# The Knapsack Problem
*A Survey of Solution Approaches*

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

April 16, 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 that the below holds true:

maximize $\sum\limits_{i=1}^{n} p_i x_i$

on the condition $\sum\limits_{i=1}^{n} w_i x_i \leq C, i = 1, ..., n$

where $x_i$ is the number of copies of each item.

The *fractional* knapsack problem allows placing a fraction $x_i$ of object $i$ into the knapsack *i.e.* $0 \leq x_i \leq 1$.

The *bounded* knapsack problem restricts each item type to an integer amount of copies *i.e.* $x_i \in \{0, ..., m_i\}$

The *unbounded* knapsack problem removes any restriction from the number of copies of each object type *i.e.* $x_i \geq 0$.

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.* $x_i \in \{0,\ 1\}$. In this paper we will survey the *0-1* knapsack problem.

# 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 solve a problem using dynamic programming, it should have two important characteristics:

- *Optimal Substructure*: This means that the overall optimal solution to a problem comprises optimal solutions to the subproblems.
- *Overlapping Subproblems*: This means that any algorithm used to solve the problem should be solving a subset of the subproblems over and over again instead of generating new subproblems. Dynamic programming stores the results of these subproblems and uses them whenever the subproblem is encountered again. This is called *memoization.*

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).*

## 2.2 Backtracking