UNIT - IV
BACKTRACKING: General method, Applications- n-queen problem, Sum of subsets problem, Graph colouring, 0/1knapsack problem, and 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 technique. In this we search for the set of solutions or optimal solution which satisfies some constraints. One way of solving a problem is by exhaustive search: we enumerate all possible solutions, and see which one produces the optimum result. For eg, for the knapsack problem, we could look at every possible subset objects, and find out one which has the greatest profit value and at the same time not greater than the weight bound. Backtracking is a variation of exhaustive search, where the search is refined by eliminating certain possibilities. Backtracking is usually faster method than an exhaustive search.

In case of greedy and dynamic programming techniques, we will evaluate all possible solutions, among which we select one solution as optimal solution. In backtracking we will get same optimal solution with a less number of steps.

Definition: It is a technique to define the algorithms. By using this technique no need to take the entire solution at a time and checking the given conditions fo the solution.

Take the solution in step by step manner. If the current solution not satisfy the given conditions then backtrack to the previous step and proceed in another possibilities.

- In the backtracking method
  1. The desired solution is expressible as an n tup[le $(x_1,x_2,\ldots x_i)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function $C(x_1,x_2,\ldots x_n)$
- The problem can be categorized into 3 categories:

  For instance – for a problem P let C be the set of constraints for P. Let D be the set

  containing all solutions satisfying C then

  Finding whether there is any feasible solution? – is the decision problem

  What is the best solution? – is the optimization problem

  Listing of all feasible solution – is the enumeration problem.

- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space(i.e set of all feasible solutions) for the given problem.
- Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.
- Explicit constraints are rules, which restrict each vector element to be chosen from the given set. Implicit constraints are rule, which determine which of the tuples in the solution space, actually satisfy the criterion function.

Recursive Backtracking algorithm:

Algorithm Backtrack(k)

//This schema describes the backtracking process 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[k-1]) do
        {
                if (Bk(x[1], x[2],..,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);
                }
        }
}
```

General Iterative Backtracking Method:

Algorithm IBacktrack(n)

//This scheme describes the backtracking process. All solutions are generated in x[1:n] and
//printed as soon as they are determined.

```
{
    k:=1;
    while (k≠0) do
    {
        if (there remains an untried 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;   //consider the next set
        }
        else k:=k-1; //backtrack to the previous set
    }
}
```

APPLICATIONS

n-QUEENS PROBLEM

We can make 8-queens problem generalized by making n queens placed on n x n chessboard. The 8-queens problem can be stated as follows: Consider a chess board of order 8X8. The problem is to place 8 queens on this board such that no two queens can attack each other. That means no two queens can be placed on the same row, column or diagonal. The solution to 8-queens problem can be obtained using backtracking method.

While solving 8-queens problem if we place row by row, we need to check for column conflicts and diagonal conflicts only.

Let position $P_1 = (i, j)$ and $P_2 = (k, l)$ then P_1 and P_2 are on the same diagonal

if $\quad i + j = k + l$ ……(1)

or $\quad i - j = k - l$ ……(2)

eqn(1) implies $\quad j-l = i-k$

eqn(2) implies $\quad j-l = k-i$

P_1 and P_2 are on the same diagonal iff $|j-l| = |i-k|$

For instance: $P_1 = (8,1)$ and $P_2=(1,8)$

$i=8, j=1$, and $k=1, l=8$

$i+j = k+l$

$8+1 = 1+8 = 9$

$j-l = k-i$

$1-8 = 1-8$

Hence P_1 and P_2 are on the same diagonal.

Now we will build a feasible sequence for 8-queens problem. The feasible sequence will be sequence of numbers 1, 2, 3…k where $k \leq 8$

Similarly, $P(i) \neq P(j)$ and $|P(j)-P(i)| \neq j-i$

The procedure to obtain feasible solution is-

Step 1: Add to the sequence numbers which are already not used.

Step 2: When a sequence generated is of length 8 and is a feasible sequence then terminate with solution. If the sequence generated is feasible but if its length is less than 8 then repeat step 1.

Step 3: If the sequence is not feasible then backtrack till the most recent value which gives the feasible sequence.

Step 4: Repeat step 1-3.

Example:

For the following feasible sequence solve 8-queens problem.

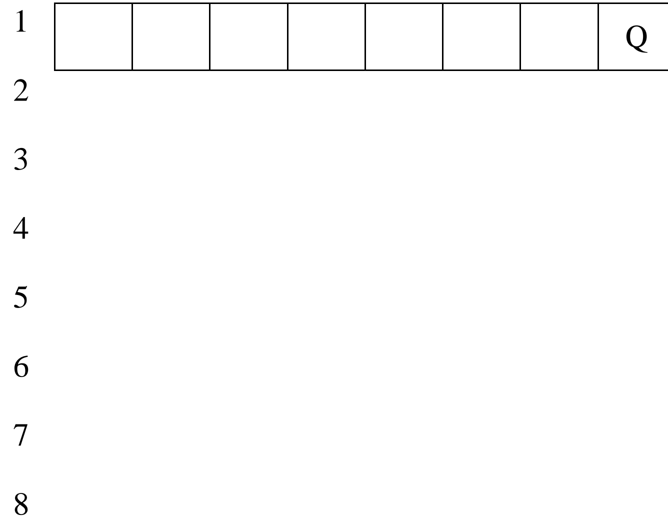
(6, 4, 7,1)

Solution: Here the given sequence represents column position and each queen is placed row by row on given column. That is place 1st queen at (1, 6) and 2nd at (2, 4) and so on..

Queens Positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				Not feasible │1-2│=5-4
6	4	7	1	3				
6	4	7	1	3	2			Not feasible 6-5=3-2
6	4	7	1	3	5			
6	4	7	1	3	5	2		
6	4	7	1	3	5	2	8	List ends, feasible sequence

Hence solution to 8-queens problem is (6, 4, 7, 1, 3, 5, 2, 8)

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



Algorithm: To check can a new queen be placed?

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 forst (k-1) value have been set. Abs[r) returns the absolute value of r.

{

for j:=1 **to** k-1 **do**

if ((x[j] = i) //two in the same column

or (Abs(x[j]-i) = Abs(j-k))) //or in the same diagonal

then return false;

return true;

}

Algorithm: All solutions to the n-queens problem

Algorithm NQueens(k, n)

//Using backtracking, this procedure prints all possible placements of n queens on an nXn
//chessboard so that they are non attacking.

{

for i:= 1 **to** n **do**

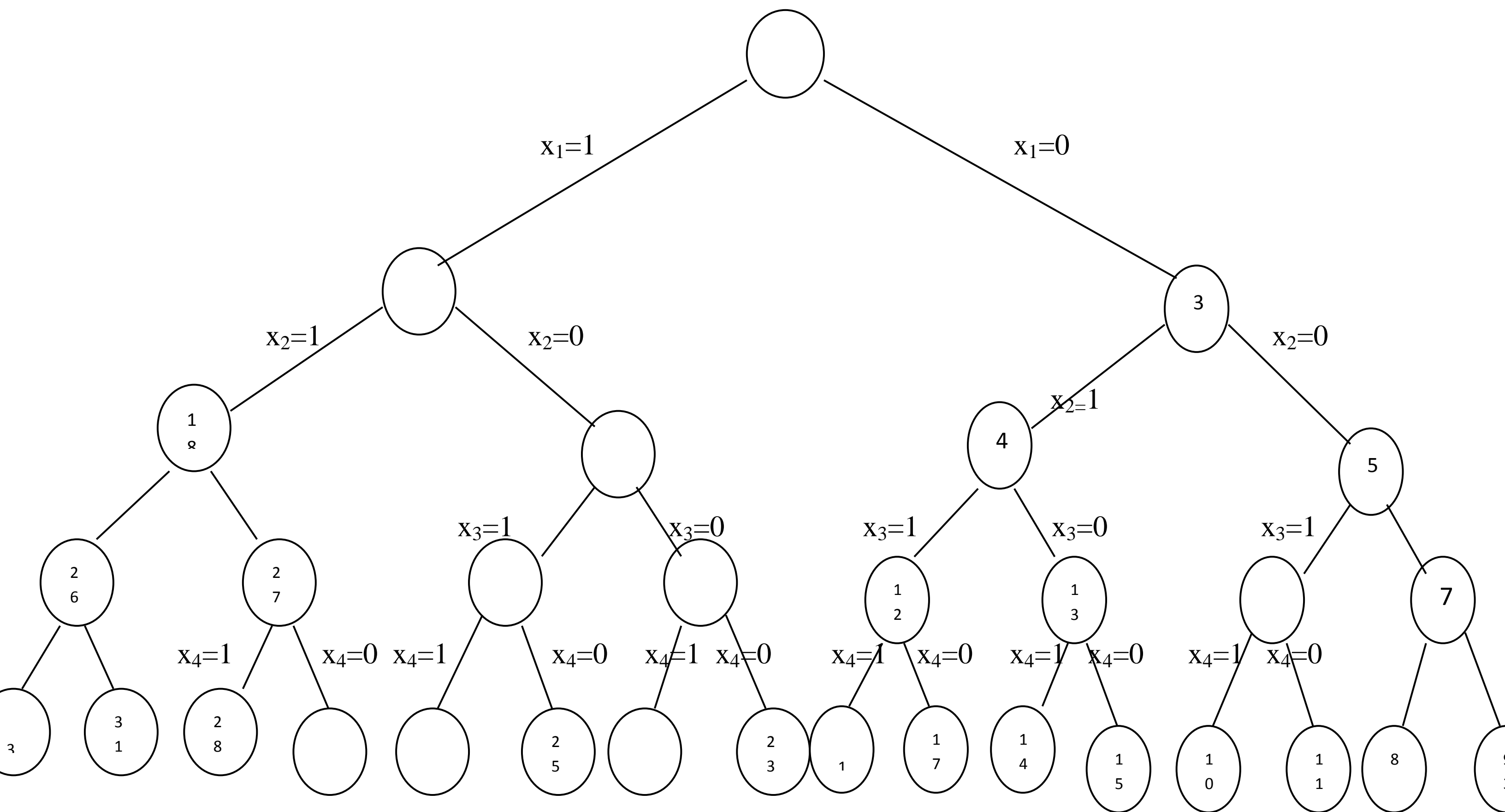
```
    {
       if Place(k, i) then
       {
              x[k] := i;
              if (k=n) then write (x[1:n]);
              else NQueens (k+1, n);
        }
     }
}
```

Exercises:

1. For the feasible sequence (7, 5, 3, 1) solve 8-queens problem using backtracking.
2. For the feasible sequence (8, 5, 4, 3) solve 8-queens problem using backtracking.

SUM OF SUBSETS PROBLEM

Suppose we are given n distinct positive numbers and we desire to find all combinations of these numbers whose sums are m. This is called the sum of subsets problem. In other words, let there be n elements given by the set w = {w_1, w_2,…,w_n} then find out all the subsets from whose sum is m. The sum of subset is based on fixed size tuple.



In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.

The children of any node are easily generated(fig). For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i=0$. The left subtree of root defines all subsets containing w_1 and the right subtree defines all subsets not containing w_1 and so on.

Bounding function is a function which is used to kill live nodes without generating all their children.

A simple choice for bounding function is $B_k(a_1,\ldots a_k)$ is true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m \quad \text{---------(1)}$$

$x_1,\ldots,x_k$ cannot lead to answer if eqn(1) is not satisfied. The bounding function can be strengthened if we assume the w_i's are initially in non decreasing order. In this case $x_1,\ldots,x_k$ cannot lead to an answer node if

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

The bounding function is $B_k(a_1,\ldots a_k)$ = true iff

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

and

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

Algorithm:

Let S be a set of elements and N is the expected sum of subsets. Then

Step 1: Start with an empty set.

Step 2: Add to the subset next element from the list.

Step 3: If the subset is having sum N then stop with that subset as solution.

Step 4: If the subset is not feasible, or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

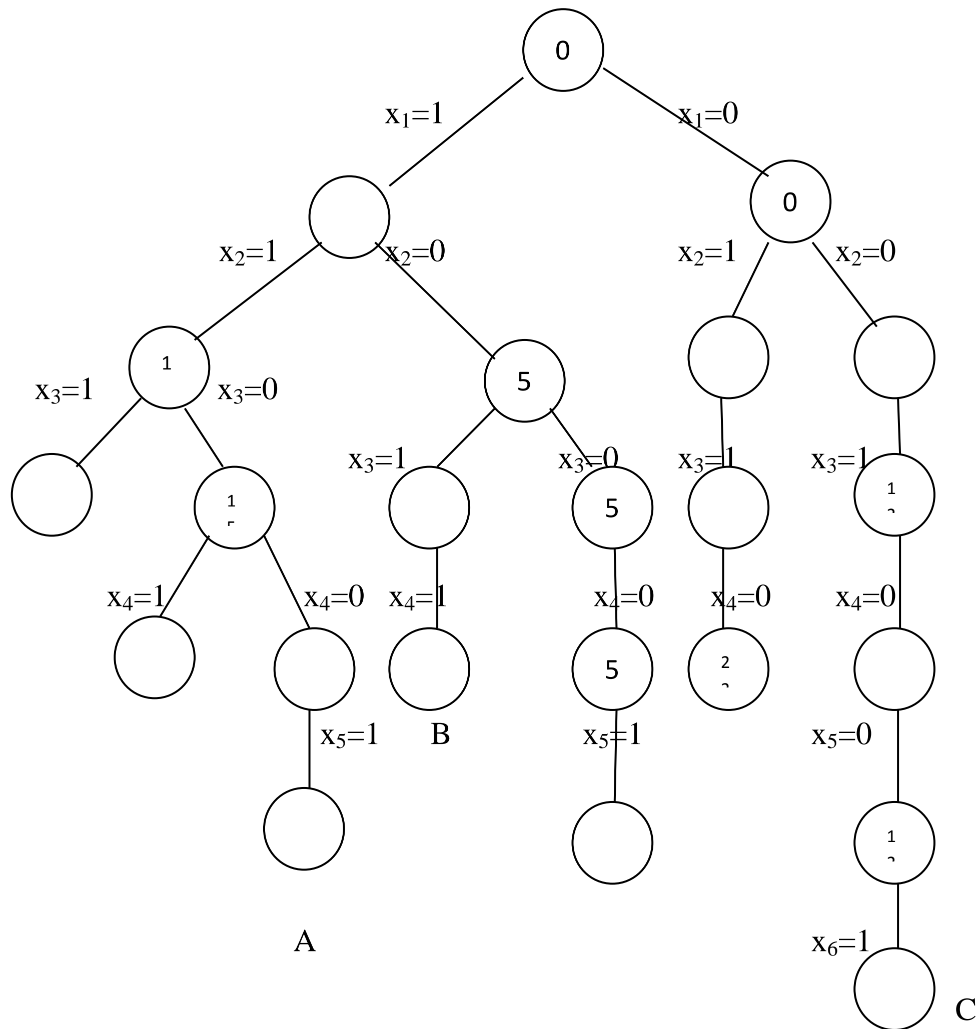
Step 5: If the subset is feasible then repeat step 2.

Step 6: If we have visited all the elements without finding a suitable subset and if no backtracking is possible, then stop with no solution.

Consider a set S = {5, 10, 12, 13, 15, 18} and N = 30

Subset{empty}	Sum	Action initially set is empty
5	5	
5,10	15	
5, 10, 12	27	
5, 10, 12, 13	40	Sum exceeds N=30. Hence backtrack
5, 10, 12, 15		Not feasible
5, 10, 12, 18		Not feasible
5, 10		List ends, backtrack
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible, backtrack
5, 10	15	
5, 10, 15	30	Solution obtained

We can represent various solutions to sum of subset by a state space tree as,



Algorithm: Recursive backtracking algorithm for sum of subsets problem.

Algorithm SumOfSub(s, k, r)

//Find all subsets of w[1:n] that sum to m. The values of x[j] $1 \leq j < k$, have already been determined. $s=\sum^{k-1}_{j=1} w[j] * x[j]$ and $r = \sum^{n}_{j=k} w[j]$. The w[j]'s are in non decreasing order. It is assumed that $w[1] \leq m$ and $\sum^{n}_{i=1} w[i] \geq m$.

{

//Generate left child. Note: $s+w[k] \leq m$ since B_{k-1} is true.

x[k]:=1;

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

```
        //there is no recursive call here as w[j]>0, 1≤j≤n.
        else if (s+w[k]+w[k+1] ≤ m)
                then SumOfSub(s+w[k], k+1, r-w[k]);
        //Generate right child and evaluate B_k
        if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
        {
                x[k] :=0;
                SumOfSub(s, k+1, r-w[k]);
        }
}
```

Exercises:

1. Let w={5, 7, 10, 12, 15, 18, 20} and m=35. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.
2. Let a) w={20, 18, 15, 12, 10, 7, 5} and b) w={15, 7, 20, 5, 18, 10, 12} and m=35. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.
3. Let m=31 and w={7, 11, 13, 24} draw a portions of state space tree. Clearly show the solutions obtained.

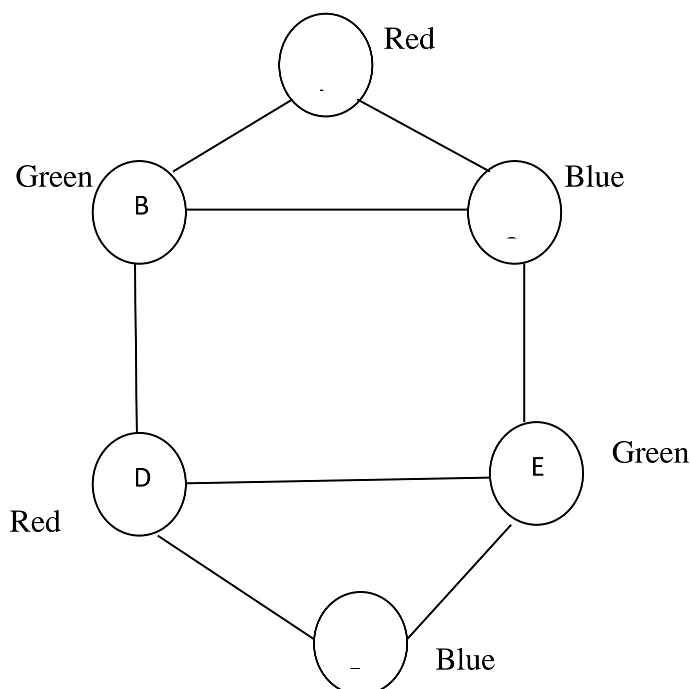
GRAPH COLORING

Let G be a graph and m be a given positive integer. Graph coloring is a problem to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is called the m-colorability decision problem.

If d is the degree of the given graph, then it can be colored with d+1 colors.

The m-colorability problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the **chromatic number** of the graph.

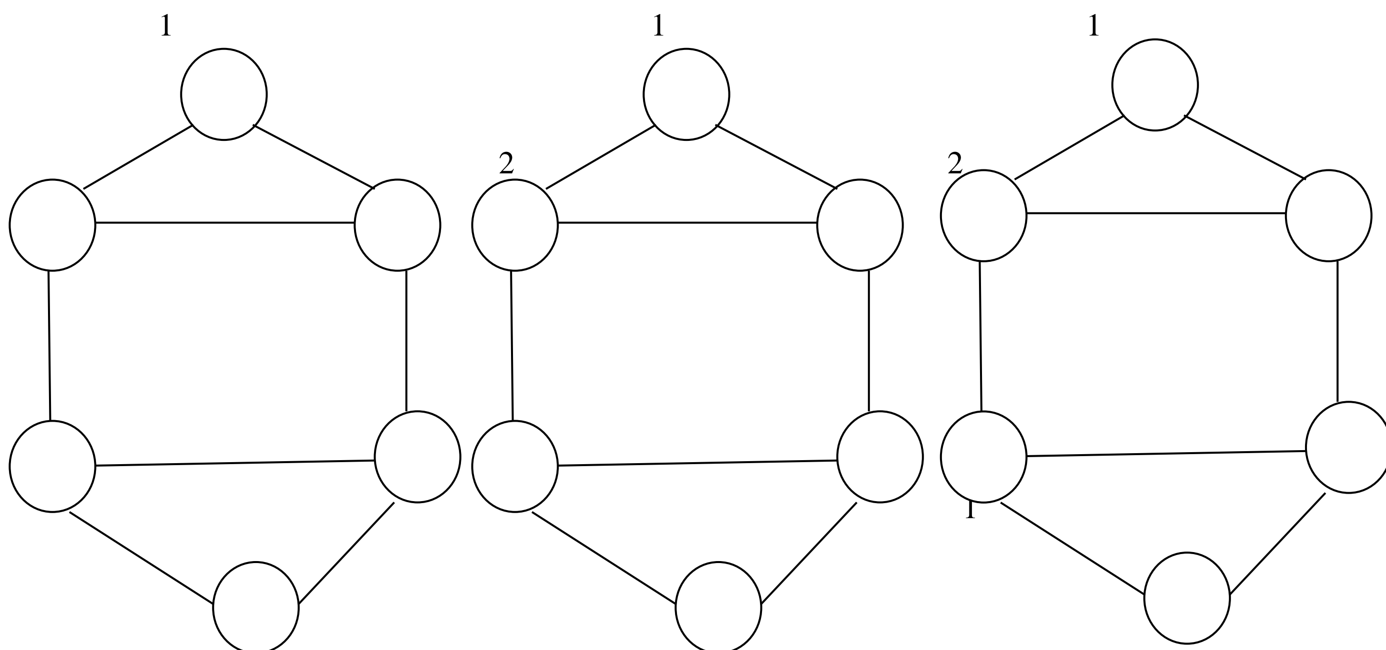
Eg:

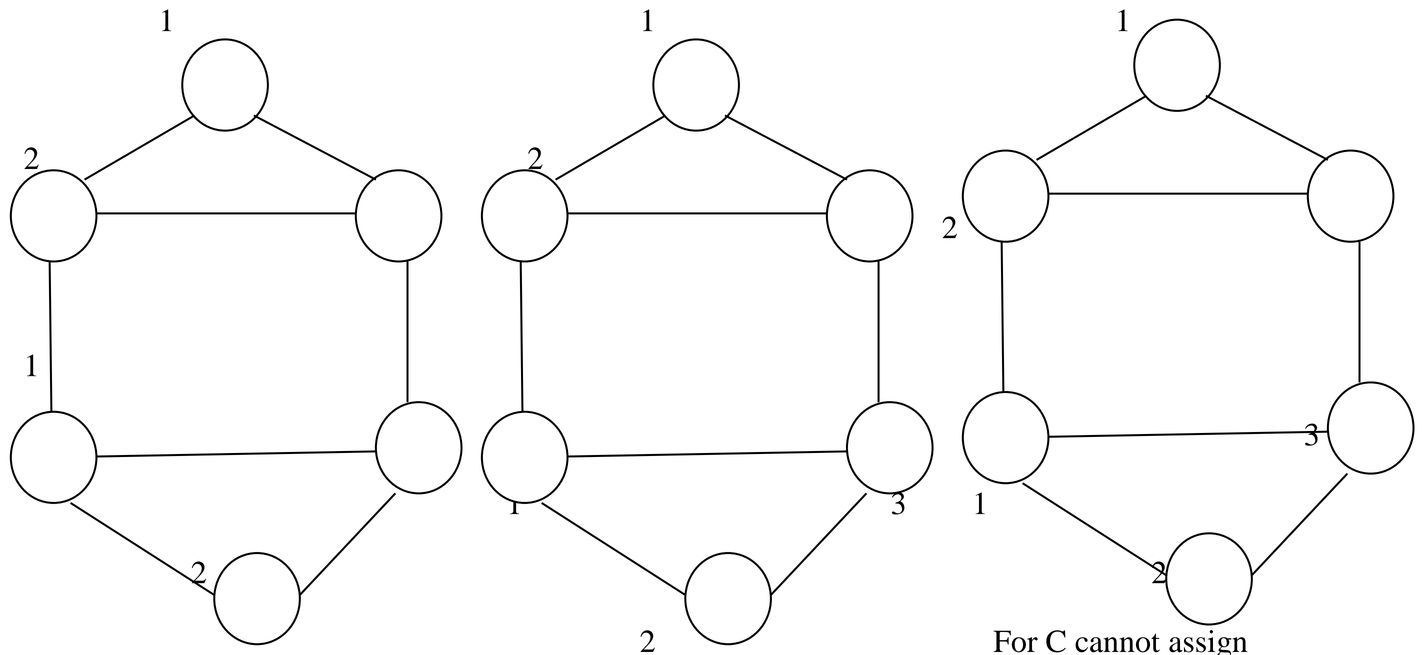


As given in fig, we require 3 colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows-

Step 1: A graph G consists of vertices from A to F. There are three colors used Red, Green, and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.

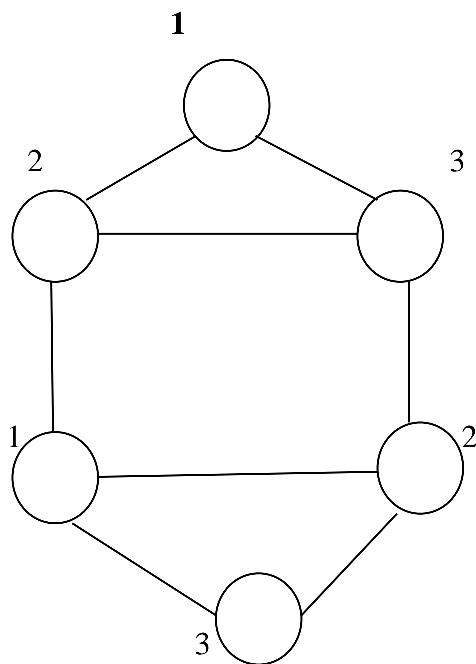
Step 2:





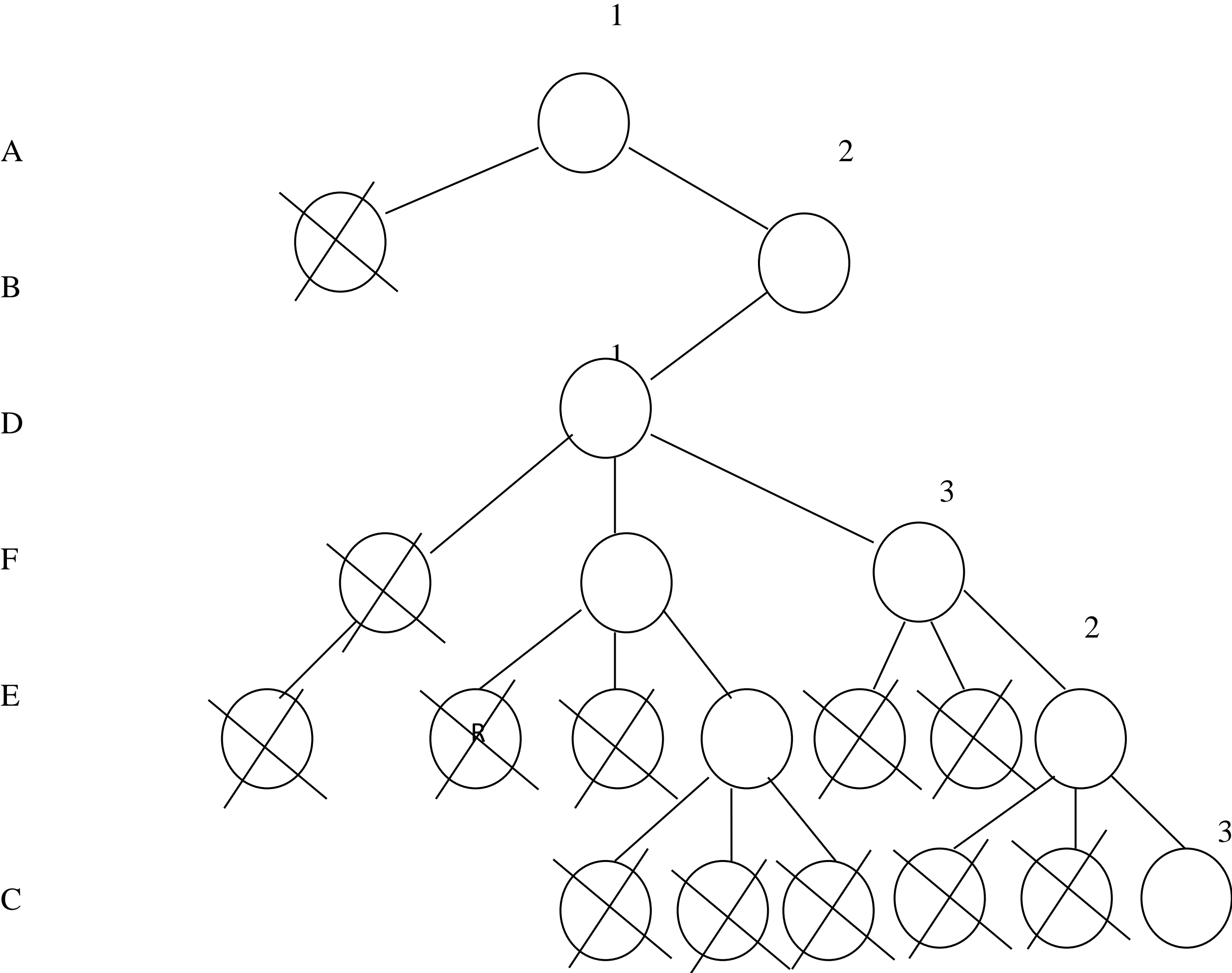
For C cannot assign
1 or 2 or 3. hence
backtrack.

Step 3:



 P. Madhuravani, Assoc.Prof

Thus the graph coloring problem is solved. The state space tree can be drawn for better understanding of graph coloring techniques using backtracking approach.



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 //the graph such that adjacent vertices are assigned distinct integers are printed. k is the index of //the next vertex to color.

{

repeat

{

// Generate all legal assignments for x[k]

NextValue(k); //Assign to x[k] a legal color.

if (x[k] = 0) **then return**; //No new color possible

if (k=n) **then** //At most m colors have been used to color the n vertices.

write (x[1:n]);

else mColoring (k+1);

} **until (false)**;

}

Generating a next color

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 the //next highest numbered color while maintaining distances from the adjacent vertices of vertex k. //If no such color exists, then x[k] is 0.

{

repeat

{

x[k] := (x[k]+1) **mod** (m+1); //Next highest color

if (x[k] = 0) **then return**;

for j:= 1 **to** n **do**

{

//check if this color is distinct from adjacent colors.

if ((G[k, j] ≠ 0) **and** (x[k] = x[j]))

//if (k, j) is and edge and if adj. vertices have the same color.

then break;

}

if (j=n+1) **then return**; //New color found.

} **until (false);** //Otherwise try to find another color.

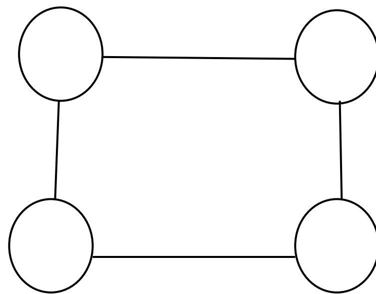
}

Analysis:

Total computing time is $O(nm^n)$.

Exercises:

A 4-node graph and all possible 3-colorings.



0/1 Knapsack:

Given n positive weights wi, n positive profits pi, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

$$\sum_{1 \le i \le n} w_i \; x_i \le m \;\; and \;\; \sum_{1 \le i \le n} p_i \; x_i \; \text{is maximized.}$$

The xi's constitute a zero–one-valued vector.

The solution space for this problem consists of the 2n distinct ways to assign zero or one values to the xi's.

Bounding functions are needed to kill some live nodes without 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. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node Z the values of xi, 1 < i < k, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements xi = 0 or 1.

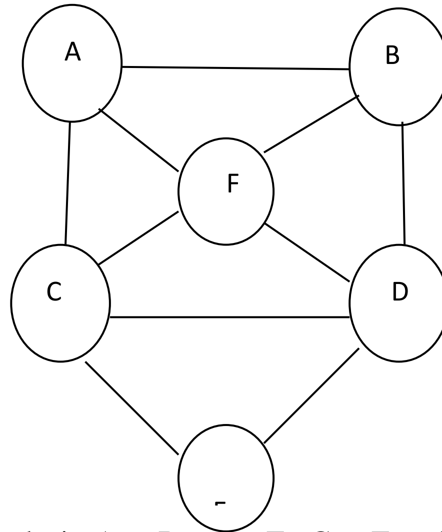
(Knapsack problem using backtracking is solved in branch and bound chapter)

HAMILTONIAN CYCLES

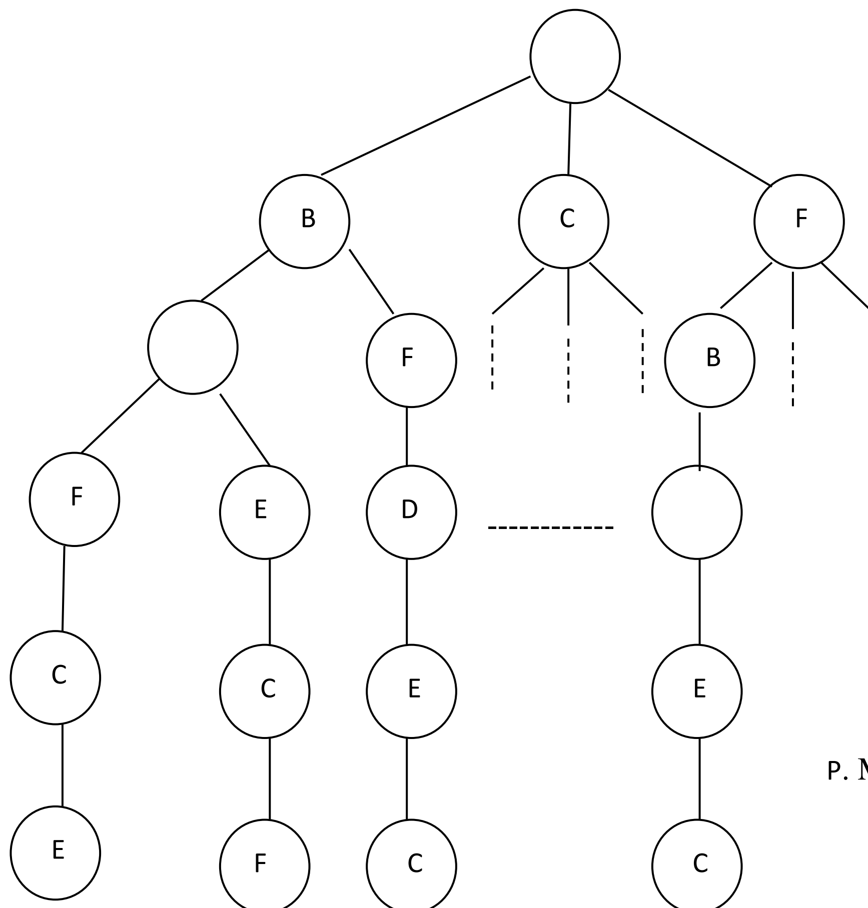
Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once and returns to its starting position.

In other words, If a Hamiltonian cycle begins at some vertex v_1 € G and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$ then the 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.

Example:



Then the Hamiltonian cycle is A – B – D- E- C – F – A. This problem can be solved using backtracking approach. The state space tree is generated in order to find all the Hamiltonian cycles in the graph. The Hamiltonian cycle can be identified as follows-



struck, not a Hamiltonian cycle Hamiltonian cycle

Hamiltonian cycle.

Hamiltonian cycle

For eg, A – B – D – F – C – E; here we get struck. For returning to A we have to revisit at least one vertex. Hence we backtrack from D node another path is chosen A – B – D- E- C – F – A which is Hamiltonian cycle.

Algorithm: Generating a next vertex

```
Algorithm NextValue(k)
{
    repeat
    {
    x[k] := (x[k]+1) mod (m+1); //Next highest color
    if (x[k] = 0) then return;
        if (G[x[k-1],x[k]] ≠ 0) then
        {
        for j := 1 to k-1 do
        if (j = k) then
            if ((k<n) or ((k = n) and G[x[n], x[1]]≠0)) then return;
  } until (false); //Otherwise try to find another color.
}
```

Algorithm: Finding all Hamiltonian cycles.

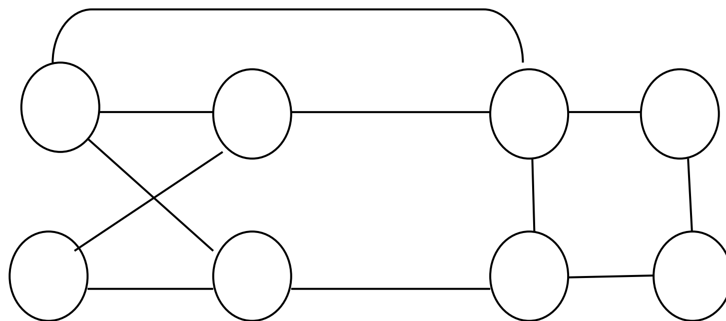
```
Algorithm Hamiltonian(k)
//This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian cycles
//of a graph. The graph is stored as an adjacency matrix G[1:n, 1:n]. All cycles begin at node 1.
{
```

```
repeat
        { //Generate values for x[k].
        NextValue(k); //Assign a legal next value to x[k]
        if (x[k] = 0) then return;
        if (k = n) then write (x[1:n]);
        else Hamiltonian (k+1);
        }until (false);
}
```

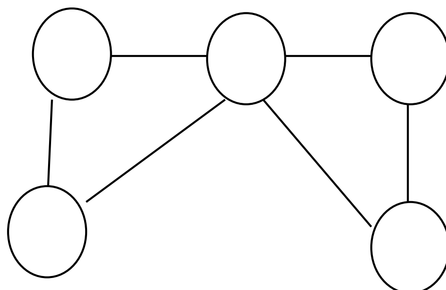
Exercises:

Draw the portion of the state space tree generated for the graph G1 and G2.

G1:



G2:



BRANCH AND BOUND

GENERAL METHOD

The backtracking algorithm is effective for decision problems, but it is not designed for optimization problems. This drawback is refined in case of branch and bound technique. In this we use bounding function, i.e. similar to backtracking. The difference between backtracking and branch and bound is – In backtracking, if we get a solution then we will terminate the search procedure, where as in branch and bound, we will continue the process until we get an optimal solution. Branch and bound is applicable only for minimization problems.

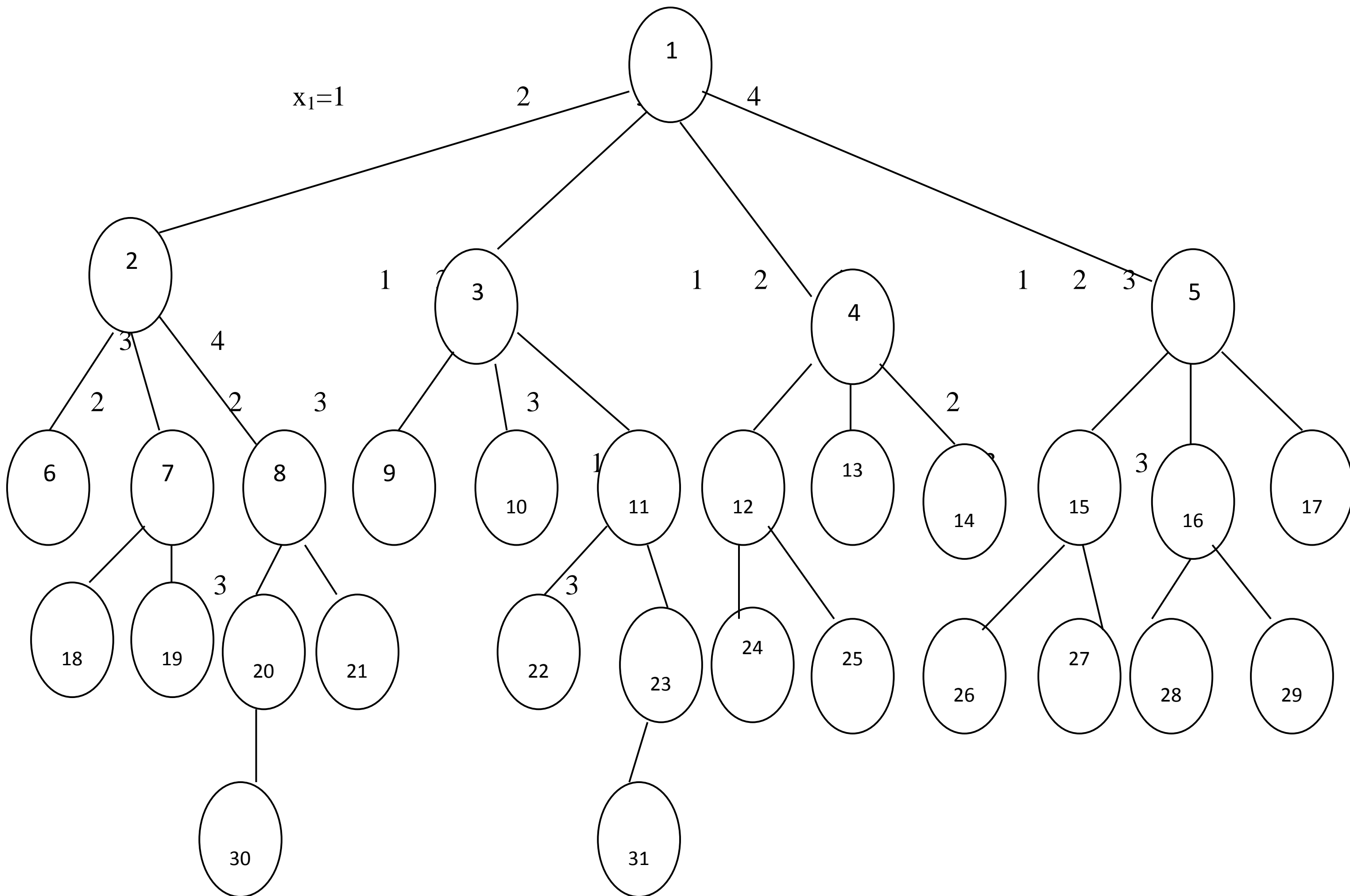
Definition: Generation of nodes in BFS with bounding function is called Branch and Bound.

The main difference between Backtracking and Branch and Bound

- In backtracking, we have to generate one child at a time for the current node by using DFS, and check the conditions for the current solution. If the conditions are not satisfied then bound the node and move back to the previous step otherwise move to next step.
- In Branch and Bound, we have to generate all children for the current parent node and check the condition in BFS method. If it is not satisfied then bound the solution.
- In Branch and Bound, a BFS like state space search will be called FIFO. Search as the list of live nodes is a FIFO list.
- A D search like state space tree- search will be called LIFO. Search as the list of live nodes is a LIFO list.
- FIFO – first generate all child nodes for the root node and select the node which is first generated and repeat the process.

State space tree for 4-queens problem.

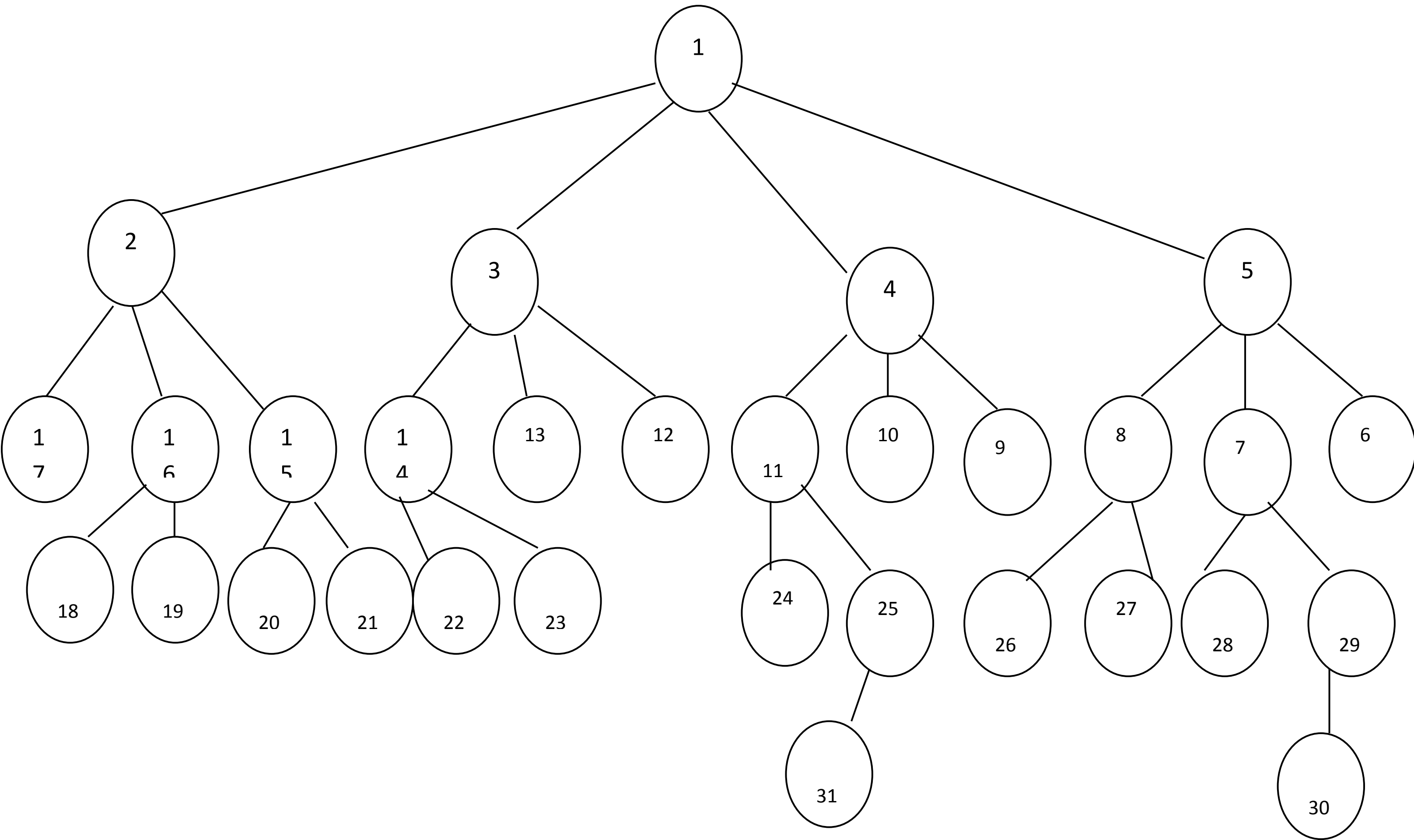
FIFO



x_1		Q_2		
x_2				Q_4
x_3	Q_1			
			Q_3	

x_4

LIFO



x_1			Q_3	
x_2	Q_1			
x_3				Q_4
x_4		Q_2		

Least Cost Search

- In branch and bound method the basic idea is selection of E-node. The selection of E-node should so perfect that we will reach to answer node quickly.
- Using FIFO and LIFO branch and bound method the selection of E-node is very complicated and somewhat blind.
- For speeding up the search process we need to intelligent ranking function for live nodes. Each time, the next E-node is selected on the basis of this ranking function. For this ranking function additional computation (normally called as cost) is needed to answer node from the live node.
- The Least Cost (LC) search is a kind of search in which least cost is involved for reaching to answer node. At each E-node the probability of being an answer node is checked.
- BFS and D-search are special cases of LC search.
- Each time the next E-node is selected on the basis of the ranking function(smallest c^(x)). Let g^(x) be an estimate of the additional effort needed to reach an answer node from x. Let h(x) to be the cost of reaching x from the root and f(x) to be any non-decreasing function such that
- c^(x) = f(h(x)) + g^(x)
- If we set g^(x) = 0 and f(h(x))to be level of node x then we have BFS.
- If we set f(h(x)) = 0 and g^(x) ≤ g^(y) whenever y is a child of x then the search is a D-search.
- In LC search, the cost function c(.) can be defined as
  - i) If x is an answer node the c(X) is the cost computed by the path from x to root in the state space tree.
  - ii) If x is not an answer node such that sub tree of x node is also not containing the answer node then c(x) = α
  - iii) Otherwise c(x) is equal to the cost of minimum cost answer node in subtree x.
- c^(.) with f(h(x)) = h(x) can be an approximation of c(.)

listnode = **record** { listnode *next, *parent; float cost; }

Algorithm LCSearch(t)

//Search t for an answer node.

```
{
if *t is an answer node then output *t and return;
E : = t; //E-node
Initialize the list of live nodes to be empty;
repeat
{
    for each child x of E do
    {
        if x is an answer node then output the path
        from x to t and return;
        Add(x); //x is a new live node.
        (x→parent) := E; //Pointer for path to root.
    }
    if there are no more live nodes then
    {
        write ("No answer node"); return;
    }
    E:=Least();
  } until(false);
}
```

Bounding

- As we know that the bounding functions are used to avoid the generation of sub trees that do not contain the answer nodes. In bounding lower bounds and upper bounds are generated at each node.

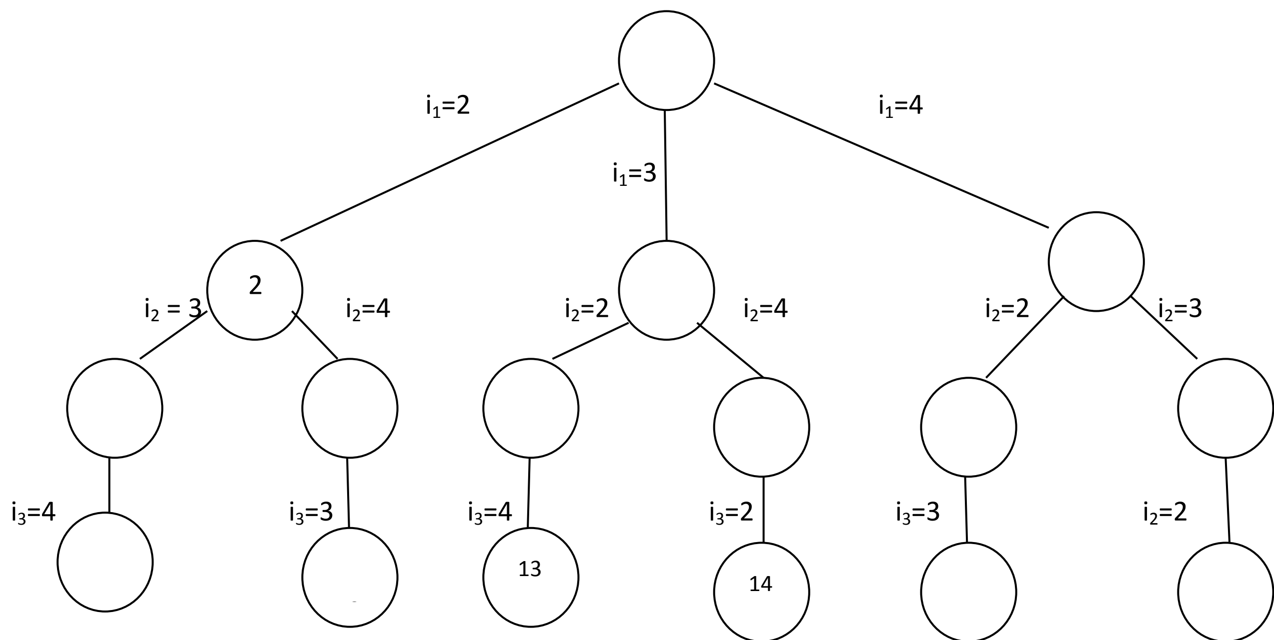
- A cost function c^(x) is such that c^(x) ≤ c(x) is used to provide the lower bounds on solution obtained from any node x.
- Let upper is an upper bound on cost of minimum-cost solution. In that case, all the live nodes with c^(x) > upper can be killed.
- At the start the upper is usually set to α. After generating the children of current E-node, upper can be updated by minimum cost answer node. Each time a new answer node can be obtained.

TRAVELLING SALES PERSON PROBLEM

If there are n cities and cost of travelling from any city to any other city is given. Then we have to obtain the cheapest round trip such that each city is visited exactly once and then returning to starting city, completes the tour.

Typically travelling salesperson problem is represented by weighted graph.

- ⇨ Let G = (V, E) be a directed graph.
- ⇨ Let C_{ij} equal the cost of edge (i, j), C_{ij} = α if (i, j) ∉ E, and let ⎮V⎮ = n.
- ⇨ Every tour starts and ends at vertex 1. So the solution space S is given by S = {1, ∏, 1⎮∏ is a permutation of (2, 3, …,n)}



State space tree for the TSP with n= 4 and i_0=i_4=1

Example:

Consider an instance for TSP is given by G as

$$G = \begin{pmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ 19 & 6 & 18 & \alpha & 3 \\ 16 & 4 & 7 & 16 & \alpha \end{pmatrix}$$

There are n = 5 nodes. Hence draw a state space tree with 5 nodes.

Tour(x) is the path that begins at root and reaching to node x in a state space tree and returns to root. In Branch and Bound strategy cost of each node x is computed. The TSP is solved by choosing the node with optimum cost.

Hence, c(x) = cost of tour(x) where x is a leaf node.

The c^(x) is the approximation cost along the path from the root to x.

Row Minimization:

To understand solving of TSP using Branch and Bound approach we will reduce the cost of the cost matrix M, by using following formula.

$$Red_Row(M) = [M_{ij} - \min \{ M_{ij} \mid 1 \le j \le n \}] \quad \text{where } M_{ij} < \alpha$$

$$M = \begin{pmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ & & & & \\ & & & & \end{pmatrix}$$

19	6	18	α	3
16	4	7	16	α

We will find the minimum of each row.

$$M = \begin{pmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ 19 & 6 & 18 & \alpha & 3 \\ 16 & 4 & 7 & 16 & \alpha \end{pmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

21 Total reduced cost.

We will subtract the row minimization value from corresponding row. Hence

$$Red_Row(M) = \begin{pmatrix} \alpha & 10 & 20 & 0 & 1 \\ 13 & \alpha & 14 & 2 & 0 \\ 1 & 3 & \alpha & 0 & 2 \\ 16 & 3 & 15 & \alpha & 0 \\ 12 & 0 & 3 & 12 & \alpha \end{pmatrix} \begin{matrix} R_1-10 \\ R_2-2 \\ R_3-2 \\ R_4-3 \\ R_5-4 \end{matrix}$$

Column Minimization:

Now we will reduce the matrix by choosing minimum from each column. The formula for column reduction of matrix is

$Red_Col(M) = [M_{ji} - \min \{ M_{ji} \mid 1 \le j \le n \}]$ where $M_{ji} < \alpha$

$$\text{Red_Col(M)} = \begin{pmatrix} \alpha & 10 & 20 & 0 & 1 \\ 13 & \alpha & 14 & 2 & 0 \\ 1 & 3 & \alpha & 0 & 2 \\ 16 & 3 & 15 & \alpha & 0 \\ 12 & 0 & 3 & 12 & \alpha \end{pmatrix}$$

↑ ↑ ↑ ↑ ↑

1 ignore 3 ignore ignore = total = 4

If row or column contains at least one zero ignore corresponding row or column.

$$\text{Red_Col(M)} = \begin{pmatrix} \alpha & 10 & 17 & 0 & 1 \\ 12 & \alpha & 11 & 2 & 0 \\ 0 & 3 & \alpha & 0 & 2 \\ 15 & 3 & 12 & \alpha & 0 \\ 11 & 0 & 3 & 12 & \alpha \end{pmatrix}$$

Full Reduction:

Let M be the cost matrix for TSP for n vertices then M is called reduced if each row and column consists of either entirely α entries or else contain at least one zero. The full reduction can be achieved by applying both row reduction and column reduction.

Thus total reduced cost will be

= cost (Red_Row(M)) + cost (Red_Col(M))

= 21 + 4 = 25

Thus

$$\begin{pmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ 19 & 6 & 18 & \alpha & 3 \\ 16 & 4 & 7 & 16 & \alpha \end{pmatrix} \xrightarrow{\text{fully reduced matrix}} \begin{pmatrix} \alpha & 10 & 17 & 0 & 1 \\ 12 & \alpha & 11 & 2 & 0 \\ 0 & 3 & \alpha & 0 & 2 \\ 15 & 3 & 12 & \alpha & 0 \\ 11 & 0 & 0 & 12 & \alpha \end{pmatrix}$$

Dynamic Reduction:

We obtained the total reduced cost as 25. That means all tours in the original graph have a length at least 25. Using dynamic reduction we can take the choice of edge i→j with optimum cost.

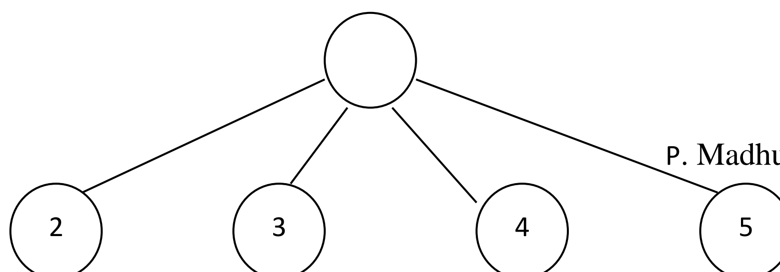
Steps in dynamic reduction technique:

1. Draw a state space tree with optimum cost at root node.
2. Obtain the cost of matrix for path i→j by making i^{th}row and j^{th} column entries as α. Also set M[i][j] = α
3. Cost of corresponding node x with path i, j, is optimum cost + reduced cost + M[i][j].
4. Set node with minimum cost as E-node and generate its children. Repeat step 1 to 4 for completing tour with optimum cost.

Thus

$$\begin{pmatrix} \alpha & 20 & 30 & 10 & 11 \\ 15 & \alpha & 16 & 4 & 2 \\ 3 & 5 & \alpha & 2 & 4 \\ 19 & 6 & 18 & \alpha & 3 \\ 16 & 4 & 7 & 16 & \alpha \end{pmatrix} \xrightarrow{\text{fully reduced matrix}} \begin{pmatrix} \alpha & 10 & 17 & 0 & 1 \\ 12 & \alpha & 11 & 2 & 0 \\ 0 & 3 & \alpha & 0 & 2 \\ 15 & 3 & 12 & \alpha & 0 \\ 11 & 0 & 0 & 12 & \alpha \end{pmatrix}$$

Optimum cost = 21 + 4 = 25



Portion of state space tree

Consider path 1, 2; node 2. Make 1st row and 2nd column = α and set M[2][1] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 11 & 2 & 0 \\ 0 & \alpha & \alpha & 0 & 2 \\ 15 & \alpha & 12 & \alpha & 0 \\ 11 & \alpha & 0 & 12 & \alpha \end{pmatrix} \begin{matrix} \rightarrow \text{ignore} \\ \rightarrow \text{ignore} \\ \rightarrow \text{ignore} \\ \rightarrow \text{ignore} \\ \rightarrow \text{ignore} \end{matrix}$$

↑ ↑ ↑ ↑ ↑

ignore ignore ignore ignore ignore

cost of node 2 is 25 + 0 + 10 = 35

↑ ↑ ↑

optimum cost, reduced cost, old value of M[1][2]

Consider path 1, 3; node 3. Make 1st row and 3rd column = α and set M[3][1] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 12 & \alpha & \alpha & 2 & 0 \\ \alpha & 3 & \alpha & 0 & 2 \\ 15 & 3 & \alpha & \alpha & 0 \\ 11 & 0 & \alpha & 12 & \alpha \end{pmatrix} \Rightarrow \begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 1 & \alpha & \alpha & 2 & 0 \\ \alpha & 3 & \alpha & 0 & 2 \\ 4 & 3 & \alpha & \alpha & 0 \\ 0 & 0 & \alpha & 12 & \alpha \end{pmatrix}$$

↑

11

cost of node 3 is 25 + 11 + 17 = 53

↑ ↑ ↑

optimum reduced old value of M[1][3]

cost cost

Consider path 1, 4; node 4. Make 1st row and 4th column = α and set M[4][1] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 12 & \alpha & 11 & \alpha & 0 \\ 0 & 3 & \alpha & \alpha & 2 \\ \alpha & 3 & 12 & \alpha & 0 \\ 11 & 0 & 0 & \alpha & \alpha \end{pmatrix}$$

cost of node 4 is 25 + 0 + 0 = 25

↑ ↑ ↑

optimum cost, reduced cost, old value of M[1][4]

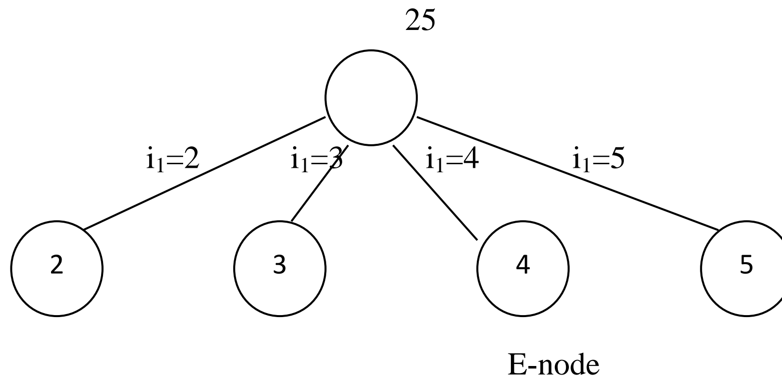
Consider path 1, 5; node5. Make 1st row and 5th column = α and set M[5][1] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 12 & \alpha & 11 & 2 & \alpha \\ 0 & 3 & \alpha & 0 & \alpha \\ 15 & 3 & 12 & \alpha & \alpha \\ \alpha & 0 & 0 & 12 & \alpha \end{pmatrix} \begin{matrix} \\ \rightarrow 2 \\ \\ \rightarrow 3 \\ \\ \end{matrix}$$

cost of node 5 is 25 + 5 + 1 = 31

↑ ↑ ↑

optimum cost, reduced cost, old value of M[1][5]



Now as cost of node 4 is optimum, we will set node 4 as E-node and generate its children nodes 6, 7, 8.

Consider path 1, 4, 2 for node 6

Set 1^{st}row, 4th row to α. Set 2nd & 4th column to α. Also M[4][1] = α, M[2][1] = α

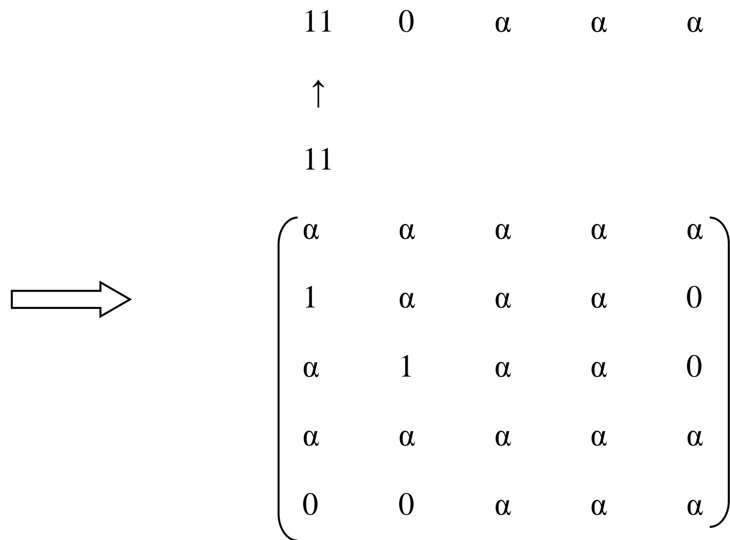
$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 11 & \alpha & 0 \\ 0 & \alpha & \alpha & \alpha & 2 \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 11 & \alpha & 0 & \alpha & \alpha \end{pmatrix}$$

cost of node 6 = 25 + M[4, 2] = 25 + 3 = 28

Consider path 1, 4, 3 for node 7.

Set 1st & 4th row to α. Set 3rd & 4th column to α. Also M[4][1] = α, M[3][1] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 12 & \alpha & \alpha & \alpha & 0 \\ \alpha & 3 & \alpha & \alpha & 2 \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ & & & & \end{pmatrix} \begin{matrix} \\ \\ \rightarrow 2 \\ \\ \\ \end{matrix}$$

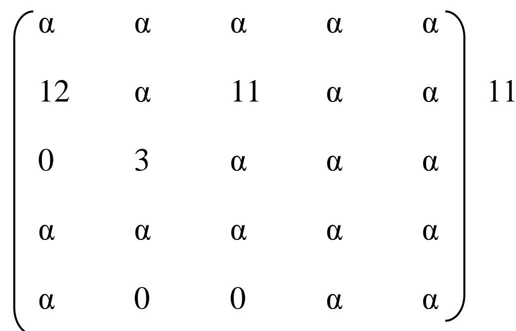


$$\begin{matrix} 11 & 0 & \alpha & \alpha & \alpha \\ \uparrow & & & & \\ 11 & & & & \end{matrix}$$

$$\Rightarrow \begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 1 & \alpha & \alpha & \alpha & 0 \\ \alpha & 1 & \alpha & \alpha & 0 \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 0 & 0 & \alpha & \alpha & \alpha \end{pmatrix}$$

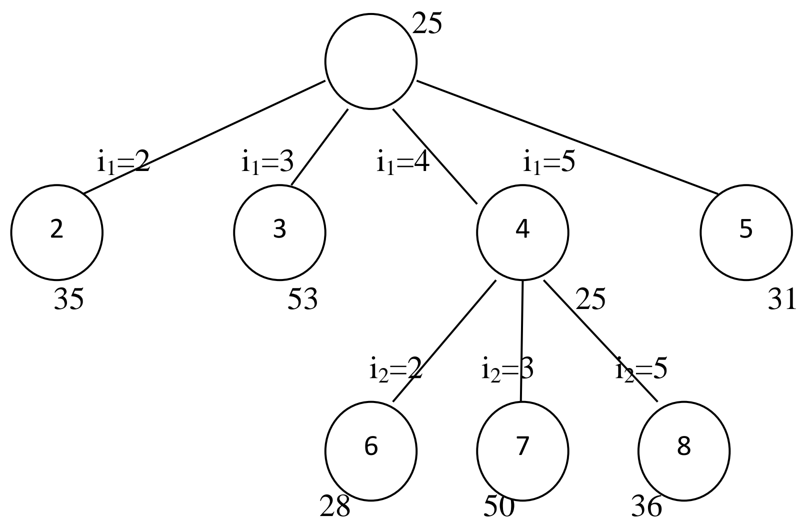
Cost of node 7 = 25 + 13 + M[4][3] = 25 + 13 + 12 = 50

Consider path 1, 4, 5 for node 8.



$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ 12 & \alpha & 11 & \alpha & \alpha \\ 0 & 3 & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & 0 & 0 & \alpha & \alpha \end{pmatrix} 11$$

cost of node 8 = 25 + 11 = 36



Now as cost of node 6 is minimum, node 6 becomes an E-node. Hence generate children for node 6. Node 9 and 10 are children nodes of node 6.

Consider path 1, 4, 2, 3 for node 9. Set 1st , 4th, 2nd row set to α. Set 4th, 2nd, 3rd coln = α

Set M[1, 4] = M[1, 2] = M[1, 3] = α

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & 2 \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 11 & \alpha & \alpha & \alpha & \alpha \end{pmatrix} \begin{matrix} \\ \\ 2 \\ \\ \\ \end{matrix}$$

$$\begin{matrix} 11 \end{matrix}$$

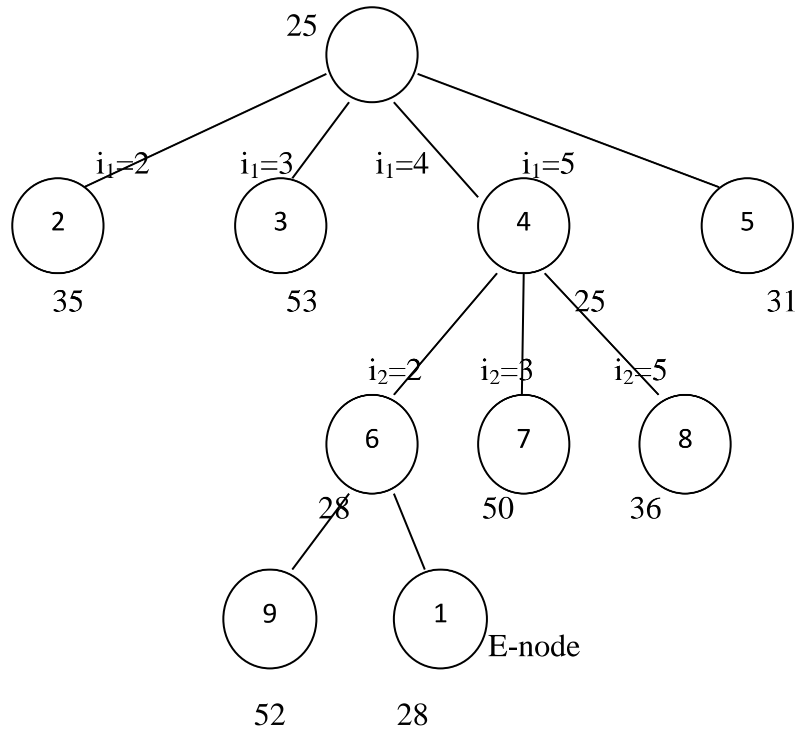
cost of node 9 = 28 + 13 + 11 = 52

Consider path 1, 4, 2, 5 for node 10. Set 1, 4, 2 rows = α. Set 4, 2, 5 columns = α

Set M[1][4] = M[1][2] = M[1][5] = α

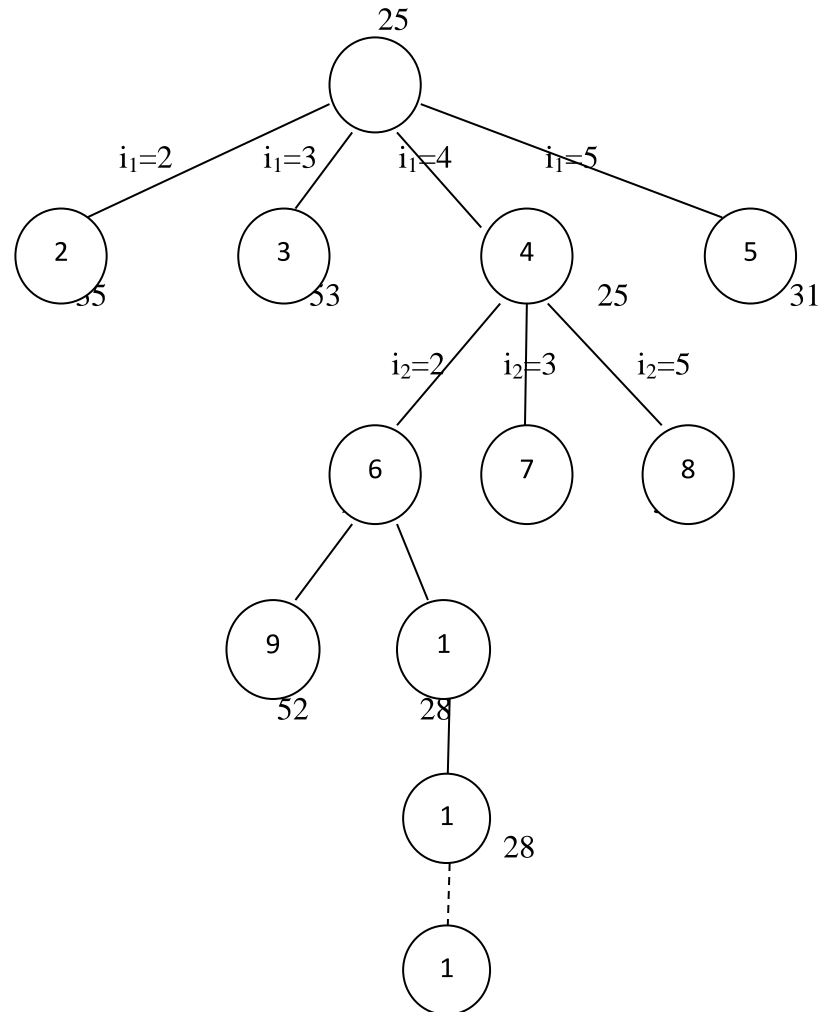
$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha & \alpha \end{pmatrix}$$

cost of node 10 = 28 + M[2][5] = 28 + 0 = 28



As for node 10 only child being generated is node 11. We set path as 1, 4, 2, 5, 3. To complete the tour we return to 1.

Hence the state space tree is



Hence the optimum cost of the tour is 28

0/1 KNAPSACK PROBLEM

The 0/1 Knapsack problem states that there are n objects given and capacity of knapsack 'm'. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned. The knapsack problem is a maximization problem. That means we will always seek for maximum $p_i x_i$ (where p_i represents profit of object x_i). We can also get $\sum p_i x_i$ maximum iff $- \sum p_i x_i$ is minimum.

The modified knapsack problem is stated as

minimize - $\sum^{n}_{i=1} p_i x_i$

subject to $\sum^{n}_{i} w_i x_i \leq m$

$x_i = 0$ or 1, $1 \leq i \leq n$

We will discuss the branch and bound strategy for 0/1 knapsack problem using fixed tuple size formulation. We will design the state space tree and compute c^(.) and u(.) where c^(x) represents the approximate cost used for computing the least cost c(x). Clearly u(x) denotes the upper bound. The upper bound is used to kill those nodes in the state space tree which cannot lead to the answer node.

Let x be the node at level j. Then we will draw the state space tree for fixed tuple formulastion having levels $1 \leq i \leq n+1$

Then we need to compute c^(x) and u(x). Such that c^(x) ≤ c(x) ≤ u(x) for every node.

Algorithm: for computing c^(x)

```
Algorithm C_Bound(total_profit, total_wt, k)
{
    pt := total_profit;
    wt := total_wt;
    for (i := k+1 to n) do
    {
        wt := wt + w[i];
        if (wt < m) then pt := pt + p[i];
        else return (pt + (1-(wt-m)/w[i])*p[i]);
    }
    return pt;
}
```

Algorithm: for computing u(x)

```
Algorithm U_Bound(total_profit, total_wt, k, m)
{
    pt := total_profit;
    wt := total_wt;
    for (i := k+1 to n) do
    {
        if (wt + w[i] <= m) then
        {
        pt := pt - p[i];
        wt := wt + w[i];
        }
    }
    return pt;
}
```

LC BRANCH AND BOUND SOLUTION

The LC Branch and Bound solution can be obtained using fixed tuple size formulation.

The steps to be followed for LCBB solution are

1. Draw the state space tree.
2. Compute C^(.) and u(.) for each node.
3. If C^(x) > upper kill node x.
4. Otherwise, the minimum cost c^(x) becomes E-node. Generate children for E-node.
5. Repeat step 3 and 4 until all the nodes get covered.
6. The minimum cost c^(x) becomes the answer node. Trace the path in backward direction from x to root for solution subset.

Example:

Consider knapsack instance n=4 with capacity m = 15

Object j	p_i	w_i
1	10	2
2	10	4
4	12	6
4	18	9

Solution:

Let us design state space tree using fixed tuple size formulation. The computation of c^(x) and u(x) for each node x is done.u(1) can be computed as

For i=1, 2, and 3, c gets incremented by 2, 4, and 6. when i=4 wt+wt[i] < = m fails.

Hence $-\sum p_i = -(10+10+12) = -32$

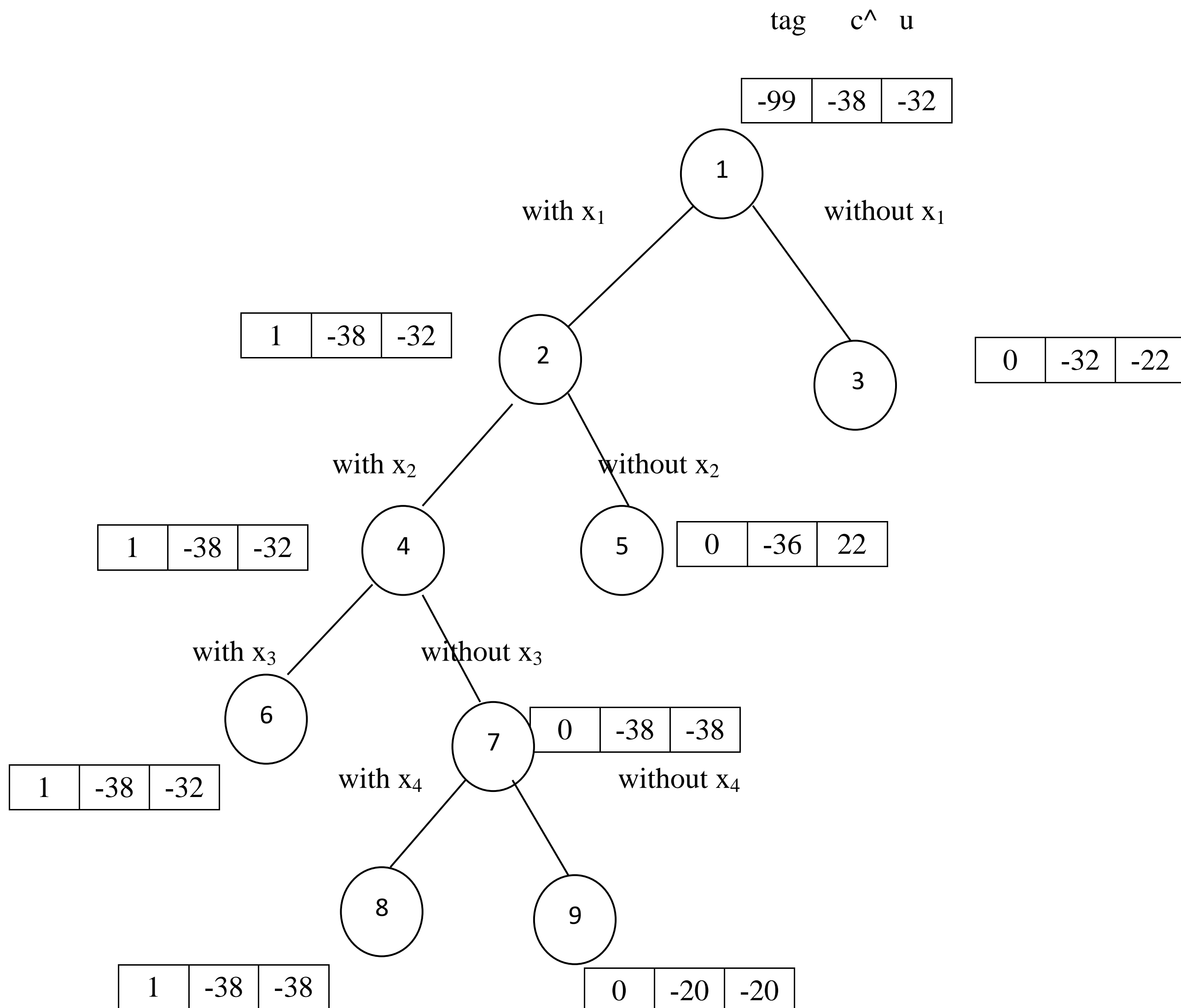
u(1) = 32

If we select i=4, then it will exceed capacity of knapsack. The computation of c^(1) can be done as

$$\hat{c}(1) = u(1) - \left(\frac{\text{m-current total weight}}{\text{Actual weight of remaining object}} \right) * [\text{Actual profit of remaining obj}]$$

c^(1) = [-32-[15-(2+4+6)]/9]*18] = -38

In this way considering each possibility of object being in knapsack or not being in knapsack. c^(x) and u(x) is computed. Each time minimum $-\sum p_i x_i$ will become E-node and we will get the answer node as node 8.



LCBB Solution space tree

In the above fig, at each node a structure is drawn in which computation of c^(.) and u(.) is given. The tag field is useful for tracing the path.

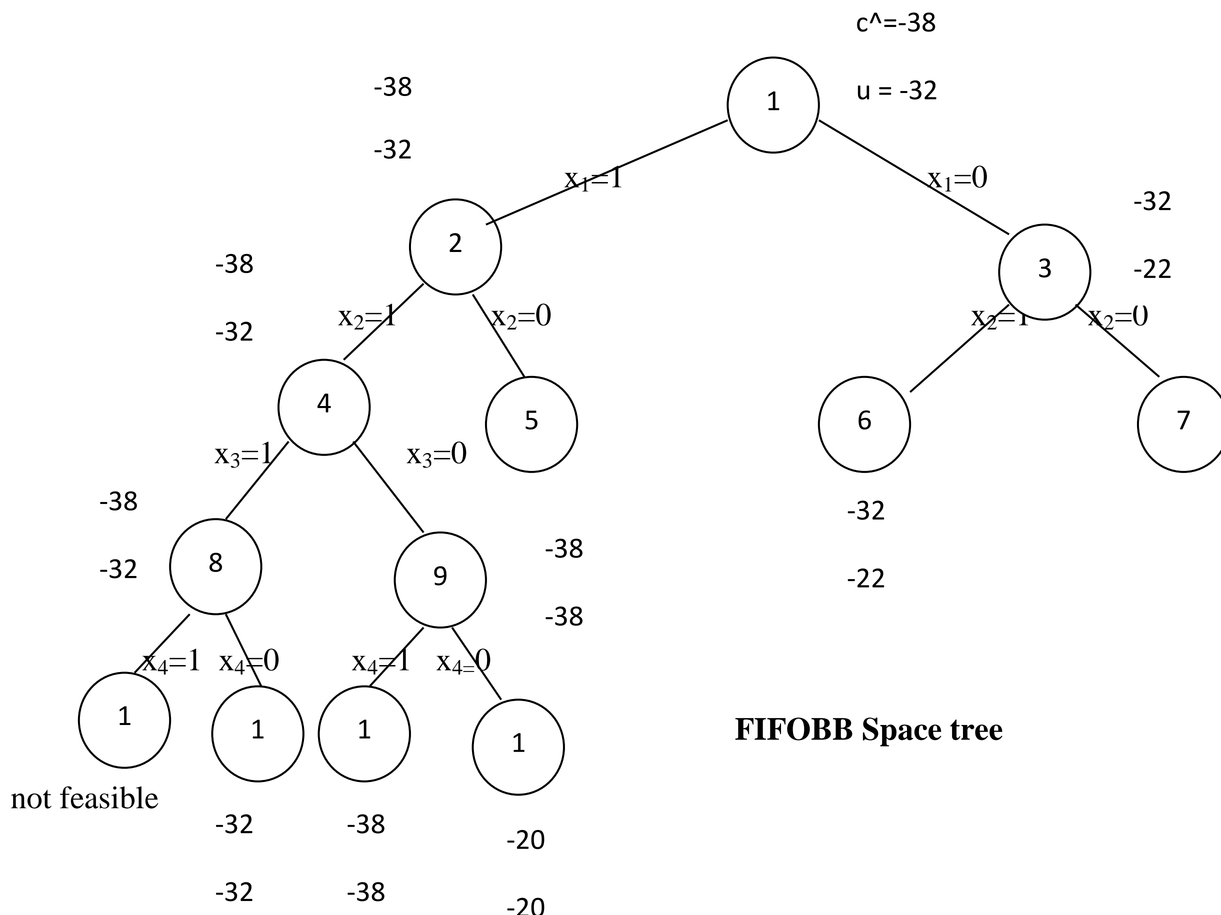
The sequence of tag bits from the answer node to the root gives the x_i values. Thus, tag(2) = tag(4) = tag(6) = tag(8) = 1 and tag(3) = tag(5) = tag(7) = tag(9) = 0.

The tag sequence for the path 8, 7, 4, 2, 1 is 1011 and so $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, $x_1 = 1$

We will select x_4, x_2 and x_1 to fill up the knapsack and gain maximum profit.

FIFO BRANCH AND BOUND SOLUTION

The space tree with variable tuple size formulation can be drawn and c^(.) and u(.) is computed.



FIFOBB Space tree

Initially upper u(1) = -32. Then children of node 1 are generated. Node 2 becomes E-node and hence children 4 and 5 are generated. Node 4 and 5 are added in the list of live nodes. Next, node 3 becomes E-node and children 6 and 7 are generated. As c^(7) > upper kill node 7. Hence node 6 will be added in the list of live nodes. Node 4 is E-node and 8 & 9 are generated. The upper is updated and it is u(9) = -38. Nodes 8 & 9 are added in the list of live nodes. Node 5 & 6 becomes the next E-node but c^(5) > upper and c^(6) > upper, kill nodes 5 & 6. Node 8 becomes next E-node and children 10 & 11 are generated. As node 10 is infeasible do not consider it. c^(11) > upper. Hence kill node 11. Node 9 becomes next E-node and upper = -38. Children 12 & 13 are generated. But c6(13) > upper. So kill node 13. Finally node 12 becomes an answer node. Solution is $x_1=1$, $x_2=1$, $x_3=0$, $x_4=1$.