Implementation of 0/1 Knapsack using Branch and Bound

We strongly recommend to refer below post as a prerequisite for this.

Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)

We discussed different approaches to solve above problem and saw that the Branch and Bound solution is the best suited method when item weights are not integers.

In this post implementation of Branch and Bound method for 0/1 knapsack problem is discussed.

How to find bound for every node for 0/1 Knapsack?
The idea is to use the fact that the Greedy approach provides the best solution for Fractional Knapsack problem.
To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can't get a better solution through the node.

Complete Algorithm:

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.
    - Extract an item from Q. Let the extracted item be u.

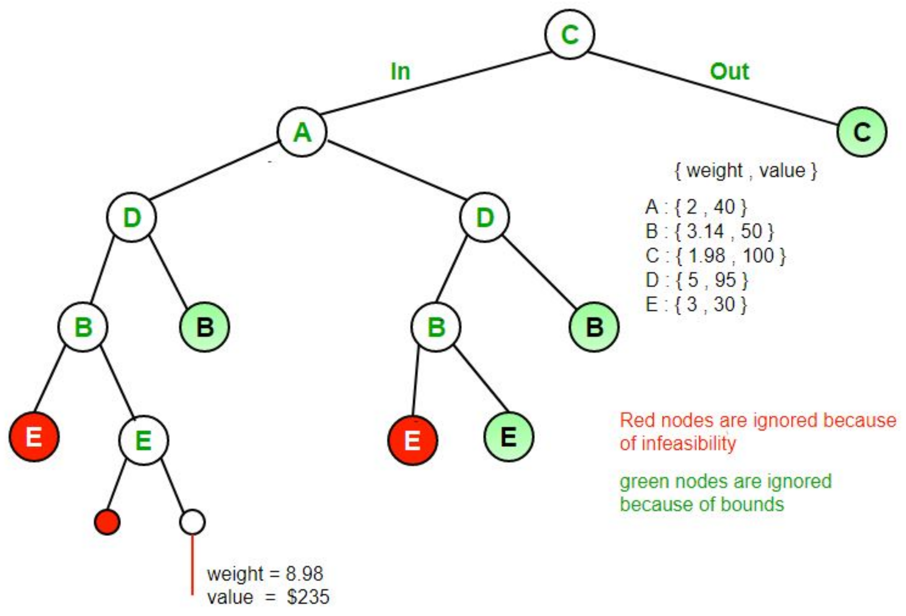
- Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

Illustration:

```
Input:
// First thing in every pair is weight of item
// and second thing is value of item
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
              {5, 95}, {3, 30}};
Knapsack Capacity W = 10

Output:
The maximum possible profit = 235

Below diagram shows illustration. Items are
considered sorted by value/weight.
```



```
Note :  The image doesn't strictly follow the
algorithm/code as there is no dummy node in the
image.
```

Following is C++ implementation of above idea.

```
// C++ program to solve knapsack problem using
// branch and bound
#include <bits/stdc++.h>
using namespace std;

// Stucture for Item which store weight and corresponding
// value of Item
struct Item
{
    float weight;
    int value;
};

// Node structure to store information of decision
// tree
struct Node
{
    // level  --> Level of node in decision tree (or index
    //             in arr[]
    // profit --> Profit of nodes on path from root to this
    //            node (including this node)
    // bound ---> Upper bound of maximum profit in subtree
    //            of this node/
    int level, profit, bound;
    float weight;
};

// Comparison function to sort Item according to
// val/weight ratio
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Returns bound of profit in subtree rooted with u.
// This function mainly uses Greedy solution to find
// an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;

    // initialize bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
```

```
        int totweight = u.weight;

        // checking index condition and knapsack capacity
        // condition
        while ((j < n) && (totweight + arr[j].weight <= W))
        {
            totweight    += arr[j].weight;
            profit_bound += arr[j].value;
            j++;
        }

        // If k is not n, include last item partially for
        // upper bound on profit
        if (j < n)
            profit_bound += (W - totweight) * arr[j].value /
                                             arr[j].weight;

        return profit_bound;
    }

    // Returns maximum profit we can get with capacity W
    int knapsack(int W, Item arr[], int n)
    {
        // sorting Item on basis of value per unit
        // weight.
        sort(arr, arr + n, cmp);

        // make a queue for traversing the node
        queue<Node> Q;
        Node u, v;

        // dummy node at starting
        u.level = -1;
        u.profit = u.weight = 0;
        Q.push(u);

        // One by one extract an item from decision tree
        // compute profit of all children of extracted item
        // and keep saving maxProfit
        int maxProfit = 0;
        while (!Q.empty())
        {
            // Dequeue a node
            u = Q.front();
            Q.pop();

            // If it is starting node, assign level 0
            if (u.level == -1)
                v.level = 0;

            // If there is nothing on next level
            if (u.level == n-1)
                continue;

            // Else if not last node, then increment level,
            // and compute profit of children nodes.
            v.level = u.level + 1;

            // Taking current level's item add current
            // level's weight and value to node u's
            // weight and value
            v.weight = u.weight + arr[v.level].weight;
            v.profit = u.profit + arr[v.level].value;

            // If cumulated weight is less than W and
            // profit is greater than previous profit,
            // update maxprofit
            if (v.weight <= W && v.profit > maxProfit)
                maxProfit = v.profit;

            // Get the upper bound on profit to decide
            // whether to add v to Q or not.
            v.bound = bound(v, n, W, arr);

            // If bound value is greater than profit,
            // then only push into queue for further
            // consideration
            if (v.bound > maxProfit)
                Q.push(v);

            // Do the same thing,  but Without taking
            // the item in knapsack
            v.weight = u.weight;
            v.profit = u.profit;
            v.bound = bound(v, n, W, arr);
            if (v.bound > maxProfit)
                Q.push(v);
        }

        return maxProfit;
    }

    // driver program to test above function
    int main()
    {
        int W = 10;   // Weight of knapsack
        Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                      {5, 95}, {3, 30}};
        int n = sizeof(arr) / sizeof(arr[0]);

        cout << "Maximum possible profit = "
             << knapsack(W, arr, n);

        return 0;
    }
```

Output :

```
Maximum possible profit = 235
```

This article is contributed Utkarsh Trivedi. If you likeGeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Recommended Posts:

0/1 Knapsack using Branch and Bound
Job Assignment Problem using Branch And Bound
N Queen Problem using Branch And Bound
8 puzzle Problem using Branch And Bound
Traveling Salesman Problem using Branch And Bound
0-1 Knapsack Problem | DP-10
Printing Items in 0/1 Knapsack
Fractional Knapsack Problem
A Space Optimized DP solution for 0-1 Knapsack Problem
Unbounded Knapsack (Repetition of items allowed)
Check if all people can vote on two machines

Article Tags : Branch and Bound knapsack

2

3.7

☐ To-do ☐ Done

Based on **15** vote(s)

Feedback Add Notes Improve Article