Branch and Bound technique

Branch and Bound Technique

- Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- The algorithm divides a big optimization problem into smaller and simpler subsets and scans for the best possible solution among the candidate solutions.
- The algorithm searches the tree entirely before arriving at the best solution. This process is considered to be a time-consuming process, especially for big or complex problems.
- This method is best when used for combinatorial problems with exponential time complexity, since it provides a more efficient solution to such problems.
- 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.

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 ith position denoting whether the ith element should be included or not. Hence, for the earlier example, the solution will be given by {1, 1, 0, 0, 1}.

Types of methods

- FIFO Branch and Bound
- LIFO Branch and Bound
- Least Cost-Branch and Bound

Advantage

- Time Complexity: The BnB algorithm does not explore all nodes in the tree. Thereby, the time complexity is significantly lesser than most other algorithms.

- Optimal Solution: If the branching is done reasonably, the algorithm can find the optimal solution in a reasonable period.

- Clear Pattern: The BnB algorithm does not repeat the notes to explore the tree for the candidate solutions; instead, it follows a minimal path to derive the optimal solution.

Disadvantage

- Time-consuming: Based on the size of the problem, the number of nodes that are computed might be too large in a worst-case scenario, making it a time-consuming process.

- Parallelization: The branching out of possible solutions provides scope for speculative parallelism. However, when alternative actions are considered for the said action, the branch and bound calculator finds difficulty.

Basis	Branch and Bound	Backtracking
Basic Function	BnB is used to solve optimization issues.	Backtracking is used to find all solutions possible to a given problem
Approach	It can travel the tree in either DFS or Breadth First Search (BFS)	Based on the Depth First Search (DFS) approach
Additional Function	Involves bounding function	Involves feasibility function
Solution	The algorithm knows that it has a better optimal solution than the pre-solution. Hence, it abandons the pre-solution.	The system recognizes that it has made a wrong choice and undoes the last choice.
Search Spectrum	It searches the tree entirely before delivering the optimal solution	It explores the tree until the answer is found.

Dynamic Programing	Branch and Bound
Constructs the solution in form of a table.	Constructs the solution in form of a tree.
Solves all possible instances of problem of size n.	Only solves promising instances from the set of instances at any given point.
Does not require a bounding function.	Needs to compute and apply a bounding function at each node.
After constructing the table, it needs to be traced back to find the solution sequence.	The solution sequence is implicit, a leaf node of the tree is the final solution.

0/1 Knapsack Problem

Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively.
Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.

We have 'n' items with value v_1 , v_2 . . . v_n and weight of the corresponding items is w_1 , w_2 . . . W_n .
Max capacity is W .

We can either choose or not choose an item. We have x_1 , x_2 . . . x_n.
Here x_i = { 1 , 0 }.

x_i = 1 , item chosen
x_i = 0 , item not chosen

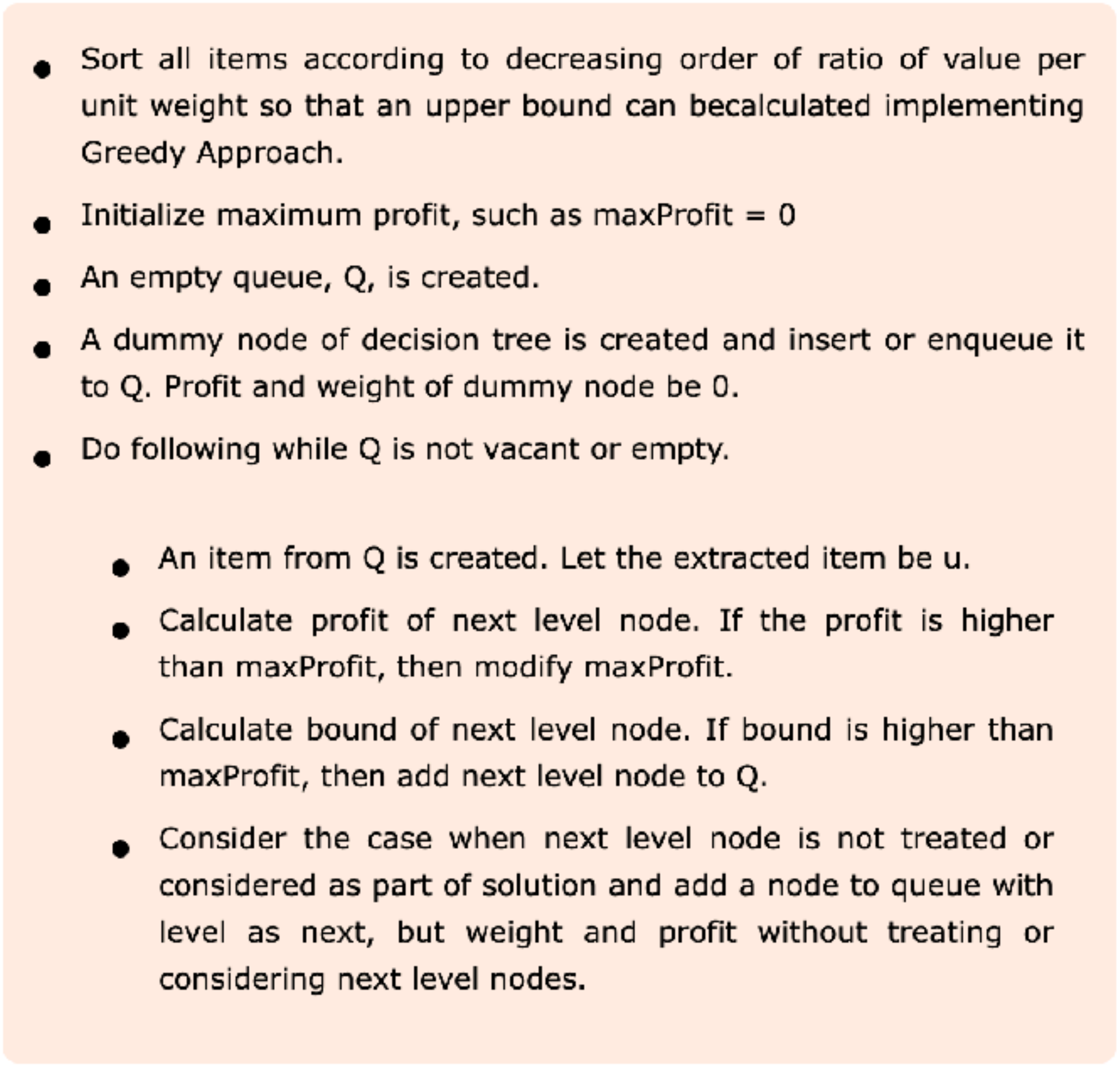
Why branch and bound ?

- **Greedy approach** works only for fractional knapsack problem.
- If weights are not integers , **dynamic programming** will not work.
- There are 2^n possible combinations of item , complexity for **brute force** goes exponentially.

➢ **These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.**

➢ **Branch and Bound solve these problems relatively quickly.**

Algorithm

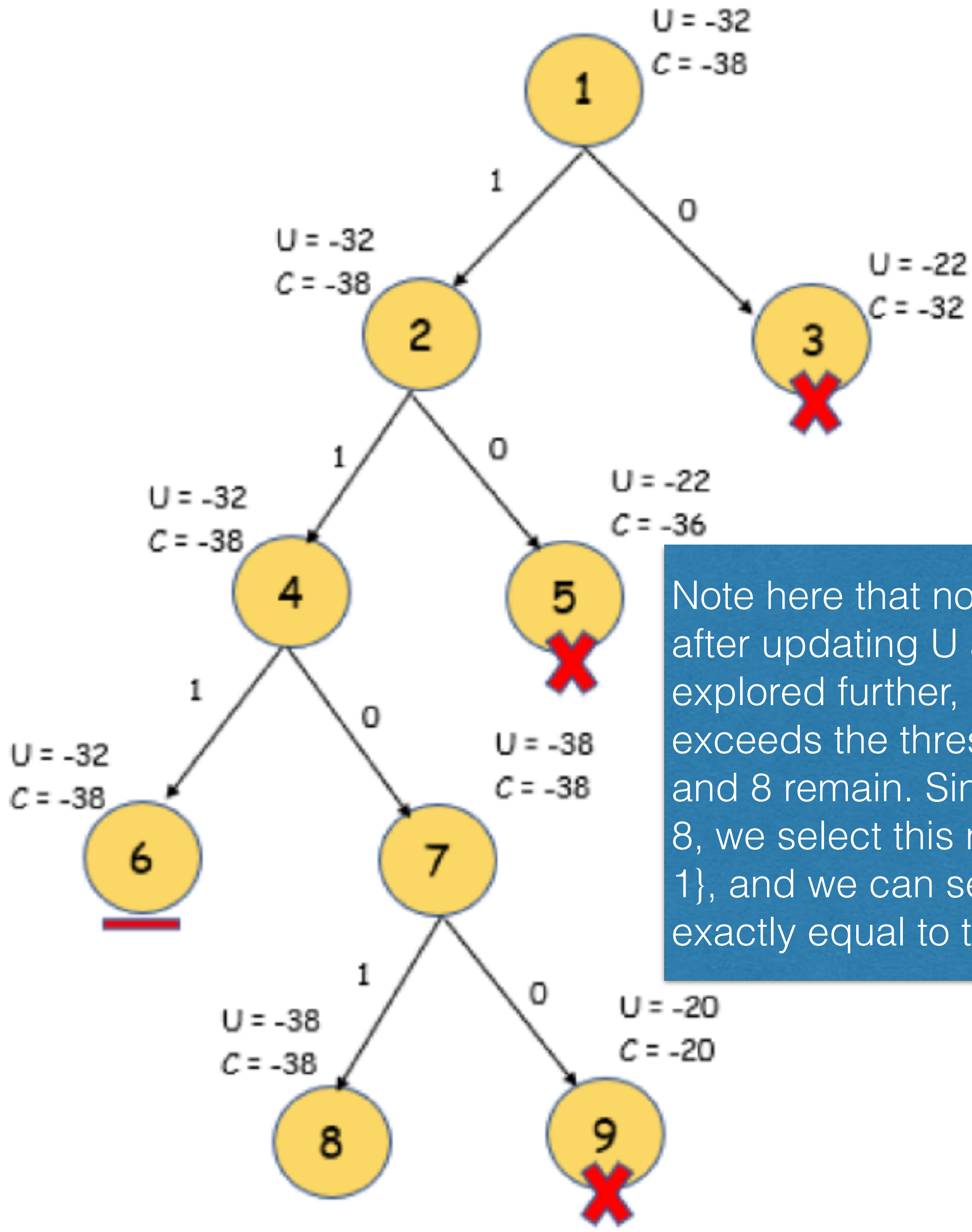


- Sort all items according to decreasing order of ratio of value per unit weight so that an upper bound can becalculated implementing Greedy Approach.
- Initialize maximum profit, such as maxProfit = 0
- An empty queue, Q, is created.
- A dummy node of decision tree is created and insert or enqueue it to Q. Profit and weight of dummy node be 0.
- Do following while Q is not vacant or empty.
    - An item from Q is created. Let the extracted item be u.
    - Calculate profit of next level node. If the profit is higher than maxProfit, then modify maxProfit.
    - Calculate bound of next level node. If bound is higher than maxProfit, then add next level node to Q.
    - Consider the case when next level node is not treated or considered as part of solution and add a node to queue with level as next, but weight and profit without treating or considering next level nodes.

Solving an Example

- Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initital upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

- After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):



Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

Time and Space Complexity

- Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by O(2^n), in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to explored, and hence its best case time complexity is given by O(n). Since this method requires the creation of the state space tree, itsspace complexity will also be exponential.

Travelling salesman Problem

- 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.
- The given statements are-
    - A set of some cities
    - Distance between every pair of cities
- Travelling Salesman Problem states-
    - A salesman has to visit every city exactly once.
    - He has to come back to the city from where he starts his journey.
    - What is the shortest possible route that the salesman must follow to complete his tour?

Algorithm

Travelling salesman problem using branch and bound (penalty) method Steps (Rule)	
Step-1:	Find out the each row minimum element and subtract it from that row. Also add each row minimum element is called row minimum.
Step-2:	Find out the each column minimum element and subtract it from that column. Also add each column minimum element is called column minimum.
Step-3:	lower bound = row minimum + column minimum
Step-4:	Calculate the penalty of all 0's penalty (of each 0) = minimum element of that row + minimum element of that column.
Step-5:	Find maximum penalty from all this penalties. And the new branch will be start from this location. If there are more than 1 such location then choose any one arbitrarily.
Step-6:	Let branch will occur at $X_{A,D}$. There are two branches. 1. If $X_{A,D} = 0$, then we have an additional cost of say t and the lower bound becomes LB + t 2. If $X_{A,D} = 1$, then we can go $A \to D$ So we can't go $D \to A$, so set it to M. Now we leave row A and column D. Again repeat the steps from step-1 using this reduced matrix, until whole path is found.
Step-7:	So finally we get total distance and final path.

Time Complexity

- Now this thing is tricky and need a deeper understanding of what we are doing. We are actually creating all the possible extenstions of E-nodes in terms of tree nodes. Which is nothing but a permutation. Suppose we have N cities, then we need to generate all the permutations of the (N-1) cities, excluding the root city. Hence the time complexity for generating the permutation is O((n-1)!), which is equal to O(2^(n-1)).

- Hence the final time complexity of the algorithm can be O(n^2 * 2^n).