UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LEONARDO FERNANDO DOS SANTOS MOURA

An Efficient Dynamic Programming Algorithm For The Unbounded Knapsack Problem

Final Report presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Profa. Dra. Luciana Buriol
Advisor

Porto Alegre, December 14th, 2012

CIP – CATALOGING-IN-PUBLICATION

Leonardo Fernando dos Santos Moura,

An Efficient Dynamic Programming Algorithm For The Unbounded Knapsack Problem /

Leonardo Fernando dos Santos Moura. – Porto Alegre: PPGC da UFRGS,

.

57 f.: il.

Final Report – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS,

. Advisor: Luciana Buriol.

1. Unbounded Knapsack Problem. 2. Cutting Stock Problem. 3. Column Generation. I. Buriol, Luciana. II. Título.

ACKNOWLEDGMENTS

First I would like to thank my parents, Vitor and Raquel, for all their support throughout undergraduation.

Secondly, I would like to thank my Advisor, Luciana Buriol, for all her attention and shared knowledge. And all the professors who guided me through this journey. A special thanks is due to my former advisor Aline Villavicencio, who inspired me to pursue an academic career.

Last but not least, I want to thank all my friends, that I saw grow academically and personally with me. Leonardo, Wagner, Júlia, Tomaz, Alessandro, André, Kauê, Cassio, among many others.

ABSTRACT

This report describes an algorithm for the Unbounded Knapsack Problem based on the algorithm EDUK (Efficient Dynamic Programming for the Unbounded Knapsack Problem). EDUK takes advantage of the problem properties of dominance and periodicity to speed up computation. This algorithm is compared with other implementations and it is tested both with randomly generated instances and with instances generated from a delayed column generation for the Cutting Stock Problem. This report also contains an analysis of unbounded knapsack instances.

Keywords: Unbounded Knapsack Problem, Cutting Stock Problem, Column Generation.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

EDUK	Efficient Dynamic Programming for the Unbounded Knapsack Problem
UKP	Unbounded Knapsack Problem
CSP	Cutting Stock Problem
FPTAS	Fully Polynomial-Time Approximation Scheme
B&B	Branch & Bound
DP	Dynamic Programming
FFD	First Fit Decreasing

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

Suppose a hardware store owner has a limited shelf space on his store, and he wants to select the most profitable products in stock that fit in his shelves. Such problems arise on a daily basis in industry and commercial applications. They usually involve a large array of decisions, many of which are dependent on each other. As problems get bigger, they tend to become impossible to solve manually. Digital computers enhanced our capabilities to solve problems, and also made possible the solution of a wider range of problems that were previously thought impossible to solve. But, which problems are easy and which problems are hard?

A considerable research effort has been directed toward the classification of problems. Stephen Cook's seminal work [6] introduces the idea of classifying problems "hardness" according to their relation with other problems. A problem A is at least as hard as a problem B if an algorithm that solves A can also be used to solve B, in which case it is said that B is reducible to A. Problems that are reducible from hard problems are called "NP-Complete". Such problems are deemed hard for a computer to solve, such that many people think all large instances of NP-Complete problems are impossible to be solved in a reasonable time. This actually is false.

Since the publication of Cook's paper, much work has been done in the classification and efficient resolution of NP-Complete problems. Different solving techniques and improvements on the computational speed of modern computers have led to the resolution of bigger and bigger instances of different problems.

The 1972 paper of Richard Karp [16] presents twenty-one problems that are classified as NP-Complete. Among those problems is the **Knapsack Problem**, a slightly different variant of the one used in this report. The knapsack problem consists in filling a knapsack of limited weight with different items, each item having a weight and a value, maximizing the total value without exceeding the weight limit of the knapsack. This kind of problem arises whenever one has to select from a list of different things the best one according to a given criterion when there is a limited amount of resources. For example, suppose one wants to invest on different companies. Each company has an estimated profitability and cost and one has a finite amount of money to invest. This problem can be modeled as a Knapsack Problem.

One of the most interesting uses of the Unbounded Knapsack problem is in delayed column generation for Linear Programming problems. Column generation is a technique used to solve Linear Programming problems that seem intractable due to their large number of constraints or variables. More details about this technique is given in Section 3.2.

1.1 Contribution

This report describes the implementation of an algorithm for the Unbounded Knapsack Problem based on the algorithm EDUK (Efficient Dynamic Programming for the Unbounded Knapsack Problem), first described in [1]. This algorithm is tested with large, randomly generated Unbounded Knapsack problem instances.

Furthermore, the implemented Unbounded Knapsack problem algorithm is integrated in a solver, described in [7], for the Cutting Stock Problem. This solver uses the delayed column generation technique, combining the commercial linear optimization solver CPLEX with different Unbounded Knapsack Problem algorithms.

Briefly, the contributions of this report are the following:

- Implementation of an algorithm for the Unbounded Knapsack Problem in an imperative programming language;
- Efficient iterative method for detecting threshold dominance;
- Test and analysis of the parameters of EDUK;
- Analysis of the characteristics of instances generated by the column generation algorithm; and
- Analysis of the behavior of the implemented algorithm coupled with a column generation algorithm for the Cutting Stock Problem.

1.2 Structure of this report

This report is structured as follows: Chapter 2 defines the Unbounded Knapsack Problem and describes some of its characteristics. In Chapter 3, the Cutting Stock problem and the delayed column generation technique are explained. Chapter 4 describes the generated instances for the Unbounded Knapsack Problem and the Cutting Stock problem. Next, the implementation details are exposed. Finally, in Chapter 6 some tests are presented to compare the proposed solutions.

2 THE UNBOUNDED KNAPSACK PROBLEM

2.1 Problem Definition

The Unbounded Knapsack Problem (UKP) is a widely studied NP-hard optimization problem [20] with an extensive range of applications in industry and financial management [24]. Informally, a set N of item types is given, each item type i has a weight w_i and a profit p_i. The goal is to fill a knapsack of limited capacity C with a linear combination of the item types in N, maximizing the sum of the profits of the selected items and respecting the capacity constraint. Consider $x_i \in \mathbb{N}$ as the decision variable that indicates the number of copies of the item type i in the solution, then the Unbounded Knapsack problem can be stated as an integer linear program:

$$\begin{aligned} \max \quad & \sum_{j \in N} x_j p_j \\ s.t. \quad & \sum_{j \in N} x_j w_j \leq C \\ & x_j \in \mathbb{N} \qquad\qquad \forall j \in N \end{aligned} \tag{2.1}$$

UKP is part of a family of problems called **Knapsack Problems**. It is called *unbounded* because each item type in N can be used as many times as needed. Other variants of knapsack problems include problems where only a limited number of items of each type can be used (Bounded Knapsack Problem), and problems where each item type can be used only once (0-1 Knapsack Problem).

Even though there is an unlimited available number of items of each type, the amount of items in a solution is naturally bounded by the knapsack capacity. So a solver for the Bounded Knapsack Problem also solves the UKP if we set the bounds b_i on each item i (the maximum number of each item type allowed in the knapsack) as the Capacity C divided by its weight w_i. The same thing can be said for the 0-1 Knapsack Problem: an UKP instance can be transformed into a 0-1 Knapsack Problem instance if we create b_i copies of each item i. Using the aforementioned transformations to solve UKP instances shows a poor performance in practice [17]. In order to use a 0-1 Knapsack Problem solver, a large number of items must be created, specially when the knapsack capacity is large, increasing the processing time and the required amount of memory needed. There are also some properties of the UKP (namely dominance and periodicity) that are not present in other variants. Such properties can be exploited by an specific algorithm for the UKP.

There are two classic, exact methods for solving UKP: Branch & Bound and Dynamic Programming. In this chapter both techniques are explained. These techniques exploit the

dominance property to solve UKP.

2.2 Dominance

If an item i is less profitable and heavier than another item j, it is never used in an optimal solution, since it is always better to replace any copy of i by one or more copies of j without decreasing the total profit.

Such relation between items is called **simple dominance** and it was first observed in [12]. Figure 2.1 illustrates how dominance relations work. An object type is represented as a triangle of width w and height p, the line at the right of the object is its "shadow", the items bellow its shadow are dominated. In Figure 2.1a, the item j simply dominates the object i since $w_j \leq w_i$ and $p_j \geq p_i$.

Some extensions of dominance were proposed in the literature. [22] presents **multiple dominance**. An item i is said to be multiply dominated by j if $\lfloor w_i/w_j \rfloor \geq p_i/p_j$, i.e., it is always better to replace one copy of i by $\lfloor w_i/w_j \rfloor$ copies of j. In Figure 2.1b, the three copies of the object type j dominate one copy of i, so j multiply dominates i.

Finally, [1] proposed another form of domination: **threshold dominance**. An item i is threshold dominated by a set of items J, if, for $\alpha \in \mathbb{N}$ and $y \in \mathbb{N}^{|J|}$, $\alpha w_i \geq \sum_{j \in J} y_j w_j$ and $\alpha p_i \leq \sum_{j \in J} y_j p_j$, i.e., it is always better to replace α copies of item i by some linear combination of the items in J. The case where $\alpha = 1$ is called **collective dominance**.

Collective dominance is shown in Figure 2.1c. The set $\{j, k\}$ collectively dominates i. And in Figure 2.1d the set $\{j, k\}$ threshold dominates i: it is better to replace three copies of i by one copy of j and another of k.

Single, multiple and collective dominance can be used to increase the speed of UKP algorithms by eliminating items that do not alter the optimal solution. In [17] the following proposition is stated:

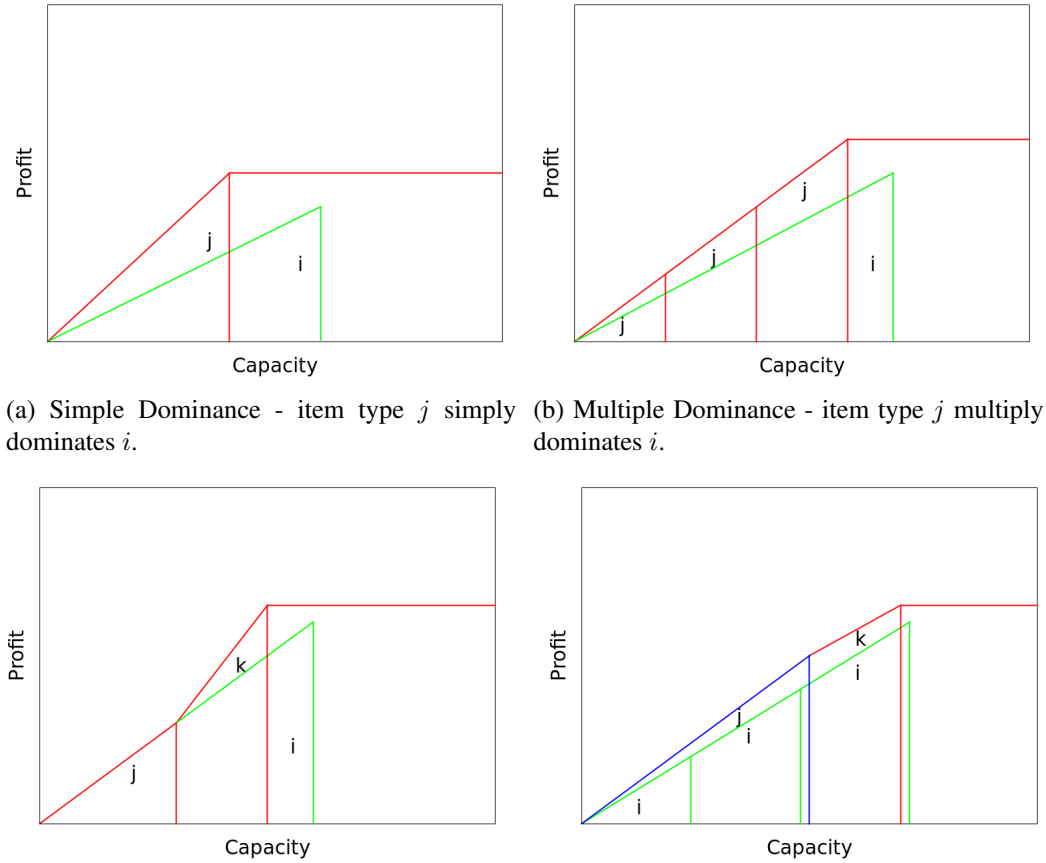
Proposition 1. *For every instance of UKP there always exists an optimal solution not containing any simply, multiply or collective dominated item types.*

That means that all simply, multiply or collective dominated item types can be discarded without changing the optimal solution, largely reducing the search space. Section 2.4.3.3 details how threshold dominance can be exploited.

2.3 Efficiency

Definition The efficiency of an item i is its profit divided by its weight.

An item type i is said to be more efficient than another item type j if the efficiency of i is larger than the efficiency of j. Some algorithms sort item types by efficiency in order to speed up computation. The algorithm proposed in this work uses efficiency to detect threshold dominance.



(a) Simple Dominance - item type j simply dominates i.

(b) Multiple Dominance - item type j multiply dominates i.

(c) Collective Dominance - set $\{j, k\}$ collectively dominates i.

(d) Threshold Dominance - set $\{j, k\}$ threshold dominates i.

Figure 2.1: Dominance Relations.

2.4 Methods used for Solving UKP

Since the publication of the column generation method, there is a growing interest in solving the Unbounded Knapsack problem efficiently. The UKP is an NP-Hard problem, nevertheless there are efficient methods to solve it optimally. Hence, relatively large instances can be solved easily.

An approximate algorithm for the UKP is shown in [18], this algorithm is briefly explained in Section 2.4.1. The most commonly used exact methods are Branch & Bound and Dynamic Programming.

Branch & Bound solutions are presented in [12, 5, 21]. This technique is better explained in Section 2.4.2.

Several special purpose dynamic programming (DP) solutions were presented in the literature [13, 15, 10, 14]. In [17], EDUK (the algorithm provided in this work) is said to be the most efficient dynamic programming solution for the UKP. EDUK2, an algorithm implemented by the same authors of EDUK, is presumably the currently state-of-the-art algorithm for the UKP [25]. The dynamic programming approach and the algorithm are explained in Section 2.4.3.

2.4.1 An Approximate Method

A fully polynomial-time approximation scheme (FPTAS) for UKP is presented in [18]. A fully polynomial approximation scheme is an approximation algorithm that has a parameterizable guarantee on the optimal solution, i.e., the solution given by this algorithm can be arbitrarily closer do the optimal solution. Given the parameter ϵ ($0 < \epsilon \leq 1$), the running time of this method is bounded on a polynomial in the size of the input and in a parameter $1/\epsilon$. Basically it guarantees a solution with an arbitrary quality. Let OPT be the value of the optimal solution for a given maximization problem, and APP be the value of the solution given by the FPTAS, the guarantee of APP is given by Equation 2.2.

$$OPT - APP = \epsilon OPT. \tag{2.2}$$

This means that the approximate solution given by this method is in a factor of $(1 - \epsilon)$ of being optimal. The smaller the parameter ϵ, the longer it will take to run the algorithm.

The main ideas of the algorithm is to divide the item profits by a factor K, that is obtained using ϵ, and to only use the most efficient items. This approximate algorithm has time and space complexity of $O(n + 1/\epsilon^3)$. A more detailed description can be found in [18].

2.4.2 Branch & Bound

The Branch & Bound method consists in enumerating every combination of item types, keeping a lower bound and an upper bound on the optimal solution. The lower bound is given by the best solution obtained during the computation, while the upper bound can be, for example, the solution for the fractional knapsack problem. When the upper bound and the lower bound are equal, the solution is optimal.

The basic Branch & Bound method works in three steps:

1. Sort items by efficiency in a non decreasing order;
2. Calculate upper bound;
3. Backtrack (Algorithm 1).

The backtrack method, shown in Algorithm 1, is used to build the optimal solution. Initially, the maximum number of copies of each item is put in the current solution. The most efficient items are put first, since the item types are sorted by efficiency. When the knapsack is full, the most efficient items are gradually replaced by less efficient items. At every replacement a lower bound on the current solution is calculated. If this lower bound is smaller than the value of a previously found solution (tested in line 8), the item currently being put on the knapsack is discarded and the next item is put on the knapsack. The algorithm proceeds until the value of the solution is equal to the upper bound or every item is used.

The standard B&B algorithm for the Unbounded Knapsack problem is MTU2, presented with more details in [21]. It was experimentally observed that most of the running time of the B&B algorithm was spent in sorting the item types. MTU2 addresses this issue by solving the problem just for some of the item types. The **core problem** is the items

```
   input: j : first item, z : profit, c : capacity
 1 for i := j to n do
 2     for m := ⌊c/w_i⌋ downto 0 do
 3         add m copies of item i to current solution;
 4         z' := z + m p_i;
 5         c' := c − m w_i;
 6         if z' = Upper Bound then
 7             return current solution ;
 8         else if z' + ⌊c' p_{i+1}/w_{i+1}⌋ ≤ Lower Bound then
 9             abandon branch;
10             remove items i from current solution;
11         else
12             update Lower Bound;
13             update best solution;
14             call Branch & Bound(i + 1, c', z');
15         end
16     end
17 end
```

Algorithm 1: Branch & Bound.

with an efficiency better than some threshold. Algorithm 1 solves the core problem, if a solution with a value equal to the upper bound is found for the core problem, this solution is optimal for all the item types. If no such solution is found, the algorithm is run again with some items added to the core. This procedure is repeated until the upper bound is reached or all the items are added to the core problem.

The Upper Bound used in MTU2, called U_3, is calculated using the three most efficient item types. Suppose that $p_1/w_1 \geq p_2/w_2 \geq p_3/w_3$. The upper bound U_3 is defined as the maximum value between two bounds U_0 and $\overline{U}_1$. Let $\overline{c} = C \mod w_1$ be the capacity left after using the maximum number of items of type 1, $c' = \overline{c} \mod w_2$ be the capacity left after using the maximum number of items of type 2 in a knapsack with capacity $\overline{c}$, and $z' = \lfloor c/w_1 \rfloor p_1 + \lfloor \overline{c}/w_2 \rfloor p_2$ be the profit of a solution that uses the maximum number of items of type 1 and the capacity left with the items of type 2. The two bounds are given by:

$$U_0 = z' + \left\lfloor c' \frac{p_3}{w_3} \right\rfloor \quad \overline{U}_1 = z' + \left\lfloor (c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1) \frac{p_2}{w_2} - p_1 \right\rfloor \tag{2.3}$$

The bound U_0 is the profit achieved by using the rest of the capacity with the item type 3 and $\overline{U}_1$ is the value of removing some items of type 1 from the solution given by z' and replacing them by items of type 2.

The performance of this approach depends on the structure of the problem instances, resulting in a hard-to-predict behavior. For some instances it can degenerate to an exponential running time. However, B&B is generally better than dynamic programming for instances with large capacities.

2.4.3 Dynamic Programming

Dynamic programming (DP) is a method used for solving large problems by using the stored solution of slightly smaller problems. Large problems solved by DP have overlapping subproblems that are used to construct their optimal solution. The solution of each subproblem is usually stored in a table, so the subproblems are never calculated more than one time. UKP can be solved by considering knapsacks of lesser capacities as subproblems, storing the best profit for each capacity. The optimal solution for a knapsack of capacity c can then be found by using the solution for the knapsacks with capacities smaller than c.

Suppose that, for example, a knapsack of capacity 11 has to be filled with two items: one of weight 5 and profit 10 and another with weight 2 and profit 3. Suppose that the solution (the largest amount of profit) for knapsacks of capacities up to 10 are known. If we use the first item, we are still going to have 6 units of weight available, so if the optimal solution for a knapsack of weight 6 is 10, using the first item is going to result in a solution with profit $10 + 10 = 20$. If we, however, use the second item, we are still going to have 9 units of weight, if the optimal solution for a knapsack of capacity 9 is 16, then using the second item results in a solution with profit of $16 + 3 = 19$. So the optimal solution for the knapsack of weight 11 is 20. And this value could be used to solve knapsacks of capacities larger than 11.

The method exemplified by the above example is summarized in the following generating function:

$$z(c) = \begin{cases} \max\limits_{i \in N | w_i \leq c} \{z(c - w_i) + p_i\} & \text{if } c > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

A vector z is used to calculate Function 2.4. This vector contains the best profit possible for each capacity c, $0 \leq c \leq C$. The classic dynamic programming algorithm for the UKP is shown in Algorithm 2. At first, in line 1, $z[0]$ is set to zero, since all items have a weight larger than zero. Then for every capacity c from 1 to C, the item type that fits in the current knapsack, and that yields the largest profit is used. Note that the previous solutions of smaller problems are never recalculated, they are fetched from vector z. After z is calculated for every c, the optimal solution can be obtained on $z[C]$.

```
1  z[0] := 0;
2  for c:= 1 to C do
3  |   z[c] := z[c − 1];
4  |   foreach item type i ∈ N do
5  |   |   if w_i ≤ c then
6  |   |   |   z[c] := max{z[c], z[c − w_i] + p_i};
7  |   |   end
8  |   end
9  end
10 return z[C];
```

Algorithm 2: Classic Dynamic Programming Algorithm for UKP.

The time complexity of this approach is O(nC), line 6 is executed n times (the number

of items) for every capacity smaller than C. Its space complexity is O(C), since the profit for every capacity smaller than C has to be stored.

The algorithm EDUK (Efficient Dynamic Programming for the Unbounded Knapsack Problem), first described in [1], is a dynamic programming solution for UKP that uses the periodicity and dominance properties, further explained in this chapter. The implementation of EDUK provided by this work is explained in Section 5.1.1. EDUK operation is similar to that of Algorithm 2, but the running time is reduced by calculating less capacities, using periodicity, and removing items that are not used, using dominance.

2.4.3.1 Periodicity

Periodicity is a property specific to the Unbounded Knapsack Problem [10]. It states that for capacities larger than some capacity $\hat{C}$, only the best item b is used. So, when such a capacity is reached by the dynamic programming method, the optimal solution can be calculated with Equation 2.5.

$$z(C) = z(\hat{C}) + \left\lceil \frac{C - \hat{C}}{w_b} \right\rceil \times p_b \tag{2.5}$$

2.4.3.2 Sparse Representation

As shown in Section 2.4.3, the classical way of calculating Equation 2.4 is to calculate every capacity from 0 to C. In [2], a method is described to only calculate the values that are needed. To implement this approach, some data structure must be used. Exploiting sparsity can have a prohibitive cost.

The algorithm proposed at this work does not implement the sparse representation. The reasons are given in Chapter 7.

2.4.3.3 Threshold Dominance

For an item i, and some set of items J, let $y = \alpha w_i$ be the smallest capacity such that $\alpha w_i \geq \sum_{j \in J} y_j w_j$ and $\alpha p_i \leq \sum_{j \in J} y_j p_j$. For capacities larger than y, the item type i will never affect the optimal solution. Since the dynamic programming algorithm calculates function 2.4 incrementally, when threshold domination is detected for an item type, it can be removed.

To detect threshold dominance, [1] introduces the function $l(i, y)$ defined in Equation 2.6. Equation 2.6 is, for an item type i and a capacity y, the largest capacity y', $y' \leq y$, in which i was the most efficient item used in an optimal solution.

$$l(i,y) = \begin{cases} 0 & \text{if } y < w_i, \\ l(i, y-1) & \text{if } y \geq w_i \text{ and } z(y) > z(y - w_i) + p_i, \\ l(i, y-1) & \text{if } y \geq w_i \text{ and } z(y) = z(y - w_i) + p_i, \\ & \text{and it exists an item type } k, k \neq i, \text{ such that } l(k, y) = y, \\ y & \text{if } y \geq w_i \text{ and } z(y) = z(y - w_i) + p_i, \\ & \text{and for every item type } k, k \neq i, l(k, y) < y, \end{cases} \tag{2.6}$$

Function l is a test to detect threshold dominance. The following proposition states that for capacities larger than $y' = l(i, y)$, the item type i can be removed and not considered when calculating Function 2.4. In the implementation of EDUK proposed in this work, a non-recursive procedure is used to detect threshold dominance based on Function 2.6. This procedure is explained in Section 5.1.1.3.

Proposition 2. *If an item type i is threshold dominated by some set J, then $y' \leq \min\{y' | l(i, y') \leq (y' - w_i)\}$.*

2.4.3.4 Obtaining the optimal solution

Function 2.4 calculates only the best profit for a given knapsack capacity. In certain applications, such as Column Generation, the information about which items were used is also required. As explained before, the dynamic programming algorithm for UKP uses a table that stores the values of Function 2.4 for each $c \in [0, C]$. Using this table, the optimal solution can be obtained without the use of any special structure nor extra space.

Algorithm 3 shows a pseudo-code for the backtrack method used to obtain the optimal solution from the vector z. A variable $current\_capacity$ is initialized with the knapsack capacity C. For every item type i, if i fits in the current knapsack ($w_i \leq current\_capacity$) and it was used in an optimal solution for a knapsack of capacity $current\_capacity$, it is added on the optimal solution and the $current\_capacity$ is updated. This procedure is repeated for every item until the capacity reaches 0 or there are no more items.

Only the items that are not simply, multiply or collective dominated must be considered (since their removal will not affect the optimal solution).

```
1 current_capacity := C;
2 foreach item i in N do
3 |   while current_capacity − w_i ≥ 0 AND
  |   z[current_capacity − w_i] + p_i = z[current_capacity] do
4 |   |   add a copy of item i to the solution;
5 |   |   current_capacity := current_capacity − w_i;
6 |   end
7 end
```

Algorithm 3: Backtracking method for obtaining the UKP optimal solution.

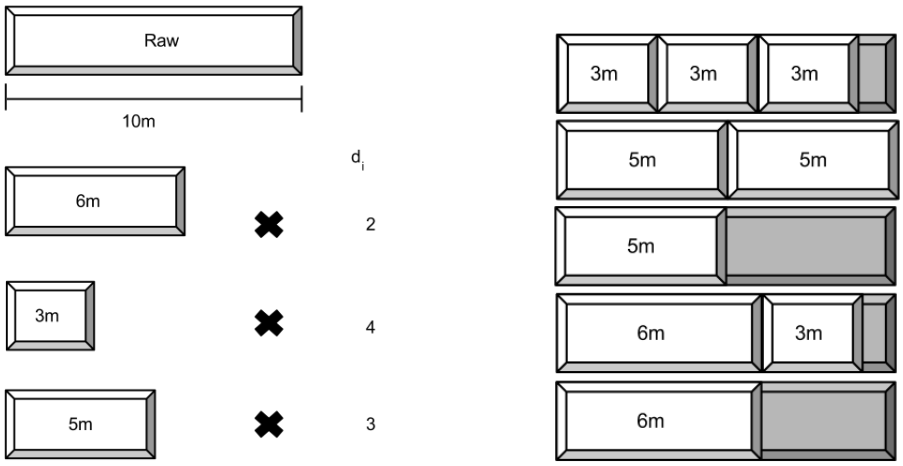
Let w_{min} be the smallest weight in a give instance of UKP, the backtrack algorithm has a worst-case, pseudo-polynomial running time of $O(C/w_{min} + n)$. In practice, the order in which items are verified influences the running time of the backtracking method. It is an open question which order is the best [1]. In the presented implementation of EDUK, the items are verified in the reverse order of their threshold domination detection, i.e., the last item for which threshold domination was detected is the first to be tested.

3 THE CUTTING STOCK PROBLEM

3.1 Problem Definition

The Cutting Stock problem (CSP) is a classical NP-Hard problem and it is subject of much interest in the literature. In [27], more than four hundred publications on cutting and packing problems are cited. CSP naturally arises in a variety of industries, such as paper, glass, steel, among others [23]. The problem consists in cutting a demand of various item sizes from an unlimited number of raws of fixed size, minimizing the number of raws used.

Formally, there is an unlimited set of raws of size L. And there is a set S of item sizes. Each item size $i \in S$ has length $s_i \leq L$ and a demand d_i that must be fulfilled. The goal is to find a way of cutting the raws in order to fulfill the demand, minimizing the number of raws used. Figure 3.1a shows an example of a Cutting Stock instance, originally shown in [17], with a raw of size 10m and three item sizes, with $s = \{6, 3, 5\}$ and $d = \{2, 4, 3\}$. A possible solution using five raws is shown in Figure 3.1b. Note that eleven meters of raw material were wasted by this solution. However, this solution has the minimum number of necessary raws.



(a) Cutting Stock Instance.

(b) Cutting Stock Solution.

Figure 3.1: Cutting Stock Example

An Integer Programming formulation, presented in [17], is given in Formulation 3.1. The value U is an upper bound on the minimum number of raws necessary (U can be

obtained by using, for example, the FFD heuristic described in Section 3.2.2). The decision variable $x_{ij} \in \mathbb{N}$ indicates the amount of items of size i produced by the raw j and $y_i \in \{0, 1\}$ is a boolean vector indicating whether the raw j is used or not.

$$\begin{aligned} \min \quad & \sum_{j=1}^{U} y_j \\ s.t. \quad & \sum_{i=1}^{n} s_i x_{ij} \leq L y_j, && j = 1, \ldots, U, \\ & \sum_{j=1}^{U} x_{ij} \geq d_i, && i \in S \\ & x_{ij} \in \mathbb{N}, y_j \in \{0, 1\} && i \in S; j = 1, \ldots, U \end{aligned} \tag{3.1}$$

This is a valid integer programming formulation for the Cutting Stock Problem. However it is solved inefficiently by integer programming solvers, so it is not commonly used in practice. The main reasons for that is:

- The LP-relaxation of 3.1 is not a good approximation of the Integer Program solution [8];
- The problem has many symmetrical solutions. Every permutation of the columns is equivalent, what makes the problem difficult for integer programming methods to solve. Symmetry can lead, for example, Branch & Bound algorithms to have several equivalent branches.

Another possible, but less compact, formulation is given by Formulation 3.2. Consider a pattern as a way of cutting the raw, defined by a vector $A \in \mathbb{N}^{|M|}$ where a_i indicates the amount of items of size i produced by the given pattern. For example, the first raw shown in Figure 3.1b would be represented by the vector $(0, 3, 0)$ (this pattern produces three items of type 2). Let M be the set of all possible patterns, and $a_{ij} \in \mathbb{N}$ be the number of copies of the size i generated by the pattern j, and $x_j \in \mathbb{N}$ be the number of times the pattern j is used.

$$\begin{aligned} \min \quad & \sum_{j \in M} x_j \\ s.t. \quad & \sum_{j \in M} a_{ij} x_j \geq d_i && \forall i \in S \\ & x_j \in \mathbb{N} && \forall j \in M \end{aligned} \tag{3.2}$$

The linear relaxation (henceforth called LP) of 3.2 gives a lower bound and a good approximation on the number of raws needed. Let $OPT(PI)$ be the optimal number of raws needed for problem 3.2. Any solution containing only integer values would also be a solution for LP, so $OPT(LP) \leq OPT(IP)$. An optimal solution for LP has at most $|S|$ non-zero variables (since LP has $|S|$ constraints). A feasible solution can be obtained by rounding up those variables, what yields a solution for 3.2 that uses $OPT(LP) + |S|$ raws in the worst case. So $OPT(LP) \leq OPT(IP) \leq OPT(LP) + |S|$.

Rounding the fractional variables up is only one possible method of transforming the solution of the linear relaxation into an integer solution. Wäsher and Gau [28] suggest rounding down the fractional variables and filling the remaining demands with the FFD heuristic (the FFD heuristic is described in Section 3.2.2). In practice this approach results

in much better approximations of the optimal solution. In the experimental results of [7], it was observed that this approach yields a solution that is no more than 4% of $OPT(IP)$.

The problem of using the linear relaxation of 3.2 is that the size of the set M grows exponentially with the number of item sizes. To overcome this problem, the delayed column generation is used.

3.2 Delayed Column Generation

Column Generation is a technique used to solve problems that have a large number of variables. In this method, presented in [11], when the simplex method needs to find the next column to enter the basis, instead of selecting over all columns, a column is generated by solving a subproblem. Column Generation has been used in the literature to solve various hard problems such as Bin Packing, Generalized Assignment, and Crew Scheduling [19, 4, 26].

As Formulation 3.2 has a large number of columns (one for each possible pattern), the main problem is decomposed in two problems: The **Master Problem** and the **Pricing Problem**. The algorithm begins by solving the master problem, a smaller problem $LP(M')$ containing, initially, just a few columns $M' \subset M$. Section 3.2.2 describes how the initial columns are selected. $LP(M')$ is solved to optimality with the revised simplex method.

Considering a minimization problem, the simplex method selects, at each iteration, a column that has a negative objective function coefficient to enter the basis. A variable with a positive coefficient could potentially worsen the solution. When all variables have a positive coefficient in the objective function, the solution is optimal. But the simplex method does not store the coefficient vector of the objective function, so the entering column must be found by solving the pricing problem.

The linear relaxation of Problem 3.2, after slack variables are introduced, can be represented by matrix notation as following:

$$\begin{array}{ll} \min & cx \\ s.t. & Ax = b \\ & x \geq 0. \end{array} \tag{3.3}$$

The vector x can be decomposed in x_B and x_N, the basic and non-basic variables, respectively. Let A_B be the matrix of columns associated with the basic variables, and let A_N be the one associated with the non-basic variables. By decomposing the vector c in the same manner we obtain the following formulation:

$$\begin{array}{ll} \min & c_B x_B + c_N x_N \\ s.t. & A_B x_B + A_N x_N = b \\ & x \geq 0. \end{array} \tag{3.4}$$

By subtracting $A_N x_N$ on both sides of the constraints equation in 3.4 we obtain:

$$A_B x_B = b - A_N x_N \tag{3.5}$$

Since A_B is nonsingular, we can multiply the inverse matrix A_B^{-1} on both sides, resulting in:

$$x_B = A_B^{-1}b - A_B^{-1}A_N x_N \tag{3.6}$$

Replacing Equation 3.6 on Equation 3.4 results in:

$$\begin{aligned} cx &= c_B(A_B^{-1}b - A_B^{-1}A_N x_N) + c_N x_N \\ &= c_B A_B^{-1}b + (c_N - c_B A_B^{-1}A_N)x_N \end{aligned} \tag{3.7}$$

Since CSP is a minimization problem we need to search for the most negative coefficient. The dual variables of LP are $y = c_B A_B^{-1}$. Let $A_j \in \mathbb{N}^{|M|}$ be the j-th column of A. By replacing the dual variables in Equation 3.7 the condition 3.8 is valid if all the reduced costs are nonnegative.

$$(\forall j \in M) \quad c_j - yA_j \geq 0 \tag{3.8}$$

Which, since all coefficients in the objective function of 3.2 are one, is equivalent to Condition 3.9.

$$\max_{j=1,\ldots,|M|} yA_j \leq 1 \tag{3.9}$$

If Equation 3.9 holds, no variable can improve the current solution, so the solution is optimal. But is not practical to generate and test every possible pattern. A column represents a pattern, a way of cutting the raw. The coefficient a_i in a given column A_k is the number of items i produced by the given pattern. So the number of produced items is limited by the raw size L. Let α_j be the number of items i produced by a pattern, the set of all possible patterns is given by Equation 3.10.

$$\sum_{j \in S} \alpha_j s_j \leq L \tag{3.10}$$

Moreover, we want to find a column that satisfies 3.10 and minimizes the reduced cost. This amounts to the pricing problem, defined as follows:

$$\begin{aligned} \max \quad & \sum_{j \in N} \alpha_j y_j \\ s.t. \quad & \sum_{j \in N} \alpha_j s_j \leq L \\ & \alpha_j \in \mathbb{N} \qquad \forall j \in M \end{aligned} \tag{3.11}$$

Essentially, the pricing problem is to generate a new column to be inserted in the master problem. Since the columns are the possible patterns, i.e., the patterns that respect the capacity constraint, and the column searched is the most negative, the pricing problem is tantamount to the unbounded knapsack problem, defined in (2.1).

The column generation method for the Cutting Stock problem is summarized in Algorithm 4. A Linear Program is generated from an initial set of columns, and a dual vector is obtained from the optimal solution of this problem. The dual vector is used to generate

an UKP instance, that is solved through dynamic programming or another method. Then if the solution obtained is less than or equal to one, the algorithm stops and a solution for the original problem is generated from the current linear program. If it is bigger than one, the generated column is added to the current LP and the program starts again.

```
generate LP with initial set of columns;
repeat
    solve LP ;
    build UKP problem using the dual variables of LP ;
    solve UKP ;
    if UKP solution is greater than one then
        add column in LP;
    end
until UKP solution is less than or equal to one ;
```

Algorithm 4: Column Generation for the Cutting Stock Problem.

3.2.1 Example

To illustrate the column generation method, let us consider an example taken from [17], and detailed in Figure 3.1a. Initially, suppose that two patterns are generated $(1, 0, 1)$, $(1, 1, 0)$ and $(0, 2, 0)$. The first step is to solve the following problem:

$$\begin{array}{llllll} \min & x_1+ & x_2+ & x_3 & & \\ s.t. & x_1+ & x_2 & & \geq 4, & \\ & & x_2+ & 2x_3 & \geq 3, & \\ & x_1+ & & & \geq 2, & \\ x_1, & x_2, & x_3 & \geq 0. & & \end{array} \tag{3.12}$$

Solving Problem 3.12, we obtain the dual variables $y_1 = \frac{1}{2}, y_2 = \frac{1}{2}, y_3 = \frac{1}{2}$. The pricing problem is presented in Problem 3.13.

$$\begin{array}{llllll} \max & \frac{1}{2}\alpha_1+ & \frac{1}{2}\alpha_2+ & \frac{1}{2}\alpha_3 & & \\ s.t. & 3\alpha_1+ & 5\alpha_2+ & 6\alpha_3 & \leq 10, & \\ & \alpha_1, & \alpha_2, & \alpha_3 & \in \mathbb{N}. & \end{array} \tag{3.13}$$

Problem 3.13 has a solution $\alpha_1 = 3, \alpha_2 = 0, \alpha_3 = 0$. This solution's objective value is $\frac{3}{2}$, which is bigger than one, so the algorithm continues. The generated column is added to the master problem, resulting in the following problem:

$$\begin{array}{llllll} \min & x_1 & +x_2 & +x_3 & +x_4 & \\ s.t. & x_1+ & x_2 & & +3x_4 & \geq 4, \\ & & x_2+ & 2x_3 & & \geq 3, \\ & x_1 & & & & \geq 2, \\ x_1, & x_2, & x_3, & x_4 & \geq 0. & \end{array} \tag{3.14}$$

This time around the the dual variables obtained are $y_1 = \frac{1}{3}, y_2 = \frac{1}{2}, y_3 = \frac{2}{3}$. And we again generate the following pricing problem:

$$\begin{array}{rllll} \max & \frac{1}{3}\alpha_1+ & \frac{1}{2}\alpha_2+ & \frac{2}{3}\alpha_3 & \\ s.t. & 3\alpha_1+ & 5\alpha_2+ & 6\alpha_3 & \leq 10, \\ & \alpha_1, & \alpha_2, & \alpha_3 & \in \mathbb{N}. \end{array} \tag{3.15}$$

With solution $\alpha_1 = 1, \alpha_2 = 0, \alpha_3 = 1$, that results in the objective value of one, so the algorithm stops and the current solution is optimal for the linear relaxation of the original problem.

The solution obtained $x_1 = 2, x_3 = \frac{3}{2}, x_4 = \frac{2}{3}$ is then rounded down to the value three and the rest of the demands of the items sizes 1 and 2 are fit into two raws with the FFD heuristic. The final result is five, that happens to be the optimal value for the integer programming problem as well.

3.2.2 Generation of the Initial Columns

The First Fit Decreasing (FFD) heuristic is used to generate the initial columns. The items are sorted by size in a decreasing order, each item is assigned to the first raw that has room for it, if no raw is available, a new one is created and the item is assigned to it. The heuristic is summarized in Algorithm 5.

```
1 sort item sizes decreasingly by size;
2 foreach size s_i in S do
3 |   if s_i fits in an available bin B then
4 |   |   put s_i in B;
5 |   else
6 |   |   create a new bin and put s_i in it;
7 |   end
8 end
```

Algorithm 5: First Fit Decreasing Heuristic.

This heuristic is a good approximation on the optimal solution. Let $OPT(L)$ be the optimal solution (the number of bins) for a bin-packing problem L, and let $FFD(L)$ be the number of bins achieved by Algorithm 5. Equation 3.16, shown in [3], states the guaranteed quality of the FFD heuristic. For example, if the optimal solution is ninety raws, FFD finds a solution having at most one hundred thirteen raws.

$$FFD(L) \leq \frac{11}{9}OPT(L) + 3 \tag{3.16}$$

4 INSTANCES GENERATED FOR THE UNBOUNDED KNAPSACK AND CUTTING STOCK PROBLEMS

This Chapter describes the different types of problem instances generated and used for the experimental tests in this report. In Section 4.1, the Unbounded Knapsack Problems instances are characterized, followed by the description of the Cutting Stock Problem instances in Section 4.2.

4.1 Unbounded Knapsack Problem Instances

4.1.1 Realistic Random Instances

Data sets that do not contain simple dominance are called **Realistic random**. They are constructed, as described in [1], by generating n random numbers in the interval $[w_{min}, w_{max}]$ and n numbers in the interval $[p_{min}, p_{max}]$. The two sets are sorted and matched in order. For example, if the sets $\{1, 5, 2\}$ and $\{12, 11, 8\}$ are generated, the generated instance is going to be $\{(1, 8), (2, 11), (5, 12)\}$.

4.1.2 Hard Instances

Instances that do not contain collective dominance are in general harder to solve. One way of generating hard instances [1] is to generate a list of n distinct random weights in the interval $[w_{min}, 2\ w_{min})$. For each item i, its profit is set to be its weight ($w_i = p_i$).

As stated in Section 2.2, an item i is collectively dominated by a set of items J, if, for a linear combination y, $y \in \mathbb{N}^{|J|}$, of the items in J, $w_i \geq \sum_{j \in J} y_j w_j$ and $p_i \leq \sum_{j \in J} y_j p_j$. Any combination that has more than two items will not satisfy the first condition, since the the maximum weight w_{max} is less than two times the minimum weight w_{min}. A combination with only one item would simply dominate the item i, but hard instances clearly do not contain simple dominance, since for each item k, $w_k = p_k$.

4.1.3 Similarity between UKP instances generated in sequence

At each iteration, the column generation method generates one UKP instance. We observed that instances generated in sequence have many items in common, i.e., many items that have the same weight and same profit. Let $(UKP)_i$ be the i-th UKP instance generated by the Column Generation algorithm. The similarity of two instances $(UKP)_i$ and $(UKP)_j$ is the percentage of equal item types between those instances.

Tables 4.1, 4.2, and 4.3 show the mean of similarity between instances generated in sequence by the column generation method on instances with Discrete Union Distributions (those instances are characterized on Section 4.2.1). Tables 4.4 and 4.5 show the mean of similarity between instances generated in sequence by the column generation method on instances with Bounded Probability Sampled Distributions (further characterized in Section 4.2.2).

To further illustrate the similarity between instances, Figure 4.2 shows the similarity between UKP instances generated in sequence by a $BS\{1, 12000, 20000, 800\}$ CSP instance with $n = 40000000$. For each iteration i it shows the percentage of equal item types between the instance $(UKP)_i$ and $(UKP)_{i-1}$. Figure 4.1 shows the same kind of graph for CSP instances with distribution $U\{1, 15000, 28000\}$ and $n = 18970$. Note that the similarity is never 100%, if the same instance were generated in two different iterations of the column generation method, the algorithm would enter in an infinite loop, generating always the same columns.

The instances of UKP generated by column generation with Discrete Uniform distributions seem to maintain much more similarity along the execution of the algorithm. If we observe the results in Section 6.4, we can see that column generation with Bounded Probabilist Sampled distributed instances spend much more time solving the linear program than the UKP, what means that more iterations of the simplex method are executed. Therefore the dual variables at the end of each iteration will change a lot, changing also the generated UKP instance.

As the presented results show, the instances can change more than 80% between two consecutive iterations. It is not clear whether taking advantage of the similarity between the UKP instances would improve the overall performance of the algorithm. One single item type that changes can invalidate all the calculated table of dynamic programming algorithms, forcing it to execute it all again. And the algorithm would have the overhead of checking which items are unchanged.

A more detailed study needs to be done to determine if the similarity between instances could be used to improve the performance of the provided algorithm.

Table 4.1: Mean of Similarity between items generated in sequence by using the column generation method on instances of type U{1, 600, 1000}.

n	Similarity Mean
600	90.09
1897	88.98
6000	88.44
18974	50.90
60000	53.41
189737	55.03
600000	58.15

Table 4.2: Mean of Similarity between items generated in sequence by using the column generation method on instances of type U{1, 6500, 10000}.

n	Similarity Mean
18974	62.96
60000	64.83
189737	70.89
600000	71.19

Table 4.3: Mean of Similarity between items generated in sequence by column generation on instances of type U{1, 15000, 28000}.

n	Similarity Mean
18970	87.44
1897370	91.12
189740	92.57

4.2 Cutting Stock Problem Instances

4.2.1 Discrete Union Distributions

In these distributions, denoted as U{j,h,k}, n items are generated with sizes s evenly distributed in the interval [j,h]. The raw has size k. Items of same size are grouped. So if, for example, the generated sizes are $(10, 10, 14, 30, 30, 30)$, the sizes and demands are going to be $\{(10, 2), (14, 1), (30, 3)\}$.

4.2.2 Bounded Probability Sampled Distributions

These distributions, presented in [9], denoted $BS\{h, j, k, m\}$ are built by generating m different sizes between h and j. A random value in the interval $[0.1, 0.9]$ is assigned for each size. The demands of each size are generated by dividing its assigned random value by the sum of all values and multiplying it by n. So the total number of items is close to n.

Table 4.4: Mean of Similarity between items generated in sequence by using the column generation method on instances of type BS{1, 6000, 10000, 400}.

n	Similarity Mean
4000	21.73
12640	12.47
40000	5.873
126490	8.231
400000	6.586
1264910	7.334
4000000	7.22

Table 4.5: Mean of Similarity between items generated in sequence by using the column generation method on instances of type BS{1, 12000, 20000, 800}.

n	Similarity Mean
40000	17.37
126400	15.98
400000	16.03
1264900	14.8
4000000	15.54
12649100	15.77
40000000	14.37

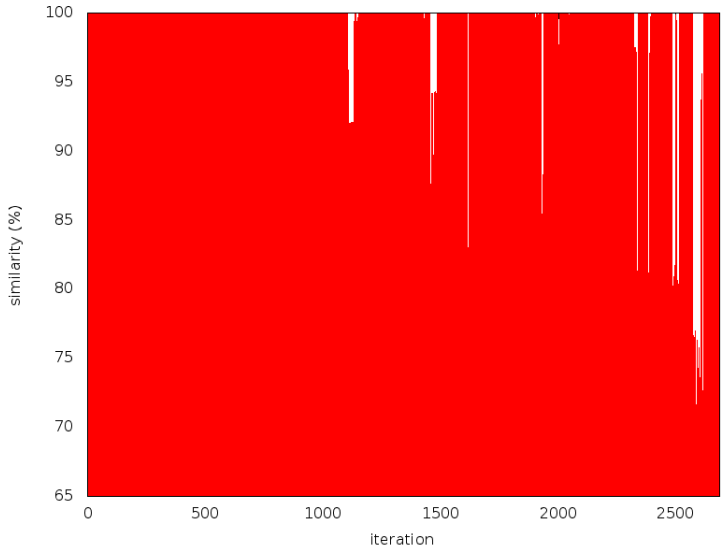


Figure 4.1: Similarity between UKP instances generated in sequence by column generation on a instance of type U{1,15000,28000} with $n = 18970$.

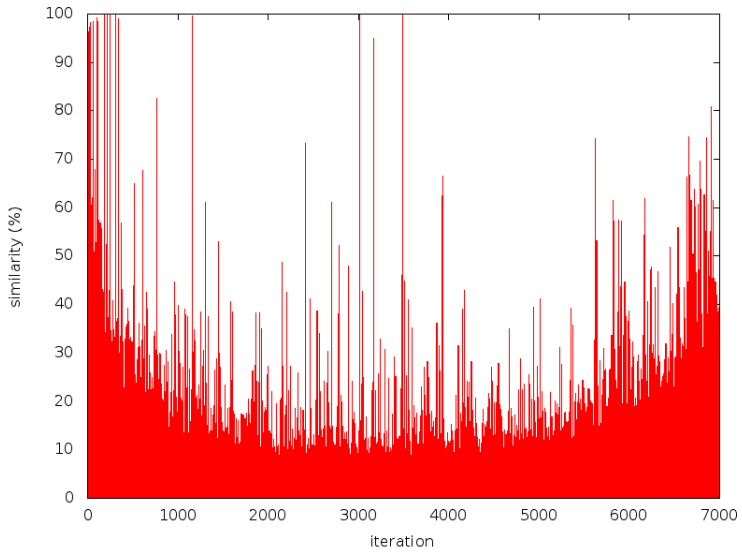


Figure 4.2: Similarity between UKP instances generated in sequence by column generation on a instance of type BS{1,12000,20000,800} with $n = 40000000$.

5 IMPLEMENTATION

5.1 Unbounded Knapsack Problem

5.1.1 myEDUK

The algorithm implemented in this work, myEDUK, is a dynamic programming algorithm for the Unbounded Knapsack Problem based on the algorithm described in [1]. The original algorithm is implemented in the O'caml language. myEDUK was implemented in C++.

This algorithm, summarized in Algorithm 7, computes Function 2.4 by filling a vector z of size C. For each $c \leq C$, $z[c]$ stores the best profit possible for the capacity c. The algorithm is divided in two phases: The **reduction phase** and **the standard phase**.

5.1.1.1 Preprocessing

In dynamic programming solutions for the Unbounded Knapsack problem, there is usually a preprocessing phase before the main computation where dominated items are detected. myEDUK does not preprocess the items. Each item type is processed after every capacity smaller than its weight is calculated, i.e., $z[c]$ for $0 \leq c \leq w_i$ is already calculated for every item j with $w_j \leq w_i$. The advantage of this approach is that single, multiple, and collective dominance can be detected with a single test (executed on line 13), executed in $O(1)$ for each item type: if, for an item i, $z[w_i] \geq p_i$ the item i will never be used, so it is not inserted in the set F of non-dominated items.

5.1.1.2 Reduction Phase

In the reduction phase, shown from line 2 to line 19, the items are processed in slices of t items sorted by increasing weights. A list of the undominated items F sorted decreasingly by efficiency is maintained. In the beginning F is empty. $z[0]$ is set to zero. For each item $i \in N$, $z[c]$ for $w_{i-1} < c \leq w_i$ is calculated using Equation 2.4. If the profit of item i is larger than the current $z[w_i]$, then the item is added to F. Otherwise, it is threshold dominated, and it is discarded.

5.1.1.3 Threshold Dominance Detection

At the end of each item slice, threshold domination is tested for every item. An item is threshold dominated for capacities larger than some y if it is not the most efficient item

for some knapsack with capacity c, $c \in [y - w_i, y]$.

To efficiently detect this, a vector l is stored, holding, for each item i, the last capacity in which i was the most efficient item used in an optimal solution. To maintain this vector valid, F is sorted by efficiency, if $z[c - w[i]] + p[i] > z[c]$, $l[i]$ is set to be c. So, if $z[c']$ is fully calculated, any item for which $l[i] < c' - w[i]$ can be removed. Algorithm 6 summarizes threshold detection.

```
1 d := last capacity calculated;
2 foreach i in F do
3 |   if l[i] < d − w_i then
4 |   |   remove i from F;
5 |   end
6 end
```

Algorithm 6: myEDUK threshold dominance checking.

5.1.1.4 Standard Phase

In the standard phase, shown from line 19 to line 34, $z[c]$ is calculated in slices of q capacities. At the end of each slice the threshold dominance is verified for each item in the same fashion as described above. If the cardinality of F reaches one, the algorithm stops and Formula 2.5 is used to calculate the optimal solution. If this never happens, the algorithm stops when the capacity reaches C.

5.1.1.5 Parameters

The provided algorithm has two parameters:

- t - the size of the item slices in the reduction phase;
- q - the size of the capacity slices in the standard phase.

At the end of each slice, threshold domination is tested for every item in F. So if the values of the parameters are too small, the verification is going to be made more times than needed. If they are too large, the items are going to take more time to be removed, causing the program to take longer to run. Different values of t and q are tested in Section 6.1.

5.1.1.6 Complexity

The number of primitive operations executed by myEDUK is dependent on the knapsack capacity C and the number of items n. The algorithm can be seen as the computation of a $C \times n$ table. Even though on most real cases the periodicity is reached before the computation of C, the worst case (one in which the periodicity is not reached) still occurs in some rare cases. So myEDUK is $O(Cn)$. Note that, Since myEDUK complexity is bounded not by the problem size (the number of items), but by the numeric value of the input, its time complexity is called pseudo-polynomial.

```
1  sort items in ascending order by weight ;
   /* Reduction Phase:                                                    */
2  foreach item slice do
3      foreach item j in the current slice do
4          foreach capacity d ∈ (w_{j−1}, w_j] do
5              z[d] := 0;
6              foreach item f in F do
7                  if z[d − w_i] + p_i > z[d] then
8                      l[f] := d;
9                      z[d] := z[d − w_f] + p_f;
10                 end
11             end
12         end
13         if z[w_j] ≤ p_j (j is not dominated by F) then
14             z[w_j] := p_j;
15             F := F ∪ {j};
16         end
17     end
18     checks threshold dominance;
19 end
   /* Standard Phase:                                                     */
20 foreach capacity slice do
21     if |F| = 1 then
22         periodicity achieved;
23     end
24     foreach capacity d in the current slice do
25         z[d] := 0;
26         foreach item f in F do
27             if z[d − w_i] + p_i > z[d] then
28                 l[f] := d;
29                 z[d] := z[d − w_f] + p_f;
30             end
31         end
32     end
33     checks threshold dominance ;
34 end
```

Algorithm 7: myEDUK.

5.2 Column Generation for the Cutting Stock Problem

To test the implemented UKP solver in column generation, the solver described in [7] was used. This solver integrates the IBM®ILOG®CPLEX®Optimization Studio V12.4, a commercial, closed source, optimization tool, and unbounded knapsack problems solvers.

The solver is implemented in the C programming language. It operates as described in Section 3.2: It builds an initial solution for the Cutting Stock Problem using the FFD

heuristic. Then it constructs the linear program from the initial solution. The linear problem is solved using CPLEX. It generates an UKP instance using the dual variables given by CPLEX. If the value of the solution of this UKP instance is bigger than one, this solution is transformed into a column and inserted in the linear program. When the solution for the Unbounded Knapsack problem is less than or equal to one, the linear program solution is rounded down and the sizes that are left are fit in raws using the FFD heuristic (as described in Section 3.1).

This solver contains the following UKP solvers:

- DP - A dynamic programming solution, it operates similarly to Algorithm 2. But it tests for simple, multiple, and collective dominance using a test similar to that of myEDUK in line 13. It does not use threshold dominance, nor periodicity.
- B&B - a Branch & Bound solver based on [21], the algorithm is described in Section 2.4.2.

Both solvers are experimentally compared with myEDUK in Chapter 6.

6 RESULTS

This Chapter presents the experimental results for myEDUK with the instances described in Section 4. Section 6.1 presents the experimental tests used to obtain the best parameters of myEDUK. Using the best parameters, myEDUK is compared with other UKP solver implementations described in the box below. In Section 6.2, the provided algorithm is compared experimentally with other implementations using the benchmark set provided by the authors of EDUK, large randomly generate instances, and hard instances. In Section 6.3, myEDUK is compared with the Branch & Bound and Dynamic Programming algorithms using instances with Discrete Uniform Distributions and Bounded Probability Sampled Distributions. Finally, in Section 6.4, the amount of time used in solving the UKP subproblem of the column generation method is analysed.

- myEDUK - version of EDUK implemented for this work in C++.
- EDUK2 - Hybrid B&B and dynamic programming solver described in [25], implemented by the authors in O'caml and available on-line.
- EDUK - Dynamic Programming solver described in [1], implemented by the authors in O'caml and available on-line.
- DP - Dynamic Programming using dominance, implemented in C, by the authors of [7].
- B&B - Branch & Bound solver based in [21], implemented in C, by the authors of [7].

In all tests, time is expressed in seconds. The compiler used was g++ 4.6.3 for C++ and C, and ocamlc 3.12 for O'caml. The machine used was an Intel Core i7 930 2.8GHz running Ubuntu 12.04.

6.1 Parameters Testing

As explained in Section 5.1.1.5, myEDUK has two parameters: t and q, the size of the slice of items and the size of the slice of capacities, respectively. In order to discover the best parameters, myEDUK was run with different parameters for the large, randomly generated, realistic random instances described in this chapter on Section 6.2.1.2. The parameter t, the size of the item slice of the reduction phase, was set as a percentage of n

(the number of item types), so for example, if $t = 0.1$ and $n = 100$ the item slices will have size 10. The size of the capacity slice q in the standard phase is set as the ratio of the heaviest item, so, if the heaviest item has $w = 1000$ and $q = 2$, the capacity slice is going to be 2000. The smaller the value of the parameters t and q are, more times the threshold dominance is checked.

The tests were executed for every combination of $t \in \{1\%, 5\%, 10\%, 20\%, 25\%, 30\%\}$ and $q \in \{1, 1.5, 2, 4\}$. Table 6.1 shows the the mean running time of myEDUK over all large instances for every combination of the parameters. The detailed list can be seen at the end of this work on Table 7.1.

Table 6.1: Parameters Tests on Large Realistic Random Instances.

q \ t	1%	5%	10%	20%	25%	30%
1	2.034	2.023	2.025	2.020	2.033	2.022
1.5	2.028	2.021	2.027	**2.018**	2.022	2.019
2	3.112	3.095	3.111	3.104	3.111	3.100
4	5.393	5.379	5.411	5.389	5.402	5.386

The best values found were $t = 20\%$ and $q = 1.5$. Those values were used for all other tests.

6.2 Unbounded Knapsack Problem Tests

In this section myEDUK is tested for large, randomly generated instances, hard instances and instances provided by the authors of EDUK.

6.2.1 Realistic Random instances

6.2.1.1 Benchmark set used by the authors

The first experimental results are shown in Table 6.2. The instances used were made available by the authors of [1]. They were generated as explained in Section 4.1.1 with the parameters indicated in the table and $p_{min} = 1$. The value n is the number of item types, the values w_{min} and w_{max} are, respectively, the minimum and maximum weight used to generate the item types, and p_{max} is the maximum possible profit for the item types, and c is the total knapsack capacity.

Table 6.2: Benchmark set used by the authors of EDUK.

n	w_{min}	w_{max}	p_{max}	c	myEDUK	EDUK2	EDUK	DP	B&B
2000	100	10000	10000	889304	0.020	0.028	0.112	0.008	**0.004**
2000	100	10000	10000	933367	**0.000**	0.072	0.024	0.008	**0.000**
2000	100	10000	10000	914968	0.020	0.156	0.208	0.008	**0.004**
2000	100	10000	10000	934160	0.080	0.068	0.580	0.008	**0.004**
2000	200	10000	10000	894645	0.070	0.088	0.516	0.008	**0.004**
2000	200	10000	10000	917598	0.080	0.184	0.528	0.008	**0.004**
20000	1	100000	200000	1624196	**0.020**	0.240	0.324	0.056	0.424

The running times of myEDUK for those instances are always better than EDUK, and is only worse than EDUK2 for one instance. It is aways worse than B&B and DP but for the last instance. As those instances are small compared with the instances used to set the parameters, this behavior was expected.

6.2.1.2 Large randomly generated instances

For the next tests, realistic random instances were generated with the following parameters:

- $n \in \{10000, 15000, 20000, 25000, 30000\}$;
- $w_{min} \in \{1, 3, 10, 30, 100, 300, 1000, 3000\}$;
- $p_{max} = 200,000$;
- capacity $C = 200,000,000$;
- $w_{max} = 100,000$; and
- $p_{min} = 2$;

The results are shown in Table 6.3, n is the number of items, w_{min} is the minimum item size. A "> 300" indicates that the implementation took more than 300s to run.

myEDUK runs always faster than EDUK, even without implementing sparsity. The worse time over all instances was 12.32s, while EDUK2 worse time was 91.83s. Those results show how dynamic programming has a more predictable behavior than Branch & Bound, since for the largest instance, EDUK2 was faster then myEDUK. This was achieved by EDUK2 due to a good upper bound on the optimal solution.

The parameter w_{min}, used to generate realistic random instances, seems to have a larger impact on the running times than the number of item types. According to [17] as w_{min} gets larger, there are more non-dominated item types. The extreme case is shown in Section 6.2.2, that uses instances that do not contain collective dominance.

6.2.2 Hard Instances

On this section, the tests on hard instances described in Section 4.1.2 are analyzed. All instances have a capacity $C = 200,000,000$. The results are shown in Table 6.4.

Table 6.3: Realistic Random Instances.

n	w_{min}	myEDUK	EDUK2	EDUK	DP	B&B
10000	1	0.010	**0.008**	**0.008**	14.205	1.796
10000	3	0.010	**0.004**	0.008	5.696	0.604
10000	10	0.010	**0.008**	**0.008**	28.870	0.028
10000	30	**0.000**	0.020	0.004	13.209	0.068
10000	100	0.070	0.028	0.256	>300.000	**0.020**
10000	300	**4.180**	4.860	13.329	>300.000	>300.000
10000	1000	**4.750**	12.221	16.021	>300.000	>300.000
10000	3000	**5.540**	15.777	21.145	>300.000	>300.000
15000	1	0.010	**0.008**	**0.008**	9.069	0.464
15000	3	**0.000**	0.004	0.012	5.264	0.464
15000	10	**0.000**	0.016	0.008	18.069	0.072
15000	30	**0.000**	0.012	0.008	23.485	0.096
15000	100	0.010	0.028	**0.008**	189.372	0.020
15000	300	**0.860**	1.592	6.892	>300.000	>300.000
15000	1000	**6.840**	10.857	32.006	>300.000	>300.000
15000	3000	**6.650**	16.733	33.914	>300.000	>300.000
20000	1	**0.000**	0.004	0.008	10.645	0.464
20000	3	**0.010**	0.024	0.012	7.896	0.092
20000	10	**0.010**	0.072	0.116	>300.000	0.048
20000	30	**0.010**	0.100	0.072	>300.000	0.096
20000	100	**0.030**	0.196	0.284	>300.000	1.308
20000	300	**6.100**	25.042	36.970	>300.000	>300.000
20000	1000	**5.980**	26.682	41.431	>300.000	>300.000
20000	3000	**8.880**	24.626	60.052	>300.000	>300.000
25000	1	0.010	**0.004**	0.008	17.281	0.244
25000	3	**0.010**	0.104	0.152	>300.000	0.476
25000	10	0.010	0.020	**0.008**	16.149	0.228
25000	30	**0.010**	0.116	0.032	>300.000	0.036
25000	100	**0.010**	0.096	0.104	>300.000	0.868
25000	300	**0.100**	0.712	1.088	>300.000	>300.000
25000	1000	3.810	**0.040**	36.754	>300.000	0.068
25000	3000	**10.310**	91.834	98.026	>300.000	>300.000
30000	1	0.010	0.020	**0.008**	21.669	0.620
30000	3	**0.000**	0.028	0.012	70.812	0.036
30000	10	**0.000**	0.020	0.016	182.147	0.036
30000	30	**0.070**	0.236	1.084	>300.000	2.232
30000	100	**0.100**	1.056	1.340	>300.000	>300.000
30000	300	**3.630**	32.818	34.698	>300.000	>300.000
30000	1000	**12.320**	57.924	129.540	>300.000	>300.000
30000	3000	11.820	**2.436**	130.156	>300.000	>300.000

Table 6.4: Hard UKP Instances.

n	w_{min}	myEDUK	EDUK2	EDUK	DP	B&B
10000	1000	0.030	**0.012**	0.272	>300.000	0.016
15000	1000	0.030	**0.008**	0.280	>300.000	0.012
20000	1000	0.030	0.020	0.304	>300.000	**0.012**
5000	1000	0.020	**0.000**	0.268	>300.000	0.020
10000	5000	0.320	**0.012**	6.412	>300.000	0.192
15000	5000	0.380	**0.004**	7.440	>300.000	0.220
20000	5000	0.410	**0.008**	8.905	>300.000	0.232
5000	5000	0.200	**0.004**	3.988	>300.000	0.200
10000	10000	0.800	**0.008**	15.569	>300.000	**0.008**
15000	10000	1.050	**0.008**	21.697	>300.000	**0.008**
20000	10000	1.250	0.012	26.302	>300.000	**0.008**
5000	10000	0.450	**0.004**	7.672	>300.000	0.008
10000	20000	1.840	**0.008**	31.510	>300.000	0.564
15000	20000	2.550	**0.008**	49.963	>300.000	0.984
20000	20000	3.190	**0.012**	69.796	>300.000	1.416
5000	20000	1.000	**0.000**	14.181	>300.000	0.184

For those instances, DP aways took more than three hundred seconds to run. EDUK2 and B&B show a better performance for hard instances. It can be concluded that the EDUK algorithm is sensible to the amount of non-dominated items.

6.3 Cutting Stock Problem Tests

In this section, myEDUK is compared with DP and B&B when integrated in the column generation solver described in Section 5.2. The instances used have Discrete Uniform and Bounded Probability Sampled distributions. EDUK and EDUK2, the algorithms implemented by the original authors, were not tested in this section because the profits of the item types in the UKP instances generated by the column generation method have double precision variables. EDUK and EDUK2 only accept integer profits.

6.3.1 Discrete Uniform Distributions

Tables 6.5, 6.6, 6.7, and 6.8 show the execution time in seconds of Branch & Bound, Dynamic Programming, and myEDUK on four different sets of instances with discrete uniform distributions. The parameters used to generate the instances are indicated in the captions of the tables.

myEDUK is always faster than the previous dynamic programming implementation, and for larger instances it almost always beats Branch & Bound. The advantage of dynamic programming over B&B is that its running time is bounded by the size of the problem (the knapsack capacity and the number of items), while B&B is not bounded and has a hard to predict behavior.

Table 6.5: U{1, 600, 1000}.

n	B&B	DP	myEDUK
600	**0.13**	1.45	0.19
1897	**0.60**	6.28	0.72
6000	11.43	3.02	**0.35**
18974	**3.41**	8.14	3.56
60000	**4.48**	7.58	5.16
189737	**7.61**	13.10	8.65
600000	**4.91**	10.28	5.20

Table 6.6: U{1, 6500, 10000}.

n	B&B	DP	myEDUK
6000	**0.02**	**0.02**	**0.02**
18974	427.16	915.02	**353.56**
60000	109.95	275.35	**109.60**
189737	157.69	274.77	**152.76**
600000	**41.49**	121.28	52.04

Table 6.7: U{1, 15000, 28000}.

n	B&B	DP	myEDUK
6000	**0.02**	**0.02**	**0.02**
18970	2550.91	7345.78	**317.85**
60000	**0.25**	**0.25**	**0.25**
189740	**11.31**	11.52	11.87
600000	4.66	4.71	**4.39**
1897370	65.48	545.64	**59.95**
6000000	18.99	337.72	**18.47**

Table 6.8: U{200, 600, 1000}.

n	B&B	DP	myEDUK
600	**0.20**	0.40	0.25
1897	0.17	0.26	**0.15**
6000	0.72	0.34	**0.20**
18974	0.73	0.51	**0.30**
60000	0.56	0.37	**0.24**
189737	0.59	0.44	**0.25**
600000	0.79	0.43	**0.28**

6.3.2 Bounded Probability Sampled Distributions

Tables 6.9, 6.10, and 6.11 show the running time of the implementations on cutting stock instances generated with distributions indicated in the label under the tables. The instances are identified by the parameter n, the expected number of items, and the mean of the demand for each size.

Table 6.9: BS{1, 6000, 10000, 400}.

n	demand mean	B&B	DP	myEDUK
4000	9.51	**0.39**	2.83	0.59
12640	31.13	**1.45**	6.12	2.43
40000	99.49	**2.64**	9.05	4.72
126490	315.70	**2.24**	8.28	3.88
400000	999.50	**2.98**	9.26	4.90
1264910	3161.77	**3.28**	9.93	5.65
4000000	9999.50	**3.39**	10.61	5.30

Table 6.10: BS{1, 12000, 20000, 800}.

n	demand mean	B&B	DP	myEDUK
40000	12.83	**222.67**	1225.94	269.87
126400	41.63	**806.64**	2281.59	955.71
400000	132.84	**1245.05**	2906.54	1489.09
1264900	421.13	**1692.85**	3791.66	1926.33
4000000	1332.83	**2059.27**	4220.45	2285.34
12649100	4215.86	**2076.06**	4307.71	2410.98
40000000	13332.84	**2309.85**	4460.83	2645.32

Table 6.11: BS{2000, 6000, 10000, 2000}.

n	demand mean	B&B	DP	myEDUK
400	1.000	**12.96**	98.67	15.35
1264	1.000	**15.60**	116.28	18.87
4000	1.690	11.55	77.63	**11.13**
12649	5.843	333.28	93.32	**33.89**
40000	19.499	554.70	163.54	**97.35**
126491	62.748	692.47	162.60	**98.28**
400000	199.493	601.89	170.43	**108.25**

For this set of instances, myEDUK is again better than the previous DP implementation for all test cases. For instances with $w_{min} = 1$, B&B is always faster than myEDUK. For the instances with distribution BS{2000, 6000, 10000, 2000}, myEDUK performs

better than B&B. Again, when w_{min} is large, 2000 in that instance, there are less non-dominated items. It seems to be the case that myEduk is less sensible to the number of non-dominated items for instances with small capacity.

6.4 Linear Programming time versus UKP

The running time of the column generation method is divided between solving the master problem, with CPLEX, and solving the pricing problem, with the UKP algorithm. An important aspect to be known is whether the linear programming solver, CPLEX, occupies most of the computation time. If this is the case the optimization focus should be the linear programming solver.

The tables 6.12, 6.13, 6.14, and 6.15 show the percentage of the total time spent on CPLEX and on the knapsack solver for discrete uniform distributed instances, and 6.16 and 6.17, for bounded probability sampled distribution. The first column **total** is the total running time in seconds, **CPLEX** and **myEDUK** are, respectively, the percentage of the running time spent solving the Linear Program and the Knapsack Problem. Note that there still time that is not used neither by myEDUK nor CPLEX.

Table 6.12: Linear Programming time versus UKP: U{1, 600, 1000}.

n	total	CPLEX (%)	myEDUK (%)
600	0.18	**40.00**	46.60
1897	0.71	50.80	**36.70**
6000	0.29	61.60	**27.30**
18974	3.19	89.40	**8.50**
60000	4.22	94.10	**4.10**
189737	7.70	95.10	**3.90**
600000	5.60	93.70	**4.50**

Table 6.13: Linear Programming time versus UKP: U{1, 6500, 10000}.

n	total	CPLEX (%)	myEDUK (%)
18974	310.10	95.90	**3.20**
60000	91.47	96.60	**2.30**
189737	143.46	98.00	**1.20**
600000	45.82	95.90	**2.40**

It can be seen that a long of time is spent solving the linear programming problem. For the larger instances tested almost always more than 80%. It is still worth, though, to optimize the UKP solution since for some instances the time spent solving it is around 50%.

Table 6.14: Linear Programming time versus UKP: U{1, 15000, 28000}.

n	total	CPLEX (%)	myEDUK (%)
18970	283.50	83.70	**13.60**
1897370	55.68	92.30	**3.60**
6000000	17.54	85.00	**8.00**

Table 6.15: Linear Programming time versus UKP: U{200, 600, 1000}.

n	total	CPLEX (%)	myEDUK (%)
600	0.29	50.00	**44.40**
1897	0.17	66.60	**26.10**
6000	0.23	50.00	**41.30**
18974	0.35	60.20	**34.00**
60000	0.27	**44.10**	48.50
189737	0.29	**44.40**	45.80
600000	0.34	49.40	**44.70**

Table 6.16: Linear Programming time versus UKP: BS{1, 6000, 10000, 400}.

n	total	CPLEX (%)	myEDUK (%)
4000	0.70	**42.60**	52.80
12640	2.50	50.70	**47.30**
40000	4.90	60.10	**38.50**
126490	3.85	59.40	**38.40**
400000	5.11	54.50	**43.90**
1264910	5.54	56.60	**41.50**
4000000	5.52	57.00	**41.80**

Table 6.17: Linear Programming time versus UKP: BS{1, 12000, 20000, 800}.

n	total	CPLEX (%)	myEDUK (%)
40000	294.09	84.40	**15.20**
126400	992.81	85.80	**14.00**
400000	1476.24	86.60	**13.20**
1264900	2100.45	88.50	**11.40**
4000000	2284.21	88.50	**11.40**
12649100	2347.37	87.80	**12.10**
40000000	2736.76	88.20	**11.60**

7 CONCLUSION AND FUTURE WORK

In this report an efficient dynamic programming algorithm for the Unbounded Knapsack problem was described and tested for different instances. This algorithm was implemented in an imperative programming language and it was experimentally compared with other implementations. The parameters of the algorithm were experimentally tested.

This work shows an efficient iterative method to detect threshold dominance. To the best of our knowledge this approach was never described in the literature.

Different methods of generating UKP and CSP instances were explained and used to generate results.

The implementation provided in this work shows an improvement over the original EDUK implementation over all instances tested, and a reasonable performance compared with the state-of-the-art EDUK2. When integrated in the Cutting Stock solver provided by [7], the proposed implementation runs faster than the former dynamic programming implementation over all instances. For large instances it outperform the Branch & Bound implementation.

Future research directions include:

- To take advantage of the the similarity between UKP instances generated in sequence by the column generation method, detected in Section 4.1.3. As the instances are not always very similar, it is not clear if this approach could achieve an improvement on the overall performance of the method. One possible way of using the similarity between consecutive instances is to store dominance relations and use them to remove items.

- Implementation of sparsity (see Section 2.4.3.2) and its comparison with the non sparse version. The results of this work show that the implementation of EDUK provided by this work shows a better performance on the instances tested, even though it does not implement sparsity. However it is not clear whether this phenomenon is a product of the programming language used, as functional programming languages are allegedly slower, or if the cost of applying sparsity is indeed too expensive on instances with large capacities.

- The Column Generation method used is an approximation of the integer CSP. A possible research direction is the integration of the implemented algorithm in a Branch and Prince algorithm in order to solve the problem optimally.

REFERENCES

[1] R. Andonov, V. Poirriez, and S. Rajopadhye. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2):394 – 407, 2000.

[2] Rumen Andonov and Sanjay Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *In International Conference on Application-Specific Array Processors (ASAP-94*, pages 302–313. IEEE, 1994.

[3] Brenda S Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6(1):49 – 70, 1985.

[4] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res*, 1998.

[5] A. Victor Cabot. An enumeration algorithm for knapsack problems. *Operations Research*, 18(2):06–311, 1970.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[7] David L. Applegate, Luciana S. Buriol, Bernard L. Dillard, David S. Johnson and Peter W. Shor. The cutting-stock approach to bin packing: Theory and experiments. In *Proceedings of ALENEX 2003*, pages p. 1–15, 2003.

[8] J.M. Valerio de Carvalho. Lp models for bin packing and cutting stock problems. *European Journal of Operational Research*, 141(2):253 – 273, 2002.

[9] Zeger Degraeve and Marc Peeters. Optimal integer solutions to industrial cutting-stock problems: Part 2, benchmark results. *INFORMS J. on Computing*, 15(1):58–81, January 2003.

[10] R. Garfinkel and G.L. Nemhauser. *Integer programming*. Series in decision and control. Wiley, 1972.

[11] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

[12] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem – part II. *Operations Research*, 11:863, 1963.

[13] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14(6):1045–1074, 1966.

[14] Harold Greenberg. An algorithm for the periodic solutions in the knapsack problem. *Journal of Mathematical Analysis and Applications*, 111(2):327 – 331, 1985.

[15] T.C. Hu. *Integer programming and network flows*. Addison-Wesley Pub. Co., 1969.

[16] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[17] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[18] Eugene L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

[19] Marco Lubbecke and Jacques Desrosiers. Selected topics in column generation. *Operations Research*, 53:1007–1023, 2004.

[20] G.S Lueker. Two NP-complete problems in nonnegative integer programming. *Report No. 178, Computer Science Laboratory, Princeton University, Princeton, NJ.*, 1975.

[21] Silvano Martello and Paolo Toth. An exact algorithm for large unbounded knapsack problems. *Oper. Res. Lett.*, 9(1):15–20, January 1990.

[22] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[23] Kazuki Matsumoto, Shunji Umetani, and Hiroshi Nagamochi. On the one-dimensional stock cutting problem in the paper tube industry. *Journal of Scheduling*, 14:281–290, 2011.

[24] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1995. Technical Report 95-1.

[25] Vincent Poirriez, Nicola Yanev, and Rumen Andonov. A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization*, 6(1):110 – 124, 2009.

[26] G.Terry Ross and RichardM. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming*, 8:91–103, 1975.

[27] P.E. Sweeney and E.R. Paternoster. Cutting and packing problems: an updated literature review. *Working Paper*, (654), 1991.

[28] Gerhard Wäscher and Thomas Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *Operations-Research-Spektrum*, 18:131–144, 1996.

Table 7.1: Detailed Parameters Tests on Large Realistic Random Instances.

n	w_{wmin}	1%/1	5%/1	10%/1	20%/1	25%/1	30%/1	1%/1.5	5%/1.5	10%/1.5	20%/1.5	25%/1.5	30%/1.5	1%/2	5%/2	10%/2	20%/2	25%/2	30%/2
5000	1	**0.000**	0.01	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	**0.000**
5000	3	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	**0.000**	0.01	**0.000**	0.01	0.02	0.01	0.01
5000	10	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	0.01	0.01	0.01	**0.000**
5000	30	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
5000	100	0.01	0.01	0.01	**0.000**	0.01	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.02
5000	300	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01
5000	1000	0.25	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24	**0.230**	0.24	0.27	0.27	0.26	0.29	0.27	0.27
5000	3000	2.7	2.68	2.68	2.69	2.7	**2.660**	**2.660**	2.67	2.68	2.68	2.68	2.7	4.04	4.04	4.05	4.05	4.02	4.04
10000	1	0.01	0.01	**0.000**	0.01	**0.000**	0.01	**0.000**	0.01	0.01	**0.000**	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	3	0.01	0.01	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	**0.000**	**0.000**	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01
10000	10	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.000**	**0.000**	0.01	**0.000**	**0.000**	0.02	0.01	0.01	0.01	0.01	0.01
10000	30	**0.000**	**0.000**	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01
10000	100	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	**0.060**	0.1	0.08	0.08	0.09	0.08	0.09
10000	300	4.24	4.29	4.26	4.22	4.22	4.2	4.17	4.18	4.18	4.19	4.14	4.16	6.57	6.56	6.58	6.57	6.52	6.49
10000	1000	4.83	4.82	4.81	4.81	4.81	4.78	4.81	4.83	4.85	4.81	4.81	4.8	7.42	7.42	7.41	7.43	7.4	7.43
10000	3000	5.59	5.56	5.53	5.54	5.62	5.61	5.63	5.61	5.66	5.63	5.6	5.65	8.54	8.57	8.56	8.6	8.5	8.55
15000	1	0.01	**0.000**	0.01	**0.000**	0.01	**0.000**	**0.000**	**0.000**	0.01	**0.000**	0.01	0.01	0.01	**0.000**	0.01	0.01	**0.000**	**0.000**
15000	3	**0.000**	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	**0.000**	**0.000**	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01
15000	10	0.01	0.01	**0.000**	0.01	0.01	**0.000**	**0.000**	0.01	0.01	**0.000**	**0.000**	0.01	0.02	0.01	0.01	0.02	0.02	**0.000**
15000	30	0.02	0.01	0.01	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.02	**0.000**	0.01	0.02	**0.000**
15000	100	0.01	0.01	0.01	**0.000**	**0.000**	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.02	0.01	0.02
15000	300	0.89	0.84	0.85	**0.830**	0.88	0.86	0.85	0.86	0.85	0.86	0.87	0.87	0.97	0.97	0.95	0.97	0.97	0.96
15000	1000	7.23	6.97	7.07	7.08	7.09	7.07	7.08	7.11	6.9	6.86	6.86	6.83	10.83	10.85	10.77	10.8	10.77	10.79
15000	3000	6.65	6.43	6.62	6.42	6.64	6.46	6.66	6.39	6.66	6.44	6.65	6.47	10.4	10.03	10.38	10.14	10.4	10.14
20000	1	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.01	**0.000**	0.01	0.01	0.02	**0.000**	0.01	0.01
20000	3	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.01
20000	10	**0.010**	0.02	**0.010**	**0.010**	0.02	0.02	0.02	0.02	**0.010**	**0.010**	**0.010**	**0.010**	0.03	0.02	0.02	**0.010**	0.02	0.02
20000	30	0.02	0.01	0.01	0.01	0.02	0.01	0.01	0.02	**0.000**	0.01	0.02	0.01	0.02	0.01	0.01	0.02	0.01	0.01
20000	100	0.03	0.03	0.03	0.03	0.03	**0.020**	0.03	0.03	0.03	0.03	0.04	**0.020**	0.03	0.03	**0.020**	0.03	**0.020**	0.03
20000	300	6.05	6.05	5.99	6.01	5.99	5.99	6	6.03	5.99	5.99	6.03	5.99	9.48	9.28	9.68	9.37	9.59	9.38
20000	1000	5.95	5.98	5.93	5.92	5.98	5.99	5.97	5.98	5.96	5.97	5.94	6	9.64	9.62	9.63	9.69	9.64	9.67
20000	3000	8.76	8.87	8.86	8.85	8.9	8.86	8.82	8.83	8.83	8.84	8.77	8.75	14.05	14.08	14.04	14.15	14.11	14.19
25000	1	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.000**
25000	3	**0.000**	0.01	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.02	0.02	0.02	0.02	0.02	0.01
25000	10	**0.000**	0.01	**0.000**	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.000**	0.01	0.01	0.01	0.01
25000	30	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.000**	0.01	**0.000**	0.01	0.01	0.02	0.01	**0.000**
25000	100	**0.010**	**0.010**	0.02	**0.010**	0.02	**0.010**	**0.010**	**0.010**	**0.010**	**0.010**	**0.010**	**0.010**	0.02	**0.010**	**0.010**	0.03	**0.010**	**0.010**
25000	300	0.09	**0.080**	**0.080**	**0.080**	0.09	**0.080**	0.09	0.09	0.09	**0.080**	0.09	0.09	0.09	0.1	**0.080**	**0.080**	0.09	0.09
25000	1000	3.77	3.81	3.75	3.75	3.81	3.78	3.83	**3.730**	3.84	3.81	3.81	3.86	5.68	5.57	5.59	5.58	5.75	5.64
25000	3000	10.33	**10.280**	10.29	10.31	10.32	**10.280**	**10.280**	**10.280**	10.31	10.3	10.31	**10.280**	15.79	15.74	15.81	15.76	15.75	15.76
30000	1	0.01	0.01	0.01	0.01	**0.000**	0.01	0.01	**0.000**	0.01	**0.000**	**0.000**	0.01	0.02	0.01	0.01	0.01	0.01	**0.000**
30000	3	0.01	**0.000**	**0.000**	0.01	**0.000**	0.01	0.01	**0.000**	**0.000**	0.01	**0.000**	0.01	**0.000**	0.01	0.01	0.01	**0.000**	0.01	0.01
30000	10	0.01	0.01	0.01	0.01	0.01	0.01	**0.000**	**0.000**	**0.000**	0.02	0.01	0.01	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	0.01
30000	30	0.08	**0.060**	0.07	0.09	0.08	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.08	0.08	0.07	0.08	0.08	0.09
30000	100	0.1	0.1	0.1	**0.090**	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.11	0.11	0.11	0.11	0.11	0.12
30000	300	3.62	3.61	3.6	3.61	3.62	3.64	3.63	3.61	3.61	**3.590**	3.61	3.62	4.16	4.16	4.18	4.13	4.15	4.15
30000	1000	12.37	**12.310**	12.35	12.32	12.39	**12.310**	12.33	12.32	12.36	**12.310**	12.34	12.35	19.06	19.02	19.03	19.02	19.07	19.02
30000	3000	11.83	11.81	11.83	11.82	11.82	**11.790**	11.82	11.81	11.86	11.82	11.82	11.8	18.65	18.63	18.64	18.61	18.65	18.6