

Branch and Bound: General Method, FIFO Branch and Bound, LC Branch and Bound, Applications: 0/1 knapsack Problem, Travelling Salesperson Problem.

Introduction to Branch and Bound Technique

- The Branch and Bound Technique is a problem solving strategy, which is most commonly used in optimization problems, where the goal is to minimize a certain value.
- The optimized solution is obtained by means of a state space tree (A state space tree is a tree where the solution is constructed by adding elements one by one, starting from the root. Note that the root contains no element).
- This method is best when used for combinatorial problems with exponential time complexity, since it provides a more efficient solution to such problems.

The Algorithm: Branch and Bound Technique

- In this technique, the first step is to create a function U (which represents an upper bound to the value that node and its children shall achieve), that we intend to minimize.
- We call this function the objective function.
- Note that the branch and bound technique can also be used for maximization problems, since multiplying the objective function by -1 converts the problem to a minimization problem.
- Let this function have an initial maximum value, according to the conditions of the given problem. Also, let U_0 be the initial value of U .
- We also calculate a cost function C which will give us the exact value that the particular node shall achieve.
- The next question is the order in which the tree is to be searched. For this, there exist multiple types of branch and bound, which we shall discuss in detail later.
- For now, let us assume that there is a set S consisting of subsets of the given set of values in the order in which they need to be searched.

The algorithm of the branch and bound method for this problem will be as follows:

For each subset s in S , do the following:

1. Calculate the value of $U(s)$.
2. If $U(s) < U_0$, then $U_0 = U(s)$.

In the end, the subset s for which the current value of U_0 is obtained will be the best solution to the given problem, and the value of the cost function at that node will give us the solution. Here, after each level, the value of U_0 tells us that there shall be a node with cost less than that value.

Types of Solutions:

For a branch and bound problem, there are two ways in which the solution can be represented:

- **Variable size solution:** This solution provides the subset of the given set that gives the optimized solution for the given problem. For example, if we are to select a combination of elements from {A, B, C, D, E} that optimizes the given problem, and it is found that A, B, and E together give the best solution, then the solution will be {A, B, E}.
- **Fixed size solution:** This solution is a sequence of 0s and 1s, with the digit at the i^{th} position denoting whether the i^{th} element should be included or not. Hence, for the earlier example, the solution will be given by {1, 1, 0, 0, 1}.

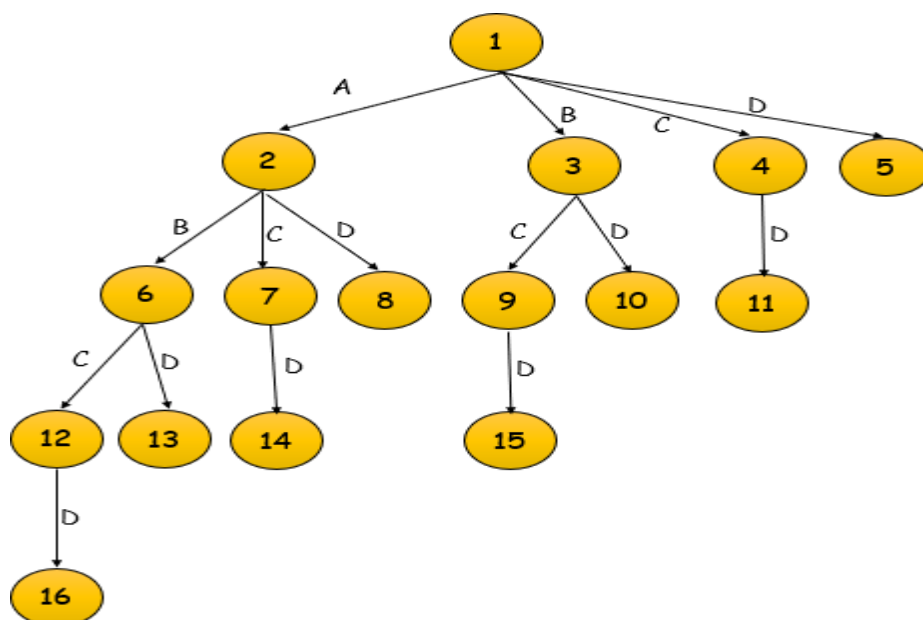
Types of Branch and Bound:

There are multiple types of the Branch and Bound method, based on the order in which the state space tree is to be searched. We will be using the variable solution method to denote the solutions in these methods.

1. FIFO Branch and Bound

The First-In-First-Out approach to the branch and bound problem follows the **queue** approach in creating the state-space tree. Here, breadth first search is performed, i.e., the elements at a particular level are all searched, and then the elements of the next level are searched, starting from the first child of the first node in the previous level.

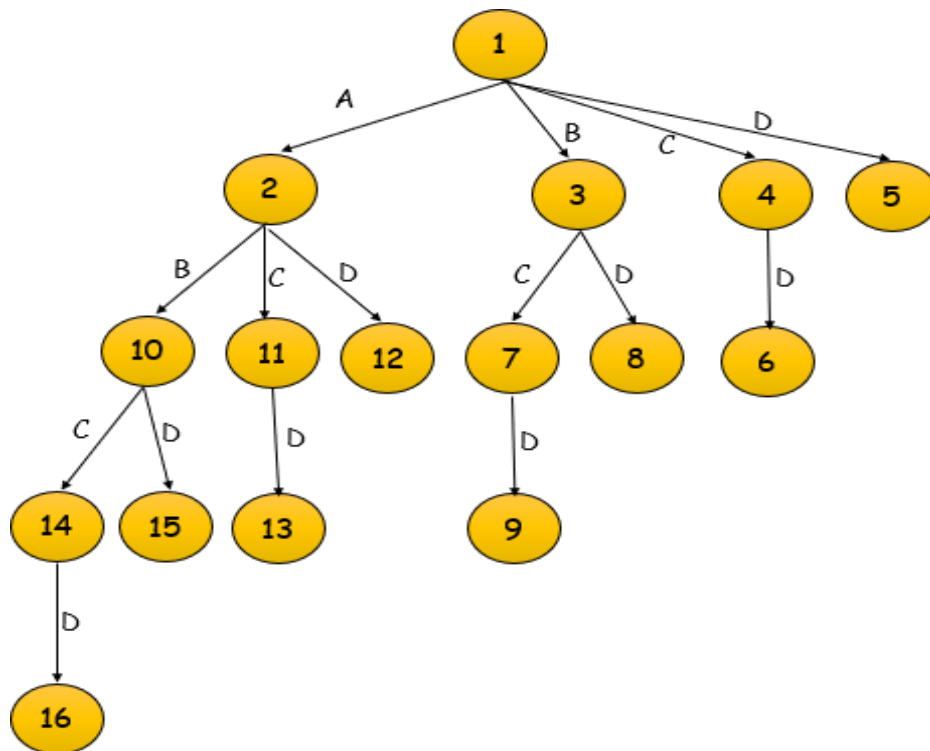
For a given set {A, B, C, D}, the state space tree will be constructed as follows:



Here, note that the number assigned to the node signifies the order in which the tree shall be constructed. The element next to the set denotes the next element to be added to the subset. Note that if an element is getting added, it is assumed here that all elements in the set preceding that element are not added. For example, in node 4, D is getting added. This implies that elements A, B and C are not added.

2. LIFO Branch and Bound

The Last-In-First-Out approach to this problem follows the **stack** approach in creating the state space tree. Here, when nodes get added to the state space tree, think of them as getting added to a stack. When all nodes of a level are added, we pop the topmost element from the stack and then explore it. Hence, the state space tree for the same example {A, B, C, D} will be as follows:



Here, one can see that the main difference lies in the order in which the nodes have been explored.

3. Least Cost-Branch and Bound

- This method of exploration uses the cost function in order to explore the state space tree.
- Although the previous two methods calculate the cost function at each node, this is not used as a criterion for further exploration.
- In this method, after the children of a particular node have been explored, the next node to be explored would be that node out of the unexplored nodes which has the least cost.
- For example, in the previous example, after reaching node 5, the next node to be explored would be that which has the least cost among nodes 2, 3, 4, 5.

Why use Branch and Bound?

- The Branch and Bound method is preferred over other similar methods such as backtracking when it comes to optimization problems.
- Here, the cost and the objective function help in finding branches that need not be explored.
- Suppose the cost of a particular node has been determined. If this value is greater than that of U_0 , this means that there is no way this node or its children shall give a solution. Hence, we can kill this node and not explore its further branches.
- This method helps us rule out cases not worth exploring, and is therefore more efficient.

Problems that can be solved using Branch and Bound

The Branch and Bound method can be used for solving most combinatorial problems. Some of these problems are given below:

1. **Job Sequencing:** Suppose there is a set of N jobs and a set of N workers. Each worker takes a specific time to complete each of the N jobs. The job sequencing problem deals with finding that order of the workers, which minimizes the time taken to complete the job.
2. **0/1 Knapsack problem:** Given a set of weights of N objects and a sack which can carry a maximum of W units. The problem deals with finding that subset of items such that the maximum possible weight is carried in the sack. Here, one cannot take part of an object, i.e., one can either take an object or not take it. This problem can also be solved using the backtracking, brute force and the dynamic programming approach.
3. **Traveling Salesman Problem:** Here, we are given a set of N cities, and the cost of traveling between all pairs of cities. The problem is to find a path such that one starts from a given node, visits all cities exactly once, and returns back to the starting city.

Applications:

1. Travelling Salesperson Problem.(LCBB)
2. 0/1 knapsack Problem.(LCBB,FIFOBB ,LIFOBB)

1. Travelling Salesperson Problem using LCBB:

- Travelling Salesman Problem (TSP) is an interesting problem. Problem is defined as “given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with the constraint of minimizing the travelling distance.”
- TSP has many practical applications. It is used in network design, and transportation route design.
- The objective is to minimize the distance. We can start tour from any random city and visit other cities in any order. With n cities, $n!$ Different permutations are possible.
- Exploring all paths using brute force attacks may not be useful in real life applications.

LCBB using Static State Space Tree for Travelling Salesman Problem

- Branch and bound is an effective way to find better, if not best, solution in quick time by pruning some of the unnecessary branches of search tree.
- It works as follow :

Consider directed weighted graph $G = (V, E, W)$, where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C , where

C_{ij} = cost of edge, if there is a direct path from city i to city j

$C_{ij} = \infty$, if there is no direct path from city i to city j .

2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.
3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.
4. Prepare state space tree for the reduce matrix
5. Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.
6. If $\langle i, j \rangle$ edge is to be included, then do following :
 - (a) Set all values in row i and all values in column j of A to ∞
 - (b) Set $A[j, 1] = \infty$
 - (c) Reduce A again, except rows and columns having all ∞ entries.

7. Compute the cost of newly created reduced matrix as,

$$\text{Cost} = L + \text{Cost}(i, j) + r$$

Where, L is cost of original reduced cost matrix and r is $A[i, j]$.

8. If all nodes are not visited then go to step 4.

Reduction procedure is described below :

Raw Reduction:

Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow:

$$M_{\text{RowRed}} = \{M_{ij} - \min \{M_{ij} \mid 1 \leq j \leq n, \text{ and } M_{ij} < \infty\}\}$$

Consider the following distance matrix:

M =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Find the minimum element from each row and subtract it from each cell of matrix.

Find the minimum element from each row and subtract it from each cell of matrix.

$$M =$$

∞	20	30	10	11	$\rightarrow 10$
15	∞	16	4	2	$\rightarrow 2$
3	5	∞	2	4	$\rightarrow 2$
19	6	18	∞	3	$\rightarrow 3$
16	4	7	16	∞	$\rightarrow 4$

Reduced matrix would be:

$$M_{\text{RowRed}} =$$

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Row reduction cost is the summation of all the values subtracted from each rows:

Row reduction cost (M) = $10 + 2 + 2 + 3 + 4 = 21$

Column reduction:

Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$$M_{\text{ColRed}} = \{M_{ji} - \min \{M_{ji} \mid 1 \leq j \leq n, \text{ and } M_{ji} < \infty\}\}$$

To reduced above matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

$$M_{\text{RowRed}} = \begin{array}{c|ccccc} \hline \infty & 10 & 20 & 0 & 1 \\ \hline 13 & \infty & 14 & 2 & 0 \\ \hline 1 & 3 & \infty & 0 & 2 \\ \hline 16 & 3 & 15 & \infty & 0 \\ \hline 12 & 0 & 3 & 12 & \infty \\ \hline \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \hline 1 & 0 & 3 & 0 & 0 \\ \hline \end{array}$$

Column reduced matrix M_{ColRed} would be:

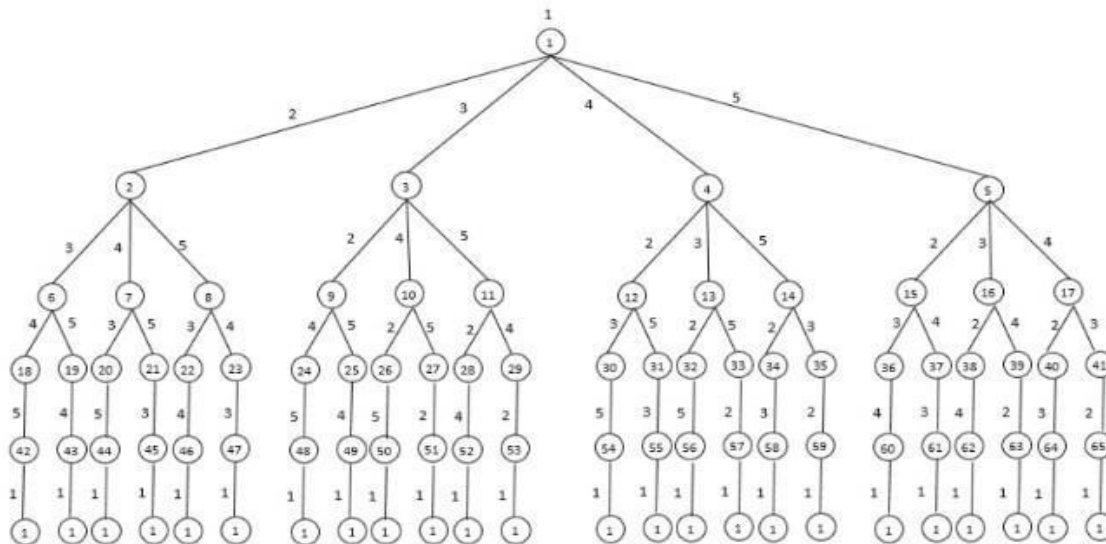
$$M_{\text{ColRed}} = \begin{array}{c|ccccc} \hline \infty & 10 & 17 & 0 & 1 \\ \hline 12 & \infty & 11 & 2 & 0 \\ \hline 0 & 3 & \infty & 0 & 2 \\ \hline 15 & 3 & 12 & \infty & 0 \\ \hline 11 & 0 & 0 & 12 & \infty \\ \hline \end{array}$$

Each row and column of M_{ColRed} has at least one zero entry, so this matrix is reduced matrix.

Column reduction cost (M) = $1 + 0 + 3 + 0 + 0 = 4$

State space tree for 5 city problem is depicted in Fig. 6.6.1. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.

State space tree for 5 city problem is depicted in Fig. 6.6.1. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.



Example

Example: Find the solution of following travelling salesman problem using branch and bound method.

Cost Matrix =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Solution:

- I. The procedure for dynamic reduction is as follow:
- II. Draw state space tree with optimal reduction cost at root node.
- III. Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞ .
Set $M[j][i] = \infty$

- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + $M[j][i]$
- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
- Reduce above cost matrix by subtracting minimum value from each row and column.

∞	20	30	10	11	\rightarrow 10	∞	10	20	0	1	$= M'_1$
15	∞	16	4	2	\rightarrow 2	13	∞	14	2	0	
3	5	∞	2	4	\rightarrow 2	1	3	∞	0	2	
19	6	18	∞	3	\rightarrow 3	16	3	15	∞	0	
16	4	7	16	∞	\rightarrow 4	12	0	3	12	∞	
						\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
						1	0	3	0	0	

M'_1 is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

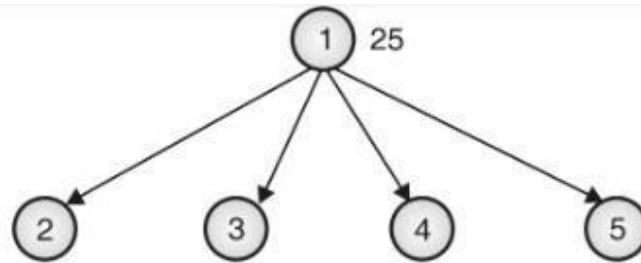
$= M_1$

Cost of $M_1 = C(1)$

= Row reduction cost + Column reduction cost

= $(10 + 2 + 2 + 3 + 4) + (1 + 3) = 25$

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree

Let us find cost of edge from node 1 to 2, 3, 4, 5.

Select edge 1-2:

Set $M_1[1][] = M_1[][2] = \infty$

Set $M_1[2][1] = \infty$

Reduce the resultant matrix if required.

∞	∞	∞	∞	∞	$\rightarrow x$
∞	∞	11	2	0	$\rightarrow 0$
0	∞	∞	0	2	$\rightarrow 0 = M_2$
15	∞	12	∞	0	$\rightarrow 0$
11	∞	0	12	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
0	x	0	0	0	

M_2 is already reduced.

Cost of node 2 :

$$C(2) = C(1) + \text{Reduction cost} + M_1[1][2]$$

$$= 25 + 0 + 10 = 35$$

Select edge 1-3

$$\text{Set } M_1[1][] = M_1[][3] = \infty$$

$$\text{Set } M_1[3][1] = \infty$$

Reduce the resultant matrix if required.

$$M_1 \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow 0 \\ \rightarrow 0 \\ \rightarrow 0 \\ \rightarrow 0 \end{array} \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{array} = M_3$$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 $11 \quad 0 \quad x \quad 0 \quad 0$

Select edge 1-4:

$$\text{Set } M_1[1][] = M_1[][4] = \infty$$

$$\text{Set } M_1[4][1] = \infty$$

Reduce resultant matrix if required.

$$M_1 \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow 0 \\ \rightarrow 0 \\ \rightarrow 0 \\ \rightarrow 0 \end{array} = M_4$$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 $0 \quad 0 \quad 0 \quad x \quad 0$

Matrix M_4 is already reduced.

Cost of node 4:

$$C(4) = C(1) + \text{Reduction cost} + M_1[1][4]$$

$$= 25 + 0 + 0 = 25$$

Select edge 1-5:

Set $M_1[1][] = M_1[][5] = \infty$

Set $M_1[5][1] = \infty$

Reduce the resultant matrix if required.

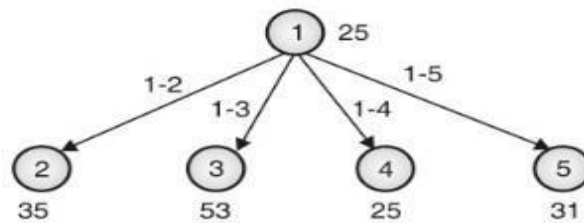
$M_1 \Rightarrow$	∞	∞	∞	∞	∞	$\rightarrow x$	\Rightarrow	∞	∞	∞	∞	∞	$= M_5$
	12	∞	11	2	∞	$\rightarrow 2$		10	∞	9	0	∞	
	0	3	∞	0	∞	$\rightarrow 0$		0	3	∞	0	∞	
	15	3	12	∞	∞	$\rightarrow 3$		12	0	9	∞	∞	
	∞	0	0	12	∞	$\rightarrow 0$		∞	0	0	12	∞	
								\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
								0	0	0	0	x	

Cost of node 5:

$C(5) = C(1) + \text{reduction cost} + M_1[1][5]$

$= 25 + 5 + 1 = 31$

State space diagram:



Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5. Let's explore all three nodes.

Select path 1-4-2 : (Add edge 4-2)

Set $M_4[1][] = M_4[4][] = M_4[][2] = \infty$

Set $M_4[2][1] = \infty$

Reduce resultant matrix if required.

$M_4 \Rightarrow$	∞	∞	∞	∞	∞	$\rightarrow x$	\Rightarrow	∞	∞	∞	∞	∞	$= M_6$
	∞	∞	11	∞	0	$\rightarrow 0$		∞	∞	∞	∞	∞	
	0	∞	∞	∞	2	$\rightarrow 0$		0	∞	∞	∞	∞	
	∞	∞	∞	∞	∞	$\rightarrow x$		∞	∞	∞	∞	∞	
	11	∞	0	∞	∞	$\rightarrow 0$		∞	∞	∞	∞	∞	
								\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
								0	0	0	x	0	

Matrix M_6 is already reduced.

Cost of node 6:

$$C(6) = C(4) + \text{Reduction cost} + M_4 [4] [2] \\ = 25 + 0 + 3 = 28$$

Select edge 4-3 (Path 1-4-3):

Set $M_4 [1] [] = M_4 [4] [] = M_4 [] [3] = \infty$

Set $M [3][1] = \infty$

Reduce the resultant matrix if required.

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty & \\ \hline 12 & \infty & \infty & \infty & 0 & \\ \hline \infty & 3 & \infty & \infty & 2 & \\ \hline \infty & \infty & \infty & \infty & \infty & \\ \hline 11 & 0 & \infty & \infty & \infty & \\ \hline \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow 0 \\ \rightarrow 2 \\ \rightarrow \infty \\ \rightarrow 0 \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty & \\ \hline 12 & \infty & \infty & \infty & 0 & \\ \hline \infty & 1 & \infty & \infty & 0 & \\ \hline \infty & \infty & \infty & \infty & \infty & \\ \hline 11 & 0 & \infty & \infty & \infty & \\ \hline \end{array} = M'_7$$

$$\begin{array}{c} \downarrow \downarrow \downarrow \downarrow \downarrow \\ 11 \ 0 \ x \ x \ 0 \end{array}$$

M'_7 is not reduced. Reduce it by subtracting 11 from column 1.

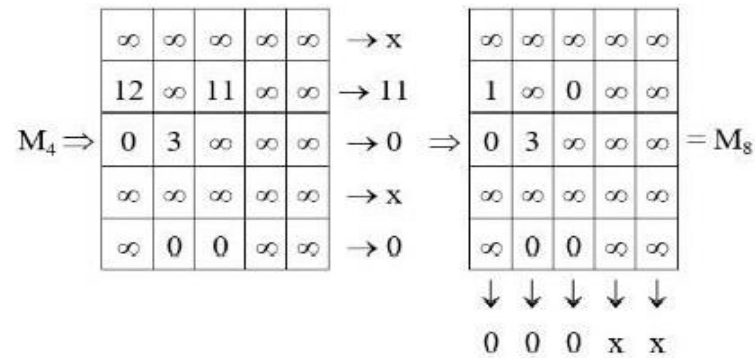
$$\therefore M'_7 \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 1 & \infty & \infty & \infty & 0 \\ \hline \infty & 1 & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & 0 & \infty & \infty & \infty \\ \hline \end{array} = M_7$$

Cost of node 7:

$$C(7) = C(4) + \text{Reduction cost} + M_4 [4] [3]$$

$$= 25 + 2 + 11 + 12 = 50$$

Select edge 4-5 (Path 1-4-5):



Matrix M_8 is reduced.

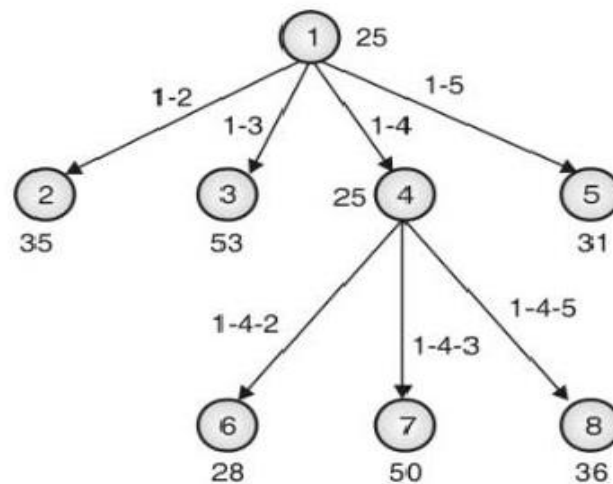
Cost of node 8:

$$C(8) = C(4) + \text{Reduction cost} + M_4[4][5]$$

$$= 25 + 11 + 0 = 36$$

State space tree

Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.



Add edge 2-3 (Path 1-4-2-3):

$$\text{Set } M_6[1][] = M_6[4][] = M_6[2][]$$

$$= M_6[][3] = \infty$$

$$\text{Set } M_6[3][1] = \infty$$

Reduce resultant matrix if required.

$$\begin{array}{c}
 \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \mathbf{2} \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{11} & \infty & \infty & \infty & \infty \end{array} & \begin{array}{l} \rightarrow \mathbf{x} \\ \rightarrow \mathbf{x} \\ \rightarrow \mathbf{0} \\ \rightarrow \mathbf{x} \\ \rightarrow \mathbf{11} \end{array} & \Rightarrow & \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \mathbf{2} \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \infty \end{array} = \mathbf{M}'_9 \\
 & & & \begin{array}{ccccc} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \mathbf{0} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{2} \end{array}
 \end{array}$$

$$\therefore \mathbf{M}'_9 \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \mathbf{0} \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \infty \end{array} = \mathbf{M}_9$$

Cost of node 9:

$$C(9) = C(6) + \text{Reduction cost} + M_6[2][3]$$

$$= 28 + 11 + 2 + 11 = 52$$

Add edge 2-5 (Path 1-4-2-5):

$$\text{Set } M_6[1][] = M_6[4][] = M_6[2][] = M_6[][5] = \infty$$

$$\text{Set } M_6[5][1] = \infty$$

Reduce resultant matrix if required.

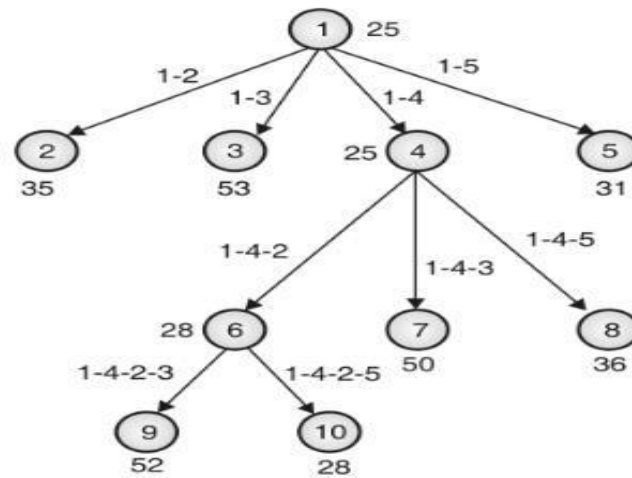
$$\therefore \mathbf{M}_6 \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \mathbf{0} & \infty & \infty \end{array} = \mathbf{M}_{10}$$

Cost of node 10:

$$C(10) = C(6) + \text{Reduction cost} + M_6[2][5]$$

$$= 28 + 0 + 0 = 28$$

State space tree



Add edge 5-3 (Path 1-4-2-5-3):

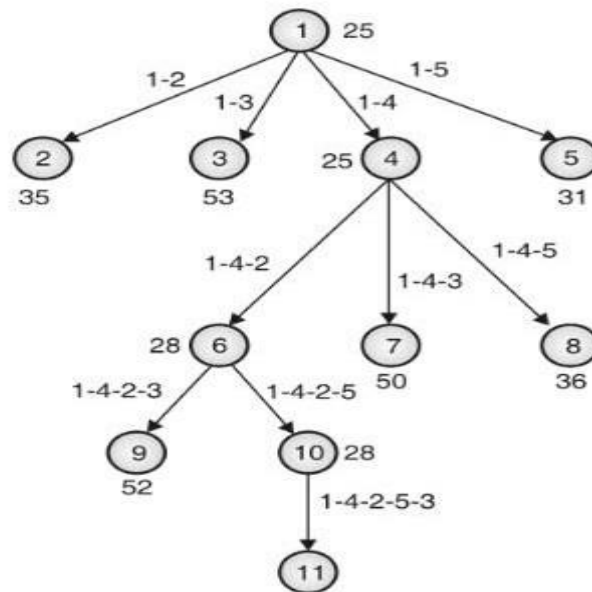
$$\therefore M_{10} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \end{array} = M_{11}$$

Cost of node 11:

$$C(11) = C(10) + \text{Reduction cost} + M_{10}[5][3]$$

$$= 28 + 0 + 0 = 28$$

State space tree:



2. 0/1 knapsack Problem (LCBB, FIFOBB, LIFOBB):

Knapsack Problem using Branch and Bound is discussed in this article. LC and FIFO, both variants are described with example.

- As discussed earlier, the goal of knapsack problem is to maximize $\sum_{i=1}^n p_i x_i$ given the constraints $\sum_{i=1}^n w_i x_i \leq M$, where M is the size of the knapsack. A maximization problem can be converted to a minimization problem by negating the value of the objective function.

- The modified knapsack problem is stated as,

minimize $-\sum_{i=1}^n p_i x_i$ subjected to $\sum_{i=1}^n w_i x_i \leq M$,

Where, $x_i \in \{0, 1\}$, $1 \leq i \leq n$

- Node satisfying the constraint $\sum_{i=1}^n w_i x_i \leq M$ in state space tree is called the answer state, the remaining nodes are infeasible.
- For the minimum cost answer node, we need to define $\hat{c}(x) = -\sum_{i=1}^n p_i x_i$ for every answer node x_i .
- Cost for infeasible leaf node would be ∞ .
- For all non-leaf nodes, cost function is recursively defined as

$$c(x) = \min\{c(\text{LChild}(x)), c(\text{RChild}(x))\}$$
- For every node x , $\hat{c}(x) \leq c(x) \leq u(x)$.
- Algorithm for knapsack problem using branch and bound is described below :
- For any node N , upper bound for feasible left child remains N . But upper bound for its right child needs to be calculated.

i. 0/1 Knapsack using LCBB:

LC branch and bound solution for knapsack problem is derived as follows:

1. Derive state space tree.
2. Compute lower bound $\hat{c}(x)$ and upper bound $u(x)$ for each node in state space tree.
3. If lower bound is greater than upper bound then kill that node.
4. Else select node with minimum lower bound as E-node.
5. Repeat step 3 and 4 until all nodes are examined.
6. The node with minimum lower bound value $\hat{c}(x)$ is the answer node. Trace the path from leaf to root in the backward direction to find the solution tuple.

Variation:

The bounding function is a heuristic computation. For the same problem, there may be different bounding functions. Apart from the above-discussed bounding function, another very popular bounding function for knapsack is,

$$ub = v + (W - w) * (v_{i+1} / w_{i+1})$$

where,

v is value/profit associated with selected items from the first i items.

W is the capacity of the knapsack.

w is the weight of selected items from first i items

Example:

Solve the following instance of knapsack using LCBB for knapsack capacity M = 15.

i	P _i	W _i
1	10	2
2	10	4
3	12	6
4	18	9

Solution:

Let us compute $u(1)$ and $\hat{c}(1)$. If we include first three item then $\sum w_i \leq M$, but if we include 4th item, it exceeds knapsack capacity. So,

$$u(1) = -\sum p_i \text{ such that } \sum w_i \leq M$$

$$= -(p_1 + p_2 + p_3)$$

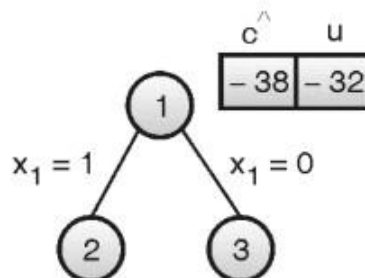
$$= -(10 + 10 + 12) = -32$$

$$\hat{c}(1) = u(1) -$$

$$\frac{M - \text{Weight of selected items}}{\text{Weight of remaining items}} * \text{Profit of remaining items}$$

That gives,

$$\hat{c}(1) = -32 - \frac{15 - (2 + 4 + 6)}{9} * 18 = -38$$

State Space Tree:**Node 2 : Inclusion of item 1 at node 1**

Inclusion of item 1 is compulsory. So we will end up with same items in previous step.

$$u(2) = -(p_1 + p_2 + p_3) = -(10 + 10 + 12) = -32$$

$$\hat{c}(2) = u(2) -$$

$$\frac{M - \text{Weight of selected items}}{\text{Weight of remaining items}} * \text{Profit of remaining items}$$

$$\hat{c}(2) = -32 - \frac{15 - (2 + 4 + 6)}{9} * 18 = -38$$

Node 2 is inserted in list of live nodes.

Node 3 : Exclusion of item 1 at node 1

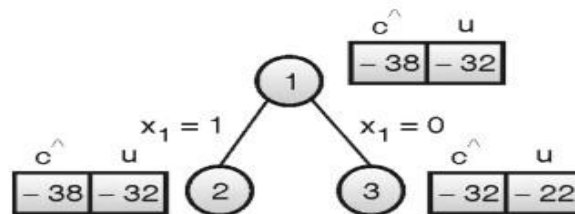
We are excluding item 1 and including 2 and 3. Item 4 cannot be accommodated in knapsack along with items 2 and 3. So,

$$u(3) = -(p_2 + p_3) = -(10 + 12) = -22$$

$$\hat{h}atc(3) = u(3) -$$

$$\frac{M - Weight; of; selected; items}{Weight; of; remaining; items} * Profit; of; remaining; items \hat{h}atc(3) = -22 - \frac{15 - (4 + 6)}{9} * 18 = -32$$

Node 3 is inserted in the list of live nodes.

State-space tree:

At level 1, node 2 has minimum $\hat{h}atc(.)$, so it becomes E-node.

Node 4 : Inclusion of item 2 at node 2

Item 1 is already added at node 2, and we must have to include item 2 at this node. After adding items 1 and 2, the knapsack can accommodate item 3 but not 4.

$$u(4) = -(p_1 + p_2 + p_3) = -(10 + 10 + 12) = -32$$

$$\hat{h}atc(4) = u(4) -$$

$$\frac{M - Weight; of; selected; items}{Weight; of; remaining; items} * Profit; of; remaining; items \hat{h}atc(4) = -32 - \frac{15 - (2 + 4 + 6)}{9} * 18 = -38$$

Node 5: Exclusion of item 2 at node 2

At node 5, item 1 is already added, we must have to skip item 2. Only item 3 can be accommodated in knapsack after inserting item 1. $u(5) = -(p_1 + p_3) = -(10 + 12) = -22$

$$\hat{h}atc(5) = u(5) -$$

$$\frac{M - Weight; of; selected; items}{Weight; of; remaining; items} * Profit; of; remaining; items \hat{h}atc(5) = -22 - \frac{15 - (2 + 6)}{9} * 18 = -36$$

Node 7 : Exclusion of item 2 at node 2

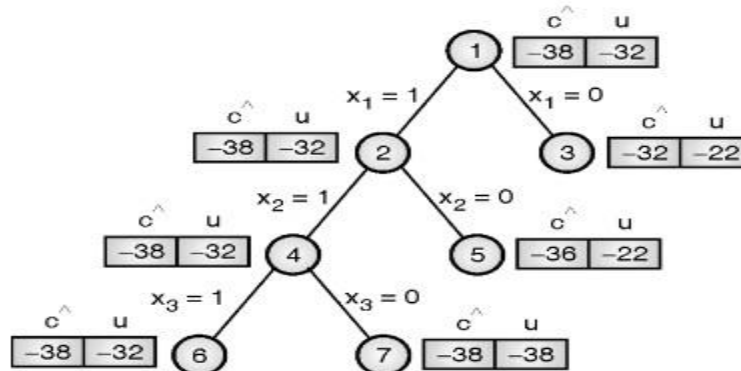
On excluding item 3, we can accommodate item 1, 2 and 4 in knapsack.

$$u(7) = -(10 + 10 + 18) = -38$$

$$hatc(7) = u(7) -$$

$$\frac{M - Weight; of; selected; items}{Weight; of; remaining; items} * Profit; of; remaining; items$$

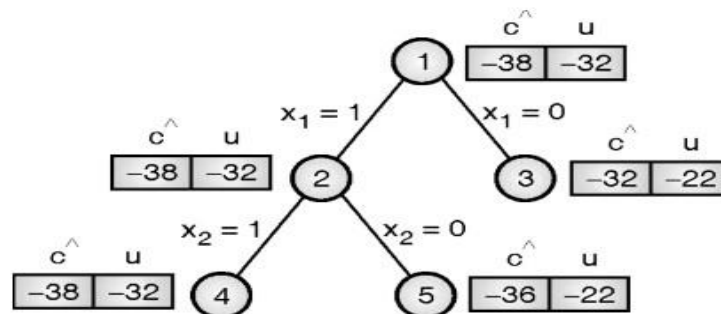
$$hatc(7) = -38 - \frac{15 - (2 + 4 + 9)}{6} * 12 = -38$$

State Space Tree:

When there is tie, we can select any node. Let's make node 7 E-node.

Node 8 : Inclusion of item 4 at node 7

$$u(8) = -(10 + 10 + 18) = -38$$

State Space Tree:

At level 2, node 4 has minimum $hatc(\cdot)$, so it becomes E-node.

Node 6 : Inclusion of item 3 at node 4

At node 4, item 1 and 2 are already added, we have to add item 3. After including item 1, 2 and 3, item 4 cannot be accommodated. $u(6) = -(10 + 10 + 12) = -32$

$$hatc(6) = u(6) -$$

$$\frac{M - Weight; of; selected; items}{Weight; of; remaining; items} * Profit; of; remaining; items$$

$$hatc(6) = -32 - \frac{15 - (2 + 4 + 6)}{9} * 18 = -38$$

$$\hat{c}(8) = u(8) -$$

$$\frac{M - \text{Weight of selected items}}{\text{Weight of remaining items}} * \text{Profit of remaining items}$$

$$\hat{c}(8) = -38 - \frac{15 - (2 + 4 + 9)}{6} * 12 = -38$$

Node 9 : Exclusion of item 4 at node 7

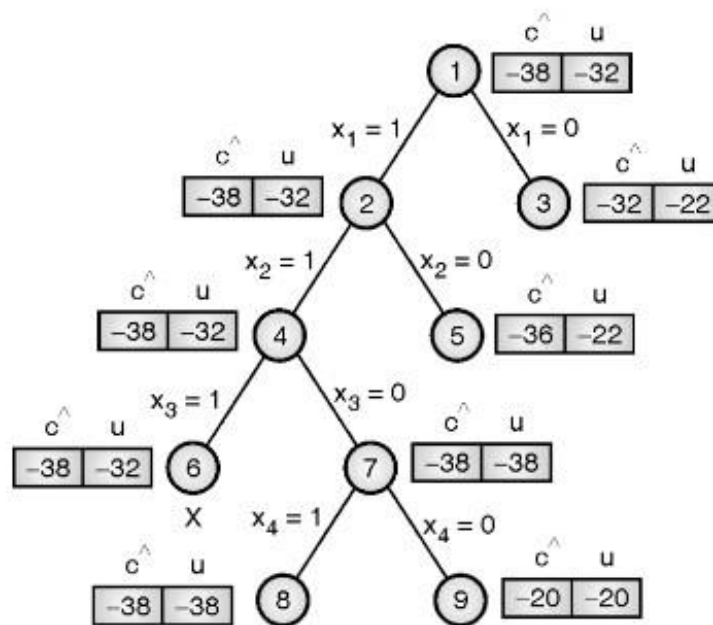
This node excludes both the items, 3 and 4. We can add only item 1 and 2.

$$u(9) = -(10 + 10) = -20$$

$$\hat{c}(9) = u(9) -$$

$$\frac{M - \text{Weight of selected items}}{\text{Weight of remaining items}} * \text{Profit of remaining items}$$

$$\hat{c}(9) = -20 - \frac{15 - (2 + 4)}{0} * 0 = -20$$



- At last level, node 8 has minimum $\hat{c}(\cdot)$, so node 8 would be the answer node. By tracing from node 8 to root, we get item 4, 2 and 1 in the knapsack.

Solution vector $X = \{x_1, x_2, x_4\}$ and profit $= p_1 + p_2 + p_4 = 38$.

Java Program For 0/1 knapsack using LCBB:

knapsack_LCBB.java

```
import java.util.*;
class Item
{
    // Stores the weight    // of items
    float weight;

    // Stores the values    // of items
    int value;

    // Stores the index    // of items
    int idx;
    public Item(){}
    public Item(int value, float weight, int idx)
    {
        this.value = value;
        this.weight = weight;
        this.idx = idx;
    }
}

class Node
{
    // Upper Bound: Best case    // (Fractional Knapsack)

    float ub;

    // Lower Bound: Worst case // (0/1)

    float lb;

    // Level of the node in    // the decision tree
    int level;

    // Stores if the current // item is selected or not

    boolean flag;
```

```
// Total Value: Stores the // sum of the values of the // items included

float tv;

// Total Weight: Stores the sum of the weights of included items
float tw;
public Node() {}
public Node(Node cpy)
{
    this.tv = cpy.tv;
    this.tw = cpy.tw;
    this.ub = cpy.ub;
    this.lb = cpy.lb;
    this.level = cpy.level;
    this.flag = cpy.flag;
}
}

// Comparator to sort based on lower bound

class sortByC implements Comparator<Node>
{
    public int compare(Node a, Node b)
    {
        boolean temp = a.lb > b.lb;
        return temp ? 1 : -1;
    }
}

class sortByRatio implements Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        boolean temp = (float)a.value / a.weight > (float)b.value / b.weight ;
        return temp ? -1 : 1;
    }
}
```

```
}

class knapsack_LCBB
{
    private static int size;
    private static float capacity;

    // Function to calculate upper bound (includes fractional part of the items)

    static float upperBound(float tv, float tw, int idx, Item arr[])
    {
        float value = tv;
        float weight = tw;
        for (int i = idx; i < size; i++)
        {
            if (weight + arr[i].weight
                <= capacity) {
                weight += arr[i].weight;
                value -= arr[i].value;
            }
            else {
                value -= (float)(capacity
                    - weight)
                    / arr[i].weight
                    * arr[i].value;
                break;
            }
        }
        return value;
    }
}
```

// Calculate lower bound (doesn't include fractional part of items)

```
static float lowerBound(float tv, float tw, int idx, Item arr[])
{
    float value = tv;
    float weight = tw;
    for (int i = idx; i < size; i++) {
        if (weight + arr[i].weight
            <= capacity) {
            weight += arr[i].weight;
            value -= arr[i].value;
        }
        else {
            break;
        }
    }
    return value;
}
```

```
static void assign(Node a, float ub, float lb, int level, boolean flag, float tv, float tw)
{
    a.ub = ub;
    a.lb = lb;
    a.level = level;
    a.flag = flag;
    a.tv = tv;
    a.tw = tw;
}
```

```
public static void solve(Item arr[])
{
    // Sort the items based on the profit/weight ratio
    Arrays.sort(arr, new sortByRatio());

    Node current, left, right;
    current = new Node();
    left = new Node();
}
```

```
right = new Node();

// min_lb -> Minimum lower bound of all the nodes explored

// final_lb -> Minimum lower bound of all the paths that reached the final level

float minLB = 0, finalLB = Integer.MAX_VALUE;
current.tv = current.tw = current.ub = current.lb = 0;
current.level = 0;
current.flag = false;

// Priority queue to store elements based on lower bounds
PriorityQueue<Node> pq = new PriorityQueue<Node>( new sortByC());

// Insert a dummy node
pq.add(current);

// curr_path -> Boolean array to store at every index if the element is included or not

// final_path -> Boolean array to store the result of selection array when it reached the
//last level

boolean currPath[] = new boolean[size];
boolean finalPath[] = new boolean[size];

while (!pq.isEmpty())
{
    current = pq.poll();
    if (current.ub > minLB || current.ub >= finalLB)
    {
        // if the current node's best case/ value is not optimal than minLB, then there is no
        //reason to explore that node. Including finalLB eliminates all those paths whose best
        //values is equal to the finalLB.
        continue;
    }
}
```

```

    if (current.level != 0)
        currPath[current.level - 1] = current.flag;

    if (current.level == size)
    {
        if (current.lb < finalLB)
        {
            // Reached last level
            for (int i = 0; i < size; i++)
                finalPath[arr[i].idx] = currPath[i];
            finalLB = current.lb;
        }
        continue;
    }

    int level = current.level;

    // right node -> Excludes current item/ Hence, cp, cw will obtain the value
    // of that of parent

    assign(right, upperBound(current.tv,current.tw, level + 1, arr), lowerBound(current.tv,
    current.tw, level + 1, arr), level + 1, false, current.tv, current.tw);

    if (current.tw + arr[current.level].weight <= capacity)
    {
        // left node -> includes current item c and lb should be calculated
        // including the current item.
        left.ub = upperBound(current.tv - arr[level].value, current.tw + arr[level].weight,
            level + 1, arr);
        left.lb = lowerBound( current.tv - arr[level].value, current.tw + arr[level].weight,
            level + 1, arr);
        assign(left, left.ub, left.lb, level + 1, true, current.tv - arr[level].value, current.tw
            + arr[level].weight);
    }

    // If the left node cannot/ be inserted

```

```

    else {

        // Stop the left node from
        // getting added to the
        // priority queue
        left.ub = left.lb = 1;
    }

    // Update minLB
    minLB = Math.min(minLB, left.lb);
    minLB = Math.min(minLB, right.lb);

    if (minLB >= left.ub)
        pq.add(new Node(left));
    if (minLB >= right.ub)
        pq.add(new Node(right));
}
System.out.println("Items taken"+ "into the knapsack are");
for (int i = 0; i < size; i++) {
    if (finalPath[i])
        System.out.print("1 ");
    else
        System.out.print("0 ");
}
System.out.println("\nMaximum profit" + " is " + (-finalLB));
}

// Driver code
public static void main(String args[])
{
    size = 4;
    capacity = 15;

    Item arr[] = new Item[size];
    arr[0] = new Item(10, 2, 0);
    arr[1] = new Item(10, 4, 1);
    arr[2] = new Item(12, 6, 2);

```

```
arr[3] = new Item(18, 9, 3);  
  
    solve(arr);  
}  
}
```

output=

Items taken into the knapsack are :

1 1 0 1

Maximum profit is : 38

ii. 0/1 Knapsack using FIFOBB

- In FIFO branch and bound approach, variable tuple size state space tree is drawn. For each node N, cost function $\hat{c}(\cdot)$ and upper bound $u(\cdot)$ is computed similarly to the previous approach. In LC search, E node is selected from two child of current node.
- In FIFO branch and bound approach, both the children of siblings are inserted in list and most promising node is selected as new E node.
- Let us consider the same example :

Example: Solve following instance of knapsack using FIFO BB.

i	Pi	Wi
1	10	2
2	10	4
3	12	6
4	18	9

Solution:

Let us compute $u(1)$ and $\hat{c}(1)$.

$$u(1) = \sum p_i \text{ such that } \sum w_i \leq M$$

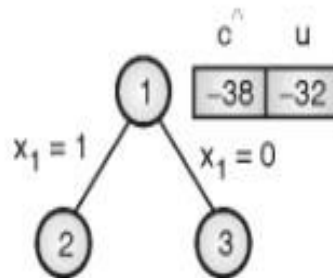
$$= 10 + 10 + 12 = 32$$

$$\text{Upper} = u(1) = 32$$

If we include the first three items then $\sum w_i \leq M$, but if we include 4th item, it exceeds the knapsack capacity.

$$\begin{aligned} \hat{c}(1) &= u(1) - \frac{M - \text{Weight of selected items}}{\text{Weight of remaining items}} * \text{Profit of remaining items} \\ \hat{c}(1) &= 32 - \frac{15 - (2 + 4 + 6)}{9} * 18 = 38 \end{aligned}$$

State space tree:



Node 2 : Inclusion of item 1 at node 1

$$u(2) = -(10 + 10 + 12) = -32$$

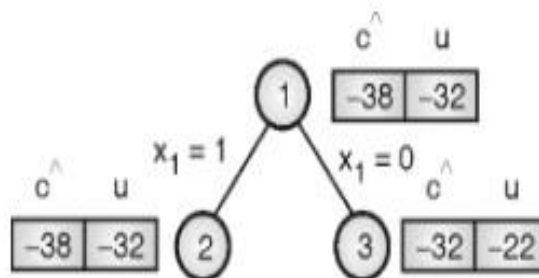
$$hat{c}(2) = -32 - \frac{15}{9} - (2 + 4 + 6) = -38$$

Node 3 : Exclusion of item 1 at node 1

We are excluding item 1, including 2 and 3. Item 4 cannot be accommodated in a knapsack along with 2 and 3.

$$u(3) = -(10 + 12) = -22$$

$$hat{c}(3) = -22 - \frac{15}{9} - (4 + 6) = -32$$



In LC approach, node 2 would be selected as E-Node as it has minimum (*). But in FIFO approach, all child of node 2 and 3 are expanded and the most promising child becomes E-node.

Node 4 : Inclusion of item 2 at node 2

$$u(4) = -(10 + 10 + 12) = -32$$

$$hatc(4) = -32 - \frac{15}{9} - (2 + 4 + 6) * 18 = -38$$

Node 5 : Exclusion of item 2 at node 2

We are excluding item 1, including 2 and 3. Item 4 cannot be accommodated in a knapsack along with 2 and 3.

$$u(5) = -(10 + 12) = -22$$

$$hatc(5) = -22 - \frac{15}{9} - (4 + 6) * 18 = -32$$

Node 6 : Inclusion of item 2 at node 3

$$u(6) = -(p_2 + p_3) = -(10 + 12) = -22$$

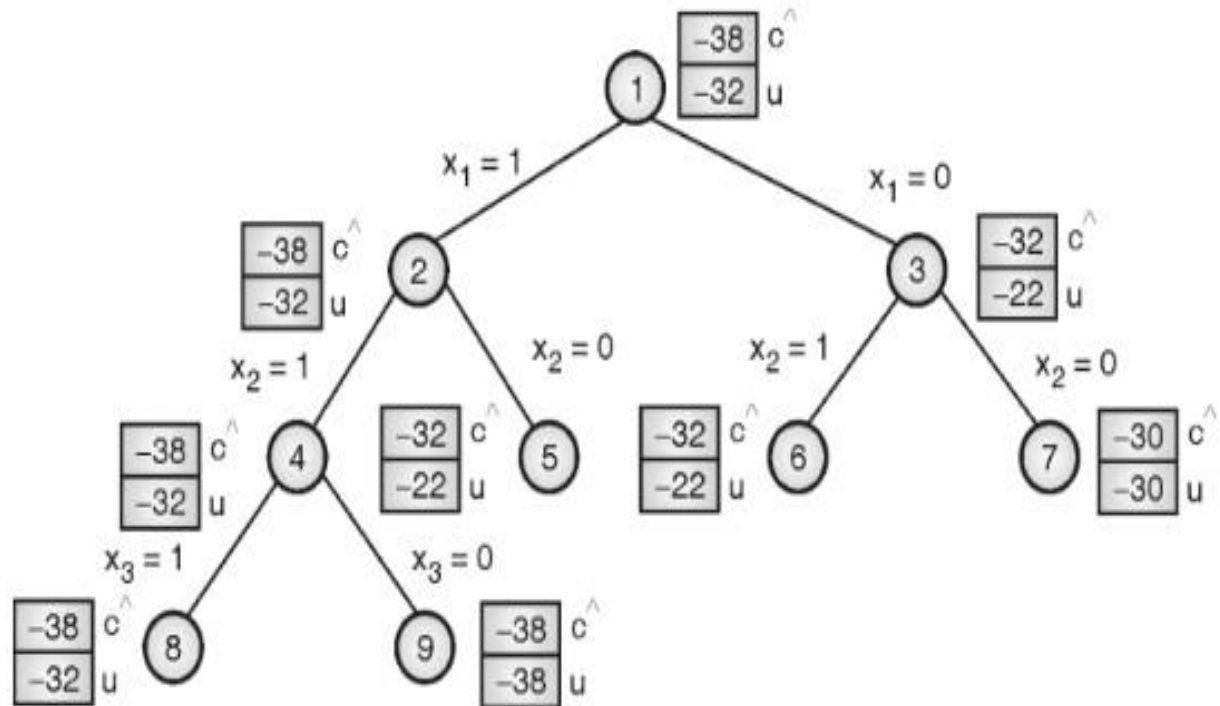
$$hatc(6) = -22 - \frac{15}{9} - (4 + 6) * 18 = -32$$

Node 7 : Exclusion of item 2 at node 3

We are excluding item 1, including 2 and 3. Item 4 cannot be accommodated in a knapsack along with 2 and 3.

$$u(7) = -(p_3 + p_4) = -(12 + 18) = -30$$

$$hatc(7) = -30 - 0 = -30$$



Out of node 4, 5, 6 and 7, $hatc(7) > upper$, so kill node 7. Remaining 3 nodes are live and added in list.

Out of 4, 5 and 6, node 4 has minimum $hatc$ value, so it becomes next E-node.

Node 8 : Inclusion of item 3 at node 4

$$u(8) = -(10 + 10 + 12) = -32$$

$$hatc(8) = -32 - \frac{15}{9} - (2 + 4 + 6) * 18 = -38$$

Node 9 : Exclusion of item 3 at node 4

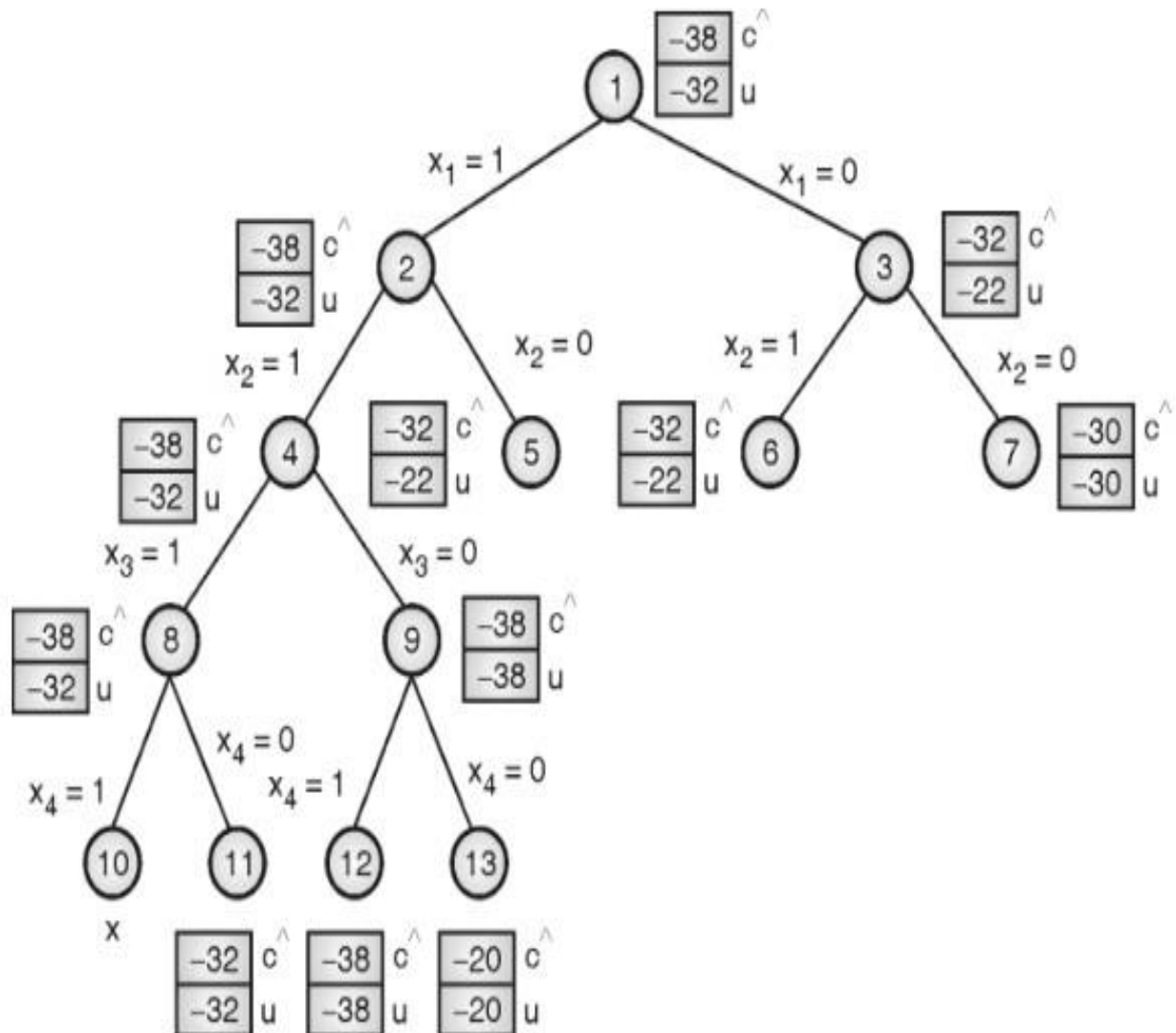
We are excluding item 3, including 1 and 2 and 4.

$$u(9) = -(p_1 + p_2 + p_4) = -(10 + 10 + 18) = -38$$

$$hatc(9) = -38 - 0 = -38$$

$$Upper = u(9) = -38.$$

$\hat{h}atc(5) > \text{upper}$ and $\hat{h}atc(6) > \text{upper}$, so kill them. If we continue in this way, final state space tree looks as follow :



Node 12 has minimum cost function value, so it will be the answer node.

Solution vector = $\{x_1, x_2, x_4\}$,

profit = $10 + 10 + 18 = 38$