A Proposed Solution to Knapsack Problem Using Branch & Bound Technique

Somya Goyal - Assistant Professor, Computer Science & Engineering, PDM college of Engineering, Bahadurgarh, Jhajjar, India
Email. somyagoyal1988@gmail.com

Anubha Parashar - Assistant Professor, Computer Science & Engineering, Manipal University, Jaipur, India
Email. anubhaparashar1025@gmail.com

Abstract: Knapsack problem comes under combinatorial optimization problem. It says if we have a set of items, containing a weight and a value, then we need to include in a collection of all items and find total count of all instances so that the total weight is less than or equal to a given limit and the total value is as large as possible. As this problem is NP – hard we need to find exact solution techniques so that we have reasonable solution times for nearly all instances encountered in practice, despite of having exponential time bounds for a number of highly contrived problem instances. In this paper we proposed a solution by modeling the solution space as a tree and then traversing the tree exploring the most promising subtrees first. Here we try to develop an algorithm where worst case complexity is bounded by some appropriate measure of the "hardness" of a problem.
Key Words: Knapsack, Dynamic Programming, Branch and Bound Technique, LCBB.

Introduction: Problem Statement:
_ Given a knapsack with maximum capacity W, and a set S consisting of n items
_ Each item i has some weight wi and benefit value bi (all wi , vi and W are integer values)
_ Problem: How to pack the knapsack to achieve maximum total value of packed items?

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W,[1]

maximize $$\sum_{i=1}^{n} vi\, xi$$

subject to $$\sum_{i=1}^{n} wi\, xi <= W$$

and $xi \in \{0,1\}$

Here x_i represents the number of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

The **fractional knapsack problem** removes the restriction to carry the object entirely. In this version of problem, the items can be broken into smaller piece, so the thief may decide to carry only a fraction x_i of object i, where $0 \leq x_i \leq 1$.

maximize $$\sum_{i=1}^{n} vi\, xi$$

subject to $\sum_{i=1}^{n} wi\, xi <= W$

Dynamic programming and Greedy Algorithms give existing solution to the respective versions of the problem. In this Paper, we focus on 0-1 Knapsack Problem.

Existing Solution Using Dynamic Programming: Existing method includes Dynamic programming that is a method for solving optimization problems. The idea: Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later. Remark: We trade space for time. The Idea of Developing a DP Algorithm involves *Step1:* Structure: Characterize the structure of an optimal solution. – Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.*Step2:* Principle of Optimality: Recursively define the value of an optimal solution. – Express the solution of the original problem in terms of optimal solutions for smaller problems. *Step 3*: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure. *Step 4:* Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined [2].

Recursive Formula for subproblems

Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight w, or
2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Figure 1. Formula used in DP approach

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 1 to n
   B[i,0] = 0
for i = 1 to n
   for w = 0 to W
        if wi <= w // item i can be part of the solution
                if bi + B[i-1,w-wi] > B[i-1,w]
                        B[i,w] = bi + B[i-1,w- wi]
                else
                        B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // wi > w
```

Figure 2. Knapsack Algorithm

Example: DP Algorithm for 0/1 Knapsack Problem

n = 4 (# of elements) [3]
W = 5 (max weight)
Elements (weight, benefit): (2,3), (3,4), (4,5), (5,6)

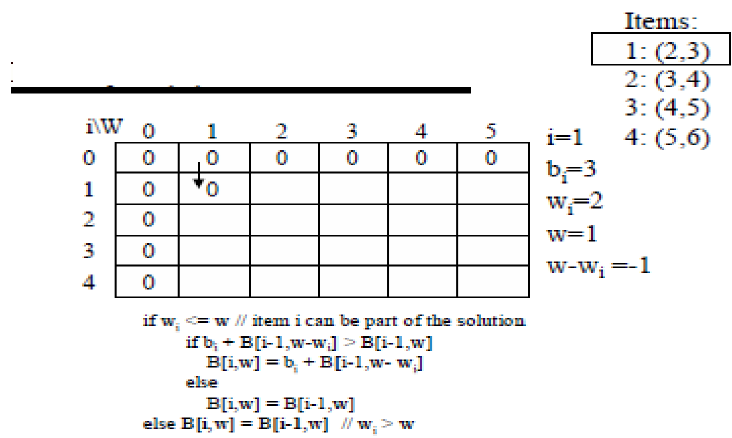

Figure 3. Example Solved with DP Solution (Iteration 1)

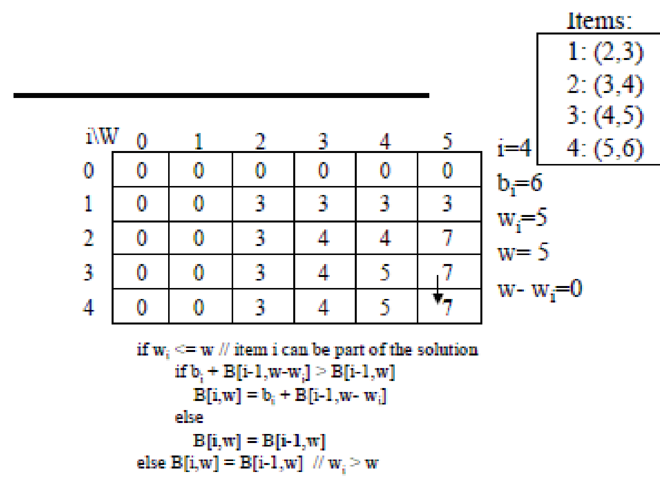

Figure 4. Example Solved with DP Solution

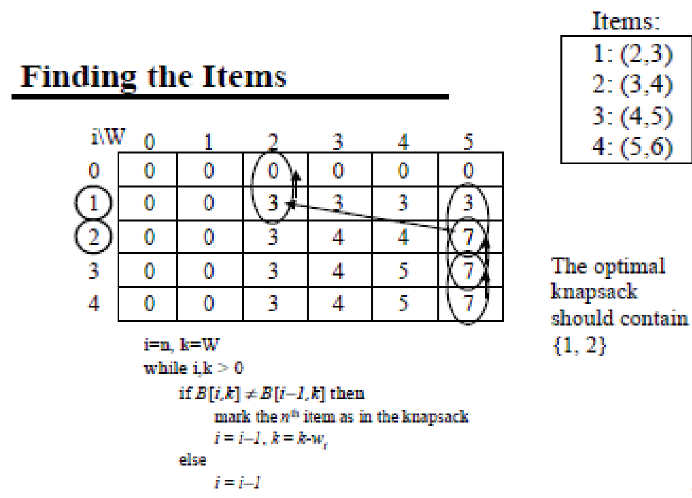

Figure 5. Step to find the Solution vector.

The solution is (1, 1, 0, and 0).

Proposed Approach Using Branch And Bound: The method we are proposing to solve the problem is Branch and Bound Method. The term branch and bound refers to all state space search methods in which all the children of E-node are generated before any other live node can become the E-node. E-node is the node, which is being expended. State space tree can be expended in any method i.e. BFS or DFS. Both start with the root node and generate other nodes. A node which has been generated and all of

whose children are not yet been expanded is called live-node. A node is called dead node, which has been generated, but it cannot be expanded further [4].

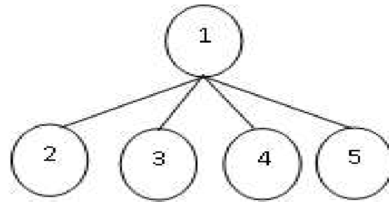

Figure 6. Live Node: 2, 3, 4, and 5

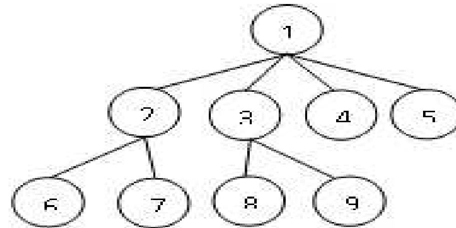

Figure 7. FIFO Branch & Bound (BFS) Children of E-node are inserted in a queue.

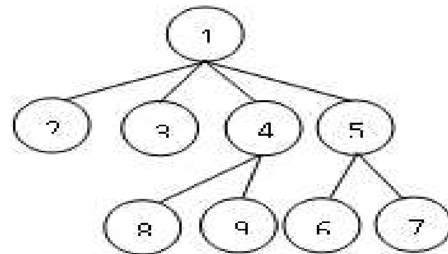

Figure 8. LIFO Branch & Bound (D-Search) Children of E-node are inserted in a stack.

General Method

Step-I. If this information is available, we can compare a node's bound value with the value of the best solution seen so far:

Step-II.If the bound value is not better than the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is non-promising and can be terminated (some people say the branch is pruned) because no solution obtained from it can yield a better solution than the one already available [6] [7] [8] [9].

Solution to the Problem

Step.1. To apply Branch & Bound to 0/1 Knapsack Problem, it is first up necessary to conceive state space tree. For this, we use Fixed-sized-tuple formulation. In this the element x_i of the solution vector is either one or zero depending upon whether the weight included or not [10],

The children of any node can be easily generated as, for any node at level I the left child corresponds to x_i =1 and the right to x_i =0.

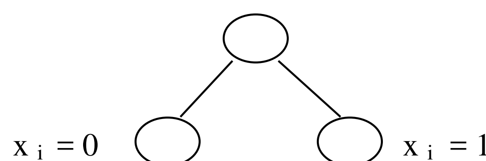

Figure 9. Fixed tuple formulation

Step.2. We cannot apply B & B directly because this problem is maximization problem, whereas B&B is suitable for minimization problem. So, this is overcome by replacing the objective function

$$\text{maximize} \quad \sum_{i=1}^{n} pi\, xi \quad \text{to} \qquad \text{minimize} \quad -\sum_{i=1}^{n} pi\, xi$$

subject to

$$\sum_{i=1}^{n} wi\, xi <= m\,, \quad x_i = 0 \text{ or } 1\,, \quad 1<= i <= n$$

where p_i, w_i *are profit and weight of item* x_i *and,* m *is capacity of knapsack*

Step.3. Every leaf node in the state space tree representing an assignment for which $\sum_{1<=i<=n} w_i x_i <= m$ *is an answer node.*

All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define c(x) = $-\sum_{1<=i<=n} p_i x_i$ *for every answer node x.*

we need two functions ĉ(x) and u(x) such that

ĉ(x) <= c(x) <= u(x).

Algorithms for defining functions

```
Algorithm UBound (cp ,cw, k, m)
 // definition for function u (x) [11-16]
// cp is the current profit total, cw is the current weight total,
// k is the index of the last removed item, m is knapsack size.
{
    b := cp;  c := cw;
    for i := k + 1  to  n  do
    {   if ( c + w[i]  <= m ) then
            {    c := c + w[i] ;   b := b - p[i];        }
    }
  return b;
}
```

Figure 10. Algorithm to compute upper bound value

```
Algorithm UBound (cp, cw, k, m)
// definition for function c (x)
// cp is the current profit total, cw is the current weight total,
// k is the index of the last removed item, m is knapsack size.
{
    b := cp;   c := cw;
    for i := k + 1  to  n  do
    {   c := c + w[i] ;
        if ( c < m ) then  b := b - p[i];
        else return b - (( 1 - ( c - m) / w[i] )  * p[i]) ;
    }
  return b;
}
```

Figure 11. Algorithm to compute bound value

LCBB Solution To The Problem Instance: The Consider an instance of 0/1 Knapsack Problem:
number of items : n = 4
profit vector : (p1 , p2, p3, p4) = (10, 10, 12, 18)
weight vector : (w1, w2, w3, w4) = (2, ,4, 6, 9)
and, knapsack capacity: m =15

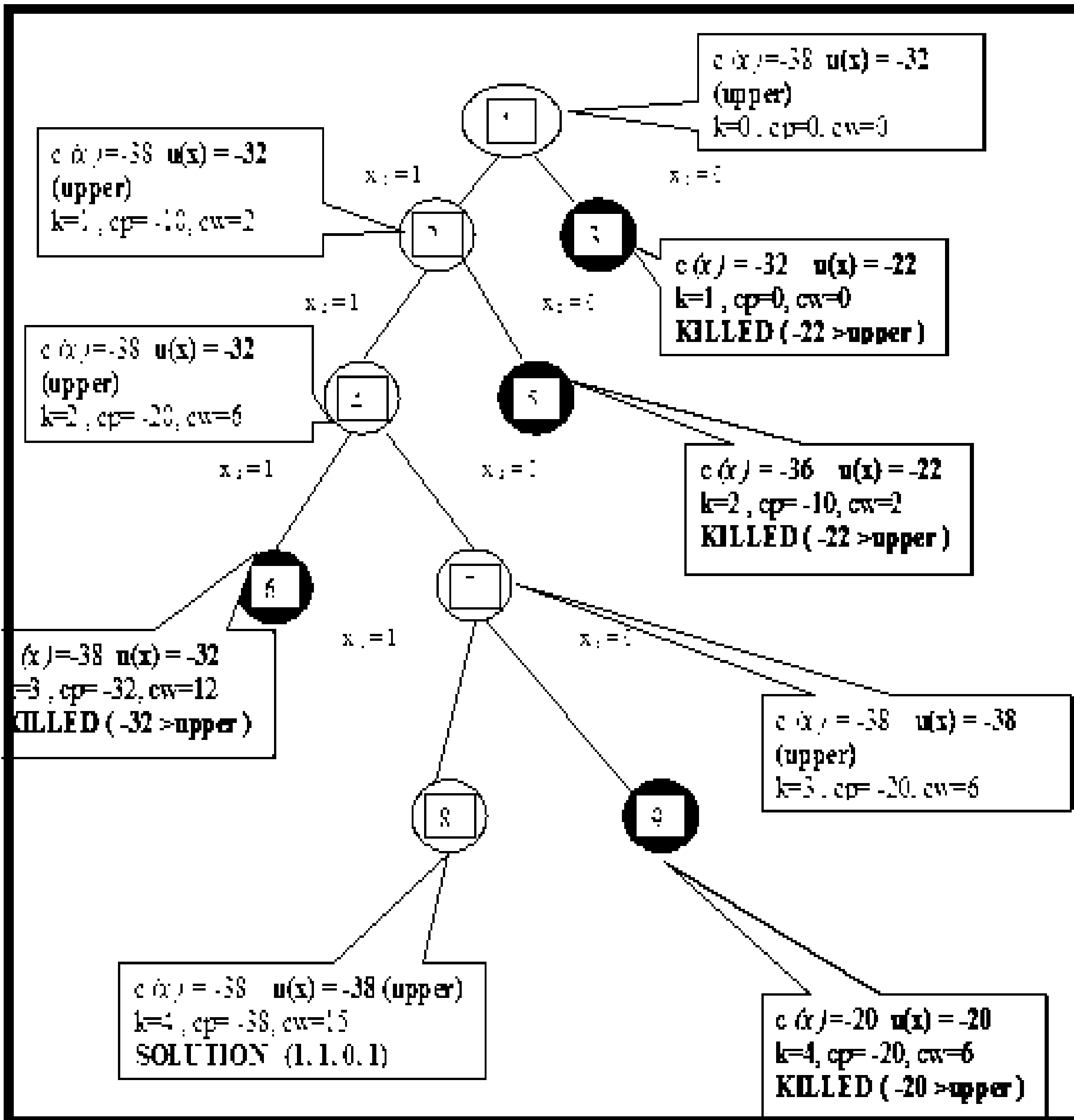

Figura 12. Solution space for instance

Hence, the solution to the instance is (1, 1, 0, 1) and, profit is 38 with weight 15.

Conclusion:

To solve 0/1 Knapsack Problem using B&B technique, we need to specify (1) State space mechanism, (2) Cost functions, 3) how to generate children of a node and (4) how to recognise a solution node. And, the definition to calculate bound values to prune the non-promising nodes is the key of this method. This technique gives better results with exponential complexity. In this paper, we have proposed an implementation of the branch and bound method. Computational results show that our approach is efficient since we have obtained stable speedups around 20 for difficult knapsack problems. Our approach permits also one to solve problems with size 500.

References:

1. Horowitz, S. Sahni, S. Rajasekaran, Fundamentals of Computer Algorithms, 2nd ed. , University Press, Galgotia publications
2. Thomas H Cormen, Charles E Leiserson and Ronald L Rivest Introduction to Algorithms, 1990, TMH
3. Aho A.V. Hopcroft J.E., the Design and Analysis of Computer Algorithm, 1974, Addison Wesley
4. Berlion, P.Bizard, P., Algorithms-The Construction, Proof and Analysis of Programs1986. Johan Wiley & Sons
5. Goodman, S.E. & Hedetnieni, Introduction to Design and Analysis of Algorithm, 1997, MGH.
6. http://paralleltsp.googlecode.com/files/teamDharmaPresentation.pdf.
7. http://lcm.csa.iisc.ernet.in/dsa/node187.html
8. www.dtic.mil/dtic/tr/fulltext/u2/a126957.pdf
9. http://retis.sssup.it/~bini/teaching/optimDisc2
10. 010/bbtsp.pdf
11. http://www.nd.edu/~dgalvin1/30210/30210_F0
12. D. Ulm, “Dynamic programming implementations on SIMD machines - 0/1 knapsack problem,” M.S. Project, George Mason University, 1991.
13. http://www.csd.uoc.gr/~hy583/papers/ch11.pdf
14. D. Ulm, “Dynamic programming implementations on SIMD machines - 0/1 knapsack problem,” M.S. Project, George Mason University, 1991.
15. http://ab.inf.unituebingen.de/teaching/ws04/phylo/script/30_11.pdf
16. V. Boyer, D. El Baz, and M. Elkihel, “Solving knapsack problems on gpu,” Computers and Operations Research, vol. 39, pp. 42–47, 2012
17. D. Ulm, “Dynamic programming implementations on SIMD machines - 0/1 knapsack problem,” M.S. Project, George Mason University, 1991.