Solution to 0-1 Knapsack problem : A Branch and Bound Approach

Harsh Aryan* ,Anup Bedia†
Indian Institute of Information Technology,Allahabad
Allahabad-211012

Email: *itm2017003@iiita.ac.in, †ism2017002@iiita.ac.in

Abstract*—This Paper introduces an algorithm to solve the 0-1 Knapsack problem. The idea is to find out an efficient algorithm such that the time complexity, as well as space complexity, is minimized. This problem can easily be expressed and solved in a plethora of ways. The task here is to use the branch and bound technique to look for the optimal solution. The method proposed in this paper gives a solution which is approximately optimal and also an upper bound on the value of optimal solution.

***Keywords* — 0-1 Knapsack problem , branch and bound, Knapsack, Time Complexity, Backtracking .**

I. Introduction

This paper presents a branch and bound algorithm for the solution of 0-1 Knapsack problem, which in itself is a special type of combinatorial optimisation problem.

Branch and bound (BnB or BB or B&B) is an algorithm design paradigm used to solve combinatorial optimization problems. Combinatorial optimization problems are usually of exponential time complexity and many a times require exploring all the possible combinations in the worst case which results to the exponential time complexity.
The approach used here (Branch and Bound) typically solves these problems in much lesser time, mainly due to the fact that not all the branches need to be explored.

The problem popularly known as knapsack problem (or rucksack problem) arises in various cargo loading situations and is usually of the form “given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible”. 0-1 Knapsack problem is a variation to the above problem where each object can be taken only 0 or 1 times. Another variation to the problem is fractional Knapsack problem where fractional parts of the objects can also be taken, but our problem does not allow that.
Branch and Bound approach makes sure of the fact that computation may be stopped for a branch which will not lead us to an optimum solution. Further, we can deduce that the efficiency of the algorithm used increases as we increase the number of objects taken. But as is the case with different branch and bound algorithms, the memory and time complexities are huge when we involve many items.

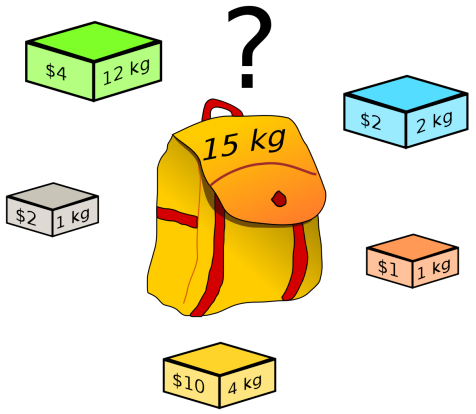

Fig 1.1: figure illustrating 0-1 Knapsack problem

Here we will try to develop an efficient algorithm to solve the problem using the branch and bound.

This algorithm is tested multiple times for time complexity, space complexity and accuracy. In our case, we tend to focus on establishing a rule or a relationship between the time and input data.

This report further contains -

II. Algorithm Design and Analysis

III. Illustration

IV. Alternate Approaches

V. Result

VI. Conclusion

VII. References

VIII. Appendix

II. ALGORITHM DESIGN AND ANALYSIS

An efficient algorithm can only be created by considering every possibility. So we have to consider every possibility while designing the algorithm.

Before designing the algorithm, the following points should be kept in mind.

1. Fractional object is not to be considered.
2. Only one of each item can be taken at max, that is, no multiple entries are allowed.

Steps for designing the algorithm include:

1. Sort items in their efficiency in decreasing order, that is, on the basis of their value to weight ratio
2. declare Profitmax and initialise it to zero
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and push it to Q. Set its profit and weight as zero. This dummy node just acts as the root.
5. While Q is not empty, Pop an item from Q. Let the popped item be u. Also,
6. Find profit of next level node, if its greater, update value of Profitmax
7. Calculate bound of next level, if it comes out to be greater, add the next level nodes to queue
8. For the case when it is not a part of solution, add a node to queue with level as next, but weight and profit without looking into next level nodes.

Variables

```
def Node: struct
    level:int, profit: int, bd:int
    wt:float

def Object:float
    wt:float
    val:int
```

Function to compare efficiency of 2 objects

```
def compar (a:Object, b:Object)
    rr:double, rs:double
    rr ← (double)b.val / b.wt
    rs ← (double)a.val / a.wt
    return rs > rr
```

Time Analysis: 13 units
Worst Case: O(1)
Best Case: Ω(1)
Average Case: Θ(1)

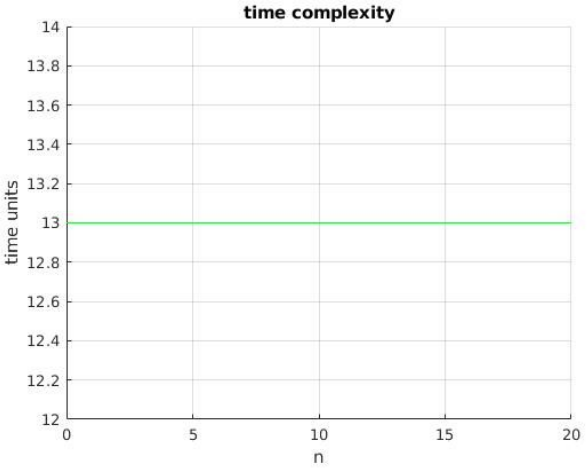

Fig 2.1: Graph depicting constant time complexity

Function to Bound

```
def bd(u:Node , n:int , W:int ,arr[] : Object)
    if (u.wt >= W)
        return 0
    profit_bd : int
    profit_bd ← u.profit
    j: int
    j ← u.level + 1
    totwt : int
    totwt ← u.wt
    while ((j < n) && (totwt + arr[j].wt <= W)) {
        totwt ← totwt + arr[j].wt
        profit_bd ← profit_bd + arr[j].val
        j←j+1
    if (j < n)
        profit_bd ← profit_bd + (W - totwt)

       * arr[j].val / arr[j].wt

    return profit_bd
```

Time Analysis:
Worst Case: O(7+20*n) = O(n)
Best Case: Ω(1)
Average Case: Θ(n)

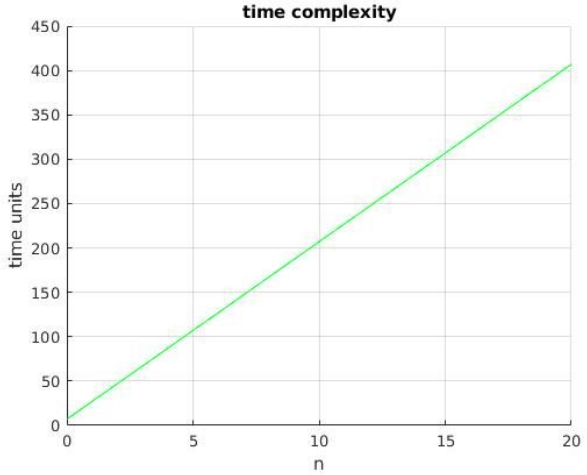

Fig 2.2: Graph depicting linear worst case time complexity

Knapsack Function

```
def knapsack(W:int, arr[]:Object[], n:int)
    sort(arr, arr + n, compar)
    Q : queue<Node>
    u :Node , v:Node
    u.level ← -1
    u.profit ← 0
    u.wt ← 0
    Q.push(u)
    Profitmax:int
    Profitmax ← 0

    while (!Q.empty())
        u = Q.front()
        Q.pop()
        if (u.level = -1)
            v.level ← 0
        if (u.level = n-1)
            continue
        v.level ← u.level + 1
        v.wt ← u.wt + arr[v.level].wt
        v.profit ← u.profit + arr[v.level].val

        if (v.wt <= W && v.profit > Profitmax)
            Profitmax ← v.profit


        v.bd ← bd(v, n, W, arr)

        if (v.bd > Profitmax)
            Q.push(v)

        v.wt ← u.wt
        v.profit ← u.profit

        v.bd ← bd(v, n, W, arr);
        if (v.bd > Profitmax)
            Q.push(v)

    return Profitmax
```

Time Analysis:
Worst Case:
In worst case we have to traverse the entire tree and no branch is pruned, so the time taken is proportional to number of leaf nodes, that is 2^n.
Hence, O(2^n)
Best Case:
Ω(n)

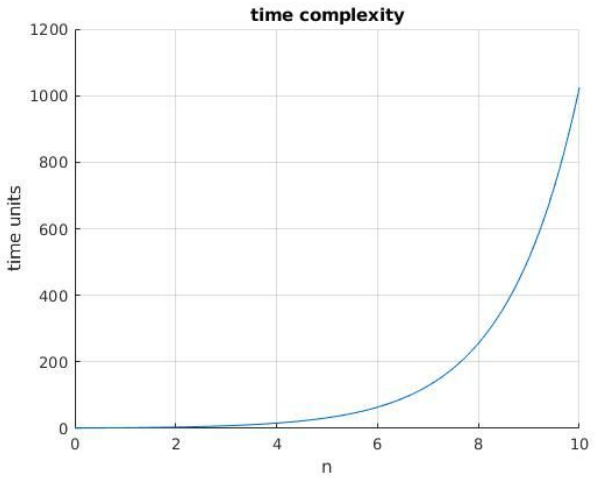

Fig 2.3: Graph depicting exponential worst case time complexity

III. ILLUSTRATION

Let us consider a case with 3 items, with respective values 1,2 and 5 and weights 2,3 and 4. The maximum weight allowed for Knapsack is 6.

So, as in our algo, we first make a dummy node with both value and weight as zero. Next we consider two branches one including abject 1, one excluding it. We do the same with other nodes, till we obtain the tree.

The maximum value is with the branch containing Item 1 and 3 and omitting object 2. Hence this is the solution. We also got two nodes where total weight exceeded the capacity of knapsack. As we can see, the entire tree is formed hence complexity reaches O(2^n), where n is 3.

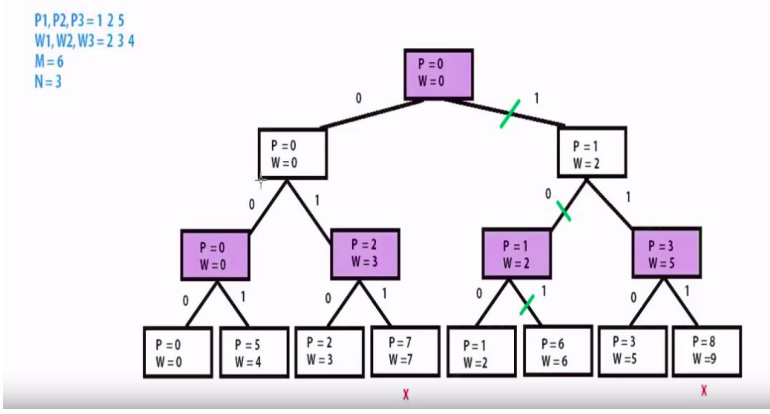

Fig 3.1: Figure illustrating an example of 0-1 Knapsack problem

IV. ALTERNATE APPROACHES AND DISCUSSSION

Greedy Approach :

What a Greedy approach does is select items in order of value per unit weight (decreasing order) . The Greedy approach can only be used properly for fractional Knapsack problem, hence can't be used here.

Dynamic Programming (DP):

DP can be used for 0/1 Knapsack problem. In DP, we use a 2D table of size n x W. The downside to this approach is the fact that DP would stop working for fractional, or say, non-integral weights. Since this can't be used for above case, we need to find other algo.

Brute Force:

Brute force checks the value at each node and finds maximum.

So for n item, the total number of probable solutions can be 2n. That is 2n variations are to be generated. These can be represented as a tree.

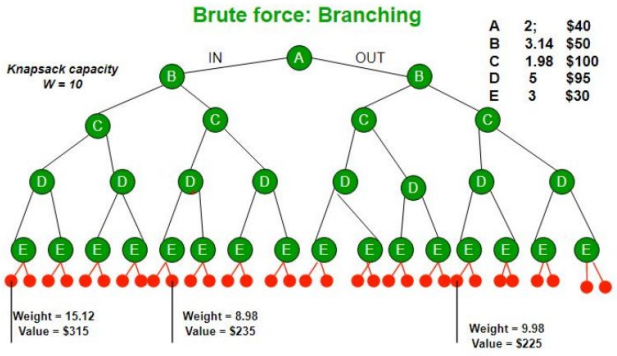

Fig 4.1: figure illustrating Brute force approach for 0-1 knapsack problem

Backtracking:

Backtracking can be used to optimize the brute force solution. We simply need to stop finding those nodes where the answer isn't feasible anymore and backtrack. This would save time as compared to the Brute Force approach.

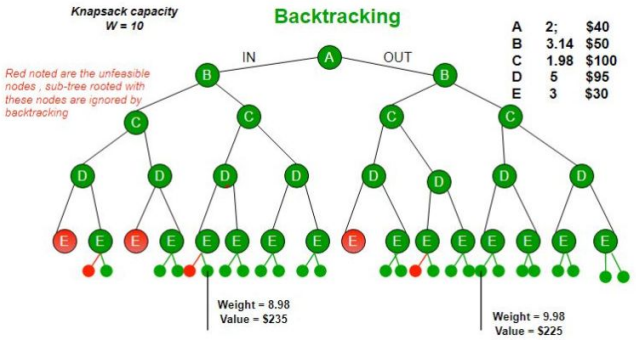

Fig 4.2: figure illustrating Backtracking approach for 0-1 knapsack problem

We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. Branch and bound is very useful technique for searching a solution but in worst case, we need to fully calculate the entire tree. At best, we only need to fully calculate one path through the tree and prune the rest of it.

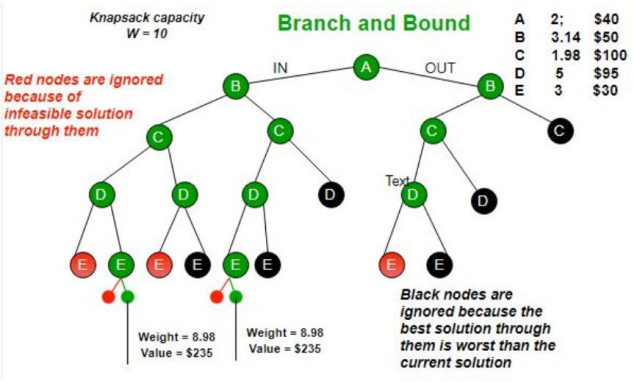

Fig 4.3: figure illustrating Branch and bound approach for 0-1 knapsack problem

V. RESULT

Overall Time Complexity

Time Complexity of the algorithm is not equal to the actual time required to execute particular code but the number of times a statement executes.

We have here used the Branch and Bound technique to solve this 0-1 Knapsack problem.

In the worst case, the Branch and Bound algorithm will generate all intermediate stages and all leaves. Hence, the tree will be complete and then the Time complexity will be proportional to the number of leaf nodes, that is O (2^n).

In the average case, the algorithm will prune many branches and hence the effort of going to each node will be saved and the time complexity will be much less than O (2^n). Branch and Bound is an optimal and effective method to find the solution of 0-1 Knapsack Problem.

VI. CONCLUSION

So here the algorithm used to solve the 0-1 Knapsack Problem has been developed by using the method of the branch and bound. It can be thought of as an efficient algorithm to solve the problem as the other approaches i.e. use of backtracking or Greedy Algorithm takes more time to solve the same problem and Greedy Approach does not work properly for situations where fractional objects are not allowed.

At first sight, backtracking may seem convenient and easy but on studying deeper it is found that backtracking causes troubles and takes a lot of time as compared to the branch and bound. Branch and bound eliminates or bounds the branches which can't give optimal solution at each level and hence reduces the number of calculations and gives an optimal solution in much lesser time.

Thus, the use of the branch and bound approach to solve the problem is the best possible solution to the given problem.
We can conclude that the branch and bound and dynamic programming algorithms outperform the greedy and backtracking algorithm in term of the total value it generated.

VII. REFERENCES

[1] Wikipedia : "Knapsack Problem" [Online], Available at : https://en.wikipedia.org/wiki/Knapsack_problem [Accessed at 21 April 2019]

[2] Columbia university article : "A branch and bound algorithm for the knapsack problem", Peter J. Kolesar , [Online] , Available at : https://www0.gsb.columbia.edu/mygsb/faculty/research/pubfiles/4407/kolesar_branch_bound.pdf [Accessed On : 21 April,2019]

[3] Thomas H. Corman, Introduction to Algorithms, 3rd Edition 2003

[4] Code to flowchart [Online] , Available at : https://code2flow.com/ [Accessed on 22 April,2019]

[5] GeeksforGeeks Article : "0-1 Knapsack problem", Utkarsh Trivedi, [Online] , Available at : https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/ [Accessed On :21 April,2019]

VIII. APPENDIX

Code:

```
#include <bits/stdc++.h>
using namespace std;

//A solution to knapsack problem using Bound and Branch Approach
//Group 47
//Endsem labtest

struct Node{
    int level, profit, bd;
    float wt;
};

struct Object {
    float wt;
    int val;
};

//this fnc compares which of the two objects is more efficient by taking thir value by weight ratio
bool compar(Object a, Object b) {
    double rr = (double)b.val / b.wt;
    double rs = (double)a.val / a.wt;
    return rs > rr;
}


int knapsack(int W, Object arr[], int n)
{
    sort(arr, arr + n, compar);
    queue<Node> Q;
    Node u, v;

    //this is the dumy node
    u.level = -1;

    u.profit = u.wt = 0;
    Q.push(u); //dummy node pushed to array

    int Profitmax = 0;

    while (!Q.empty()){
            u = Q.front();
            Q.pop();

            if (u.level == -1)
                    v.level = 0;

            if (u.level == n-1)
                    continue;

            v.level = u.level + 1;
            v.wt = u.wt + arr[v.level].wt;
            v.profit = u.profit + arr[v.level].val;


    if (v.wt <= W && v.profit > Profitmax)
                    Profitmax = v.profit;

            v.bd = bd(v, n, W, arr);

            if (v.bd > Profitmax)
                    Q.push(v);

            v.wt = u.wt;
            v.profit = u.profit;

            v.bd = bd(v, n, W, arr);
            if (v.bd > Profitmax)
                    Q.push(v);
    }

    return Profitmax;
}
```

```
int bd(Node u, int n, int W, Object arr[]) {
    if (u.wt >= W) return 0;

    int profit_bd = u.profit;
    int j = u.level + 1;
    int totwt = u.wt;

    while ((j < n) && (totwt + arr[j].wt <= W)) {
            totwt += arr[j].wt;
            profit_bd += arr[j].val;
            j++;
    }

    if (j < n){
            profit_bd += (W - totwt) * arr[j].val / arr[j].wt;
    }

    return profit_bd;
}

int main()
{
    int W;
    int n;
    cout<<"How many objects do you have? ";
    cin>>n;
    Object arr[n];
    cout<<"Enter the weight of objects"<<"\n";

for (int i = 0; i < n; ++i)
    {
            cin>>arr[i].wt;

    }
    cout<<"Enter the value of objects"<<"\n";
    for (int i = 0; i < n; ++i)
    {
            cin>>arr[i].val;
    }


cout<<"What is the maximum weight of Knapsack?";
    cin>>W;


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

Code execution:

```
harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$ ./a.out
How many objects do you have?3
Enter the weight of objects7
3
9
Enter the value of objects6
70
10
What is the maximum weight of Knapsack?10
Maximum possible profit = 76harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$ ./a.out
How many objects do you have?3
Enter the weight of objects7
3
9
Enter the value of objects6
70
10
What is the maximum weight of Knapsack?12
Maximum possible profit = 80harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$
```

```
harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$ g++ knapsack_cpp.cpp
harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$ ./a.out
How many objects do you have? 5
Enter the weight of objects
2
3.14
1.98
5
3
Enter the value of objects
40
50
100
95
30
What is the maximum weight of Knapsack?10
Maximum possible profit = 235harsh@harsh-Lenovo-ideapad-500-14ISK:~/Downloads$
```