The Knapsack Problem

A Survey of Solution Approaches

Sayak Biswas
UNIVERSITY *of* **FLORIDA**
UFID: 54584911

April 17, 2016

Abstract

This paper surveys existing literature for different approaches to solve the knapsack problem. The Knapsack Problem is a combinatorial optimization problem in which one has to maximize the profits gained by packing a set of objects in a knapsack without exceeding its capacity. The problem is $\mathcal{NP}$-hard, thus there is no known polynomial time algorithm for a large input.

Specifically, we take a look at the *0-1 Knapsack Problem* and provide a qualitative comparison between the three well-known approaches towards solving the problem: dynamic programming, backtracking and branch & bound algorithms.

1 Introduction

The *Knapsack Problem* is an optimization problem, which at a high level is to choose the most profitable subset from a collection of available items without overloading the knapsack. The problem is formally defined as follows:

Given a knapsack of maximum capacity C and n items each weighing w_i and with an associated profit of p_i, the *Knapsack Problem* is to choose a subset of the items such to maximize $\sum_{i=1}^{n} p_i x_i$ on the condition $\sum_{i=1}^{n} w_i x_i \leq C, i = 1, ..., n$ where x_i is the number of copies of each item.

The *0-1* knapsack problem restricts the copy count to either zero or one, meaning the object is either included in the knapsack or not, *i.e.* $x_i \in \{0, 1\}$.

2 Algorithms

A naïve brute force approach would be to consider all 2^n possible combinations of items for the knapsack and choose the one that yields te maximum profit. Such an approach would lead to exponential complexity and hence is not desirable.

2.1 Dynamic Programming

Dynamic programming solves optimization problems by breaking it into smaller subproblems and then solving those subproblems to find the overall solution. To design a dynamic programming solution for the problem, we first define a function *knap(1, n, C)* which finds the optimal solution, $f_n(C)$ for a knapsack of capacity C using objects from *1* to *n*. We divide this into subproblems denoted by *knap(1, j, y)* which finds the optimal solution for a knapsack of capacity y using objects from *1* to *j*. Let the solution to this be defined by $f_j(y)$.

At any point in the problem state, the solution depends on making a decision on whether to use the current object or not. So, we obtain the top-down recurrence relation

$$f_j(y) = max \quad \{f_{j-1}(y), f_{j-1}(y - w_j) + p_j\}, y \geq w_j \tag{1}$$

Let us consider an example with a knapsack of capacity $C = 6$, three given objects with weights 2,3,4 and profits 1,2,5 respectively. Using the relation 1 we calculate all possible values and store them in a table as follows

	0	1	2	3	4	5	6
$f_1(y)$	0	0	1	1	1	1	1
$f_2(y)$	0	0	1	2	2	3	3
$f_3(y)$	0	0	1	2	5	5	6

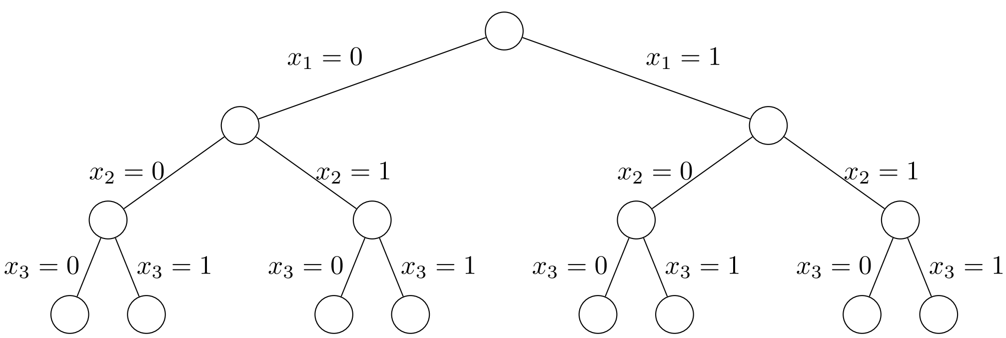
So, we have a matrix of size *n* x *C* which we fill in row-wise manner as values in a row depend on the previous row values. The running time is *O(nC)* and the space requirement is *O(C)*. Below is the pseudo-code:

Algorithm 1 DynamicKnapsack($p,\ w,\ n,\ C$)

```
for j from 0 to C do
    result[0, j] ← 0
end for
for i from 1 to n do
    for j from 0 to C do
        if w[i − 1] > j then
            result[i, j] ← result[i − 1, j]
        else
            result[i, j] ← max(result[i − 1, j], result[i − 1, j − w[i − 1]] + p[i − 1])
        end if
    end for
end for
```

2.2 Backtracking

A backtracking algorithm builds a set of possible solution all the while discarding any partial solution candidates which it determines to be not feasible. As discussed earlier the solution space for *0-1* Knapsack problem consists of 2^n ways of assigning 0 or 1 to x_i. Below is a possible solution space when $n = 3$.



This space is searched in depth-first manner from the start node. From each E-node we check if the next node will result in a feasible solution. If yes, the next node becomes the E-node. If no, we backtrack to the previous node and kill the E-node. To do this, we use an upper bound on the best solution that can be achieved from this node. We obtain this bound by relaxing the constraint from $x_i \in \{0,\ 1\}$ to $0 \leq x_i \leq 1$ and using the greedy algorithm (of sorting the items in non-decreasing order of profit per unit weight and adding to the knapsack) for the rest of the problem. The pseudo-code for the bounding algorithm and the backtracking solution is given below.

Algorithm 2 GreedyBound($p,\ w,\ n,\ C,\ k,\ p_c,\ w_c$)

```
for i from k+1 to n do
    w_c ← w_c + w_i
    if w_c < C then
        p_c ← p_c + p_i
    else
        return p_c + (1 − (w_c − C)/w_i) ∗ p_i
    end if
end for
return p_c
```

Algorithm 3 BacktrackingKnapsack($p,\ w,\ n,\ C,\ k,\ p_c,\ w_c$)

```
if w_c + w_k ≤ C then
    y_k ← 1
    if k < n then
        BacktrackingKnapsack(p, w, n, C, k+1, p_c + p_k, w_c + w_k)
    end if
    if p_c + p_k > p_t and k = n then
        p_t ← p_c + p_k
        w_t ← w_c + w_k
        for j from 1 to k do
            x_j ← y_j
        end for
    end if
end if
if GreedyBound(p, w, n, C, k, p_c, w_c) ≥ p_t then
    y_k ← 0
    if k < n then
        BacktrackingKnapsack(p, w, n, C, k+1, p_c, w_c)
    end if
    if p_c > p_t and k = n then
        p_t ← p_c
        w_t ← w_c
        for j from 1 to k do
            x_j ← y_j
        end for
    end if
end if
```