Branch and Bound: General Method, FIFO Branch and Bound, LC (least cost) Branch and Bound, Applications: Travelling Salesperson Problem,0/1 Knapsack 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).
- Branch and bound is a systematic way of exploring all possible solutions to a problem by dividing the problem space into smaller sub-problems and then applying bounds or constraints to eliminate certain subproblems from consideration.
- This method is best when used for combinatorial problems with exponential time complexity, since it provides a more efficient solution to such problems.

Characteristics of Branch and Bound:

- Optimal solution: The algorithm is designed to find the optimal solution to an optimization problem by searching the solution space in a systematic way.
- Upper and lower bounds: The algorithm uses upper and lower bounds to reduce the size of the search space and eliminate subproblems that cannot contain the optimal solution.
- Pruning: The algorithm prunes the search tree by eliminating subproblems that cannot contain the optimal solution or are not worth exploring further.
- Backtracking: The algorithm uses backtracking to return to a previous node in the search tree when a dead end is reached or when a better solution is found.

Applications of Branch and Bound:

- Traveling salesman problem: Branch and bound is used to solve the traveling salesman problem, which involves finding the shortest possible path that visits a set of cities and returns from where it started to visit the set of cities.
- Knapsack problem: Branch and bound is used to solve the knapsack problem, which involves finding the optimized combination of items to pack into a knapsack of limited capacity.
- Resource allocation: Branch and bound is used to solve resource allocation problems, like scheduling work on machines or assigning work to workers.
- Network optimization: Branch and bound is used to solve network optimization problems, it helps in finding the optimized path or flow through a network.
- Game playing: Branch and bound is used in some of the game-playing algorithms, like chess or tic-tac or 16 puzzle problem, to explore the various possible moves and find the optimized strategies.

Advantages of Branch and Bound:

- Optimal solution: Branch and bound algorithm is created to find the best answer to an optimization issue by methodically searching the solution space.
- Reduces search space: The algorithm uses lower and upper bounds to cut down on the size of the search area and get rid of sub-problems that cannot have the best answer.
- We don't explore all the nodes in a branch and bound algorithm. Due to this, the branch and the bound algorithm have a lower time complexity than other algorithms.
- Whenever the problem is small and the branching can be completed in a reasonable amount of time, the algorithm finds an optimal solution.
- By using the branch and bound algorithm, the optimal solution is reached in a minimal amount of time. When exploring a tree, it does not repeat nodes.

Disadvantages of Branch and Bound:

- Exponential time complexity: The branch and bound algorithm's worst-case time complexity is exponential in the size of the input, making it unsuitable for handling complex optimization issues.
- Memory-intensive: To store the search tree and the current best answer, the method needs a lot of memory. When dealing with numerous instances of the issue, this may become a problem.
- Sensitivity to problem-specific parameters: The quality of the problem-specific constraints utilized determines how well the method performs, and sometimes it might be challenging to discover good bounds.
- Limited scalability: Due to the size of the search tree which expands exponentially with the size of the problem, the Branch and Bound technique may not scale effectively for problems with huge search spaces.

Types of Solutions:

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

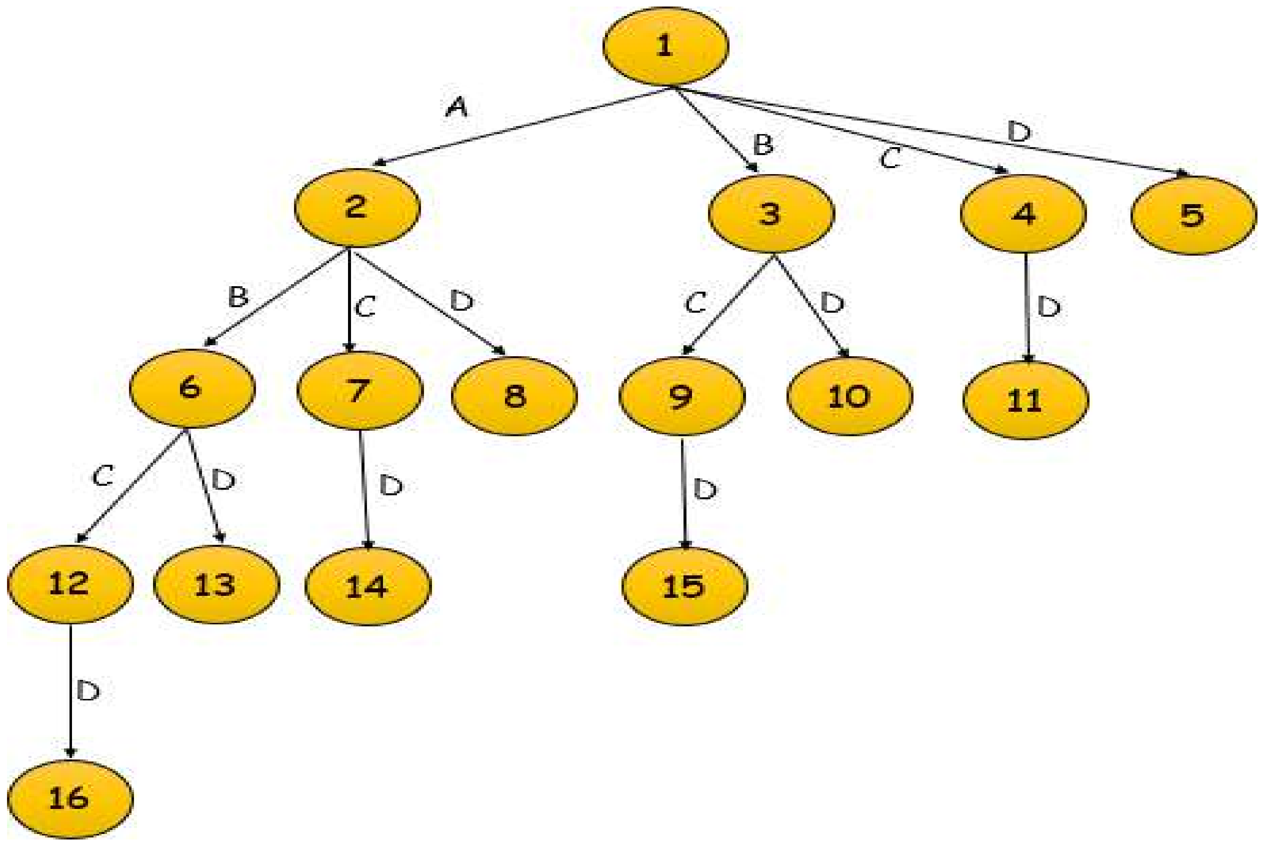
- **Variable size solution**: This solution provides the subset of the given set that gives the optimized solution for the given problem. For example, if the we are to select a combination of elements from {A, B, C, D, E} that optimizes the given problem, and it is found that A, B, and E together give 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

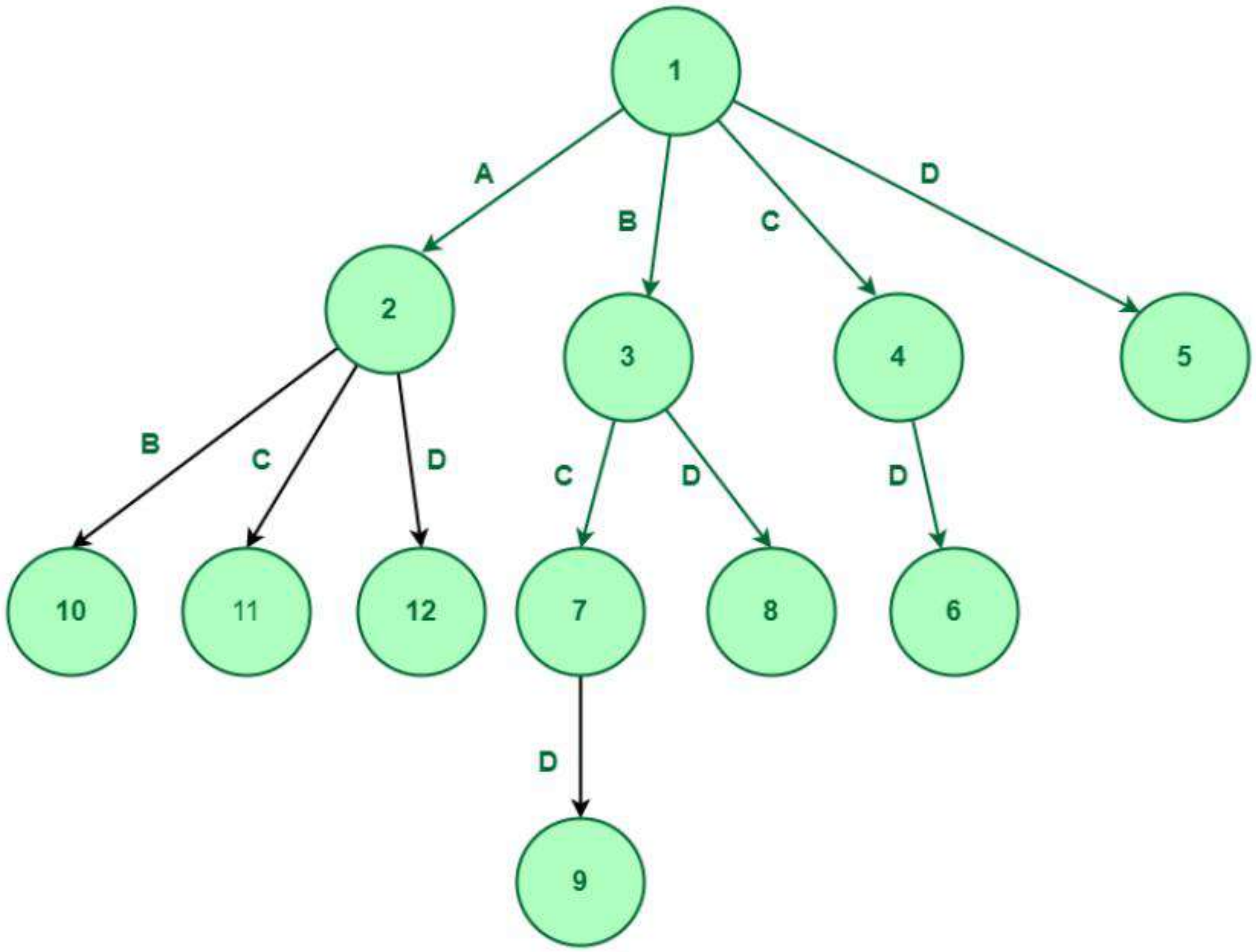
The First-In-First-Out approach to the branch and bound problem follows the **queue** approachin creating the state-space tree. Here, breadth first search is performed, i.e., the elements at a particular level are all searched, and then the elements of the next level are searched, starting from the first 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 5, 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.

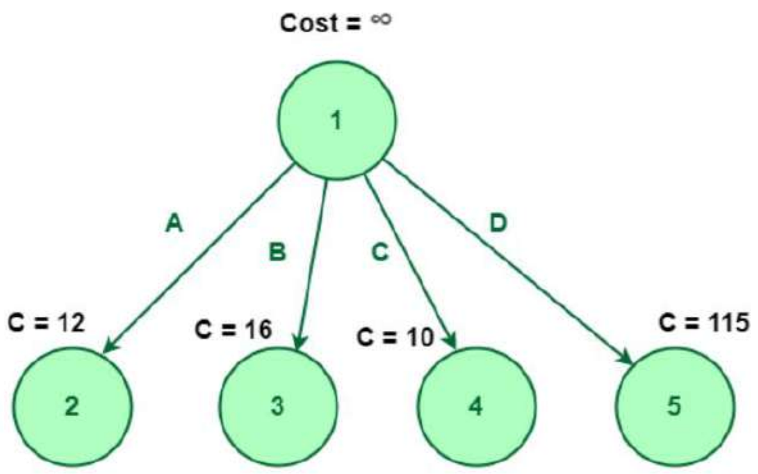
3. Least Cost-Branch and Bound

To explore the state space tree, this method uses the cost function. The previous two methods also calculate the cost function at each node but the cost is not been used for further exploration.

In this technique, nodes are explored based on their costs, the cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.
Now, consider a node whose cost has been determined. If this value is greater than U0, this node or its children will not be able to give a solution. As a result, we can kill this node and not explore its further branches. As a result, this method prevents us from exploring cases that are not worth it, which makes it more efficient for us.

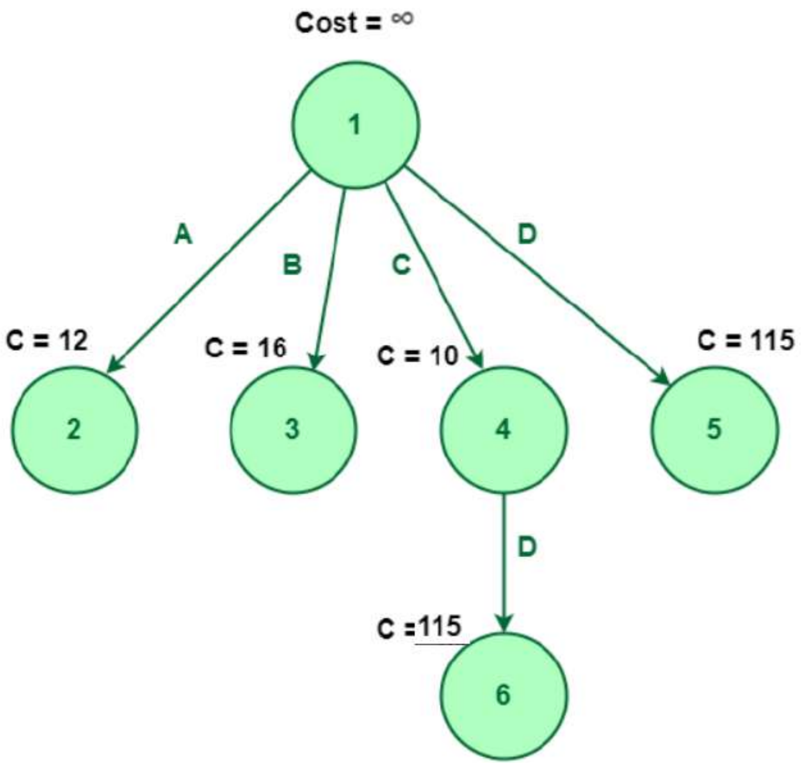
Let’s first consider node 1 having cost infinity shown below:

In the following diagram, node 1 is expanded into four nodes named 2, 3, 4, and 5.



In this method, we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 4. So, we will explore node 4 having a cost of 10.

During exploring node 4 which is element C, we can notice that there is only one possible element that remains unexplored which is D (i.e, we already decided not to select elements A, and B). So, it will get expanded to one single element D, let's say this node number is 6.



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. <u>Travelling Salesperson Problem using LCBB:</u>

- 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

- <u>Branch and bound</u> 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} = ∞, 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 <i, j> 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, i] = ∞
   (c) Reduce A again, except rows and columns having all ∞ entries.

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

Cost = L + Cost(i, j) + r

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

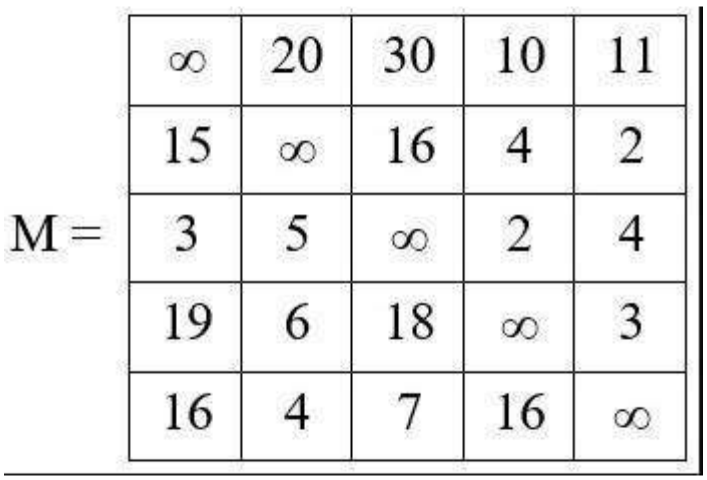
8. If all nodes are not visited then go to step 4.

Reduction procedure is described below :

Row Reduction:

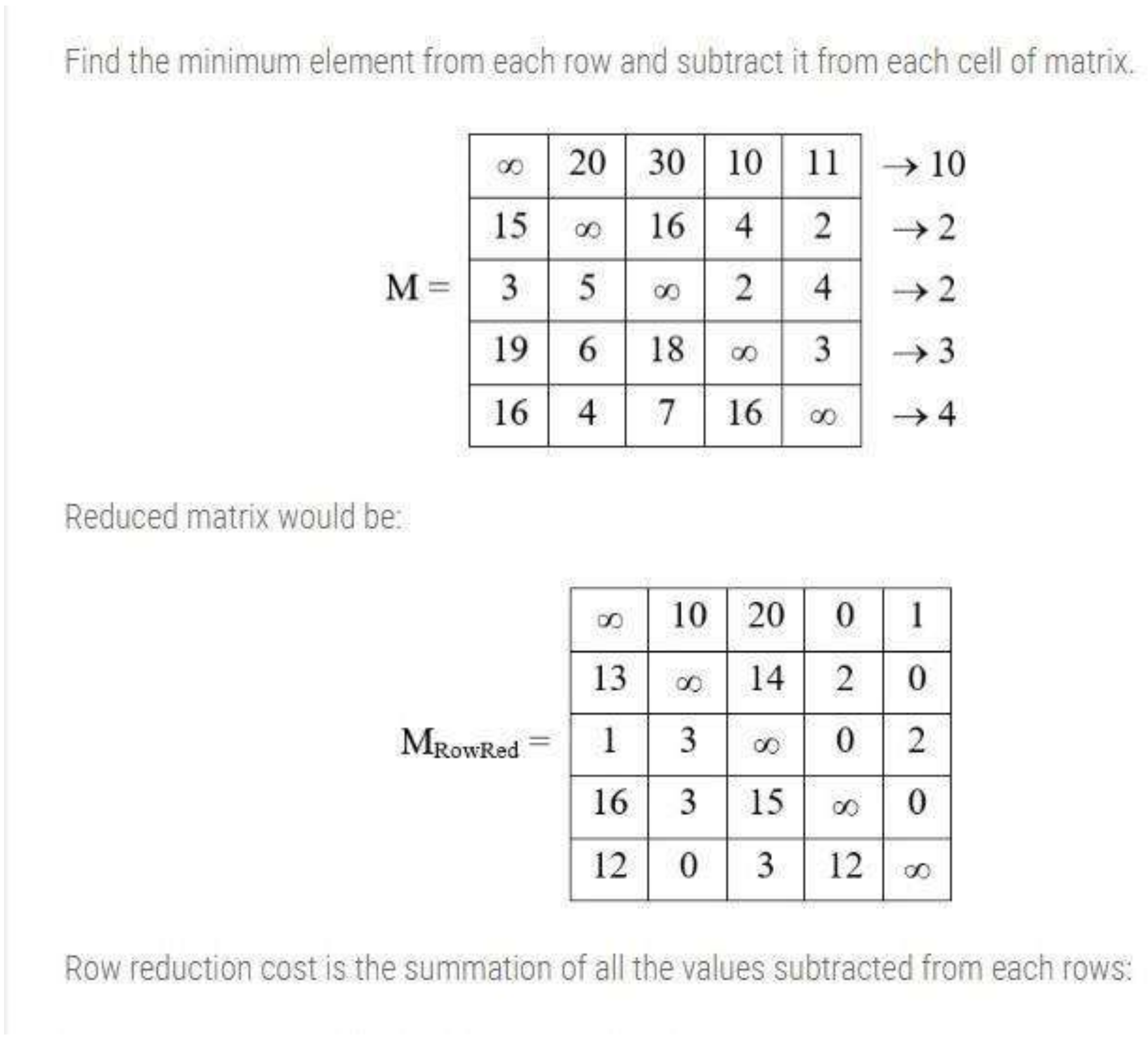
Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow:

$M_{RowRed} = \{M_{ij} - \min \{M_{ij} \mid 1 \le j \le n, \text{ and } M_{ij} < \infty \}\}$
Consider the following distance matrix:



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

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.

$$M = \begin{array}{|c|c|c|c|c|l} \hline \infty & 20 & 30 & 10 & 11 & \rightarrow 10 \\ \hline 15 & \infty & 16 & 4 & 2 & \rightarrow 2 \\ \hline 3 & 5 & \infty & 2 & 4 & \rightarrow 2 \\ \hline 19 & 6 & 18 & \infty & 3 & \rightarrow 3 \\ \hline 16 & 4 & 7 & 16 & \infty & \rightarrow 4 \\ \hline \end{array}$$

Reduced matrix would be:

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

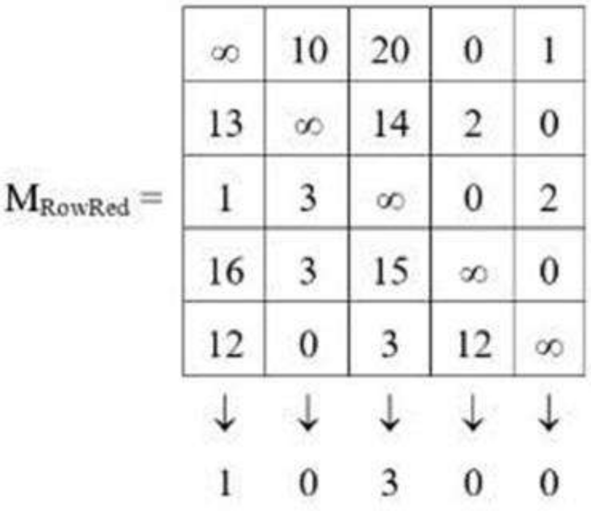
Row reduction cost is the summation of all the values subtracted from each rows:

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

Column reduction:

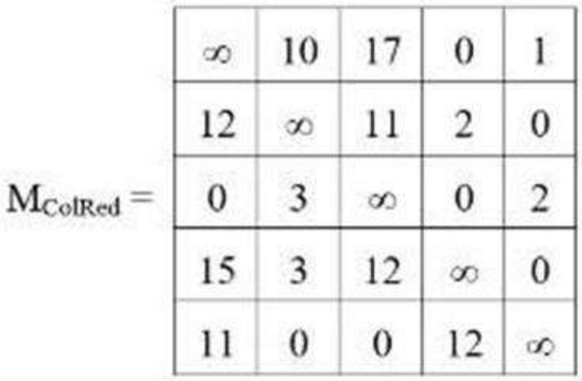
Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.
$M_{ColRed} = \{M_{ji} – \min \{M_{ji} \mid 1 \le j \le n, \text{ and } M_{ji} < \infty \}\}$

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



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

Column reduced matrix M_{ColRed} would be:



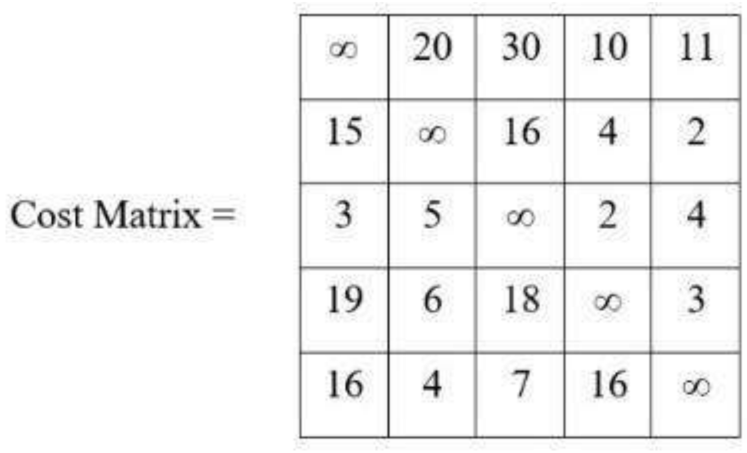
$$M_{ColRed} = \begin{array}{|c|c|c|c|c|} \hline \infty & 10 & 17 & 0 & 1 \\ \hline 12 & \infty & 11 & 2 & 0 \\ \hline 0 & 3 & \infty & 0 & 2 \\ \hline 15 & 3 & 12 & \infty & 0 \\ \hline 11 & 0 & 0 & 12 & \infty \\ \hline \end{array}$$

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

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] = ∞

- 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.

$$\begin{array}{|c|c|c|c|c|}\hline \infty & 20 & 30 & 10 & 11 \\ \hline 15 & \infty & 16 & 4 & 2 \\ \hline 3 & 5 & \infty & 2 & 4 \\ \hline 19 & 6 & 18 & \infty & 3 \\ \hline 16 & 4 & 7 & 16 & \infty \\ \hline \end{array} \begin{array}{c} \to 10 \\ \to 2 \\ \to 2 \\ \to 3 \\ \to 4 \end{array} \Rightarrow \begin{array}{ccccc} \begin{array}{|c|c|c|c|c|}\hline \infty & 10 & 20 & 0 & 1 \\ \hline 13 & \infty & 14 & 2 & 0 \\ \hline 1 & 3 & \infty & 0 & 2 \\ \hline 16 & 3 & 15 & \infty & 0 \\ \hline 12 & 0 & 3 & 12 & \infty \\ \hline \end{array} \end{array} = M_1'$$

$$\begin{array}{ccccc} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 0 & 3 & 0 & 0 \end{array}$$

M_1' is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

$$\begin{array}{|c|c|c|c|c|}\hline \infty & 10 & 17 & 0 & 1 \\ \hline 12 & \infty & 11 & 2 & 0 \\ \hline 0 & 3 & \infty & 0 & 2 \\ \hline 15 & 3 & 12 & \infty & 0 \\ \hline 11 & 0 & 0 & 12 & \infty \\ \hline \end{array} = M_1$$

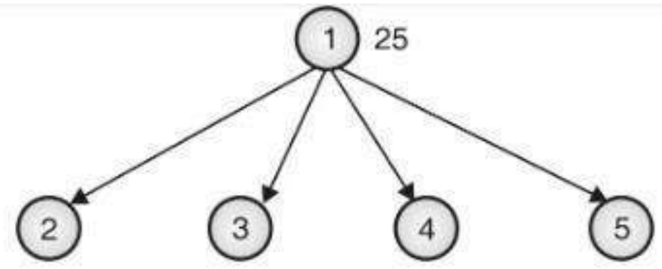
Cost of M_1 = C(1)

= 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] = ∞

Set M_1 [2] [1] = ∞

Reduce the resultant matrix if required.

$$\begin{array}{|c|c|c|c|c|l}
\hline
\infty & \infty & \infty & \infty & \infty & \rightarrow x \\
\hline
\infty & \infty & 11 & 2 & 0 & \rightarrow 0 \\
\hline
0 & \infty & \infty & 0 & 2 & \rightarrow 0 \\
\hline
15 & \infty & 12 & \infty & 0 & \rightarrow 0 \\
\hline
11 & \infty & 0 & 12 & \infty & \rightarrow 0 \\
\hline
\multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \\
\multicolumn{1}{c}{0} & \multicolumn{1}{c}{x} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} &
\end{array} = M_2$$

M_2 is already reduced.

Cost of node 2 :

C(2) = C(1) + Reduction cost + M_1 [1] [2]

= 25 + 0 + 10 = 35

Select edge 1-3

Set M_1 [1][ ] = M_1 [ ] [3] = ∞

Set M_1 [3][1] = ∞

Reduce the resultant matrix if required.

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

Cost of node 3:

C(3) = C(1) + Reduction cost + M1[1][3]

= 25 + 11 + 17 = 53

Select edge 1-4:

Set M_1 [1][ ] = M_1[ ][4] = ∞

Set M_1 [4][1] = ∞

Reduce resultant matrix if required.

$$M_1 \Rightarrow \begin{array}{|c|c|c|c|c|l}
\hline
\infty & \infty & \infty & \infty & \infty & \rightarrow x \\
\hline
12 & \infty & 11 & \infty & 0 & \rightarrow 0 \\
\hline
0 & 3 & \infty & \infty & 2 & \rightarrow 0 \\
\hline
\infty & 3 & 12 & \infty & 0 & \rightarrow 0 \\
\hline
11 & 0 & 0 & \infty & \infty & \rightarrow 0 \\
\hline
\multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \\
\multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{x} & \multicolumn{1}{c}{0} &
\end{array} = M_4$$

Matrix M_4 is already reduced.

Cost of node 4:

C(4) = C(1) + Reduction cost + M_1 [1] [4]

= 25 + 0 + 0 = 25

Select edge 1-5:

Set M_1 [1] [ ] = M_1 [ ] [5] = ∞

Set M_1 [5] [1] = ∞

Reduce the resultant matrix if required.

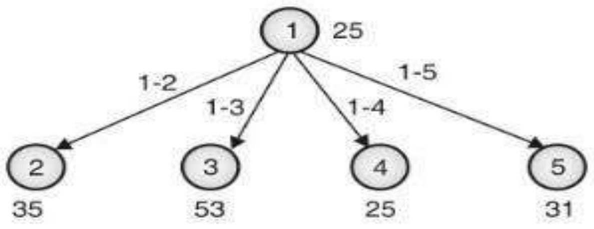
$$M_1 \Rightarrow \begin{array}{|c|c|c|c|c|l}\hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline 12 & \infty & 11 & 2 & \infty & \rightarrow 2 \\ \hline 0 & 3 & \infty & 0 & \infty & \rightarrow 0 \\ \hline 15 & 3 & 12 & \infty & \infty & \rightarrow 3 \\ \hline \infty & 0 & 0 & 12 & \infty & \rightarrow 0 \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline 10 & \infty & 9 & 0 & \infty \\ \hline 0 & 3 & \infty & 0 & \infty \\ \hline 12 & 0 & 9 & \infty & \infty \\ \hline \infty & 0 & 0 & 12 & \infty \\ \hline \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} \\ \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{x} \end{array} = M_5$$

Cost of node 5:

C(5) = C(1) + 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] = ∞

Set M_4 [2] [1] = ∞

Reduce resultant matrix if required.

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|l}\hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline \infty & \infty & 11 & \infty & 0 & \rightarrow 0 \\ \hline 0 & \infty & \infty & \infty & 2 & \rightarrow 0 \\ \hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline 11 & \infty & 0 & \infty & \infty & \rightarrow 0 \\ \hline \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \\ \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{x} & \multicolumn{1}{c}{0} & \end{array} = M_6$$

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.

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|l}\hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline 12 & \infty & \infty & \infty & 0 & \rightarrow 0 \\ \hline \infty & 3 & \infty & \infty & 2 & \rightarrow 2 \\ \hline \infty & \infty & \infty & \infty & \infty & \rightarrow \infty \\ \hline 11 & 0 & \infty & \infty & \infty & \rightarrow 0 \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline 12 & \infty & \infty & \infty & 0 \\ \hline \infty & 1 & \infty & \infty & 0 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 11 & 0 & \infty & \infty & \infty \\ \hline \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} \\ \multicolumn{1}{c}{11} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{x} & \multicolumn{1}{c}{x} & \multicolumn{1}{c}{0} \end{array} = M_7'$$

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):

$$M_4 \Rightarrow \begin{array}{ccccc|c} \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ 12 & \infty & 11 & \infty & \infty & \rightarrow 11 \\ 0 & 3 & \infty & \infty & \infty & \rightarrow 0 \\ \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \infty & 0 & 0 & \infty & \infty & \rightarrow 0 \end{array} \Rightarrow \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 0 & 0 & x & x \end{array} = M_8$$

Matrix M_8 is reduced.

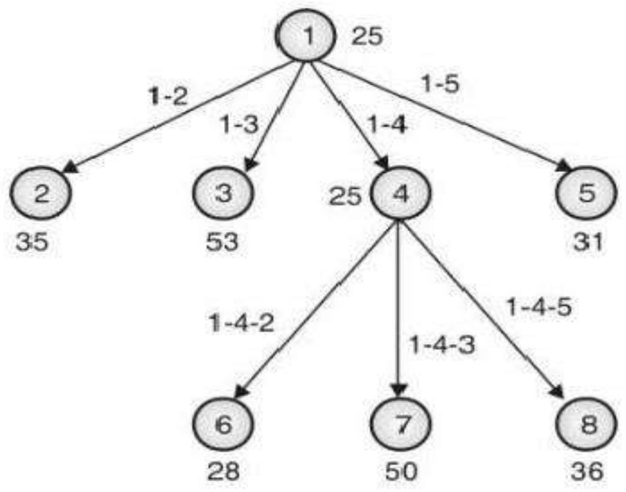
Cost of node 8:

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

$= 25 + 11 + 0 = 36$

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):

Set $M_6\,[1][\,] = M_6\,[4][\,] = M_6\,[2][\,]$

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

Set $M_6\,[3][1] = \infty$

Reduce resultant matrix if required.

$$M_6 \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 11 & \infty & \infty & \infty & \infty \\ \hline \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow x \\ \rightarrow 0 \\ \rightarrow x \\ \rightarrow 11 \end{array} \Rightarrow \begin{array}{ccccc} \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & \infty \\ \hline \end{array} \\ \downarrow\ \ \downarrow\ \ \downarrow\ \ \downarrow\ \ \downarrow \\ 0\ \ \ x\ \ \ x\ \ \ x\ \ \ 2 \end{array} = M_9'$$

$$\therefore M_9' \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & 0 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & \infty \\ \hline \end{array} = M_9$$

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):

Set $M_6\,[1][\,] = M_6\,[4][\,] = M_6\,[2][\,] = M_6\,[\,][5] = \infty$

Set $M_6\,[5][1] = \infty$

Reduce resultant matrix if required.

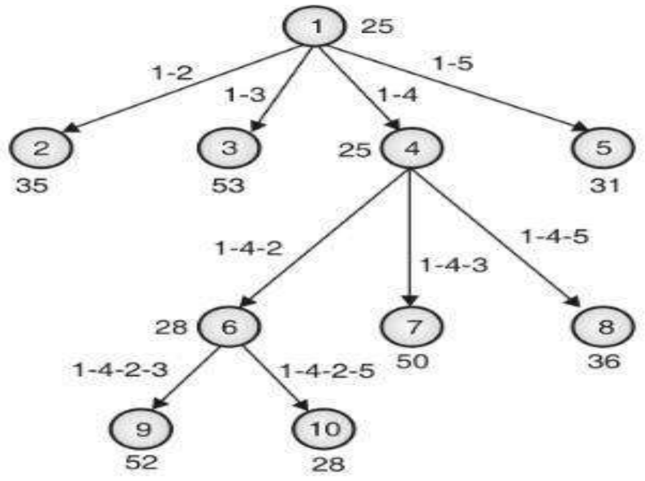
$$\therefore M_6 \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & 0 & \infty & \infty \\ \hline \end{array} = M_{10}$$

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):

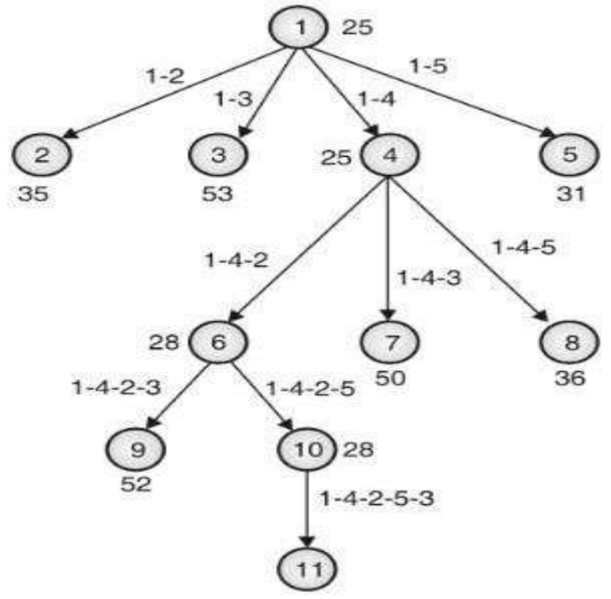
$$\therefore\ M_{10} \Rightarrow \begin{array}{|c|c|c|c|c|}\hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty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

Cost of node 11:

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

$= 28 + 0 + 0 = 28$

State space tree:



2. <u>0/1 knapsack Problem:</u>

Knapsack Problem using Branch and Bound with LC.

- As discussed earlier, the goal of knapsack problem is to maximize sum_{i}^{} p_i x_i*sumipi xi* given the constraints sum_{i}^{} w_i x_i le M*sumiwi xileM*, 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}^{} p_i x_i−*sumipi xi* subjected to sum_{i}^{} w_i x_i le M*sumiwi xileM*,
Where, x_i in {0, 1}, 1 le i le n*xiin*0,1,1*leilen*

- Node satisfying the constraint sum_{i}^{} w_i x_i le M*sumiwi xileM* 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}^{} p_i x_i*hatc*(*x*)=−*sumipi xi* 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(LChild(x)), c(RChild(x)) }
- For every node x, hat{c}(x) le c(x) le u(x)*hatc*(*x*)*lec*(*x*)*leu*(*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. <u>0/1 Knapsack using LCBB:</u>

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

1. Derive state space tree.
2. Compute lower bound/cost hat{c}(.)*hatc*(.) and upper bound u(.)*u*(.) for each node in state spacetree.
3. If lower bound is greater than upper bound than 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}(.)*hatc*(.) 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,

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

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 $hatc(1)$. If we include first three item then $sumw_ileM$, but if we include 4th item, it exceeds knapsack capacity. So,

u(1) $= -sump_i$ such that $sumw_ileM$

$= -(p_1 + p_2 + p_3)$

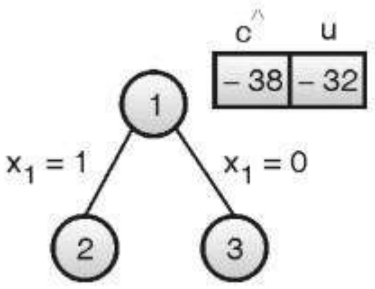
= -(10 + 10 + 12) = − 32

$hatc(1) = u(1) - fracM - Weight;of;selected;itemsWeight;of;remaining;items * Profit;of;remaining;items$

That gives,

$hatc(1) = -32 - frac15 - (2+4+6)9 * 18$ = -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.

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

$hatc(2) = u(2) - fracM - Weight;of;selected;itemsWeight;of;remaining;items * Profit;of;remaining;items$ $hatc(2) = -32 - frac15 - (2+4+6)9 * 18$ = -38

Node 2 is inserted in list of live nodes.

Node 3 : Exclusion of item 1 at node 1

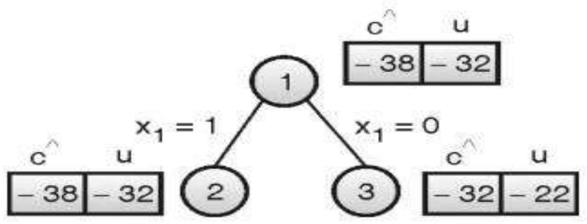
We are excluding item 1 and including 2 and 3. Item 4 cannot be accommodated in knapsack along with items 2 and 3. So,

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

$hatc(3) = u(3) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(3) = -22 - frac15 - (4 + 6)9 * 18$ = -32

Node 3 is inserted in the list of live nodes.

State-space tree:



At level 1, node 2 has minimum $hatc(.)$, 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

$hatc(4) = u(4) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(4) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -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

$hatc(5) = u(5) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(5) = -22 - frac15 - (2 + 6)9 * 18$ = -36

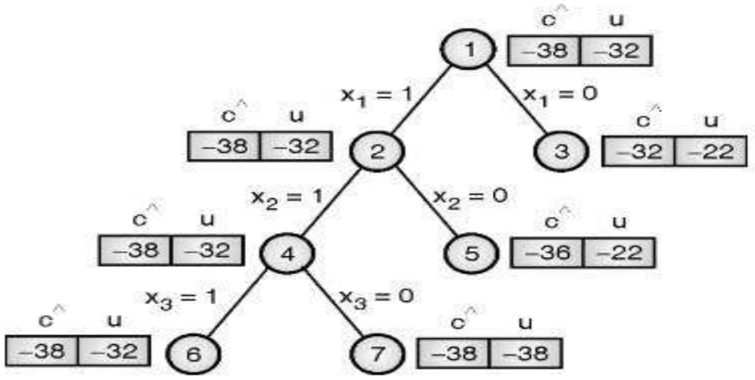
Node 7 : Exclusion of item 2 at node 2

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

u(7) = -(10 + 10 + 18) = − 38

$hatc(7) = u(7) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(7) = -38 - frac15 - (2 + 4 + 9)6 * 12$ = -38

State Space Tree:

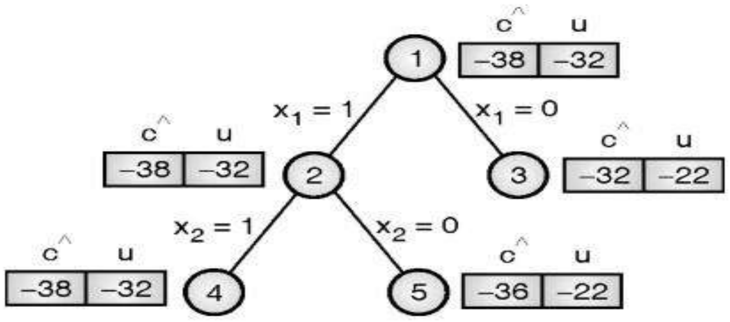


When there is tie, we can select any node. Let's make node 7 E-node.

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 $hatc(.)$, 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

$hatc(6) = u(6) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(6) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -38

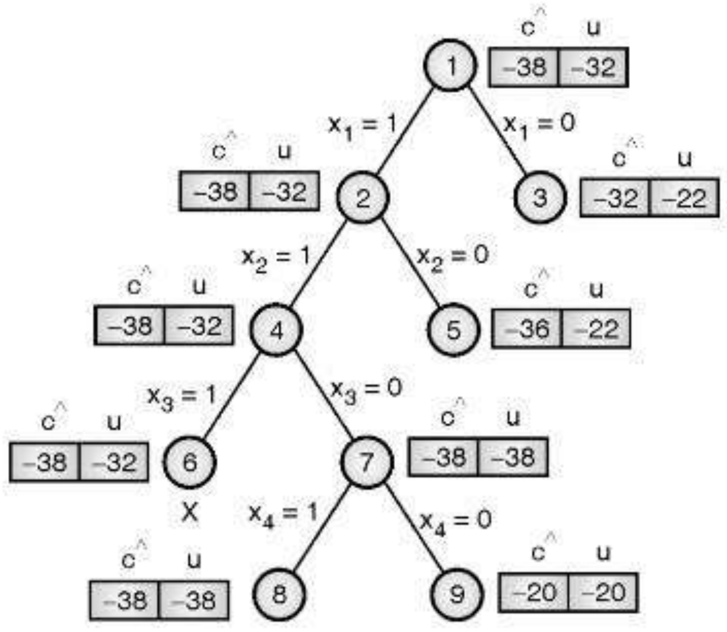
$hatc(8) = u(8) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(8) = -38 - frac15 - (2 + 4 + 9)6 * 12$ = -38

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

$hatc(9) = u(9) - fracM - Weight; of; selected; itemsWeight; of; remaining; items * Profit; of; remaining; items\ hatc(9) = -20 - frac15 - (2 + 4)0 * 0$ = -20



- At last level, node 8 has minimum hat{c}(.)$hatc(.)$, 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 kanpsack using LCBB: **knapsack_LCBB.java**

```
import java.util.*;

class Item {
    int value;
    int weight;

    Item(int value, int weight) {
        this.value = value;
        this.weight = weight;
    }
    public String toString()
    {
        return "[" + value + "," + weight + "]";
    }
}

class Node {
    int level, profit, bound;
    int weight;

    Node(int level, int profit, int weight) {
        this.level = level;
        this.profit = profit;
        this.weight = weight;
    }
}

public class Knapsack_LCBB {
    static Comparator<Item> itemComparator = (a, b) -> {
        double ratio1 = (double) a.value / a.weight;
        double ratio2 = (double) b.value / b.weight;
        // Sorting in decreasing order of value per unit weight
        return Double.compare(ratio2, ratio1);
    };

    static int bound(Node u, int n, int W, Item[] arr) {
        //System.out.println("level " + u.level + " n " + n + " profit " + u.profit + " weight " +
u.weight);
        if (u.weight >= W)
            return 0;
        int profitBound = u.profit;
        int j = u.level + 1;
        int totalWeight = u.weight;

        while (j < n && totalWeight + arr[j].weight <= W) {
            totalWeight += arr[j].weight;
            profitBound += arr[j].value;
            j++;
        }

        if (j < n)
            profitBound += (int) ((W - totalWeight) * arr[j].value / arr[j].weight);

        return profitBound;
    }

    static int knapsack(int W, Item[] arr, int n) {
        System.out.println("Before sort " + Arrays.toString(arr));
        Arrays.sort(arr, itemComparator);
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>((a, b) ->
Integer.compare(b.bound, a.bound));
        Node u, v;
        System.out.println("After sort " + Arrays.toString(arr));

        u = new Node(-1, 0, 0);
        priorityQueue.offer(u);

        int maxProfit = 0;

        while (!priorityQueue.isEmpty()) {
            u = priorityQueue.poll();

            if (u.level == -1)
                v = new Node(0, 0, 0);
            else if (u.level == n - 1)
                continue;
            else
                v = new Node(u.level + 1, u.profit, u.weight);

            v.profit += arr[v.level].value;
            v.weight += arr[v.level].weight;

            //System.out.println("level " + v.level + " profit " + v.profit + " weight " + v.weight);
            // Compute profit of next level node.
            // If the profit is more than maxProfit, then update maxProfit.
            if (v.weight <= W && v.profit > maxProfit)
                maxProfit = v.profit;

            // Compute bound of next level node.
            // If bound is more than maxProfit, then add next level node to Q.
            v.bound = bound(v, n, W, arr);
            //System.out.println("Item included level " + v.level + " bound " + v.bound + " maxProfit "
+ maxProfit);

            if (v.bound > maxProfit)
                priorityQueue.offer(v);

            v = new Node(u.level + 1, u.profit, u.weight);
            v.bound = bound(v, n, W, arr);

            //System.out.println("Item not included level " + v.level + " bound " + v.bound + "
maxProfit " + maxProfit);

            if (v.bound > maxProfit)
                priorityQueue.offer(v);
        }

        return maxProfit;
    }

    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        int size = sc.nextInt();
        int wt = sc.nextInt();

        Item arr[] = new Item[size];
        for(int i=0;i<size;i++)
            arr[i] = new Item(sc.nextInt(), sc.nextInt());

        int maxProfit = knapsack(wt, arr, size);
        System.out.println("Maximum possible profit = " + maxProfit);
    }
}
```

input=

```
4
15
12 6
10 2
10 4
18 9
```

output= 38

ii. <u>0/1 Knapsack using FIFOBB</u>

- In FIFO branch and bound approach, variable tuple size state space tree is drawn. For each node N, cost function hat{c}(.)*hatc*(.) and upper bound u(.) 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 $hatc(1)$.

u(1) = $sump_i$ such that $sumw_i leM$

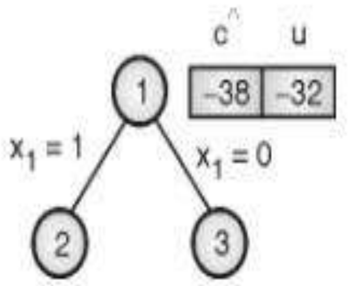
= − (10 + 10 + 12) = − 32

Upper = u(1) = − 32

If we include the first three items then $sumw_i leM$, but if we include 4th item, it exceeds the knapsack capacity.

$hatc(1) = u(1) - fracM - Weight;of;selected;itemsWeight;of;remaining;items * Profit;of;remaining;items$ $hatc(1) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -38

State space tree:



Node 2 : Inclusion of item 1 at node 1

u(2) = -(10 + 10 + 12) = – 32

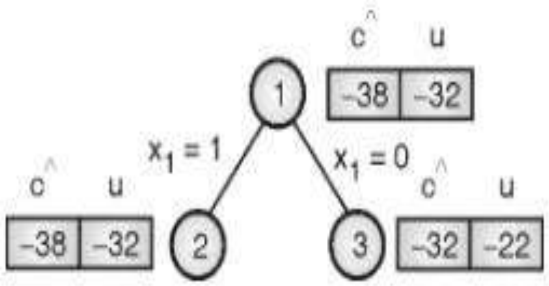
$hatc(2) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -38

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

$hatc(3) = -22 - frac15 - (4 + 6)9 * 18$ = -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

$hatc(4) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -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

$hatc(5) = -22 - frac15 - (4 + 6)9 * 18$ = -32

Node 6 : Inclusion of item 2 at node 3

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

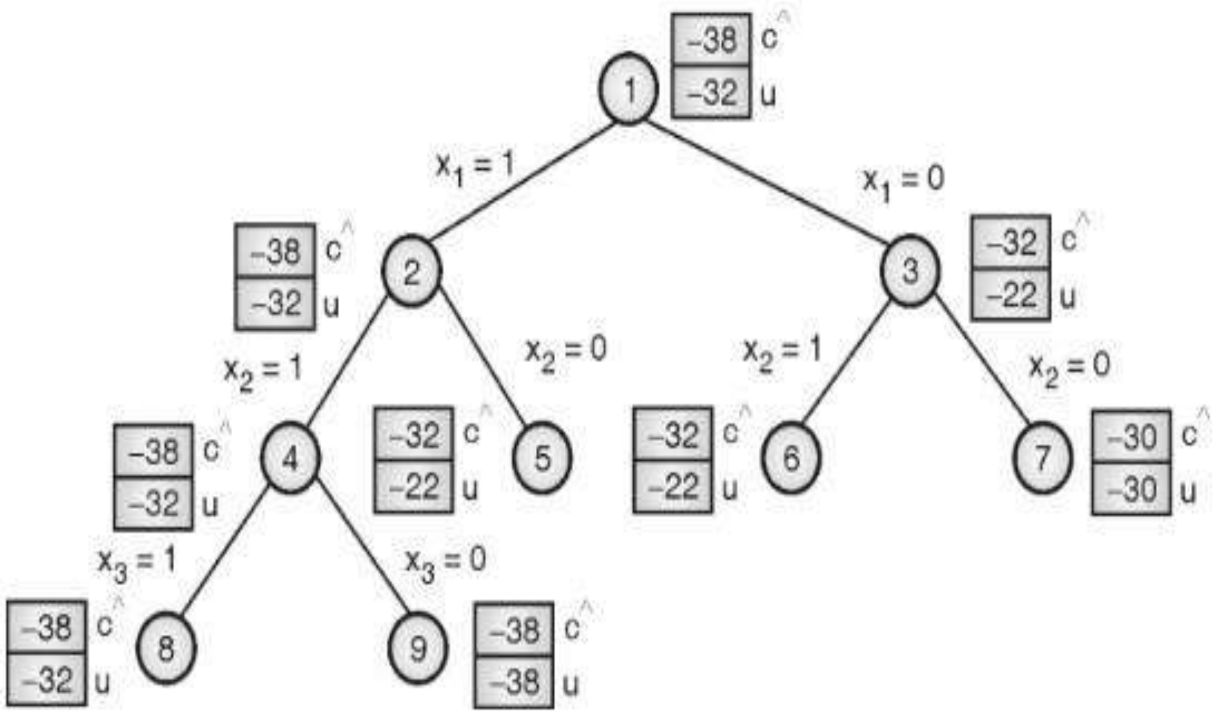
$hatc(6) = -22 - frac15 - (4 + 6)9 * 18$ = -32

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

$hatc(8) = -32 - frac15 - (2 + 4 + 6)9 * 18$ = -38

Node 9 : Exclusion of item 3 at node 4

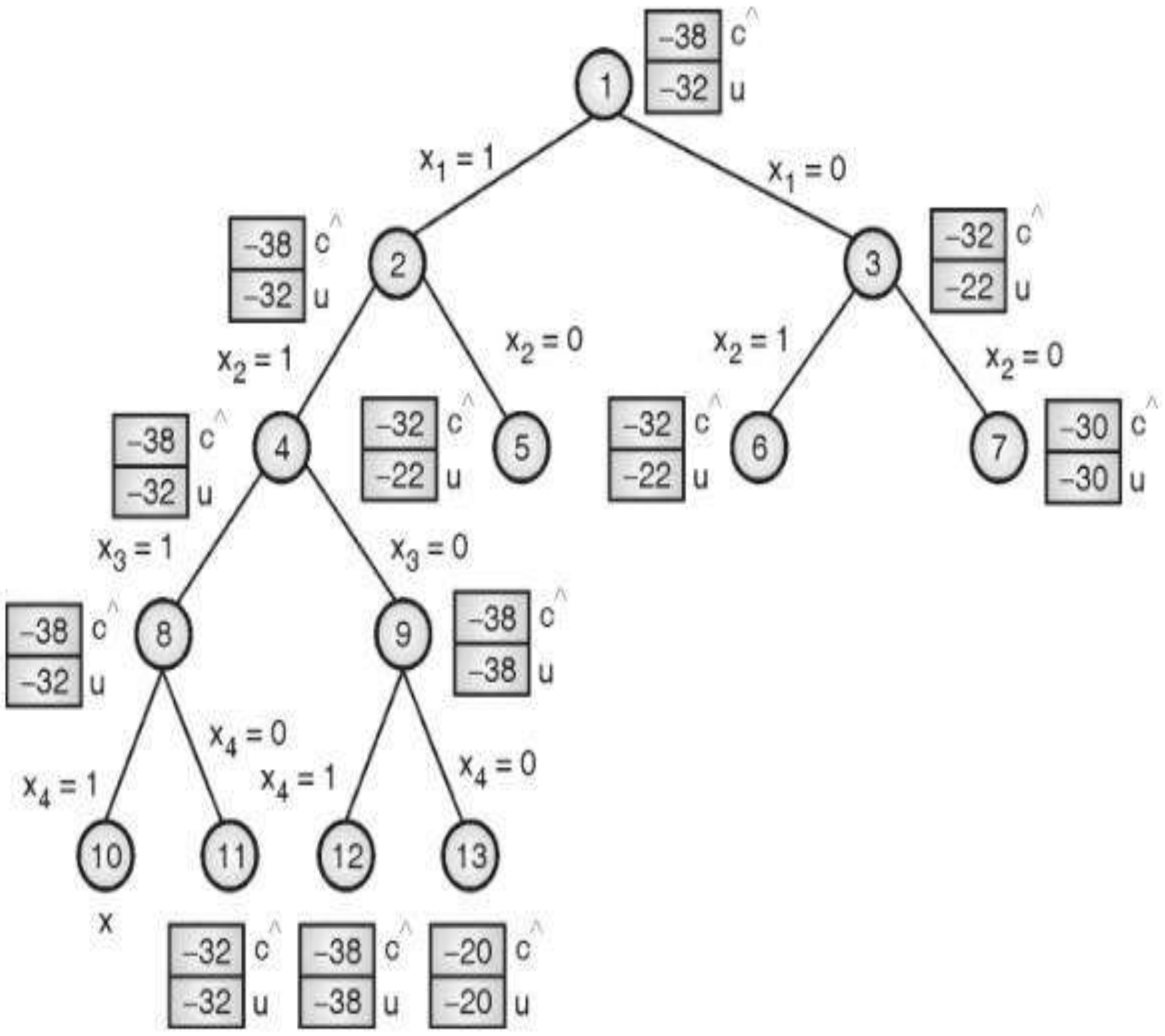
We are excluding item 3, including 1 and 2 and 4.

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

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

Upper = u(9) = − 38.

$hatc(5)$ > upper and $hatc(6)$ > 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.

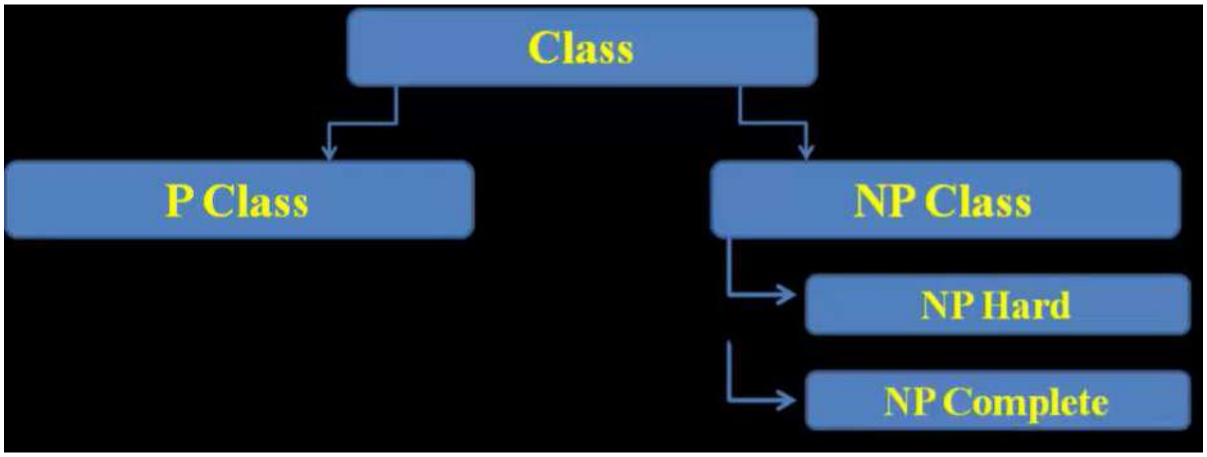
Solution vector = $\{x_1, x_2, x_4\}$.

profit = 10 + 10 + 18 = 38

NP-Hard and NP-Complete problems:

Syllabus: NP-Hard and NP-Complete problems: Basic concepts, non-deterministic algorithms, NP-Hard and NP-Complete classes, Cook's theorem.

Classification of algorithms:
Every algorithm has a time complexity and based on the time complexity of the algorithm we can classify then into different categories as shown below.



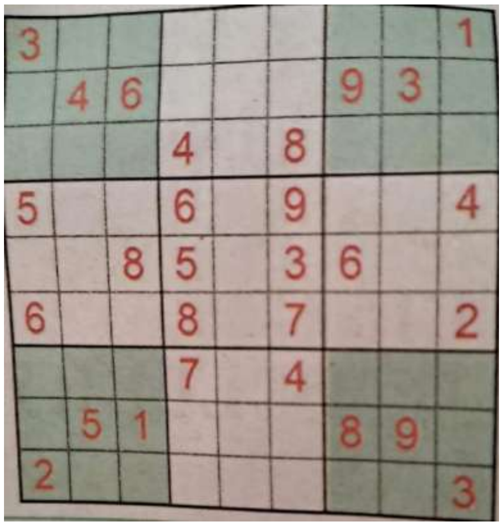
P Class
A problem that can be solved in a polynomial time is called as P Class algorithm.
An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^K)$for some non negative integer K

Insertion sort $O(n^2)$
Merge sort O(nlogn)
Sequential search O(n)

In a P class algorithm, there are a number of statements and all the statements are deterministic (very clear without any confusion).

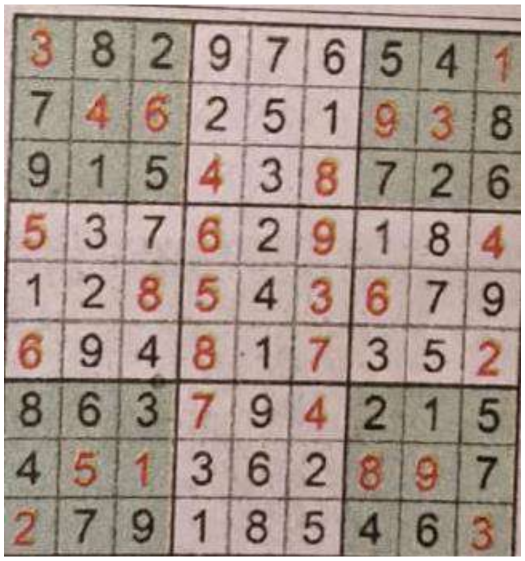
NP class
It is a non-deterministic polynomial time algorithm.
It cannot be solved in polynomial time
However, NP problems are checkable in polynomial time. It means for a given solution of problem; we can check whether the solution is correct or not in polynomial time.

Example - Suduku game: 9 * 9 board which has few numbers given. We need to fill remaining with 1 to 9 numbers such that no number is repeated vertically or horizontally. In addition, each smaller square of 3*3 should contain 1 to 9 numbers.



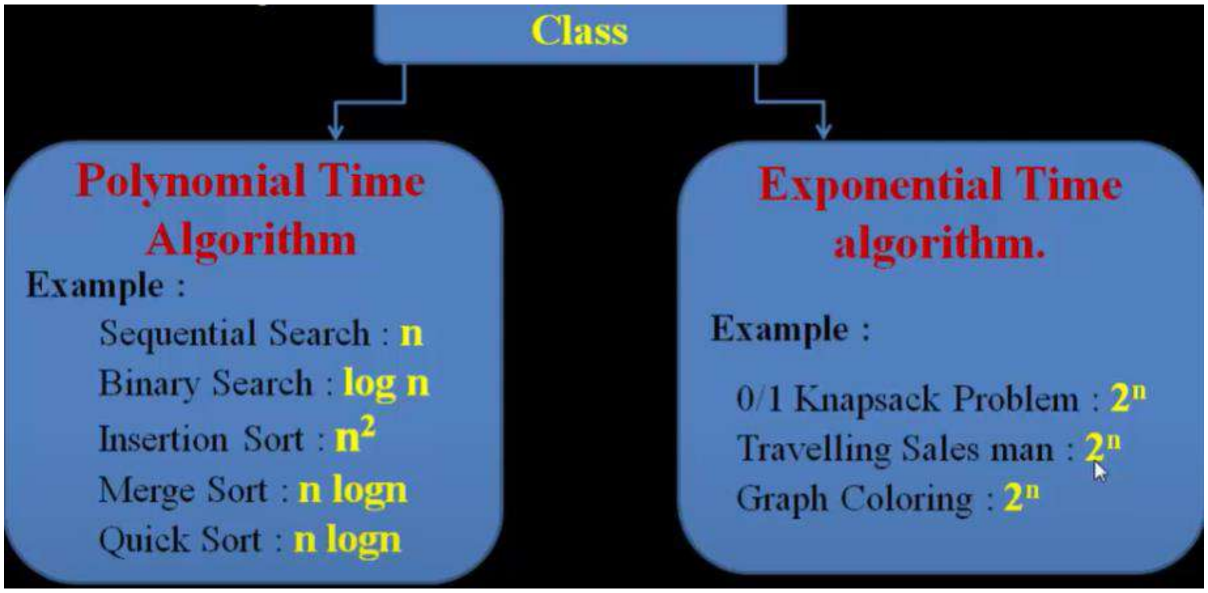
3								1
	4	6				9	3	
			4		8			
5			6		9			4
		8	5		3	6		
6			8		7			2
			7		4			
	5	1				8	9	
2								3

Time complexity to solve the game is exponential.



3	8	2	9	7	6	5	4	1
7	4	6	2	5	1	9	3	8
9	1	5	4	3	8	7	2	6
5	3	7	6	2	9	1	8	4
1	2	8	5	4	3	6	7	9
6	9	4	8	1	7	3	5	2
8	6	3	7	9	4	2	1	5
4	5	1	3	6	2	8	9	7
2	7	9	1	8	5	4	6	3

Once we have the solution as shown above, we can verify the same in polynomial time. However, it can be solved only in exponential time.



Deterministic and non-deterministic algorithms

Deterministic: The algorithm in which every operation is uniquely defined is called deterministic algorithm.

Non-Deterministic: The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithm is called non-deterministic algorithm.

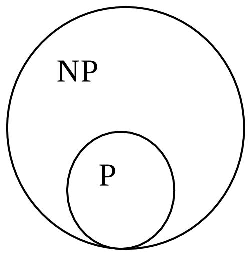
The non-deterministic algorithms use the following functions:

1. Choice: Arbitrarily chooses one of the elements from given set.
2. Failure: Indicates an unsuccessful completion
3. Success: Indicates a successful completion

A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. Whenever, there is a set of choices that leads to a successful completion, then one such set of choices is selected and the algorithm terminates successfully. In case the successful completion is not possible, then the complexity is O(1). In case of successful signal completion then the time required is the minimum number of steps needed to reach a successful completion of O(n) where n is the number of inputs.

The problems that are solved in polynomial time are called tractable problems and the problems that require super polynomial time are called non-tractable problems. All deterministic polynomial time algorithms are tractable and the non-deterministic polynomials are intractable.

A nondeterministic algorithm is considered as a superset for deterministic algorithm i.e., the deterministic algorithms are a special case of nondeterministic algorithms.



Example1: A non-deterministic algorithm for searching an element Key in the given set of elements A[1:n] in the time complexity O(1).

```
Algorithm Search_NP(Key)
{
        int array[100], i;
        i = check_array(Key);

         if(array(i)== Key) then
        {
                Write(i);
                Success();
        }
        else
        {
                Write(0);
                Failure();
        }
}
```

Reducability:
A problem Q1 can be reduced to Q2 if any instance of Q1 can be easily rephrased as an instance of Q2. If the solution to the problem Q2 provides a solution to the problem Q1, then these are said to be reducable problems.

Let L1 and L2 are the two problems. L1 is reduced to L2 if there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time and is denoted by $L1 \alpha\ L2$.

If we have a polynomial time algorithm for L2 then we can solve L1 in polynomial time. Two problems L1 and L2 are said to be polynomially equivalent if $L1 \alpha\ L2$ and $L2\ \alpha\ L1$.

Example: Let P1 be the problem of selection and P2 be the problem of sorting. Let the input have n numbers. If the numbers are sorted in array A[ ] the i^{th} smallest element of the input can be obtained as A[i]. Thus P1 reduces to P2 in O(1) time.

Decision Problem:
Any problem for which the answer is either yes or no is called decision problem. The algorithm for decision problem is called decision algorithm.

Example: Max clique problem, sum of subsets problem.

Optimization Problem: Any problem that involves the identification of an optimal value (maximum or minimum) is called optimization problem.

Example: Knapsack problem, travelling salesperson problem.

In decision problem, the output statement is implicit and no explicit statements are permitted.

The output from a decision problem is uniquely defined by the input parameters and algorithm specification.

Many optimization problems can be reduced by decision problems with the property decision problem can be solved in polynomial time iff the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time, then the optimization problem cannot be solved in polynomial time.

NP HARD AND NP COMPLETE
Polynomial Time algorithms
Problems whose solutions times are bounded by polynomials of small degree are called polynomial time algorithms

Example: Linear search, quick sort, all pairs shortest path etc.

Non- Polynomial time algorithms
Problems whose solutions times are bounded by non-polynomials are called non-polynomial time algorithms

Examples: Travelling salesman problem, 0/1 knapsack problem etc

It is impossible to develop the algorithms whose time complexity is polynomial for non-polynomial time problems, because the computing times of non-polynomial are greater than polynomial. A problem that can be solved in polynomial time in one model can also be solved in polynomial time.

NP-Hard and NP-Complete Problem:
Let P denote the set of all decision problems solvable by deterministic algorithm in polynomial time. NP denotes set of decision problems solvable by nondeterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of nondeterministic algorithms, $P \subseteq NP$. The nondeterministic polynomial time problems can be classified into two classes. They are

1. NP Hard and
2. NP Complete

NP-Hard: A problem L is NP-Hard if satisfiability reduces to L i.e., any nondeterministic polynomial time problem is satisfiable and reducable then the problem is said to be NP-Hard.

<u>Example</u>: Halting Problem, Flow shop scheduling problem

NP-Complete: A problem L is NP-Complete if L is NP-Hard and L belongs to NP (nondeterministic polynomial).

A problem that is NP-Complete has the property that it can be solved in polynomial time if all other NP-Complete problems can also be solved in polynomial time. (NP=P)

If an NP-hard problem can be solved in polynomial time, then all NP- complete problems can be solved in polynomial time. All NP-Complete problems are NP-hard, but some NP-hard problems are not known to be NP- Complete.

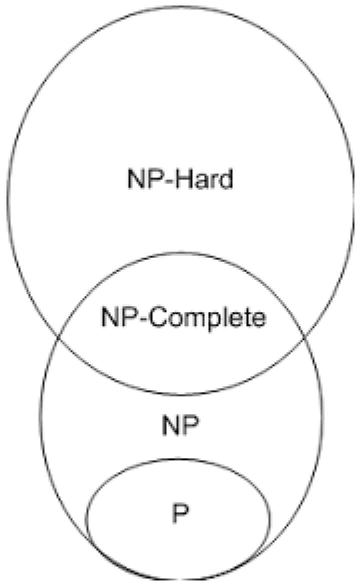
Normally the decision problems are NP-complete but the optimization problems are NP-Hard.

However, if problem L1 is a decision problem and L2 is an optimization problem, then it is possible that L1 α L2.

<u>Example</u>: Knapsack decision problem can be reduced to knapsack optimization problem. There are some NP-hard problems that are not NP-Complete.

Relationship between P, NP, NP-hard, NP-Complete

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-complete respectively. Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:



Problem conversion

A decision problem D1 can be converted into a decision problem D2 if there is an algorithm which takes as input an arbitrary instance I1 of D1 and delivers as output an instance I2 of D2 such that I2 is a positive instance of D2 if and only if I1 is a positive instance of D1. If D1 can be converted into D2, and we have an algorithm which solves D2, then we thereby have an algorithm which solves D1. To solve an instance I1 of D1, we first use the conversion algorithm to generate an instance I1 of D2, and then use the algorithm for solving D2 to determine whether or not I2 is a positive instance of D2. If it is, then we know that I is a positive instance of D1, and if it is not, then we know that I is a negative instance of D1. Either way, we have solved D1 for that instance. Moreover, in this case, we can say that the computational complexity of D1 is at most the sum of the computational complexities of D2 and the conversion algorithm. If the conversion algorithm has polynomial complexity, we say that D1 is at most polynomially harder than D2. It means that the amount of computational work we have to do to solve D1, over and above whatever is required to solve D2, is polynomial in the size of the problem instance. In such a case the conversion algorithm provides us with a feasible way of solving D1, given that we know how to solve D2.

Cook's theorem

Cook's Theorem shows that the Boolean Satisfiability Problem (SAT) is NP-complete. To formally connect this to "P = NP," here is the logical reasoning:

1. **Key Insight from Cook's Theorem**:
   - Cook's Theorem states that SAT is NP-complete. That means SAT is one of the hardest problems in NP.
   - If SAT can be solved in polynomial time (SAT $\in$ P), then every NP problem can also be solved in polynomial time, making P = NP.
2. **Proof Breakdown**:
   - Suppose SAT$\in$P:
     - By the definition of NP-completeness, every problem in NP can be reduced to SAT in polynomial time.
     - If SAT can be solved in polynomial time, the solution to every NP problem can be derived in polynomial time (via reduction to SAT + solving SAT).
     - Therefore, P=NP.
   - Conversely, suppose P=NP:
     - By definition, every NP-complete problem, including SAT, would belong to P, as all problems in NP would have polynomial-time solutions.
     - Thus, SAT$\in$P.
3. **Conclusion**:
   - SAT is in P if and only if P = NP.

This is why Cook's Theorem is often tied to the P vs. NP question—it identifies SAT as the "representative" problem of NP-completeness. Solving SAT efficiently would imply efficient solutions for all problems in NP.