Title

Pedro Henrique Centenaro[*]
Supervisor: Luiz Rafael Santos

Abstract. TBD

[*]Undergraduate Control, Automation and Computation Engineering student at UFSC/Campus Blumenau.

1 Introduction

TBD

2 Knapsack problems

Informally, knapsack problems (KPs) may be described as finding the optimal combination of items from a set. Usually, this decision involves a certain value intrinsic to each item, and the objective is to maximize the total value of the selected items. To complicate this decision, items may have certain limiting values associated with them, which means not all of them may be selected.

The simplest example of a KP is the binary knapsack problem (BKP), also known as 0-1 knapsack problem [1]. It consists of a list of n items, each with a value v_i and a weight w_i associated with them, for $i \in \{1, \dots, n\}$. A subset of items must be selected in order to maximize the total value, subject to a maximum weight capacity[1] C. One may thus formulate the BKP as Integer Program (1),

$$\begin{aligned} \max \sum_{i=1}^{n} x_i v_i & \\ \text{s. t.} & \\ \sum_{i=1}^{n} x_i w_i &\leq C, \\ x_i &\in \{0,1\}, \end{aligned} \tag{1}$$

where x_i are decision variables that specify whether item $i \in \{1, \dots, n\}$ has been selected or not.

By introducing slight variations to this formulation, it is possible to obtain many different uses for the KP:

- Unbounded knapsack problem (UKP): By letting $x_i \in \mathbb{N}$, the number of units of each item available is unlimited.
- Fractional knapsack problem (FKP): By letting $x_i \in [0, 1]$, a fraction of each item can be selected.
- Subset sum problem (SSP): When $v_i = w_i \, \forall i \in \{1, \dots, n\}$, the objective can be interpreted as maximizing the weight of the knapsack. Furthermore, if we demand that $\sum_{i=1}^{n} x_i w_i = C$, then the problem becomes finding the combination of items that produces sum C exactly [4].

[1]This limiting factor, which can be interpreted as a bag or a container, is what gives the KP its name.

The KP can be further complicated by taking into account multiple knapsacks and constraints. In this section, we focus on the latter.

2.1 Simple algorithms for the Binary Knapsack Problem

The selection of subsets implies that an upper bound on the number of solutions to the BKP is 2^n. Therefore, a natural first solution to the problem is a recursive binary tree algorithm [3]. Let $I = \{1, \dots, n\}$ be the set of items to be put in the knapsack. Then, by representing the values and weights of items as entries in vectors $v, w \in \mathbb{R}^n$, respectively, and the remaining knapsack weight capacity as $c \in \mathbb{R}$, we arrive at Algorithm 1 for an instance $\phi(n, c, X, s)$ of the problem, where $X \subseteq I$ is the set of selected items and s is the sum of said items' values at the node.

Algorithm 1 Recursive binary tree algorithm for the KP: $\phi(n, c, X, s)$

```
1: if $n = 0$ then
2:     return $(X, s)$
3: $X_1, s_1 := \phi(n-1, c, X, s)$
4: $X_2, s_2 \leftarrow (\emptyset, \emptyset)$
5: if $w_n \leq c$ then
6:     $X_2, s_2 := \phi(n-1, c - w_n, X \cup \{n\}, s + v_n)$
7: $s := max(s_1, s_2)$
8: $X_3 \leftarrow 0$
9: if $v = v_1$ then
10:     $X_3 := X_1$
11: else
12:     $X_3 := X_2$
13: return $(X_3, s)$
```

The algorithm consists of a top-bottom phase, in which the nodes are generated, and a bottom-top phase, in which the optimal solution is found. The top-bottom phase begins at Line 3, where a new instance of ϕ is called with item n discarded from the solution. Subsequently, if there is enough capacity left in the knapsack, a second node is generated, with n included in the solution.

When $n = 0$ (Line 1), it is impossible to generate a new node. Thus, the only remaining action is to return the solution set at the node and its value sum. Once the whole tree has been generated, Line 7 is reached. From this point on, the bottom-top phase begins, with each node selecting the best solution its children have to offer. Hence, the optimal solution is returned by the root node.

Due to the combinatorial nature of the KP, Algorithm 1 is only efficient at solving small problems [3].

Example 2.1. Suppose we have a knapsack with maximum capacity $C = 5$ and three items. The weight vector is $w = [1, 2, 3]$ and the value vector is $v = [2, 4, 3]$. By applying Algorithm 1 to this problem, we obtain the tree in Figure 1.

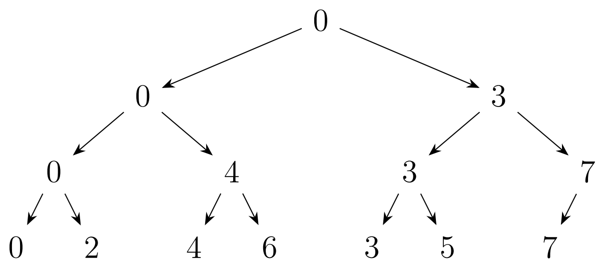

Figure 1: BKP binary tree algorithm example.

Each node in the tree represents the value sum of a feasible combination of items. It immediately becomes clear that the optimal solution in this case is 7, with the combination of items 2 and 3. However, this is not obvious to the computer, which must solve every recursion and return to the root node first.

Dynamic programming [6] approaches to the KP exist [2, 3, 5]. The idea behind these approaches is to generate a table with solutions to subproblems of the KP, using previous calculations to shorten the path to the optimal solution.

Let $T \in \mathbb{R}^{(n+1)\times(c+1)}$ be the dynamic programming table for the KP. Algorithm 2 exemplifies how T can be constructed and used to arrive at the optimal KP solution.

$T_{i,j}$ represents the optimal value for a subproblem of the KP with $i - 1$ items and knapsack capacity $j - 1$. Consequently, no set of items can satisfy $j = 1$, and so $T_{i,1} = 0\,\forall i \in \{1, \dots, n+1\}$. Furthermore, since there are no items to select when $i = 1$, it follows that $T_{1,j} = 0\,\forall j \in \{1, \dots, c+1\}$. This explains lines 1 through 4.

Lines 5 through 8 fill the remaining entries in the table. If $w_{i-1} > j - 1$, we simply keep the best solution found so far for capacity $j-1$. Otherwise, we compare said solution with what would be obtained if $i - 1$ was selected. The larger value becomes entry $T_{i,j}$.

Once table T is finished, starting at Line 10, the process of determining what items constitute an optimal solution occurs. Following the logic of lines 5 through 8, if $T_{i,j} = T_{i-1,j}$, then the current item has not been selected and we may move on to the next item. Otherwise, the item is included in the final solution and we move on to the next item with less capacity, $T_{i-1,j-w_i}$.

Algorithm 2 Dynamic programming algorithm for the KP: $\Phi(n, c)$

```
 1: for $i$ from 1 to $n+1$ do
 2:     $T_{i,1} := 0$
 3: for $j$ from 2 to $c+1$ do
 4:     $T_{1,j} := 0$
 5:     for $i$ from 2 to $n+1$ do
 6:         $T_{i,j} \leftarrow T_{i-1,j}$
 7:         if $w_i \leq j$ then
 8:             $T_{i,j} \leftarrow \max\left(T_{i,j}, v_{i-1} + T_{i-1,j-w_{i-1}}\right)$
 9: $n_i, c_i, X \leftarrow (n, c, \emptyset)$
10: while $c_i > 0$ do
11:     if $T_{n_i+1,c_i+1} \neq T_{n_i,c_i+1}$ then
12:         $X \leftarrow X \cup n_i$
13:         $c_i \leftarrow c_i - w_n$
14:     $n_i \leftarrow n_i - 1$
15: return X
```

Though better than the binary tree approach, the dynamic programming algorithm's complexity is exponential as well [3]. In fact, this approach is notably inefficient for large capacity values [5].

Example 2.2. Suppose the following problem suggested by Feofiloff [3]: We have 4 items and a capacity 5. The weights and values of items are, respectively, $w = [4, 2, 1, 3]$ and $v = [500, 400, 300, 450]$. To find the optimal solution to this BKP, we start by applying the part of Algorithm 2 to generate Table 1.

0	0	0	0	0	0
0	0	0	0	500	500
0	0	400	400	500	500
0	300	400	700	700	800
0	300	400	700	750	850

Table 1: BKP dynamic programming table example.

Next, we start looking for the solution from the table's bottom right entry. We note that $T_{5,6} \neq T_{4,6}$. This means that item $4 \in X$ (the solution set). Next, we investigate item 3. Since $4 \in X$, that means the capacity is now at $5 - w_4 = 2$. So we look at $T_{4,3}$, which is equal to $T_{3,3}$. We conclude that $3 \notin X$. Moving up a row, $T_{3,3} \neq T_{2,3}$, which means $2 \in X$. By repeating this one more time, we conclude that $X = \{2, 4\}$.

References

[1] Maram Assi and Ramzi A Haraty. “A survey of the knapsack problem.” In: *2018 International Arab Conference on Information Technology (ACIT)*. IEEE. 2018, pp. 1–6.

[2] Erik Demaine and Srini Devadas. *Polynomial Time vs Pseudo-Polynomial Time*. 2011. URL: https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf (visited on 11/23/2011).

[3] Paulo Feofiloff. *Mochila booleana*. 2020. URL: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool (visited on 11/21/2020).

[4] Paulo Feofiloff. *Subset sum*. 2020. URL: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-subsetsum (visited on 11/02/2020).

[5] Maya Hristakeva and Dipti Shrestha. “Different approaches to solve the 0/1 knapsack problem.” In: *The Midwest Instruction and Computing Symposium*. 2005.

[6] David B Wagner. “Dynamic programming.” In: *The Mathematica Journal* 5.4 (1995), pp. 42–51.