (x[k] = x[j])) then
// If (k, j) is an edge and if adjacent vertices have same color
    break;
    }
    if(j = n + 1) then return;     // New Color found
  }         // Otherwise try to find another color
}
```

Graph Coloring

- Lets trace the algorithm by drawing a State Space tree generated by **mColoring** Algorithm.
- The state space tree used is a tree of degree $\boldsymbol{m}$ and height $\boldsymbol{n+1}$.
- Each node at level $\boldsymbol{i}$ has $\boldsymbol{m}$ children corresponding to the $\boldsymbol{m}$ possible assignments to $\boldsymbol{x_i, 1 \le i \le n}$.
- Let's draw solution space tree for a graph with four nodes and 3 colors as an Example i.e. $\boldsymbol{n = 4 \; and \; m = 3}$.



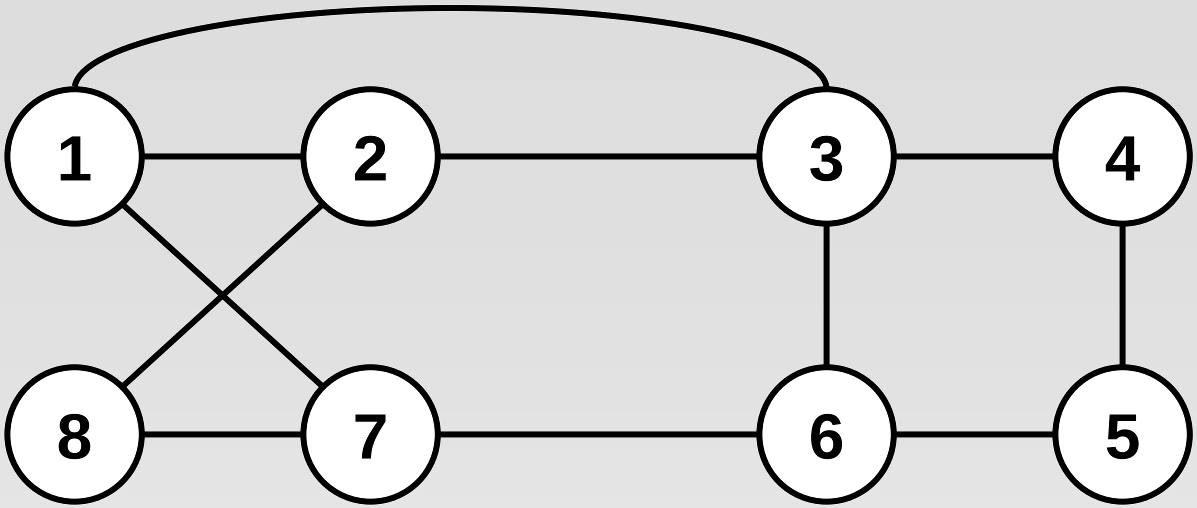


A 4-Node Graph and all possible 3-Colorings

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 and returns to its starting position.
- In other words if a Hamiltonian Cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots.$ v_{n+1} then edges (v_i, v_{i+1}) are in $E, 1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.
- The backtracking solution vector $(x_1, x_2, \ldots . x_n)$ is defined so that x_i represents the i^{th} visited vertex of proposed cycle.
- The function NextValue(k), which determines a possible next vertex for the proposed cycle.
- Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian Cycles.

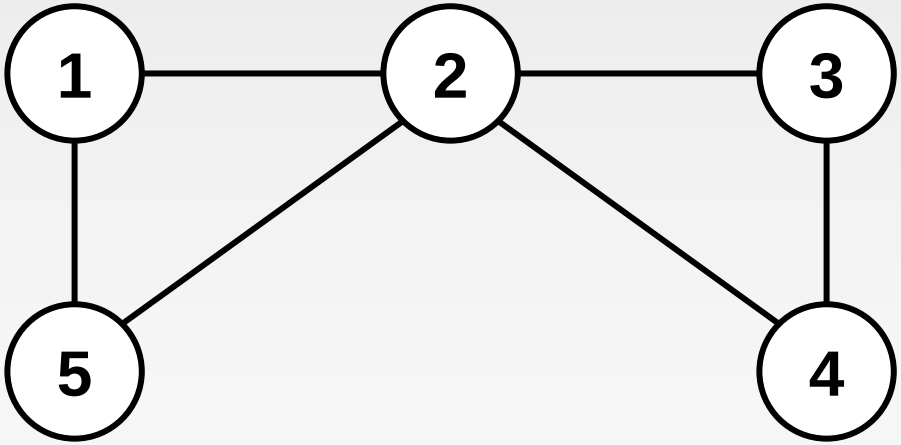
G1:



- Hamiltonian Cycle (1) → $1 - 3 - 4 - 5 - 6 - 7 - 8 - 2 - 1$.

  (2) → $1 - 2 - 8 - 7 - 6 - 5 - 4 - 3 - 1$.

G2:



Example Graphs

- No Hamiltonian Cycle

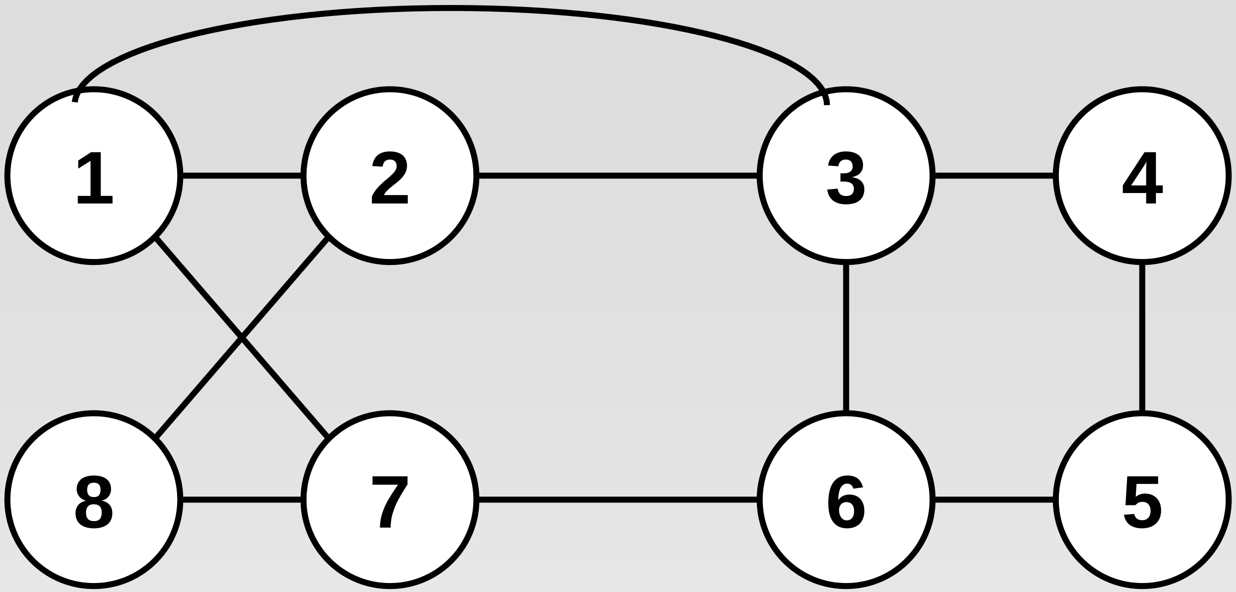
```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
//of a graph. The graph is stored as adjacency
//matrix G[1:n, 1:n]. All cyclesbegin at 1.
  {
     while(true)
     {         // Generate values for x[k].
        nextValue(k);     // Assign a legal next value of  x[k]
      if(x[k] = 0) then return;     // No new node possible
      if (k = n) then write(x[1:n]) // All nodes visited
       else  Hamiltonian(k + 1)
     }
  }
```

This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ *to Zero and* $x[1]$ *to* **1** and then executing **Hamiltonian (2)**.

```
Algorithm NextValue (k)
//x[1: k − 1] is a path of k − 1 distinct vertices.
// If x[k] = 0, then has yet been assigned to x[k].
// After execution x[k] is assigned to the next highest
// numbered vertex which doesn't 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].
  {
    while(true)
    {
      x[k] = (x[k] + 1)mod(n + 1);//Next vertex
     if(x[k] = 0)then return;
```

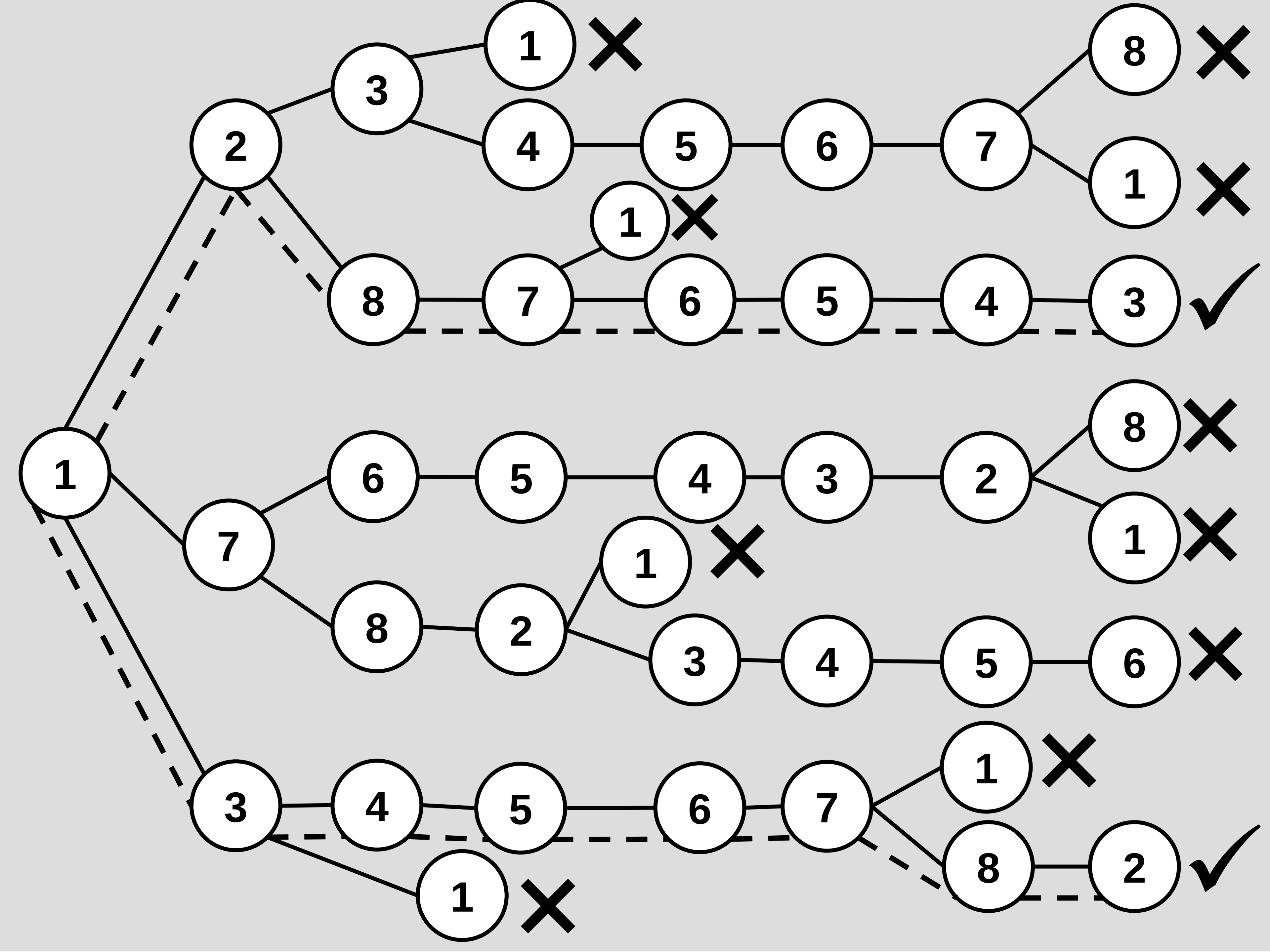
```
if (G[x[k − 1], x[k]] = 1) then
{     //Is there an edge?
 for j := 1 to k − 1 do
   if (x[j] = x[k]) then break;      // Check for distinctness
   if (j = k) then      // If true, then the vertex is distinct.
      if ((k < n) or ((k = n) and G[x[k], x[1]] = 1)) then
// If there are still nodes or
// This is last node and this node connected to first node.
            return;
 }
}
}
```

G1:

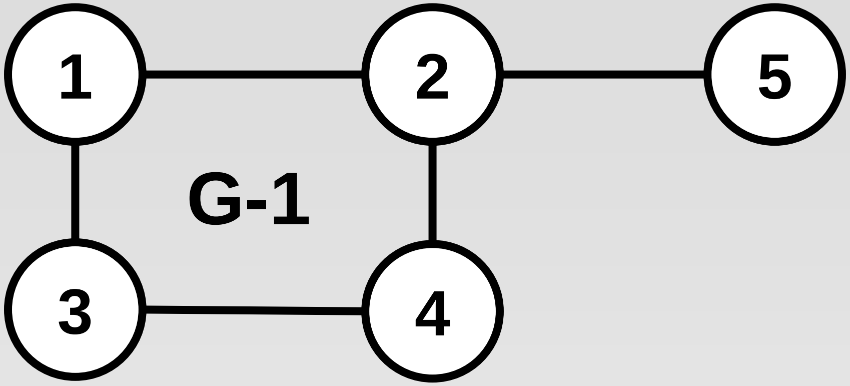


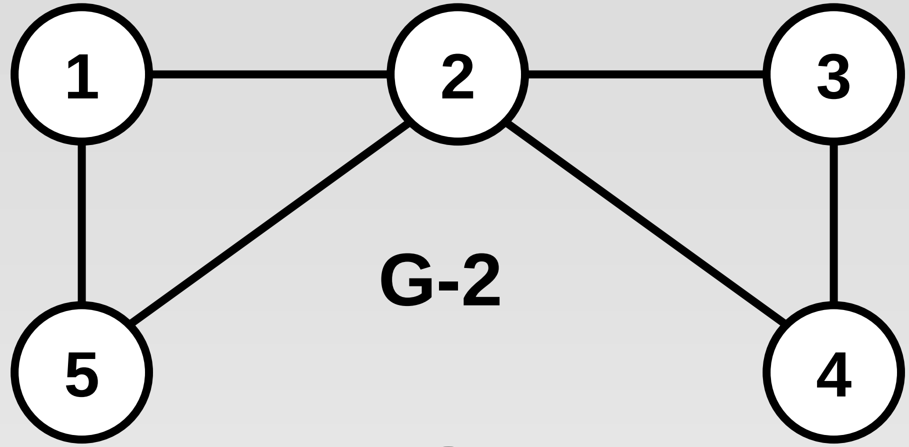
- **Hamiltonian Cycle (1)** → 1 – 3 – 4 – 5 – 6 – 7 – 8 – 2 – 1.

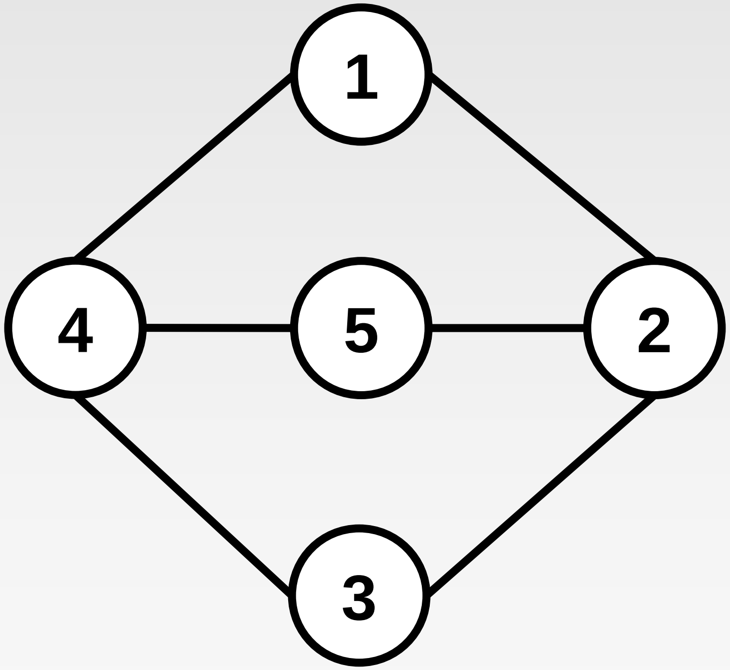
  (2) → 1 – 2 – 8 – 7 – 6 – 5 – 4 – 3 – 1.


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

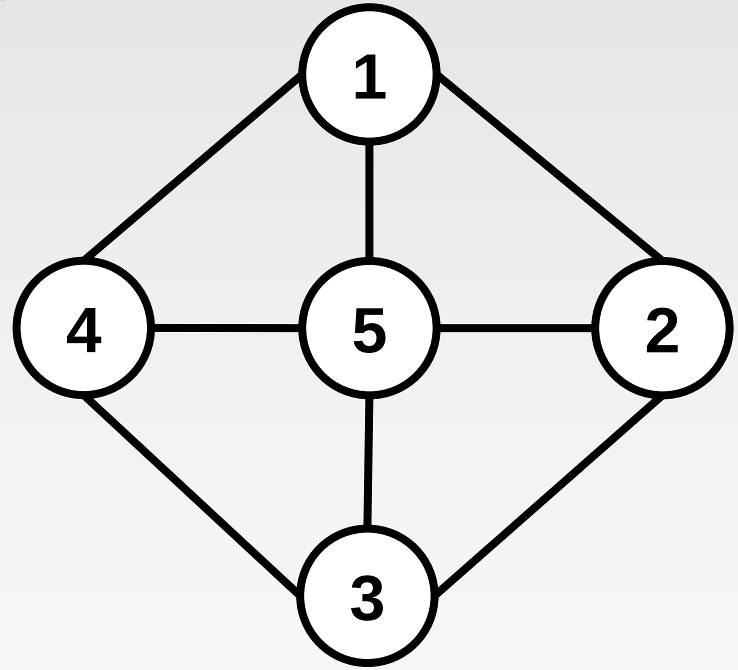
Find Hamiltonian Cycles in the following Graphs, if exists







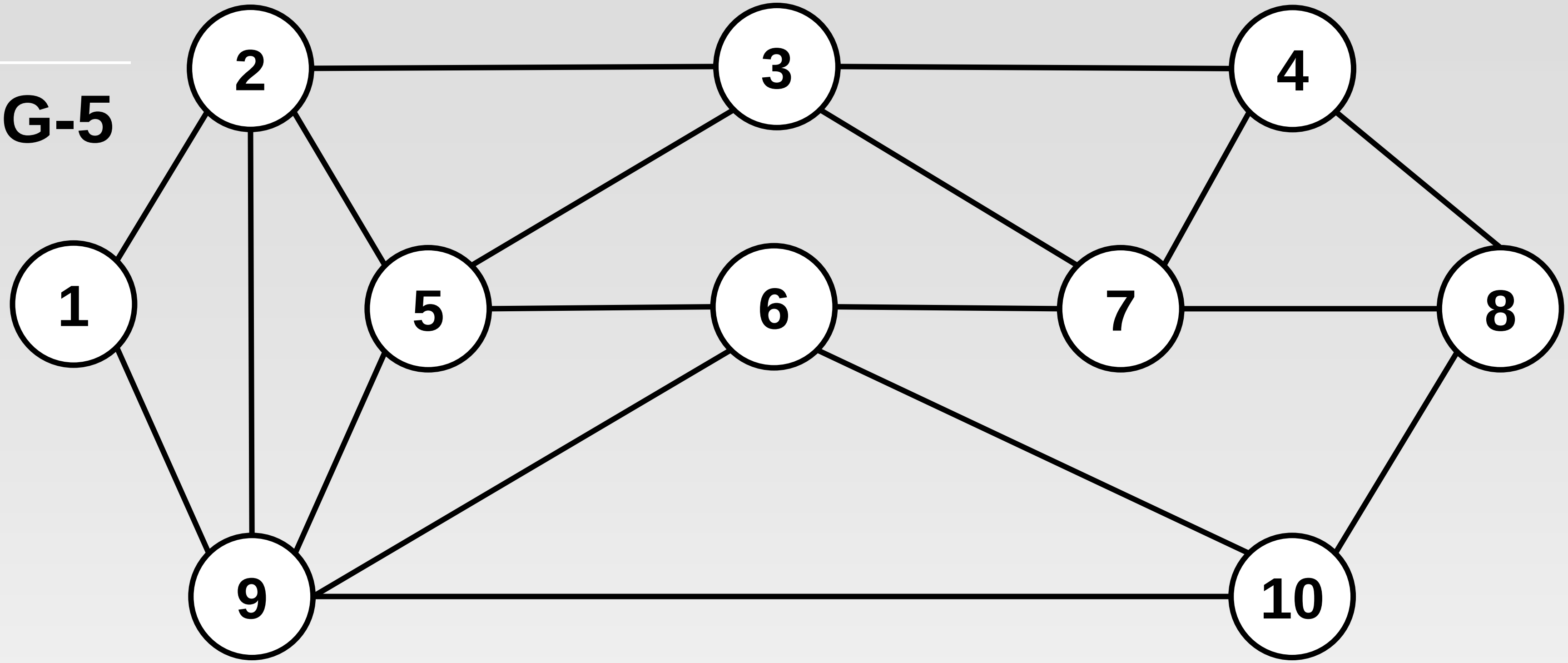
G-3

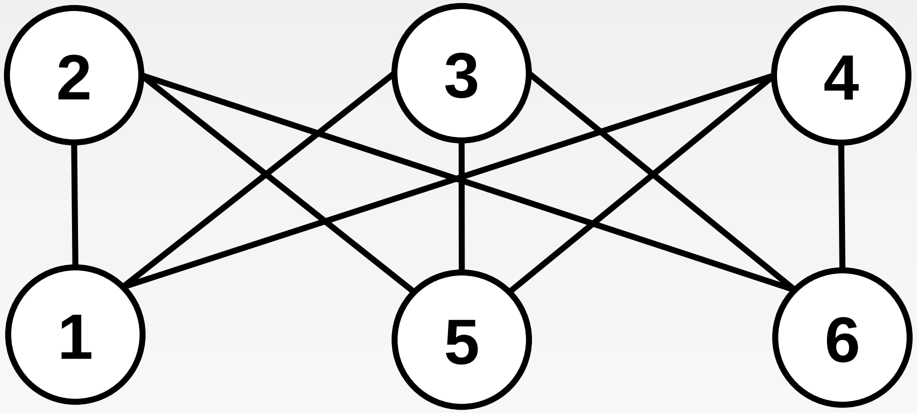


G-4

Find Hamiltonian Cycles in the following Graph, if exists

G-5

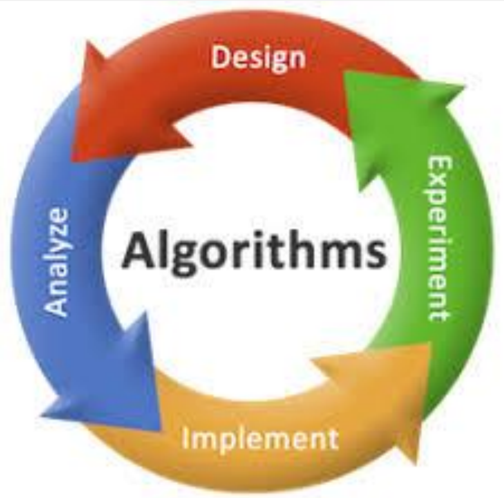


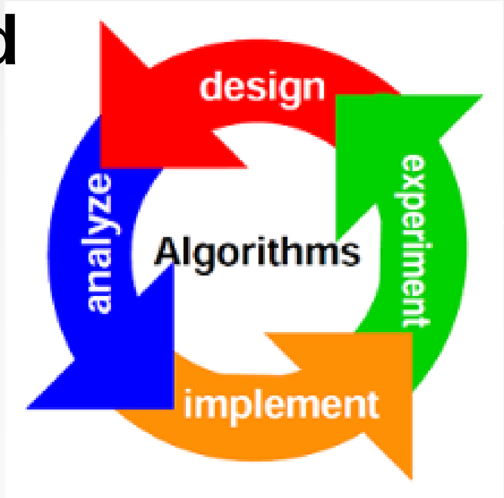


G-6

Branch-and-Bound

- General Method
- Applications
  - Travelling Sales Person Problem
  - 0/1 Knapsack Problem
    - LC Branch-and-Bound Solution
    - FIFO Branch-and-Bound Solution.





Branch-and-Bound – General Method

- The design technique known as **Branch-and-Bound** is very similar to backtracking method in that it searches a tree model of the solution space and is applicable to a wide variety of discrete combinatorial problems.
- Branch-and-Bound refers to all problems dealing with state space search methods in which all children of E-Node (Exposed node) are generated, before any other **Live Node** can become E-Node.
- **Solution states** are those problem states **'s'** for which the path from the root to **'s'** defines a tuple in the solution space.
- The leaf nodes in the combinatorial tree are the solution states.
- **Answer states** are those solution states **'s'** for which the path from the root to **'s'** defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem.

Branch-and-Bound – General Method

- 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 a **Live Node.**
- The **Live Node** whose children are currently being generated is called the **E-Node** (node being expanded).
- A **Dead Node** is a generated node, which is not to be expanded further or all of whose children have been generated.
- **Bounding functions** are used to kill live nodes without generating all their children.
- The term Branch-and-Bound refers to all state space search methods in which all children of the E-Node are generated before any other live node can become the E-node.

Branch-and-Bound – General Method

- We know already two graph search strategies
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
- In these methods, exploration of new node can not begin, until the node currently being exposed is fully explored.
- In Branch & Bound terminology, BFS is called as FIFO Search, as the list of live nodes is a FIFO List (Queue).
- DFS Search (D-Search) is called as LIFO (Last In First Out) Search, as the list of live nodes is a list of LIFO (Stack).
- Similar to backtracking, Bounding functions are applied to avoid generation of subtrees that do-not contain in answer node.

Branch-and-Bound – General Method

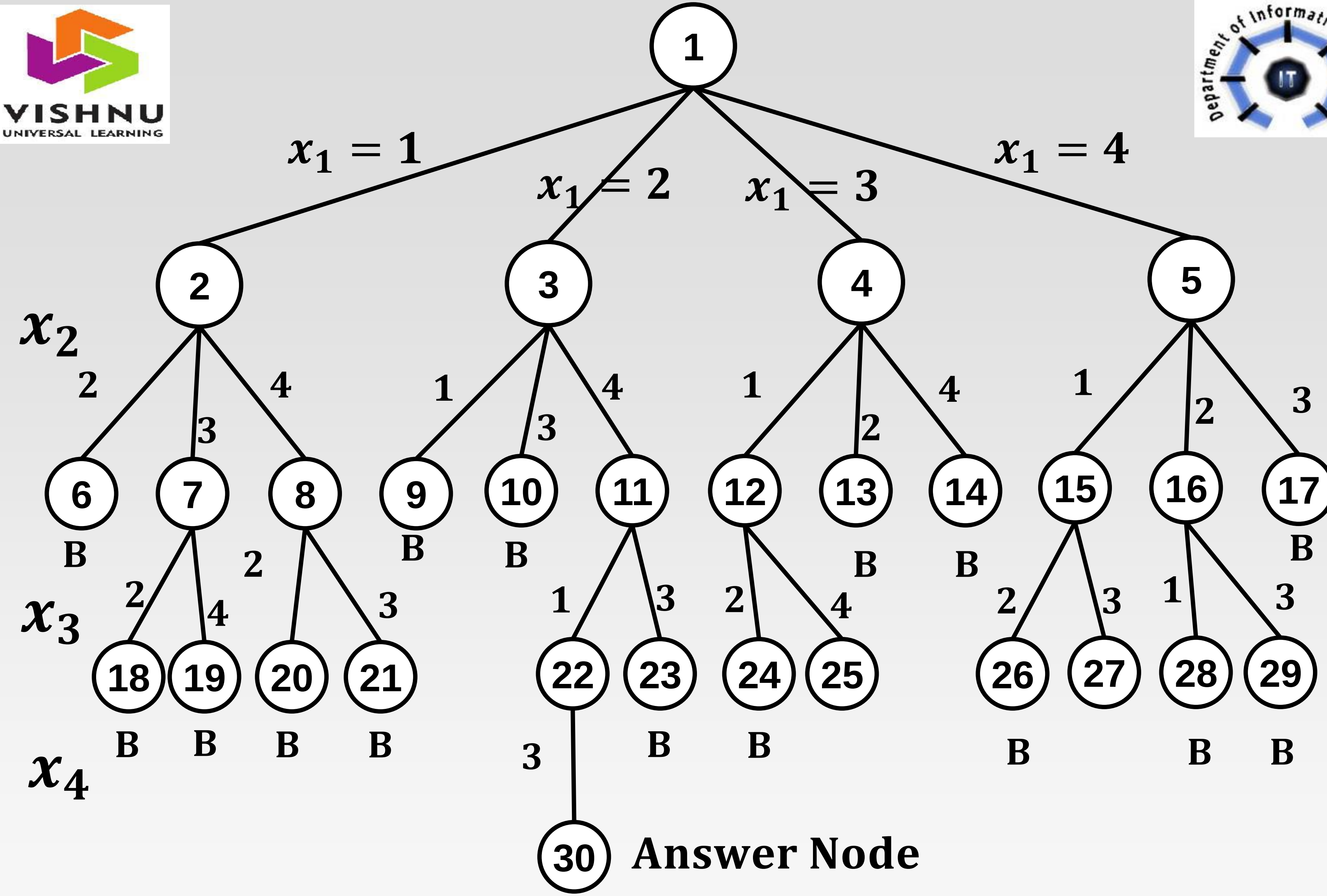
- The branch-and-bound algorithms search a tree model of the solution space to get the solution.
- However, this type of algorithms is oriented more toward optimization.
- An algorithm of this type specifies a real-valued cost function for each of the nodes that appear in the search tree.
- Usually, the goal here is to find a configuration for which the cost function is minimized.
- The branch-and-bound algorithms are rarely simple.
- They tend to be quite complicated in many cases.

Example 4-Queens Problem

- Let us see how a FIFO branch-and-bound algorithm would search the state space tree for the 4-queens problem.
- Initially, there is only one live node, Node No 1.
- This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node.
- It is expanded and its children, nodes 2, 3, 4 and 5 are generated.
- These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively.
- The only live nodes 2, 3, 4, and 5. If the nodes are generated in this order, then the next E-node are node 2.
- It is expanded and the nodes 6, 7, and 8 are generated.
- Node 6 is immediately killed using the bounding function.
- Nodes 7 and 8 are added to the queue of live nodes.

Example 4-Queens Problem

- Node 3 becomes the next E-node.
- Nodes 9, 10, and 11 are generated.
- Nodes 9 and 10 are killed as a result of the bounding functions. Node 11 is added to the queue of live nodes.
- Now the E-node is node 4.
- Figure shows the portion of the tree of previous Figure that is generated by a FIFO branch-and-bound search.
- Nodes that are killed as a result of the bounding functions are a "B" under them.
- At the time the answer node, node 30, is reached, the only live nodes remaining are nodes 25 and 27.



Portion of 4-Queens state space generated by FIFO Branch-and-Bound

Least Cost (LC) Search

- In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind.
- The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- Thus, in the previous 4 Queens Problem Example, when node 22 is generated, it should have become obvious to the search algorithm that this node will lead to answer node in one move.
- However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 22 was expanded.

Least Cost (LC) Search

- The search for answer node can be speeded by using "intelligent" ranking function $\hat{c}(.)$ for live nodes.
- The next *E*-node is selected on the basis of this ranking function.
- If in the 4-queens example we use a ranking function that assigns node 22 a better rank than all other live nodes, then node 22 will become E-node, following node 11.
- The remaining live nodes will never become E-nodes as the expansion of node 22 results in the generation of an answer node (node 30).
- The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node.

Least Cost (LC) Search

- For any node x, this cost could be
  1) The number of nodes on the sub-tree x that need to be generated before any answer node is generated or,
  2) The number of levels the nearest answer node (in the sub-tree x) is from x.
- Using cost measure (2), the cost of the root of 4 Queens problem tree is 4 (node 30 is four levels from node 1).
- The costs of nodes 3 and 4, 11 and 12, and 22 and 25 are respectively 3, 2, and 1.
- The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1.
- Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 3, 11, and 22 (in that order).
- The only other nodes to get generated are nodes 2, 4, 5, 9, 10, 23, and 30.

Least Cost (LC) Search

- The difficulty of using the ideal cost function is that computing the cost of a node usually involves a search of the sub-tree x for an answer node.
- Hence, by the time the cost of a node is determined, that sub-tree has been searched and there is no need to explore x again.
- In LC Searches, a cost function $\mathbf{c}(x)$ for an answer node x is defined as cost of reaching x from the root of the state space tree.
- If x is not an answer node, then $c(x) = \infty$ providing the subtree x contains no answer node; otherwise $\mathbf{c}(x)$ equals the cost of a minimum cost answer node in the subtree x.

0/1 Knapsack Problem

- We have seen various solutions (using Greedy Method & Dynamic Programming) to Knapsack Problem in previous Units, let us see one more way of solving this problem specially 0/1 Knapsack using Branch-and-Bound method.
- To use Branch-and-Bound technique to solve any problem, we have to construct state space tree for the problem.
- Branch-and-bound always based on minimization function, but 0/1 Knapsack problem refers to maximization problem (i.e. maximizing the profit).
- This difficulty is easily overcome by replacing objective function $\sum \boldsymbol{p_i x_i}$ by the function $-\sum \boldsymbol{p_i x_i}$
- Clearly $\sum \boldsymbol{p_i x_i}$ is maximized iff $-\sum \boldsymbol{p_i x_i}$ is minimized.

0/1 Knapsack Problem

- So, the modified 0/1 Knapsack problem is

Minimize $-\sum_{i=1}^{n} p_i x_i$

Subjected to $\sum_{i=1}^{n} w_i x_i \leq m$ $where\ x_i = 0\ or\ 1, for\ 1 \leq i \leq n$

- We continue the discussion assuming a fixed tuple size formation for the solution space (Eg: {1, 1, 0, 1}), this can be easily extended to variable tuple size formulation.
- Every leaf node in the state space tree representing an assignment for which $\sum_{i=1}^{n} w_i x_i \leq m$ is an answer node.
- All other leaf nodes are infeasible.
- For a minimum-cost answer node to correspond to any optimal solution, we need to define $\mathbf{c}(\mathbf{x}) = -\sum_{i=1}^{n} p_i x_i$ for every answer node x.
- The cost $c(x) = \infty$ for infeasible leaf nodes.
- Two bound function have to be used for Knapsack Problem

```
Algorithm Bound (cp, cw, k)
//cp is the current profit total, cw is the current weight.
//total; k is the index of next item, & m is 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 + ((m−c−w[i])/m) * p[i];
    }
   return b
}
```

Function $\hat{c}(x)$ for Knapsack Problem

```
Algorithm UBound (cp, cw, k, m)
//cp is the current profit total, cw is the current weight.
//total; k is the index of next item, & m is knapsack size.
//w[i] and p[i] are the weight and profit of i^th object.
{
    b := cp;  c := cw;
    for i := k + 1 to n do
    {
      if (c + w[i] ≤ m) then
      {
      c := c + w[i];
      b := b − p[i];
    }
   }
  return b
}
```

Function u(x) for Knapsack Problem

Least Cost Branch-and-Bound (LCBB) Solution to Knapsack Problem

- Consider the knapsack instance $\boldsymbol{n = 4,\ m = 15}$

$$(\boldsymbol{w_1, w_2,\ w_3, w_4}) = (\boldsymbol{2, 4, 6, 9}) \text{ and}$$
$$(\boldsymbol{p_1, p_2,\ p_3,\ p_4}) = (\boldsymbol{10, 10, 12, 18})$$

- Let us trace working of an LC Branch-and-Bound search using Cost $\hat{c}(.)$ and Upper Bound $\boldsymbol{u}(.)$
- The search begins with the root node as **E-Node**.
- For this node, node 1, we have cost as $\hat{c}(\boldsymbol{1}) = -\boldsymbol{38}$ **and** $\boldsymbol{upper\ bound\ as\ u(1) = -32.}$ As we converted Maximization Problem to Minimization, upper bound is -32 is larger than -38
- Let us see first, how value of $\boldsymbol{u(1)\ and}\ \hat{c}\ (\boldsymbol{1})$ is calculated.
- These values can be calculated in two ways – Using Formula or using Bound & UBound Functions.

- Let us see first, how value of $\boldsymbol{u}(\mathbf{1})$ $\boldsymbol{and}$ ĉ $(\mathbf{1})$ is calculated using Formula.
- $\boldsymbol{The\ upper\ bound\ is\ calculated\ as\ Sum\ of\ all\ Profits}$.

  $\boldsymbol{u} = -\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}} \boldsymbol{p_i x_i} \quad \leq \boldsymbol{m}$ Here values taken without Fraction.
- Similar way we will find **Cost** ĉ

  $\hat{c} = -\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}} \boldsymbol{p_i x_i}$ Here values taken by considering Fractions.

(In our solution we are not going to include Fractions, only for Computation purpose we are using fractions)

- So the $\boldsymbol{upper\ bound\ value\ of\ node\ 1}$,

$\boldsymbol{u}(\mathbf{1}) = -(\mathbf{10} + \mathbf{10} + \mathbf{12}) = -\mathbf{32}$ Profits of Weight 1, 2 & 3.

We can't take 4th as it will exceed the capacity of Knapsack.

- **Simliarly** Cost ĉ $\boldsymbol{value\ of\ node\ 1}$,

$$\hat{c}(\mathbf{1}) = -\left(\mathbf{10} + \mathbf{10} + \mathbf{12} + \frac{\mathbf{3}}{\mathbf{9}} * \mathbf{18}\right) = -(\mathbf{10} + \mathbf{10} + \mathbf{12} + \mathbf{6}) = -\mathbf{38}$$

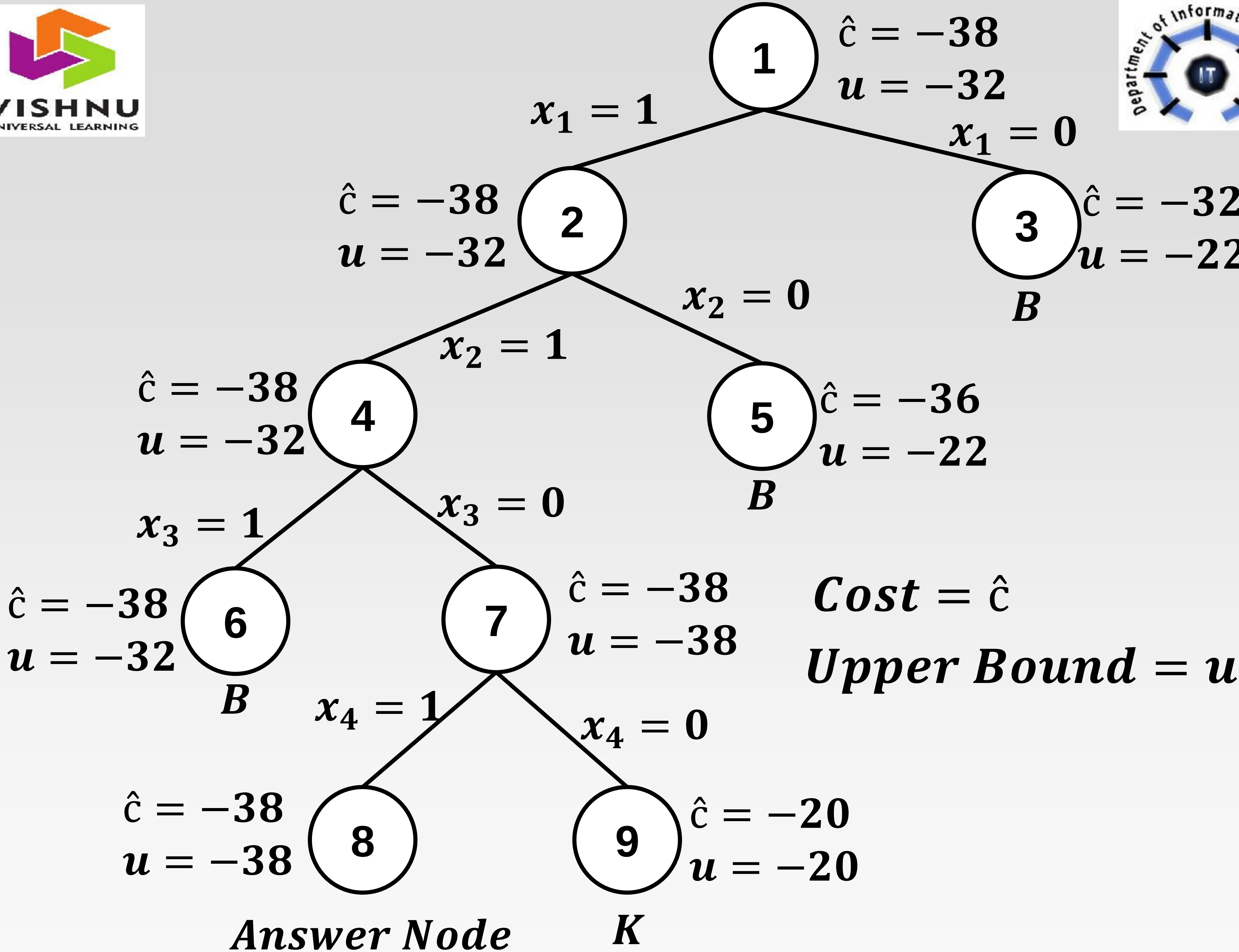
Profits of Weight 1, 2 & 3 as whole & fraction (3/9) of 4th weight to fill Knapsack Completely.

LC Branch-and Bound Solution

- Let us see second way for calculating value of $u(1)$ *and* $\hat{c}\,(1)$ using **UBound** and **Bound** Function.
- The upper bound $u(1)$ has a value Ubound(0, 0, 0, 15).
- UBound scans objects through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered.
- At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned.
- In Function **UBound**, c and b start with a value of Zero.
- For $i = 1, 2, and\ 3,\ \ c\ gets\ incremented\ by\ 2, 4, and\ 6$, respectively.
- When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32.

LC Branch-and Bound Solution

- The Function Bound is similar to UBound, except that is also considers a fraction of the first object that doesn’t fit the knapsack.
- For example, in computing Cost $\hat{c}(\mathbf{1})$, the first object that doesn’t fit the knapsack is $\mathbf{4}$ whose weight is $\mathbf{9}$.
- The total weight of the objects $\mathbf{1}, \mathbf{2},$ and $\mathbf{3}$ is $\mathbf{12}$.
- So, Bound considers a fraction $\frac{\mathbf{3}}{\mathbf{9}}$ of the object $\mathbf{4}$ and hence returns $-\mathbf{32}\ -\frac{\mathbf{3}}{\mathbf{9}} * \mathbf{18} = -\mathbf{38}$.


1
$\hat{c} = -38$
$u = -32$
$x_1 = 1$
$x_1 = 0$
2
$\hat{c} = -38$
$u = -32$
3
$\hat{c} = -32$
$u = -22$
B
$x_2 = 1$
$x_2 = 0$
4
$\hat{c} = -38$
$u = -32$
5
$\hat{c} = -36$
$u = -22$
B
$x_3 = 1$
$x_3 = 0$
6
$\hat{c} = -38$
$u = -32$
B
7
$\hat{c} = -38$
$u = -38$
$x_4 = 1$
$x_4 = 0$
8
$\hat{c} = -38$
$u = -38$
Answer Node
9
$\hat{c} = -20$
$u = -20$
K
$Cost = \hat{c}$
$Upper\ Bound = u$

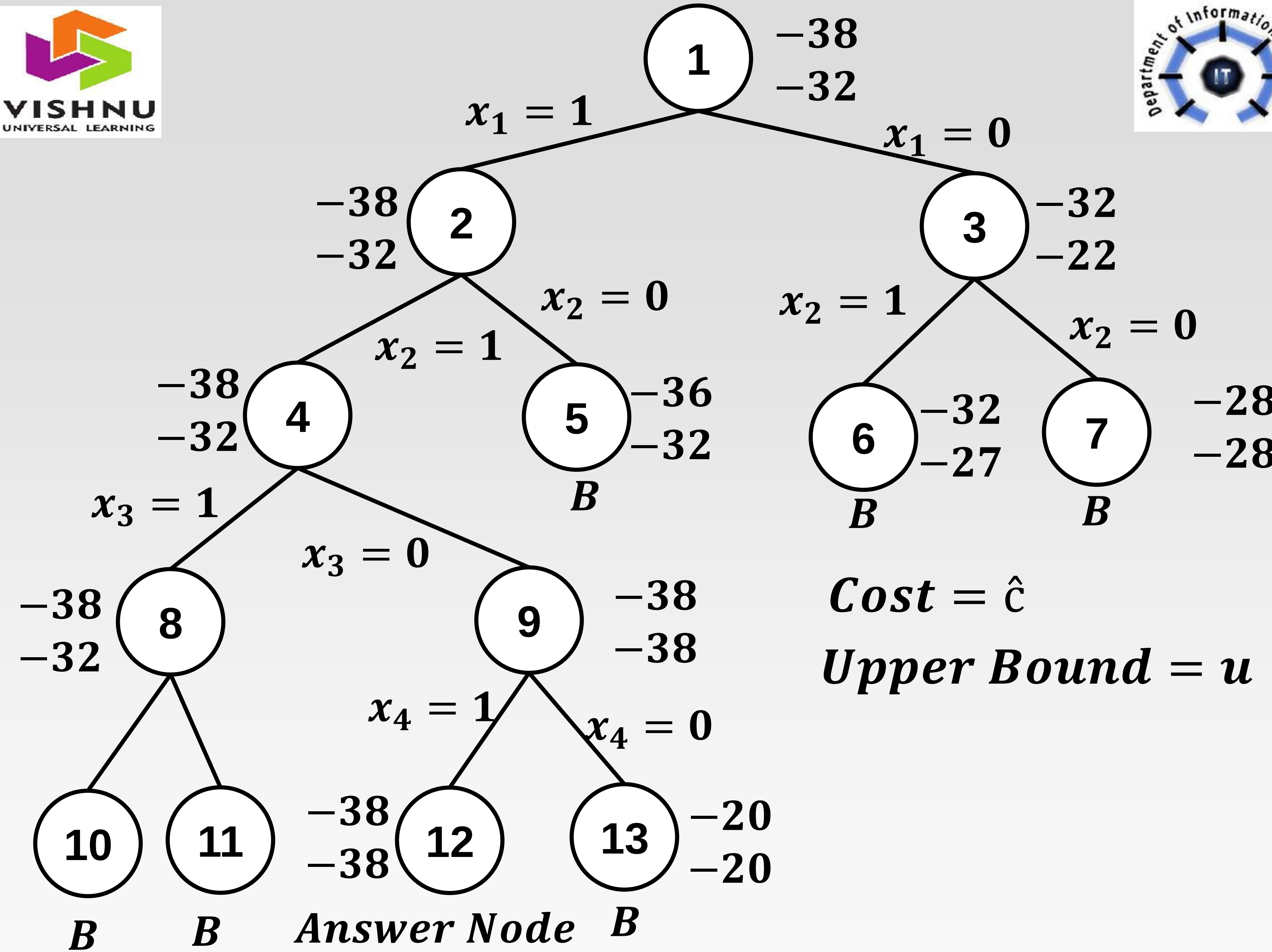
FIFO Branch-and-Bound (FIFOBB) Solution to Knapsack Problem

- Consider the same knapsack instance with $\boldsymbol{n = 4,\ m = 15}$

  $(\boldsymbol{w_1, w_2,\ w_3, w_4}) = (\boldsymbol{2, 4, 6, 9})$ and

  $(\boldsymbol{p_1, p_2,\ p_3,\ p_4}) = (\boldsymbol{10, 10, 12, 18})$

- Let us trace working of an FIFO Branch-and-Bound search using $\hat{c}(.)$ and $u(.)$.
- The search begins with the root node as **E-Node**.
- For this node, node 1 calculation of Cost & Upper Bound is same, so we have $\hat{c}(\mathbf{1}) = \mathbf{-38}$ **and** $\boldsymbol{u}(\mathbf{1}) = \mathbf{-32}$.
- **The difference between LCBB & FIFOBB is that in FIFOBB we are exploring the nodes in order of creation of nodes whereas in LCBB we are exploring nodes based on nodes with minimum cost in all nodes.**


1
−38
−32
$x_1 = 1$
$x_1 = 0$
2
−38
−32
3
−32
−22
$x_2 = 1$
$x_2 = 0$
$x_2 = 1$
$x_2 = 0$
4
−38
−32
5
−36
−32
B
6
−32
−27
B
7
−28
−28
B
$x_3 = 1$
$x_3 = 0$
8
−38
−32
9
−38
−38
$x_4 = 1$
$x_4 = 0$
10
B
11
B
12
−38
−38
Answer Node
13
−20
−20
B
$Cost = \hat{c}$
$Upper\ Bound = u$

0/1 Knapsack Problem Exercise

- Solve the following Knapsack Problems with state space tree by using LCBB.
- $n = 5,\ m = 12$

$$(w_1, w_2,\ w_3, w_4, w_5) = (4, 6, 3, 4, 2) \text{ and}$$

$$(p_1, p_2, p_3, p_4,, p_5) = (10, 15, 6, 8, 4)$$

- $n = 5,\ m = 15$

$$(p_1, p_2, p_3, p_4,, p_5) = (w_1, w_2, w_3, w_4, w_5) = (4, 4, 5, 8, 9)$$

Travelling Salesperson Problem (TSP)

- We have seen solution to Travelling Salesperson Problem using Dynamic Programming in previous Unit, let us see solution to this problem using branch-and-bound method.
- Compared to Dynamic Programming $\boldsymbol{O}(\boldsymbol{n}^2 \mathbf{2}^{\boldsymbol{n}})$, Branch-and-Bound method's good bounding functions will solve some problem instances in much less time.
- Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a directed graph with edge cost $\boldsymbol{c_{ij}}$.
- Let $\boldsymbol{c_{ij}}$ equal to cost of edge $\langle \boldsymbol{i}, \boldsymbol{j} \rangle$, and $\boldsymbol{c_{ij}} = \infty$, if $\langle \boldsymbol{i}, \boldsymbol{j} \rangle \notin$ **E**.
- Let $|\boldsymbol{V}| = \boldsymbol{n}$ (i.e. number of nodes) and assume $\boldsymbol{n} > \mathbf{1}$
- We assume, a tour of G is a directed cycle that starts & end at vertex 1 and include every vertex in $\boldsymbol{V}$.
- The traveling salesperson problem is to find a tour of minimum cost.

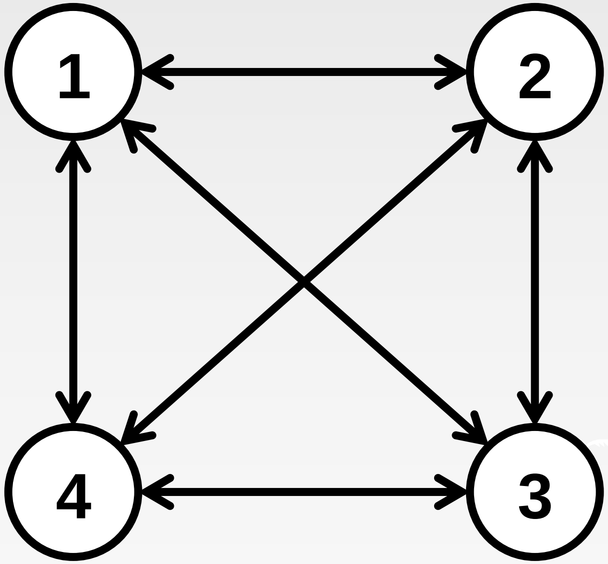
Travelling Salesperson Problem

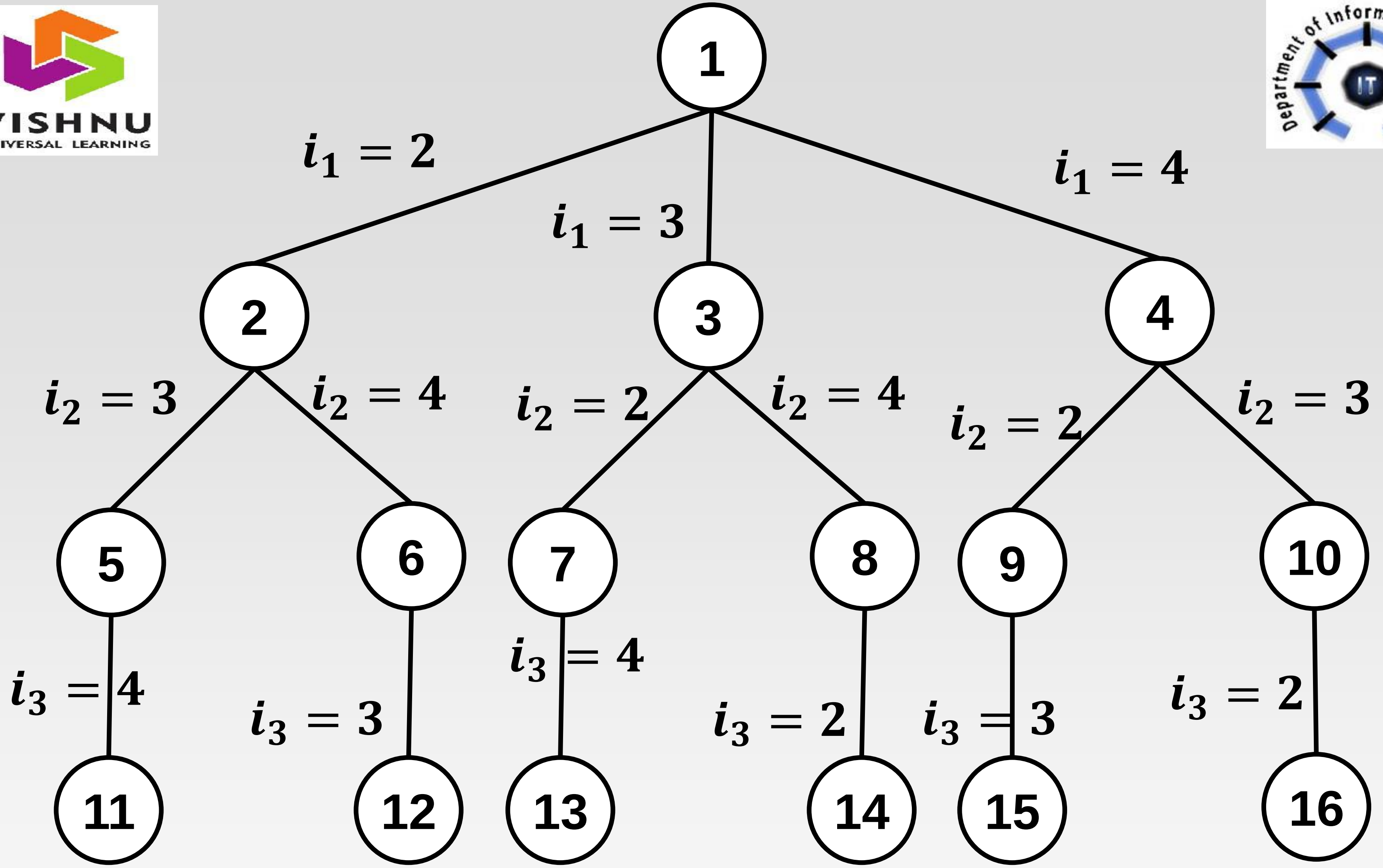
- So, the solution space **S** is given by

  $\boldsymbol{S = \{1, \pi, 1 \mid \pi\ is\ a\ permutation\ of\ (2, 3, \ldots., n)\}}$
- Then $|\boldsymbol{S}| = (\boldsymbol{n-1})!$
- The size of S can be reduced by restricting S so that

$$\boldsymbol{(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S\ \ iff\ \langle i_j, i_{j+1}\rangle \in E, 0 \le j \le n-1,}$$
$$\boldsymbol{and\ i_0 = i_n = 1}$$

- S can be organized into state space tree.
- The figure (next slide) shows the solution space tree organization for case of a complete graph as in figure with $\boldsymbol{|V| = 4\ i.e.\, n = 4.\, Assume\ \ i_0 = i_n = 1}$





State space tree for Travelling Salesperson Problem with $n = 4$ *and* $i_0 = i_4 = 1$

Travelling Salesperson Problem

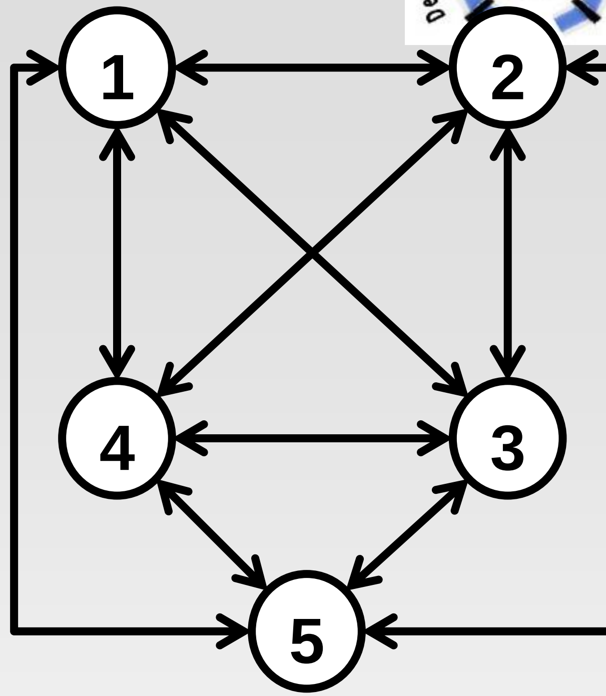
- For Least Cost Search Branch-and-Bound (LCBB), the cost function is defined as $\boldsymbol{c(x)}$ such that the solution node with least cost function corresponds to a shorter tour in G. In general

$$\boldsymbol{c(A)} = \begin{cases} \boldsymbol{length\ of\ tour\ defined\ by\ the\ path\ from\ the\ root\quad to\ A, if\ A\ is\ leaf} \\ \boldsymbol{cost\ of\ a\ minimum-cost\ leaf\ in\ the\ subtree\ A, if\ A\ is\ not\ a\ leaf} \end{cases}$$

- A simple $\hat{c}(.)$ **such that** $\hat{c}(\boldsymbol{A}) \leq \boldsymbol{c(A)}$ for all $\boldsymbol{A}$ obtained by defining $\hat{c}(\boldsymbol{A})$ to be the length of the path defined at node $\boldsymbol{A}$.
- A better $\hat{c}(\boldsymbol{A})$ can be obtained by using **reduced cost matrix** corresponding to G.
- A row (column) is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.
- A matrix is reduced cost matrix iff every row and column is reduced.

Travelling Salesperson Problem

- As an example, consider directed graph with five vertices i.e. $\boldsymbol{n = 5}$ and cost matrix of a graph G.



$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(1) Cost Matrix

- Since every tour on this graph includes exactly one edge $\langle \boldsymbol{i}, \boldsymbol{j} \rangle$ with $\boldsymbol{i = k, 1 \leq k \leq 5}$, subtracting a constant $\boldsymbol{t}$ from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly $\boldsymbol{t}$.
- A minimum-cost tour remains a minimum-cost tour following this subtraction operation.

Travelling Salesperson Problem

- If t is chosen to be the minimum entry in row i (column j), then subtracting it from all entries in row i (column j) introduces a zero into row i (column j).
- Repeating this as often as needed, the cost matrix can be reduced.
- The total amount subtracted from the columns and rows is a lower bound on the length of a minimum-cost tour and can be used as the ĉ value for the root of the state space tree.
- Subtracting $\mathbf{10, 2, 2, 3, 4, 1}$, and $\mathbf{3}$ from rows $\mathbf{1, 2, 3, 4}$, and $\mathbf{5}$ and columns $\mathbf{1}$ and $\mathbf{3}$ respectively of matrix yields the reduced matrix as shown below (next slide).
- The total amount subtracted is (**Reduced Cost**) **25**, hence, all tours in the original graph have a length at least **25**.

Travelling Salesperson Problem

- Row reduction : $\boldsymbol{R1-10},\ \boldsymbol{R2-2},\ \boldsymbol{R3-2},\ \boldsymbol{R4-3},\ \boldsymbol{R5-4}$ given reduced cost matrix of a graph G as.

(2) Reduced (Row Deduction) Cost Matrix

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

- Column reduction : $\mathbf{C1-1},\ \boldsymbol{C3-3}$ given reduced cost matrix of a graph G as.

(3) Reduced Cost Matrix L=25

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Remember this Reduced Cost Matrix, we will use this for few starting calculations.

Travelling Salesperson Problem

- Let is trace the progress of the LCBB algorithm on the problem instance of **figure (1)**.
- We will use $\mathbf{Cost}\ \hat{c}$ as done previously, starting with node 1 (First Matrix).
- The cost $\mathbf{Cost(1)} = \hat{c}(\mathbf{1}) = \mathbf{Reduced\ Cost} = \mathbf{25}$
- The initial reduced cost matrix is as shown in **figure (3)** in previous slide will be used for initial calculations and we assume $\boldsymbol{upper} = \infty$**.** (We are not going to use upper value during calculations, we can update this value as and when required).
- The portion of the state space tree that gets generated is shown in figure (next slide).
- Starting with the root node as **E-Node**, nodes **2, 3, 4, and 5** are generated.

Travelling Salesperson Problem

- Cost of any node $\boldsymbol{x}$ can be calculated by using formula

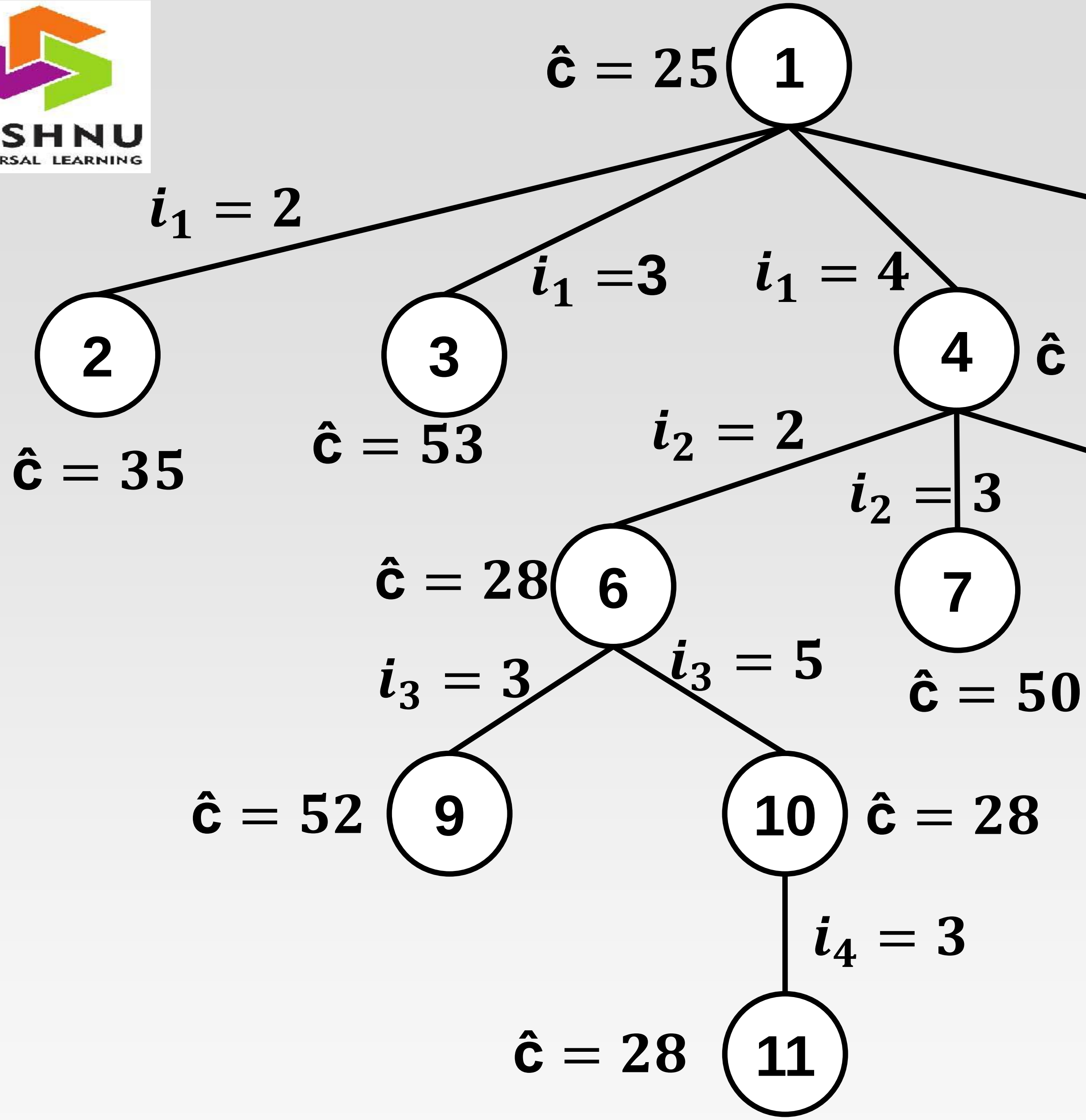
$$\boldsymbol{cost}(\boldsymbol{x}) = \hat{c}(\boldsymbol{x}) = \boldsymbol{cost}\langle \boldsymbol{i}, \boldsymbol{j} \rangle + \hat{c}(\boldsymbol{i}) + \hat{\boldsymbol{r}}$$

- Here $\boldsymbol{cost}\langle \boldsymbol{i},\ \boldsymbol{j} \rangle$ is value of edge $\langle \boldsymbol{i},\ \boldsymbol{j} \rangle$ from Cost Matrix.
- $\hat{c}(\boldsymbol{i})$ is Cost of Parent node of $\boldsymbol{x}$ & $\hat{\boldsymbol{r}}$ is the sum of reduction values for cost matrix.
- Rules to compute reduced cost $\hat{\boldsymbol{r}}$ for selected edge $\langle \boldsymbol{i},\ \boldsymbol{j} \rangle$:

1) In the reduced cost matrix of parent node $\boldsymbol{i}$, make $\boldsymbol{i^{th}}\ \boldsymbol{row}\ \infty$
2) Also make $\boldsymbol{j^{th}}\ \boldsymbol{column}\ \infty$
3) Make the entry $\langle \boldsymbol{j},\ \boldsymbol{1} \rangle$ as ∞
4) Do row reductions and column reductions, if possible.
5) The reduced cost $\hat{\boldsymbol{r}}$ is the sum of new reductions made in step 4.

- If no new reductions made then $\hat{\boldsymbol{r}} = \boldsymbol{0}$
- If new reductions are made then

$$\hat{\boldsymbol{r}} = \boldsymbol{sum\ of\ reduction\ values}.$$



Numbers outside the nodes are ĉ values

State Space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1, 2; Node 2

$$cost(2) = cost\langle 1, 2\rangle + \hat{c}(1) + \hat{r}$$

$$\hat{c}(2) = 10 + 25 + 0 = 35$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1, 3; Node 3

$$cost(3) = cost\langle 1, 3\rangle + \hat{c}(1) + \hat{r}$$

$$\hat{c}(3) = 17 + 25 + 11 = 53$$

$$Here\ we\ have\ done\ C1 - 11$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1, 4; Node 4

$$cost(4) = cost\langle 1, 4\rangle + \hat{c}(1) + \hat{r}$$

$$\hat{c}(4) = 0 + 25 + 0 = 25$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1, 5; Node 5

$$cost(5) = cost\langle 1, 5\rangle + \hat{c}(1) + \hat{r}$$

$$\hat{c}(5) = 1 + 25 + 2 + 3 = 31$$

$$Here\ we\ have\ done\ R1 - 2\ \&\ R4 - 3$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1, 4, 2; Node 6

$$cost(6) = cost\langle 4, 2\rangle + \hat{\mathbf{c}}(4) + \hat{r}$$

$$\hat{\mathbf{c}}(6) = 3 + 25 + 0 = 28$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1, 4, 3; Node 7

$$cost(7) = cost\langle 4, 3\rangle + \hat{\mathbf{c}}(4) + \hat{r}$$

$$\hat{\mathbf{c}}(7) = 12 + 25 + 11 + 2 = 50$$

$$Here\ we\ have\ done\ C1 - 11\ \&\ R3 - 2$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1, 4, 5; Node 8

$$cost(8) = cost\langle 4, 5\rangle + \hat{\mathbf{c}}(4) + \hat{r}$$

$$\hat{\mathbf{c}}(8) = 0 + 25 + 11 = 36$$

$$Here\ we\ have\ done\ R2 - 11$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1, 4, 2, 3; Node 9

$$cost(9) = cost\langle 2, 3\rangle + \hat{\mathbf{c}}(6) + \hat{r}$$

$$\hat{\mathbf{c}}(9) = 11 + 28 + 2 + 11 = 52$$

$$Here\ we\ have\ done\ R3 - 2\ \&\ R5 - 11$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \mathbf{0} & \infty & \infty \end{bmatrix}$$

(i) Path 1, 4, 2, 5; Node 10

$$\boldsymbol{cost(10) = cost\langle 2,5\rangle + \hat{c}(6) + \hat{r}}$$
$$\boldsymbol{\hat{c}(10) = 0 + 28 + 0 = 28}$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

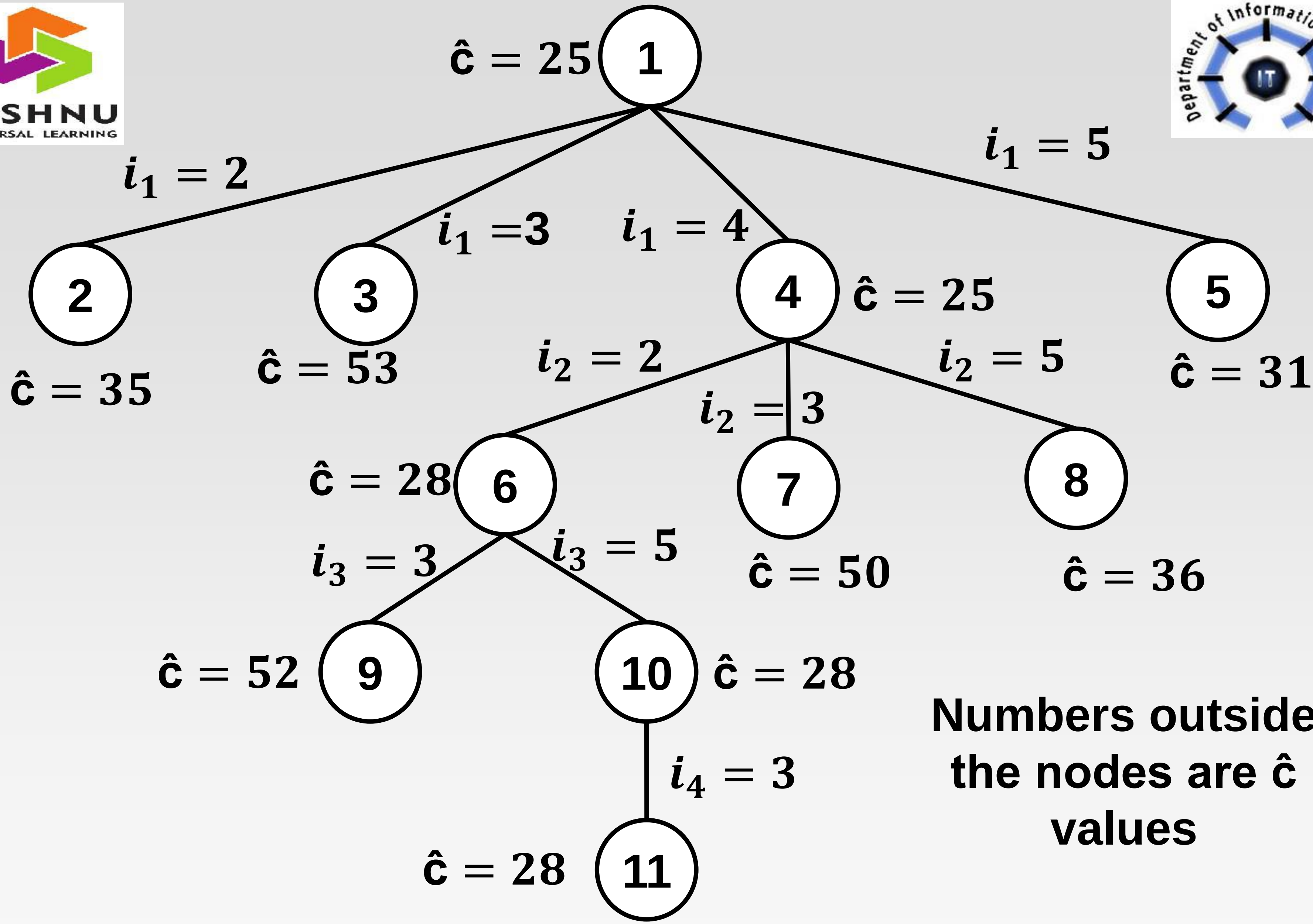
(j) Path 1, 4, 2, 5, 3; Node 11

$$\boldsymbol{cost(11) = cost\langle 5,3\rangle + \hat{c}(10) + \hat{r}}$$
$$\boldsymbol{= 0 + 28 + 0 = 28}$$

- **Tour $\mathbf{1 - 4 - 2 - 5 - 3 - 1}$**
- **Cost = 28**

- **From Cost Matrix G,**

$$\boldsymbol{Cost\ of\ Tour = Cost[(1,4) + (4,2) + (2,5) + (5,3) + (3,1)]}$$
$$\boldsymbol{= 10 + 6 + 2 + 7 + 3 = 28}$$



State Space tree generated by procedure LCBB

Travelling Salesperson Problem Exercise

- Solve the following Travelling Salesperson Problem with state space tree by using LCBB. Show all steps Calculations.

(a)

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

(b)

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

Next - Unit VI

NP-Hard and NP-Complete Problems