|  |  |
| --- | --- |
| **AssignmentNo.** | **4** |
| **Title** | To Implement 0-1 Knapsack problem using dynamic programming or branch and bound strategy. |
| **PROBLEM STATEMENT/DEFINITION** | Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy. |
| **Objectives** | **Understand and implement** 0-1 Knapsack problem using dynamic programming or branch and bound strategy. |
| **Software packages and hardware apparatus used** | Technology : Java/Python  Ubuntu /Linux  Latest Version of 64 bit Operating Systems, Open Source Fedora-GHz. 8 G.B.  RAM, 500 G.B. HDD, 15"Color Monitor, Keyboard, Mouse |
| **References** | 1. <https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/> |
|  |  |

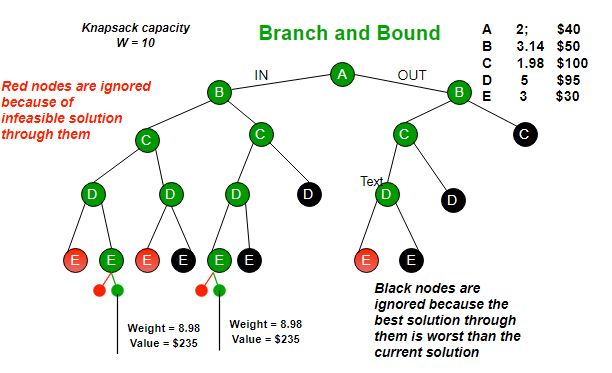
**Concepts Related Theory**

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. 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.

Let us consider below 0/1 Knapsack problem to understand Branch and Bound. 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. Let us explore all approaches for this problem.

1.A Greedy approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.

1. We can use Dynamic Programming (DP) for 0/1 Knapsack problem. In DP, we use a 2D table of size n x W. The DP Solution doesn’t work if item weights are not integers.
2. Since DP solution doesn’t always work, a solution is to use Brute Force. With n items, there are 2n solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that satisfies constraint. This solution can be expressed as tree**.**
3. We can use Backtracking to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.
4. Branch and BoundThe backtracking based solution works better than brute force by ignoring infeasible solutions. We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node. Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30
5. 

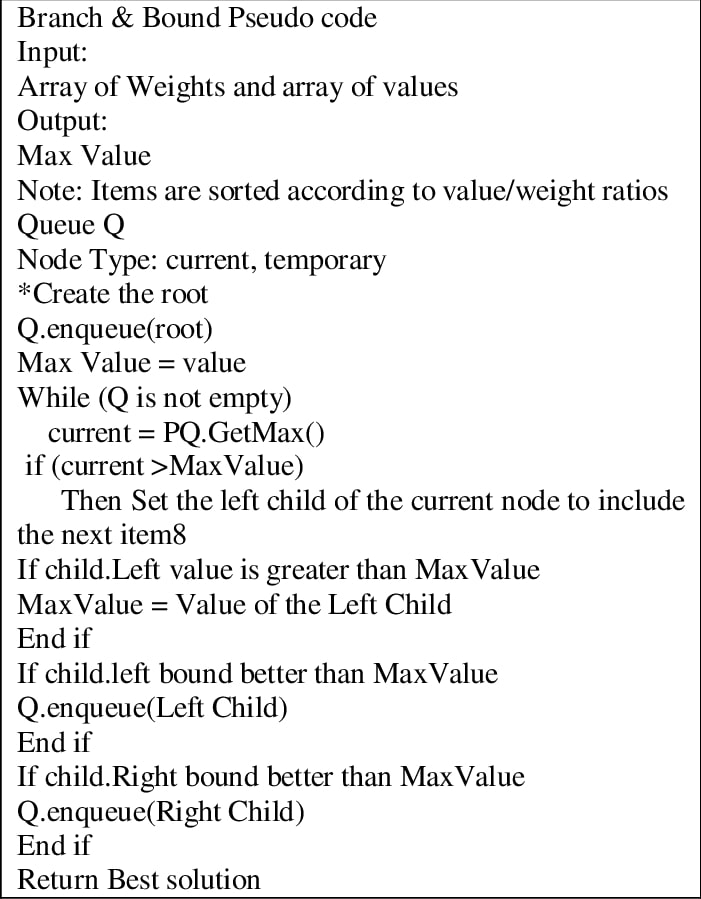
**Steps:**

* Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
* If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than *U*, then replace the value of *U* with this value. Then, kill all unexplored nodes which have cost function greater than this value.
* The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
* While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

**Algorithm:**

* Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
* Initialize maximum profit, maxProfit = 0
* Create an empty queue, Q.
* Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
* Do following while Q is not empty:

1. Extract an item from Q. Let the extracted item be u.
2. Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
3. Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
4. Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.



## Conclusion:

Even though this method is more efficient than the other solutions to this problem, its worst case **time complexity is still given by O(2n)**, 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, its space complexity will also be **exponential.**