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

HENRIQUE BECKER

The Unbounded Knapsack Problem: a critical review

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Luciana Buriol

Porto Alegre
March 2017

CIP — CATALOGING-IN-PUBLICATION

Becker, Henrique

The Unbounded Knapsack Problem: a critical review / Henrique Becker. – Porto Alegre: PPGC da UFRGS, 2017.

108 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Luciana Buriol.

1. Unbounded knapsack problem. 2. Dynamic programming. 3. Optimization. 4. Cutting stock problem. I. Buriol, Luciana. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"It isn't just you. It's the whole Galaxy.

Pirenne heard Lord Dorwin's idea of scientific research.

Lord Dorwin thought the way to be a good archaeologist was to read all the books on the subject – written by men who were dead for centuries. He thought that the way to solve archaeological puzzles was to weigh the opposing authorities.

And Pirenne listened and made no objections.

Don't you see that there's something wrong with that?"

— THE FOUNDATION, ISAAC ASIMOV

DEDICATION AND ACKNOWLEDGMENTS

To my parents for their unending patience, and for completely supporting my choices, while offering criticism.

To my advisor funnily for the exact same reasons, while in a little more specific context.

To my hometown friends for helping me recover my energy (and sanity) in the weekends, and understanding when that was not possible.

To my brother for the unfiltered discussions and exchange of ideas, and also for making me laugh.

To my college teachers for always seeing in me a future colleague of the profession.

To my friend Didi for accepting, on short notice, the job of revising my terrible English.

To everyone in the lab, for accepting me (and my stuffed ponies), and being nothing less than welcoming and helpful.

To my friend Tadeu for giving me some perspective in unrelated matters and inviting me to play tabletop games.

To Dennis (43) who would give me a ride back home everyday, while trying to convince me to play DOTA2.

ABSTRACT

A review of the algorithms and datasets in the literature of the Unbounded Knapsack Problem (UKP) is presented in this master's thesis. The algorithms and datasets used are briefly described in this work to provide the reader with basis for understanding the discussions. Some well-known UKP-specific properties, such as dominance and periodicity, are described. The UKP is also superficially studied in the context of pricing problems generated by the column generation approach applied to the continuous relaxation of the Bin Packing Problem (BPP) and Cutting Stock Problem (CSP). Multiple computational experiments and comparisons are performed. For the most recent artificial datasets in the literature, a simple dynamic programming algorithm, and its variant, seems to outperform the remaining algorithms, including the previous state-of-the-art algorithm. The way dominance is applied by these dynamic programming algorithms has some implications for the dominance relations previously studied in the literature. In this master's thesis we defend that choosing sets of artificial instances has defined what was considered the best algorithm in previous works. We made available all codes and datasets referenced in this master's thesis.

Keywords: Unbounded knapsack problem. dynamic programming. optimization. cutting stock problem.

O problema da mochila com repetições: uma revisão crítica

RESUMO

Uma revisão dos algoritmos e conjuntos de instâncias presentes na literatura do Problema da Mochila com Repetições (PMR) é apresentada nessa dissertação de mestrado. Os algoritmos e conjuntos de instâncias usados são brevemente descritos nesse trabalho, a fim de que o leitor tenha base para entender as discussões. Algumas propriedades bem conhecidas e específicas do PMR, como a dominância e a periodicidade, são explicadas com detalhes. O PMR é também superficialmente estudado no contexto de problemas de avaliação gerados pela abordagem de geração de colunas aplicada na relaxação contínua do *Bin Packing Problem (BPP)* e o *Cutting Stock Problem (CSP)*. Múltiplos experimentos computacionais e comparações são realizadas. Para os conjuntos de instâncias artificiais mais recentes da literatura, um simples algoritmo de programação dinâmica, e uma variante do mesmo, parecem superar o desempenho do resto dos algoritmos, incluindo aquele que era estado-da-arte. O modo que relações de dominância é aplicado por esses algoritmos de programação dinâmica têm algumas implicações para as relações de dominância previamente estudadas na literatura. O autor dessa dissertação defende a tese de que a escolha dos conjuntos de instâncias artificiais definiu o que foi considerado o melhor algoritmo nos trabalhos anteriores. O autor dessa dissertação disponibilizou publicamente todos os códigos e conjuntos de instâncias referenciados nesse trabalho.

Palavras-chave: problema da mochila com repetições, programação dinâmica, otimização, cutting stock problem.

LIST OF ABBREVIATIONS AND ACRONYMS

B&B	Branch and Bound
BKP	Bounded Knapsack Problem
BPP	Bin Packing Problem
CA	Consistency Approach
CPU	Central Processing Unit
CSP	Cutting Stock Problem
DP	Dynamic Programming
FP	Floating Point
KP	Knapsack Problem
PRNG	Pseudo-Random Number Generator
SCF	Set Covering Formulation
SD	Standard Deviation
UKP	Unbounded Knapsack Problem

LIST OF FIGURES

Figure 1.1 Classical graphic representation of the simple, multiple and threshold dominances. Source: (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000). It is interesting to note that the triangle slope equals to the items efficiency, and that all points in the shaded area created by an item or solution triangle are dominated. 15
Figure 1.2 Classical graphic representation of the threshold dominance. Source: (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000). The t_i is the $t$hreshold of item i, i.e. the weight of the smallest solution composed only of copies of i that is dominated by another item/solution. 15

Figure 3.1 An uncorrelated instance generated with $n = 100$, $w_{min} = 1$, $w_{min} = 1000$, $p_{min} = 1$, and $p_{max} = 1000$. 25
Figure 3.2 A 128-16 BREQ Standard instance with $n = 2048$ 37

Figure 4.1 The solutions $\{1, 2\}$ and $\{2, 1\}$ (with $w_1 = 3$ and $w_2 = 4$) are equivalent. The solution $\{3, 2, 1\}$ can be also constructed in many ways (with $w_1 = 4$, $w_2 = 5$ and $w_3 = 6$). The figure shows the solution generation with and without symmetry pruning. 47

Figure 5.1 Benchmark using PYAsUKP dataset; the UKP5, GREENDP and EDUK2 algorithms; and no timeout. 62
Figure 5.2 Comparison between MTU1 and MTU2, C++ and Fortran implementations. 66
Figure 5.3 Benchmark with the 128-16 Standard BREQ instances. 68
Figure 5.4 Total time used solving the continuous relaxation of the BPP/CSP. 72
Figure 5.5 Total time used solving pricing subproblems (UKP) in the continuous relaxation of the BPP/CSP. 73
Figure 5.6 Percentage of time taken by solving pricing subproblems (UKP) in the continuous relaxation of the BPP/CSP. 74
Figure 5.7 Total time used solving the master problem in the continuous relaxation of the BPP/CSP. 75
Figure 5.8 Quantity of pricing subproblems each method solved, relative to the method that solved the greatest amount for the same instance (in %). 76
Figure 5.9 For the four UKP5 variants, the relative difference between the number of pricing subproblems solved in the same instance. 79
Figure 5.10 Comparison of the mean times of UKP5 and EDUK2 when executed in two diferent computers, in parallel and in serial mode. 83
Figure 5.11 Parallel and Serial Runs Standard Deviation 85

CONTENTS

1 INTRODUCTION ... 11
1.1 Motivation and scope ... 11
1.2 Formulation and notation ... 12
1.3 Properties of the UKP ... 13
1.3.1 Dominance relations ... 13
1.3.2 Periodicity and periodicity bounds ... 17
2 PRIOR WORK ... 20
3 THE DATASETS ... 25
3.1 Uncorrelated items distribution datasets ... 25
3.1.1 Babayev's use of uncorrelated instances ... 26
3.1.2 Martello's use of uncorrelated instances ... 27
3.2 PYAsUKP dataset ... 28
3.2.1 Subset-sum instances ... 29
3.2.2 Instances with strong correlation between weight and profit ... 29
3.2.3 Instances with postponed periodicity ... 30
3.2.4 Instances without collective dominance ... 31
3.2.5 SAW instances ... 31
3.2.6 PYAsUKP dataset and reduced PYAsUKP dataset ... 32
3.3 CSP pricing subproblem dataset ... 32
3.4 Bottom Right Ellipse Quadrant instances ... 36
3.5 Other distributions ... 40
4 APPROACHES AND ALGORITHMS ... 41
4.1 Conversion to other KP variants ... 41
4.2 Dynamic Programming ... 42
4.2.1 The naïve algorithm ... 42
4.2.2 The algorithm of Garfinkel and Nemhauser ... 43
4.2.3 The step-off algorithms of Gilmore and Gomory ... 44
4.2.4 UKP5 ... 45
4.2.4.1 A note about UKP5 performance ... 48
4.2.4.2 Weak solution dominance ... 48
4.2.4.3 Implementation details ... 48
4.2.5 EDUK ... 49
4.3 Branch-and-Bound ... 51
4.3.1 MTU1 ... 53
4.3.2 MTU2 ... 54
4.3.3 Other B&B algorithms ... 55
4.4 Hybrid (DP and B&B) ... 56
4.4.1 GREENDP ... 56
4.4.2 EDUK2 ... 57
4.5 Consistency Approach ... 58
4.5.1 GREENDP1 ... 58
4.5.2 Babayev's algorithm ... 59
4.6 Other approaches ... 60
5 EXPERIMENTS AND ANALYSES ... 61
5.1 Setup of the first four experiments ... 61
5.2 Solving the PYAsUKP dataset ... 62
5.2.1 MTU1 and MTU2 (C++ and Fortran) ... 65
5.2.2 Algorithms implemented but not used ... 67

5.3 Solving the BREQ 128-16 Standard Benchmark 67
5.4 Solving pricing subproblems from BPP/CSP 70
5.4.1 The differences in the number of pricing subproblems solved 76
5.4.2 The only outlier 77
5.4.3 Similar methods generate different amounts of pricing subproblems 78
5.4.4 Algorithms not used in this experiment 80
5.5 The effects of parallel execution 81
5.5.1 Setup 81
5.5.2 Experiment 82
6 CONCLUSIONS AND FUTURE WORK 86
6.1 Conclusions 86
6.1.1 A critical review 86
6.2 UKP-specific knowledge contributions 88
6.3 Technological UKP-specific contributions 91
6.4 Future work 91
REFERENCES 93
APPENDIX A — TABLES 95
A.1 Data and code related to CSP pricing subproblem dataset 100
A.2 Capítulo de resumo em português 100
A.2.1 Introdução 100
A.2.2 Trabalhos relacionados 101
A.2.3 Classes de instâncias 102
A.2.3.1 Instâncias do PYAsUKP 102
A.2.3.2 Problemas de avaliação gerados a partir do BPP/CSP 103
A.2.3.3 Instâncias BREQ 104
A.2.4 Abordagens e algoritmos 105
A.2.5 Experimentos e análises 106
A.2.6 Conclusões 107

1 INTRODUCTION

The Unbounded Knapsack Problem (UKP) is a simpler variant of the well-known Bounded Knapsack Problem (BKP) and the 0-1 Knapsack Problem (0-1 KP). The only difference between the UKP and these other KP variants is that the UKP does not impose a bound on the available quantity of each item type. The UKP can also be seen as a special case of the BKP in which, for each item type, there are more copies available than is possible to fit in the knapsack capacity.

The UKP is NP-Hard and, thus, has no known polynomial-time algorithm for solving it. Nevertheless, the UKP can be solved in pseudo-polynomial time by dynamic programming algorithms.

1.1 Motivation and scope

The applied use of the UKP discussed in this thesis is: the UKP as the pricing subproblem generated by solving the continuous relaxation of the set covering formulation for the unidimensional Bin Packing Problem (BPP) and Cutting Stock Problem (CSP) using the column generation approach. The BPP and the CSP are classical problems in the area of operations research and of great importance for the industry, see (DELORME; IORI; MARTELLO, 2014) and (GILMORE; GOMORY, 1961; GILMORE; GOMORY, 1963). The best lower bounds known for the optimal solution value of the BPP and the CSP are the optimal solution value for their continuous relaxation. The tightest formulation for the BPP and the CSP has an exponential number of columns and, because of this, is solved by using the column generation approach (GILMORE; GOMORY, 1961). The UKP is the pricing subproblem of this column generation approach.

The author of this thesis is aware of the existence of many very good heuristics and approximations for solving the UKP, including the existence of fully polynomial time approximation schemes (FPTAS) for the UKP. This thesis, however, will only discuss exact methods. This was done in order to limit scope of this thesis and also because the pricing subproblem instances have to be solved exactly to guarantee that the relaxation is being solved exactly (which in turn guarantee that a branch-and-bound algorithm using the relaxation as a lower bound solves the original BPP/CSP instance exactly).

1.2 Formulation and notation

The notation presented in this section will be used in the rest of this work. An instance of the UKP is a pair of a capacity c and a list of n items. Each item can be referenced by its index in the item list $i \in \{1 \dots n\}$. Each item i has a weight value w_i, and a profit value p_i. A solution is an item multiset (i.e, a set that allows multiple copies of the same element). The sum of the items weight, or profit value, of a solution s is denoted by w_s, or p_s; and will be referred to as the weight, or profit, of the solution. A solution s is a valid solution iff $w_s \leq c$. An optimal solution s^* is a valid solution with the greatest profit among all valid solutions.

To solve an instance of the UKP is to find an optimal solution for that instance. Finding all optimal solutions for an instance of the UKP is not the focus of this work.

The mathematical formulation of UKP is:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i \tag{1.1}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq c, \tag{1.2}$$

$$x_i \in \mathbb{N}_0. \tag{1.3}$$

The quantities of each item i in an optimal solution are denoted by x_i, and are restricted to the non-negative integers, as (1.3) indicates. We assume that the capacity c, the quantity of items n and the weights of the items w_i are positive integers. The profit values of the items p_i are positive real numbers.

The terms *item* and *item type* mean two different things in this work. The term *item* will be used to refer to an specific item that has a position in the items list of an instance, and that can have duplicates in the same instance. The term *item type* will be used to refer to a pair of weight and profit value that can be shared by many items.

The efficiency of an item i is its profit-to-weight ratio ($\frac{p_i}{w_i}$), and is denoted by e_i. We use w_{min}, or w_{max}, to denote the smallest items weight, or the largest items weight, within an instance of the UKP. We refer to the item with greatest efficiency among all items of an specific instance as the *best item* (or simply b); if

more than one item shares the greatest efficiency, then the item with the lowest weight among them will be considered the best item type; if more than an item has both previously stated characteristics, then the first item with both characteristics in the items list is the best item.

The attentive reader will note that an UKP instance is defined by a list of items, instead of a set of items, as usual. Some algorithms sort the items and manipulate their indices. So having notation to refer to the items indices is convenient. Also, a set would not allow for identical items (i.e. that share the same item type). Such duplicated items exist in some instance datasets of the literature, and can affect the solving time of the algorithms.

1.3 Properties of the UKP

This section presents properties of the UKP that can be exploited to speed-up its solving. The main point of these properties is that they are only valid if we have available as many copies of each item type as we could need. Consequently, those properties are always valid for the UKP and generally not valid for other knapsack variants. Nevertheless, if every item type i of an instance of the BKP (or 0-1 KP) has at least $\lfloor \frac{c}{w_i} \rfloor$ copies available, then the instance can be solved as if it were an instance of the UKP.

1.3.1 Dominance relations

If one item i has the same or less weight than another item j ($w_i \leq w_j$), and i also has the same or more profit value than item j ($p_i \geq p_j$), then it is clear that if we replace j by i in any valid solution, the solution will remain valid (the weight of the solution can only remain the same or decrease), as for the profit value of the solution, it can only remain the same or increase. This relationship between i and j is called a dominance relation; more specifically, it is a case of simple dominance, in which i simple dominates j.

We will assume that i dominates j. There are only two possibilities: the profit value of i is greater than the profit value of j; or they are the same. If it is the former, then j cannot be part of an optimal solution. By definition, an optimal

solution is a valid solution with the greatest profit value; if j was part of an optimal solution, replacing j by i would give a valid solution with an even greater profit value (*reductio ad absurdum*). If it is the latter (i.e. i and j have the same profit value), j can be part of an optimal solution. However, we would get an optimal solution with the same or less weight, by replacing j by i in the optimal solution.

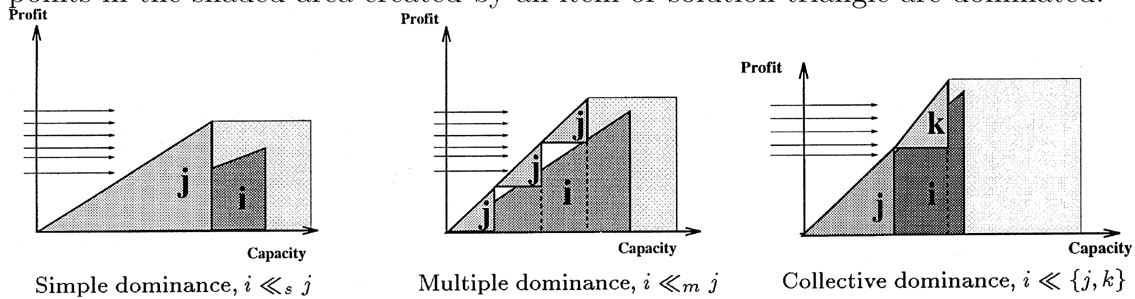
If we are interested in obtaining one optimal solution, but not all optimal solutions, we can use dominance relations to reduce the computational effort needed to find a solution. This is done by detecting dominance relations, and removing the dominated items, thus reducing the size of the problem. The detection can be done in a preprocessing phase or within the solving algorithm (reusing computation that would be needed anyway).

Dominance relations are not restricted to one single item simple dominating another single item. An item multiset (or solution[1]) can dominate a single item too. Given a solution s containing only two or more copies of an item i, and an item j, if $w_s \leq w_j$ and $p_s \geq p_j$, then it is said that item i multiple dominates item j. This dominance relation is called *multiple dominance*. Given a solution s composed of any items, and an item j, if $w_s \leq w_j$ and $p_s \geq p_j$, then it is said that solution s collective dominates item j. This dominance relation is called *collective dominance*. In multiple and collective dominance, whenever we would use item j in a solution, we can use the items that constitute s in place of j. Simple and multiple dominance are special cases of the collective dominance. Simple dominance can also be seen as a special case of multiple dominance, and some authors use 'multiple dominance' to refer to both simple and multiple dominances (POIRRIEZ; YANEV; ANDONOV, 2009).

Given a solution s composed of any items, and a solution t containing only two or more copies of an item j, if $w_s \leq w_t$ and $p_s \geq p_t$, then it is said that solution s threshold dominates item j. This dominance relation is called *threshold dominance*. If t is composed of n copies of item j, then we know that we can disregard solutions with n or more copies of item j (as each group of n copies of j can be replaced by s without loss to the value of the optimal solution). Collective dominance can be seen as a special case of threshold dominance, where $n = 1$ (i.e. the dominated solution t

[1] In this thesis, the term *solution* is used as an synonym for 'item multiset'. A solution does not need to be valid, or optimal (these qualifiers would be meaningless otherwise), and surely does not need to fill the knapsack capacity in a way that no more items can be added to it. As the UKP does not limit the quantity of each item type in a solution, we can treat any solution as a new item type, with weight and profit value equal to the weight and profit value of the solution.

Figure 1.1: Classical graphic representation of the simple, multiple and threshold dominances. Source: (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000). It is interesting to note that the triangle slope equals to the items efficiency, and that all points in the shaded area created by an item or solution triangle are dominated.



An object type is shown as a triangle of width w and height p. Its "shadow" is the region to its right and below its height. Left: i is in the shadow of j. Middle: i is in the combined shadow of $3j$. Right: i is in the combined shadow of $j + k$.

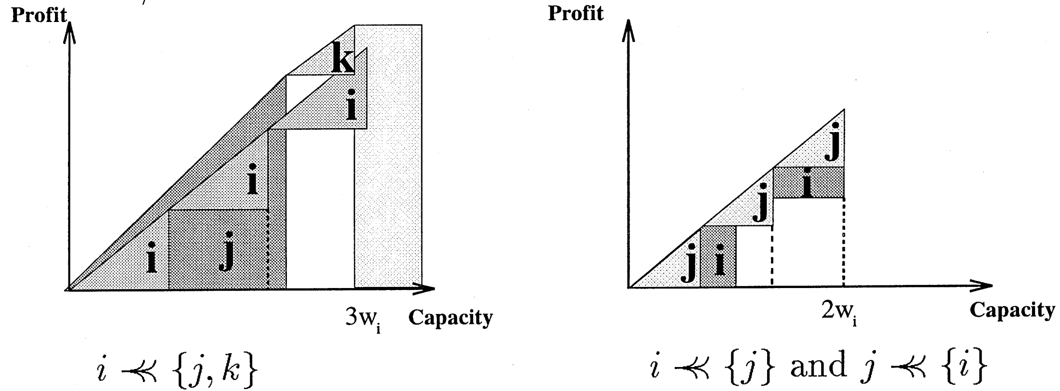
is a single item solution).

The simple and multiple dominances were deeply studied in the previous literature. Algorithms that remove all simple and multiple dominated items in $O(n^2)$, and heuristics with less complexity that do not guarantee removing all simple or multiple dominated items, were proposed, see (PISINGER, 1995) for a good review on this subject. On the other hand, the collective and threshold dominances seem too computationally expensive to be done in a preprocessing phase. However, in the context of a Dynamic Programming (DP) algorithm, in which the optimal solutions of lower capacities can be reused to detect both collective and threshold dominances (POIRRIEZ; YANEV; ANDONOV, 2009), these dominances are cheap to detect.

The Efficient Dynamic programming for the Unbounded Knapsack problem (EDUK) and the EDUK2 algorithms detect those four types of dominance relations (POIRRIEZ; YANEV; ANDONOV, 2009). In fact, threshold dominance was first proposed by the EDUK authors in (POIRRIEZ; ANDONOV, 1998), and was a primary feature of the EDUK algorithm. However, we will see that both algorithms seem to be dominated by algorithms that predate them by about forty years, as the *ordered step-off* algorithm from (GILMORE; GOMORY, 1966), and its 'improved version' from (GREENBERG; FELDMAN, 1980). These two old algorithms did not directly apply any of the four types of dominance.

After examination, it becomes clear that these two old algorithms indirectly apply all four dominances. By *indirectly*, we means that, in the course of these

Figure 1.2: Classical graphic representation of the threshold dominance. Source: (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000). The t_i is the *t*hreshold of item i, i.e. the weight of the smallest solution composed only of copies of i that is dominated by another item/solution.



Left: $t_i = 3w_i$ since 3 copies of i are in the combined shadow of j and k.
Right: equiprofitable object: $2w_i = 3w_j = t_i = t_j$.

algorithms execution, they eventually stop using dominated items to create new solutions, and that happens without testing items for each one of the four dominance relations previously explained.The approach used by these old algorithms is focused on solutions, not individual items. Consequently, it would be more adequate to say that they make use of some sort of *solution dominance*. Ideally, given a solution s and a solution t (both s and t can be composed of any items), if $w_s \leq w_t$ and $p_s \geq p_t$, then an optimized algorithm could not generate any solutions that are a superset of t (as the respective supersets of s would dominate them anyway) without loss to the value of the optimal solution.. Such dominance relation would generalize all previous dominances, as the dominated items can be seen as single item solutions (or multiple item solutions, in the case of threshold dominance).

Those old algorithms do not apply this ideal solution dominance, but a weaker version of it. This weaker version of solution dominance does not avoid generating every solution that is a superset from a dominated solution, but it can be implemented with almost no overhead. As this is algorithm-specific, it will be further discussed in Section 4.2.4.2.

In this section, the concept of dominance was introduced by explaining four dominance relations found in the literature and proposing the concept of strong solution dominance. In the literature, one of the definitions used for dominance is: “Given an instance of UKP, relative to item types set N, item type $k \in N$ is dominated if the optimal solution value does not change when k is removed

from N." (MARTELLO; TOTH, 1990b, p. 100). This definition is consistent with simple, multiple, and collective dominance, but not with threshold dominance. If an item is threshold dominated, it can still be present in all optimal solutions, it only cannot appear n or more times in all optimal solutions. Solution dominance is not covered by this definition, as it is item-centric. Finally, such broad definition of dominance does not give hints on how to design a procedure for removing the dominated items (without solving the instance), differently from the dominance relations.

1.3.2 Periodicity and periodicity bounds

The UKP has an interesting periodic property that can be stated the following way: for every set of item types, there exists a capacity y^+, for which every capacity y' ($y' > y^+$) will have an optimal solution that is an optimal solution for capacity $y' - w_b$ with one more copy of the best item added. In other words, after some capacity y^+, we can find optimal solutions simply by adding copies of the best item to optimal solutions of capacities y^+ and lower (all other items are not relevant anymore). In the literature, this periodic property is called *periodicity*. It should be clear that if we knew y^+ beforehand, and $y^+ < c$, then we can solve the UKP for the capacity $y^* = c - \lceil \frac{c-y^+}{w_b} \rceil w_b$ and fill the gap between y^* and c with exactly $\frac{c-y^*}{w_b}$ copies of the best item (instead of solving the UKP for capacity c).

The periodicity is a direct consequence of the threshold dominance. However, the periodicity was discovered much before the concept of threshold dominance. For instance, periodicity was already described in (GILMORE; GOMORY, 1966). As the concept of threshold dominance was already explained in this thesis, the author will use it to explain periodicity.

The threshold dominance property states that, if a solution s dominates solution t, and t contains only n copies of the same item type j, then solutions with n or more copies of j can be be replaced by solutions using s instead (without loss to the value of the optimal solution). The periodicity property states that after some capacity y^+ we can obtain optimal solutions only by adding copies of the best item to the optimal solutions from capacities y^+ or below (all other items are not relevant anymore). The link between these two properties is that *for a sufficiently large capacity, solutions composed of copies of the best item will threshold dominate*

solutions composed of copies of any other item.

First, we will verify the truth of the statement above. For any two positive integers a and b there will always exist at least one integer number that is divisible by both a and b (e.g. $a \times b$, or their *least common multiple*, $LCM(a, b)$). Therefore, for each non-best item j, there will exist a capacity value y that it is divisible by both w_b and w_j. Given $m_j = \frac{y}{w_b}$ and $n_j = \frac{y}{w_j}$, and by the definition of best item, we have that $m_j \times p_b \geq n_j \times p_j$. In other words, a solution composed only of m_j copies of the best item will threshold dominate a solution composed only of n_j copies of item j.

It is important to notice that, in the case of an instance in which all items have the same efficiency, the best item (that will have weight w_{min} by definition) will threshold dominate each other item at the capacity that is the *lowest common multiple* between their weights (as shown in the last paragraph). However, if the best item b is more efficient than the non-best item j (which is more common), then a smaller solution s composed only of copies of b can be more profitable than a bigger solution t composed only of copies of j. The weights of the two solutions do not have to be the same.

Now, we explain how periodicity is a direct consequence of threshold dominance. As we have seen, for each non-best item j, there will exist a positive integer n_j, in a way that solutions with n_j copies of j are dominated by solutions that use m_j copies of b instead. We will assume that there exists a solution u composed of $n_j - 1$ copies of each item j. If another copy of any item j is added to u, the resulting solution $u' = u \cup \{j\}$ could be replaced by another solution v that contains no copies of j, and m_j additional copies of b. Any solution that weight more than solution u has only two possibilities: it uses more copies of the best item; or it uses more than n_j copies of some non-best item j. In the last case, copies of j can be replaced by copies of b until the quantity of copies of j is smaller than n_j. Consequently, after the knapsack capacity $y'' = w_u$, adding copies of other items to a solution would be equal to adding copies of the best item (making any other item type except b irrelevant). Note that y'' is only an *upper bound* on y^+, the value of y^+ can be smaller than y'', as the items that the best item threshold dominate, can themselves threshold dominates other items.

It's worth mentioning that computing the exact value of y^+ is a very expensive process that equals to solving the UKP for all capacities y^+ and lower while checking

for threshold dominance. The author do not know any algorithm for solving the UKP that computes the exact value of y^+ before starting to solve the UKP. The algorithms compute an *upper bound* on the y^+ capacity value, as the one presented in the previous paragraph, that was presented in (KELLERER; PFERSCHY; PISINGER, 2004, p. 215).

An upper bound on y^+ is less valuable than y^+ itself, but it can be computed in a reasonable polynomial time, before starting the solving process. If one algorithm checks for threshold dominance periodically, it can stop when all non-best items have been threshold dominated by the best item. Such algorithm would not benefit much from computing an upper bound on the y capacity value. If this algorithm setup phase (e.g. allocating and initializing memory) is linear in the knapsack capacity c and the upper bound on the y capacity value is considerably smaller than c, then the algorithm could benefit from the upper bound.

There exist many proposed periodicity bounds, but some are time-consuming, as the $O(n^2)$ periodicity bound presented in (HUANG; TANG, 2012). Others depend on specific instance characteristics to be tight, as the ones presented in (IIDA, 2008) and (POIRRIEZ; YANEV; ANDONOV, 2009). For reasons that will be made clear in the conclusions, the author did not found relevant to present a review on periodicity bounds in this work. The UKP5 algorithm makes use of one simple periodicity bound, and it will be explained together with UKP5 in Section 4.2.4.3.

2 PRIOR WORK

It is important to note that the name “Unbounded Knapsack Problem” is more recent than the problem itself. To the best of our knowledge, this name was used for the first time in (MARTELLO; TOTH, 1990a). Earlier papers simply referred to ‘a’ or ‘the’ knapsack problem(s) the variant discussed was specified by the model presented in the paper. An earlier paper from the same author tackled both the UKP and the BKP (MARTELLO; TOTH, 1977) and called them, respectively, the General Unconstrained Knapsack Problem (GUKP) and General Constrained Knapsack Problem (GCKP). In the paper, the adjective unidimensional was also used to characterize both variants. More recently, the term ‘UKP’ seems to be well accepted. Also, unidimensional is considered the default, and the term multi-dimensional is used to differ from it. A researcher making a literature review about a specific variant of the knapsack problem should be aware of such caveat.

This literature review starts with (GILMORE; GOMORY, 1961), when the *column generation* approach was proposed. The main utility of the column generation approach was to avoid the existence of an exponential number of variables when solving the tightest linear programming model of BPP and CSP. The relationship between the UKP and the BPP/CSP was already briefly described at Section 1.1, and its technical details will be described at Section 3.3. The UKP is not solved, it is only said that “the auxiliary problem will be of the integer programming variety, but of such a special type (the ‘knapsack’ type) that it is solvable by several methods” (GILMORE; GOMORY, 1961, p. 2). Two years later, in (GILMORE; GOMORY, 1963), the authors proposed a specific algorithm for the UKP, and experiments solving BPP and CSP instances were executed. Some findings of this experiments will be discussed in Sections 3.3 and 5.4.

In (GILMORE; GOMORY, 1966), the one-dimensional and two-dimensional knapsack problems related to BPP and CSP were discussed. The author of this thesis reinvented one algorithm from (GILMORE; GOMORY, 1966) and published a paper about it, believing it was novel (BECKER; BURIOL, 2016), thus, he apologizes to the academic and scientific community for such disregard. Further information about the algorithms of (GILMORE; GOMORY, 1966) and (BECKER; BURIOL, 2016) can be found in Section 4.2. A small improvement over the algorithm of (GILMORE; GOMORY, 1966) was proposed in (GREENBERG; FELDMAN, 1980). The author

implemented the improved algorithm and its results can be seen in Section 5.2.

The papers (CABOT, 1970) and (SHAPIRO; WAGNER, 1967) were published shortly after. Both papers are behind a paywall and we did not have access to them. However, the algorithm from (CABOT, 1970) was compared indirectly by (MARTELLO; TOTH, 1977) (that will be discussed in the followig pages). The comparison showed that the algorithm from (CABOT, 1970) was dominated by the algorithm proposed in (MARTELLO; TOTH, 1977). In (GREENBERG; FELDMAN, 1980), it is implied that the proposed algorithm is an improvement over the algorithm from (SHAPIRO; WAGNER, 1967). However, the author of this thesis believes that it would be interesting to implement and execute these algorithms on recent datasets. In (MARTELLO; TOTH, 1977), empirical evidence was presented, but they used datasets considered to be small according to current standards. They also used an items distribution that we have some reservations about (see Section 3.1.2). In (GREENBERG; FELDMAN, 1980), the claims are backed up by theoretical reasoning, nevertheless empirical evidence shown in Section 5.2 revealed that the improvement had some unpredicted behavior over some instance datasets.

In the 1970's, there was a shift of attention from the DP approach to the B&B approach. The first algorithms using this approach seem to be the Cabot's enumeration method (CABOT, 1970) and the MTU1 algorithm (MARTELLO; TOTH, 1977).

MTU1 was proposed in (MARTELLO; TOTH, 1977), with the name of KP1 at the time (we will refer to this paper as the 'MTU1 paper'). Unfortunately, by current standards, the instances used in the comparison were very small (which is understandable considering the paper publishing date). The numbers of items used were 25, 50 and 100, for instance; the weights (and profits) had values between 11 and 110 (in the case of the profits, 120); the knapsack capacity was chosen between $0.2 \sum_{i \in n} w_i$ and $\sum_{i \in n} w_i$; the distributions used were uncorrelated and weakly correlated ($p_i = w_i + \alpha$, where α was randomly chosen from -10 and 10 following an uniform distribution).

The comparison presented in (MARTELLO; TOTH, 1977) was between KP1 (MTU1), the dynamic programming algorithm called 'periodic step-off' from (GILMORE; GOMORY, 1966), that we will call G.G. for short, and two B&B algorithms for the 0-1 KP (for which the UKP instances were transformed in 0-1 KP instances). The best results were from MTU1, and the second best from the G.G. algorithm. How-

ever, the instances were too small to draw strong conclusions, and the relative difference between G.G. and MTU1 average times was not even one order of magnitude apart. The G.G. algorithm was about four times slower than MTU1 in the instances with $n = 25$; about two or three times slower in the instances with $n = 50$; and less than two times slower in instances with $n = 100$. This trend could indicate that for big instances, the G.G. algorithm would have better times than MTU1 (e.g. the G.G. algorithm could have a costly initialization process but a better average-case asymptotic complexity).

The experiments described above were used by the authors on another paper to state that "The most efficient algorithms for the exact solution of the UKP [...] consist of two main steps: Step 1. Sort the item types according to (5). Step 2. Find the optimal solution through branch-and-bound." (MARTELLO; TOTH, 1990a). This comment established B&B as most efficient approach for the UKP. The paper introduced the MTU2 algorithm (and the author will refer to it as the 'MTU2 paper').

The MTU2 algorithm was designed for large instances (up to 250,000 items). Only sorting the items list was already computationally expensive for the period, and the solutions often involved only the most efficient items. The MTU2 main feature was grouping and sorting only the $k = max(100, \frac{n}{100})$ most efficient items, and calling MTU1 over them. The UKP instance consisting of this reduced items list and the original knapsack capacity was called 'tentative core problem'. If the optimal solution of the tentative core problem was proven to be optimal for the original problem, the algorithm stopped. Otherwise, the optimal solution of the tentative core problem was used as a lower bound to remove dominated items. After this, the k most efficient items outside the tentative core problem were added to it, restarting the process.

The algorithms comparison included only MTU1 and MTU2. The datasets used in the paper were large, but artificial and abundant in dominated items. A more detailed analysis of one of the datasets and the experiment setting is available in Section 3.1.2. The MTU2 was clearly the best algorithm for the chosen datasets.

MTU2 was adopted by the subsequent works as the baseline for testing new algorithms for the UKP. We believe this happened due to many factors, such as: the code of MTU2 was freely available; the algorithm was well and thoroughly explained in Martello and Toth's publications; it presented empirical evidence of dominating

other methods and, consequently, comparing with it would remove the necessity of comparing to many other algorithms; the description of MTU2 stated that it was designed for large instances. However, MTU2 does not completely dominate MTU1, it simply was better for the chosen instances (that were chosen with the purpose of evidencing this). Instances in which the MTU2 needs to repeat the process of adding items to the tentative core problem many times can be more easily solved by MTU1 than by MTU2. Unfortunately, the works that followed chose to compare their algorithms only against MTU2.

EDUK (*Efficient Dynamic programming for the Unbounded Knapsack problem*), a novel DP algorithm for the UKP, was proposed in a conference paper (POIRRIEZ; ANDONOV, 1998) and then presented in a journal paper (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000). EDUK is very different from the previous DP algorithms, and its main features are the application of threshold dominance (proposed in the same paper), and the use of a sparse representation of the iteration domain. This last feature was implemented by using lazy lists, mainly because EDUK was implemented in the functional language OCaml. EDUK is strongly based on the ideas first discussed in (ANDONOV; RAJOPADHYE, 1994).

In (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000), the authors criticize the item distributions used in previous papers, especially the uncorrelated distribution. The author of this thesis agrees with this criticism, further discussion can be found in Section 3.1. However, the solution given for this problem were new datasets of artificial instances. The new datasets do not have simple dominated items, or small efficient items, as the previous datasets, and one of them does not even have any collective dominated items. The change in the choice of items distributions benefits DP methods (and consequently EDUK), which are better suited for such kind of instances. When the new datasets are used, the comparison between MTU2 and EDUK shows that the average times of MTU2 are strongly dominated by the ones of EDUK.

The weakly and strongly correlated distributions are also used in (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000), but varying the value of w_{min}. For those instances, MTU2 dominates EDUK when the weight of the smallest item is close to one, but MTU2 times grow much faster than EDUK times when w_{min} is increased. Only one comparison is made against another DP algorithm. The DP algorithm used seems to be a naïve DP algorithm with a preprocessing phase that removes simple

dominance. The comparison uses a completely different dataset of small instances, in an effort to take into account real-world applications of the UKP, as the ones provenient from solving BPP and CSP with column generation. The average run times in this comparison are smaller than 0.1 seconds, and the difference between the average times of EDUK and the naive DP are about 20% (with EDUK being faster).

EDUK2 is an improvement of EDUK proposed in (POIRRIEZ; YANEV; ANDONOV, 2009). The main improvement brought up by EDUK2 is the hybridization of EDUK with the B&B approach. A B&B preprocessing phase was added to EDUK. If it solves the instance using less than a parametrized number of nodes, then EDUK is never executed; otherwise, the bounds computed in the B&B phase are used to reduce the number of items before EDUK execution and in intervals during its execution. The paper also proposes a new bound for a subset of the strongly correlated instances (the SAW instances), which that is the tightest bound known for such instances. Comparisons are performed with EDUK and MTU2. EDUK2 is clearly the winner, but the average solution times of the methods are few seconds, or less than a second. The experiments are then remade using the same distributions with larger coefficients. MTU2 has integer overflow problems and is left out of the comparison. Between EDUK and EDUK2, EDUK2 has the best results, as expected.

Both (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000) and (POIRRIEZ; YANEV; ANDONOV, 2009) cite (BABAYEV; GLOVER; RYAN, 1997), which presents an algorithm for solving the UKP using the Consistency Approach (CA). The algorithm described in (BABAYEV; GLOVER; RYAN, 1997) was tested against MTU2 and had better times, but the instances used in the experiment make it difficult to have an idea of what would be its performance using more recent datasets (see Section 3.1.1 for further discussion). The CA was already discussed in (GREENBERG, 1986). However, the algorithm proposed in (BABAYEV; GLOVER; RYAN, 1997) considered performance as a priority, different from previous works that treated applying CA to the UKP as an interesting theoretical problem. As the authors of (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000) and (POIRRIEZ; YANEV; ANDONOV, 2009), we tried to obtain a copy of the code from the authors of (BABAYEV; GLOVER; RYAN, 1997), but did not obtain success. The author of this thesis suggests the implementation and comparison of this algorithm as a future work.

Before ending this literature review, we would like to discuss the chapters

about the UKP in the following textbooks: (HU, 1969) and (GARFINKEL; NEMHAUSER, 1972). Those textbooks are especially relevant because they are cited by many of the papers presented at this section. The chapter about the UKP in (HU, 1969, p. 311) has a good introduction about the problem, the simplest DP method for solving it, and the basics of the periodicity.

The chapter about the UKP in (GARFINKEL; NEMHAUSER, 1972, p. 214) has an extensive bibliographical review of the works about the UKP that predates it (1972), which is very relevant as the name 'UKP' was only established some years later. In Section 4.2.3, we discuss a limitation of the review proposed in that chapter.

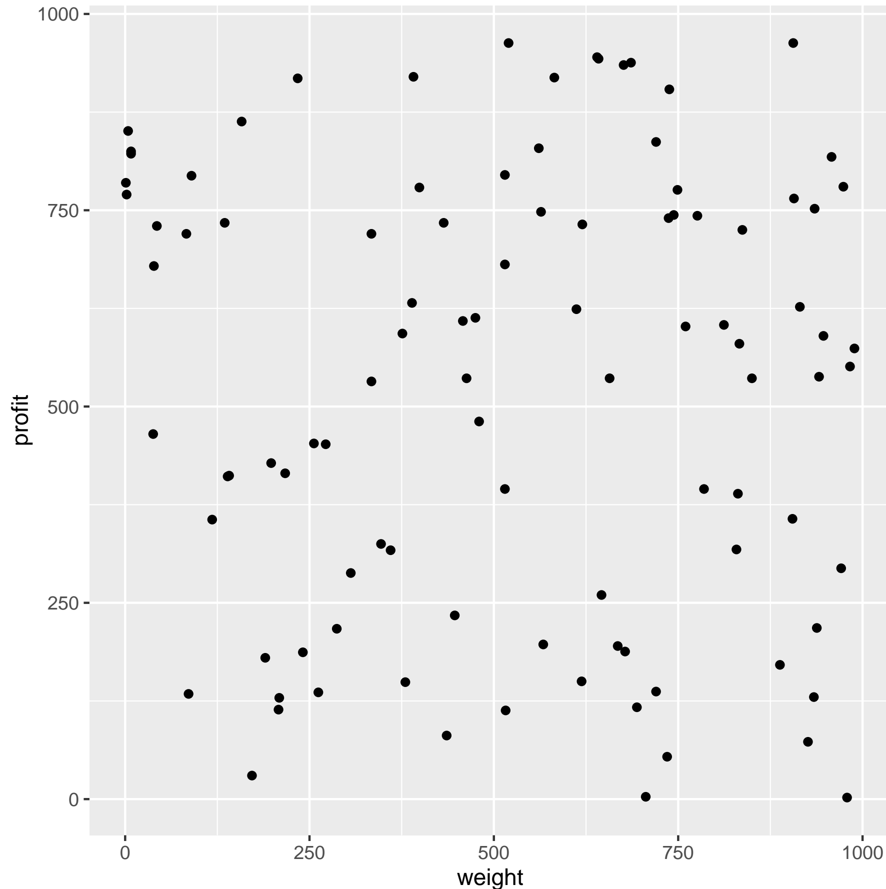
3 THE DATASETS

In this section, we describe some datasets from the literature and propose new datasets. The majority of the datasets described in this section is used in the experiments described in Section 5.

3.1 Uncorrelated items distribution datasets

A instance with an uncorrelated items distribution is an UKP instance in which the weight and the profit of its items are not correlated. The most common way of generating uncorrelated instances is to generate a value between w_{min} and w_{max} for the weight, and a value between p_{min} and p_{max} for the profit, for each of the n items of the instance, using a Pseudo-Random Number Generator (PRNG) with a uniform distribution.

Figure 3.1: An uncorrelated instance generated with $n = 100$, $w_{min} = 1$, $w_{min} = 1000$, $p_{min} = 1$, and $p_{max} = 1000$.



Source: the author.

We have chosen to not use uncorrelated instances in our experiments. The main reason behind this choice is that: *the times used by an algorithm to solve uncorrelated instances have little to do with the algorithm capability of solving a NP-hard problem, and much to do with the polynomial algorithms or heuristics it uses to remove simple and multiple dominated items.* Two uncorrelated instance datasets from the literature are analyzed in the rest of this section. Such analyses should be sufficient to illustrate the reasoning behind our choice.

3.1.1 Babayev's use of uncorrelated instances

A dataset of uncorrelated instances was used in experiments by (BABAYEV; GLOVER; RYAN, 1997)[1]. The set of parameters used to generate this dataset caused almost all instances to be trivial: almost all uncorrelated instances are solved exactly by a greedy procedure that uses only the two most efficient items to fill the knapsack[2].

The reason for this is that the items weight interval is between [1, 1000], and not between [10, 1000] (w_{min} is even smaller). An item with weight *one* has $\frac{1}{1000}$ odds of being generated; and has $\frac{500}{1000} = \frac{1}{2}$ odds of having $p > 500$; so for each item generated there is $\frac{1}{2000}$ (0.05%) odds of generating an *item that multiple-dominates all other items.*

While the maximum efficiency of an item with weight *one* is *one thousand*, the max efficiency of an item with weight *two* is *five hundred*; *three* is 333 and so on. An item i with $w_i = 1$ and $p_i = 501$ will multiple dominate an item j with $w_i = 2$ and $p_i = 1000$; and j is the most efficient item type with weight *two* that can exist. Consequently, the item i will dominate any other item with weight greater than one.

With $n = 5000$, the odds of an instance having item i are already of 91.79%; with $n = 10000$ they are of 99.32%. The dataset was composed of 108 instances; 18 instances for each n value of 5,000; 10,000; 50,000; 100,000; 150,000, and 9 instances for the n values of 200,000 and 250,000. Therefore, for the vast majority of the instances, the solution was probably composed of c copies of the same item i with weight one, and $p > 500$.

---

[1]Section “Second group of experiments - Problems with Uncorrelated Random Coefficients” of the original paper.

[2]This information is pointed out in the original paper.

The author of this thesis did not find the knapsack capacity c used in the instances.

3.1.2 Martello's use of uncorrelated instances

Datasets using an uncorrelated distribution were used in experiments in (MARTELLO; TOTH, 1990a). The uncorrelated distribution and the set of parameters used to generate the datasets created large instances with only a small number of undominated items.

One dataset had instances with twenty different values of n, from 50 to 250,000. The items weights were integer values randomly generated between 10 and 1000. This parameter choice alone already reduces the number of undominated item types to 991. The authors acknowledge the existence of only 991 undominated items in the paper first dataset and propose an additional dataset to show that "the algorithm is not affected by the number of possible undominated item types.". The additional dataset have instances with all combinations of parameters between quantity of items (10^3, 10^4 and 10^5); weight range ([10, 10^3], [10, 10^4], [10, 10^5]); and profit value range ([1, 10^3], [1, 10^4], [1, 10^5]).

In the additional dataset, five out of nine instances also have only 991 undominated items, they are all the instances with weight (or profit) range equal to [10, 10^3] (or [1, 10^3]).

Also in the additional dataset, instances with ranges [10, n] have more than 99.5% probability[3] of having at least one item i with $w_i \leq 20$ and $p_i > \frac{p_{max}}{2}$. A solution s composed of two copies of i will have $p_s > p_{max}$ and $w_s \leq 40$. Consequently, all items j with $p_j \leq p_{max}$ and $w_j \geq 40$ will be multiple dominated by item i. Such instances have only thirty undominated items (possibly less).

All instances of the additional dataset have more than 99.5% probability[4] of

[3]Calculated the following way: $\frac{11}{991}$ are the odds of generating an instance with weight smaller than or equal to twenty (i.e. [10, 20]); $\frac{1}{2}$ are the odds of one item having profit greater than half of the maximum profit value (e.g. [501, 1000]); $\frac{11}{991} \times \frac{1}{2}$ are the odds of generating an item with both previous characteristics (small weight and above average profit), and $1 - (\frac{11}{991} \times \frac{1}{2})$ are the odds of generating an item that *do not* have both characteristics; $(1 - (\frac{11}{991} \times \frac{1}{2}))^{1000} \equiv 0.00382$ are the odds of generating 1000 items, and none of them having both characteristics. These odds are about 0.38%, and therefore the odds of generating at least one item with both characteristics among 1000 items is about 99.61%.

[4]Calculated the same way, but using $\frac{9}{100} \times \frac{1}{2}$ as the chance of generating and item i with $w_i \leq \frac{w_{max}}{10}$ and $p_i > \frac{p_{max}}{2}$.

having an item i with $w_i \leq \frac{w_{max}}{10}$ and $p_i > \frac{p_{max}}{2}$, that would multiple dominate all items j with $w_j > 2 \times \frac{w_{max}}{10}$ (i.e. at least 80% of the items are multiple dominated by a single item in all instances).

3.2 PYAsUKP dataset

This section describes the instance datasets proposed in (POIRRIEZ; YANEV; ANDONOV, 2009), and reused in the comparison presented in (BECKER; BURIOL, 2016). All those datasets were artificially generated with the purpose of being "hard to solve". The adjective 'hard' can mean a different thing for each one of the datasets. Some item distributions used in these datasets were first proposed in (POIRRIEZ; YANEV; ANDONOV, 2009), others were taken from the literature.

These datasets are similar to the ones used in (POIRRIEZ; YANEV; ANDONOV, 2009). The same tool was used to generate the datasets (PYAsUKP) and the same parameters were used, unless otherwise stated. However, some instances make use of random seed values that were not provided, so the exact instances used in (POIRRIEZ; YANEV; ANDONOV, 2009) can be different. The instances are exactly the same presented in (BECKER; BURIOL, 2016).

The same code that implements EDUK and EDUK2 also implements the instance generator that generated the instances used in the experiments described in this thesis. Some datasets generated by this tool have the item list ordered by increasing weight, what gives a small advantage to EDUK that uses this ordering.

In the subsection 5.1.1 *Known "hard" instances* of (POIRRIEZ; YANEV; ANDONOV, 2009) some sets of easy instances are used to allow comparison with MTU2. However, MTU2 had integer overflow problems on harder instances. With exception of the subset-sum dataset, all datasets have a similar harder set (Subsection 5.2.1 *New hard UKP instances* (POIRRIEZ; YANEV; ANDONOV, 2009)). Thus, we considered in the runs only the harder ones.

The notation $rand(x, y)$ means an integer between x and y (including both x and y), generated by a PRNG with an uniform distribution. Also, when referring to the parameters for the generation of an instance, w_{min} will be used to denote the smallest weight that can appear in an instance, but without guarantee that an item with this exact weight will exist in the generated instance. The meaning of w_{max} is analogue. The syntax $x\overline{n}$ means x as a string concatenated with the value of n as a

string (e.g. for $n = 5000$ then $10\overline{n} = 105000$).

The dataset presented in this section comprises five smaller datasets. Each of these datasets is characterized by a formula used to generate the items, and use different combinations of parameters to generate the instances. Three parameters are present in all instance generation procedures, they are: n (number of items), c (knapsack capacity) and w_{min} (explained last paragraph). The arguments for such parameters were given to the PYAsUKP binary by means of the following flags: *-wmin* w_{min} *-cap* c *-n* $\boldsymbol{n}$. In the description of each one of the five smaller datasets, we will present any other PYAsUKP flags that were also needed to generate that dataset.

We found some small discrepancies between the formulas presented in (POIRRIEZ; YANEV; ANDONOV, 2009) and the ones used in PYAsUKP code. We opted for using the ones from PYAsUKP code, and they are presented below.

3.2.1 Subset-sum instances

Subset-sum instances are instances where $p_i = w_i = rand(w_{min}, w_{max})$. Subset-sum instances generated with the same parameters presented in (POIRRIEZ; YANEV; ANDONOV, 2009) can be solved in less than a centisecond by many of the algorithms presented at this work. The implementation of the EDUK and EDUK2 algorithm have imprecise time measuring, i.e. if the algorithm takes less than a second to solve an instance, the run time returned by the program is sometimes zeroed. Because of this, in this paper, we use a similar dataset, but with each parameter multiplied by ten. This way, the instances will take more time to solve, and the effects of the imprecise time measuring will be minimized. Therefore, 10 instances were generated for each possible combination of: $w_{min} \in \{10^3, 5 \times 10^3, 10^4, 5 \times 10^4, 10^5\}$; $w_{max} \in \{5 \times 10^5, 10^6\}$ and $n \in \{10^3, 2 \times 10^3, 5 \times 10^3, 10^4\}$, totaling 400 instances. Each instance had a random capacity in the range $[5 \times 10^6; 10^7]$. The PYAsUKP *-form ss -wmax* w_{max} flags were used.

The rationale for the study of such instance distribution is not well understood by the author of this thesis. If purely subset-sum knapsacks existed in practice, then discarding all profit values and applying a subset-sum algorithm would be the best way to solve them. Also, collective dominance is very common if w_{min} is small.

3.2.2 Instances with strong correlation between weight and profit

There are many formulae for generating items distribution that could be considered strongly correlated item distributions. In (MARTELLO; TOTH, 1990a) and (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000), the formula used was $w_i = rand(w_{min}, w_{max})$ and $p_i = w_i + \alpha$, for a given w_{min} and w_{max} and a constant $\alpha = 100$. The already described Subset-Sum instances can also be considered strongly correlated. However, in (POIRRIEZ; YANEV; ANDONOV, 2009), the formula presented below was used, because "(CHUNG; HUNG; ROM, 1988) have shown that solving this problem is difficult for B&B.". In all strongly correlated instances with $\alpha > 0$, the smallest item is also the best item, a trait that often makes an instance easier (the best item ends up multiple dominating many items).

Instances were generated using the following formula: $w_i = w_{min} + i - 1$ and $p_i = w_i + \alpha$, for a given w_{min} and α. Note that, except by the random capacity, all instances with the same α, $\mathbf{n}$, and w_{min} combination are equal. The formula does not rely on random numbers. The PYAsUKP *-form chung -step* α flags were used.

Twenty instances were generated with each combination of $n = 5 \times 10^3$, $\alpha \in \{5, -5\}$ and $w_{min} \in \{10^4, 1.5 \times 10^4, 5 \times 10^4\}$. Twenty more instances were generated with each combination of $n = 10^4$, $\alpha \in \{5, -5\}$ and $w_{min} \in \{10^4, 5{\times}10^4, 11{\times}10^4\}$, totalling 240 instances. Each instance had a random capacity in the range $[20\overline{n}; 100\overline{n}]$.

3.2.3 Instances with postponed periodicity

Many algorithms benefit from the periodicity property, explained in Section 1.3.2, by computing an upper bound on capacity y. For the instances created using the formula below and "where $c < 2 \times w_{max}$ and n is large enough, the periodicity property does not help"(POIRRIEZ; YANEV; ANDONOV, 2009, p. 13). The idea of such instances seems to be putting all algorithms on an equal footing regarding which capacity they are solving an instance.

This family of instances is generated by the following method: $\mathbf{n}$ distinct weights are generated with $rand(w_{min}, w_{max})$ and then sorted by increasing order; $p_1 = w_1 + rand(1, 500)$; and $\forall i \in [2, n]$. $p_i = p_{i-1} + rand(1, 125)$. The w_{max} is computed as $10\overline{n}$. The PYAsUKP *-form nsds2 -step 500 -wmax* w_{max} flags were used.

Two hundred instances were generated for each combination of $n = \{2 \times 10^4, 5 \times 10^4\}$ and $w_{min} = \{2 \times 10^4, 5 \times 10^4\}$. Totalling 800 instances. Each instance had a random capacity in the range $[w_{max}; 2 \times 10^6]$.

3.2.4 Instances without collective dominance

Any items distribution in which the efficiency of the items increases with their weight has no collective, multiple, or simple dominated instances. There is threshold dominance in such instances, as bigger items will probably dominate solutions composed of many copies of a smaller item. A distribution with this characteristics was proposed "to prevent a DP based solver to benefit from the variable reduction due to the collective dominance" (POIRRIEZ; YANEV; ANDONOV, 2009, p. 13), with the intent of making comparison fairer.

This family of instances is generated in the following method: **n** distinct weights are generated with $rand(w_{min}, w_{max})$ and then sorted by increasing order; $p_1 = p_{min} + rand(0, 49)$; and $\forall i \in [2, n]$. $p_i = \lfloor w_i \times ((p_{i-1}/w_{i-1}) + 0.01) \rfloor + rand(1, 10)$. The given values are: $w_{min} = p_{min} = \mathbf{n}$ and $w_{max} = 10\overline{n}$. The PYAsUKP *-form hi -pmin* p_{min} *-wmax* w_{max} flags were used.

Five hundred instances were generated for each combination of $n = w_{min} \in \{5 \times 10^3, 10^4, 2 \times 10^4, 5 \times 10^4\}$, totalling 2000 instances. Each instance had a random capacity in the range $[w_{max}; 1000\overline{n}]$.

3.2.5 SAW instances

The SAW instances were proposed in (POIRRIEZ; YANEV; ANDONOV, 2009). They include the strongly correlated instances with $\alpha > 0$, and as such share the same trait pointed out in Section 3.2.2. The main point of the authors of (POIRRIEZ; YANEV; ANDONOV, 2009) for including such instances in the benchmark seems to be testing a new general upper bound proposed in the same work. The new bound is the tightest known for the SAW instances, and is included in EDUK2.

This family of instances is generated by the following method: generate **n** random weights between w_{min} and $w_{max} = 1\overline{n}$ with the following property: $\forall i \in$

$[2, n]$. $w_i \ mod \ w_1 > 0$ (w_1 is the smallest weight); sort the weights by increasing order; then $p_1 = w_1 + \alpha$ where $\alpha = rand(1, 5)$, and $\forall i \in [2, n]$. $p_i = rand(l_i, u_i)$ where $l_i = max(p_{i-1}, q_i)$, $u_i = q_i + m_i$, $q_i = p_1 \times \lfloor w_i / w_1 \rfloor$, and $m_i = w_i \ mod \ w_1$. The PYAsUKP *-form saw -step* α *-wmax* w_{max} flags were used.

Two hundred instances were generated for each combination of $w_{min} = 10^4$, $n \in \{10^4, 5 \times 10^4, 10^5\}$. Additional 500 instances were generated with $w_{min} = 5 \times 10^3$ and $n = 5 \times 10^4$, totalling 1100 instances. Each instance had a random capacity in the range $[w_{max}; 10\overline{n}]$.

3.2.6 PYAsUKP dataset and reduced PYAsUKP dataset

The dataset described in the last five subsections, totalling 4540 instances, will be referred to in the rest of this work as *the PYAsUKP dataset*. In each of these five smaller datasets, for the same combination of parameters, the number of instances generated was always perfectly divisible by ten. The dataset composed of one tenth of the PYAsUKP dataset, following the same distribution, and totalling 454 instances, will be referred to in the rest of this work as *the reduced PYAsUKP dataset*.

3.3 CSP pricing subproblem dataset

The applied use of the UKP chosen by the author to be developed in this work was the pricing subproblem generated by solving the continuous relaxation of the set covering formulation for the classic Bin Packing Problem (BPP) and Cutting Stock Problem (CSP) using the column generation approach. A summary about this use was already given in Section 1.1, together with the explanation of its relevance. In this section, sufficient technical detail will be given, allowing the reader to understand how such instances of the UKP are generated and why it is necessary to solve them. Our explanation will allude to the simplex method, but it is not necessary to understand it to follow the explanation. For readers interested in mathematical proofs, and in longer explanations of *why* the method works, in contrast to *how* it works, we recommend reading Section 15.2 (pages 455 to 459) of (KELLERER; PFERSCHY; PISINGER, 2004) and/or the seminal paper by (GILMORE; GO-

MORY, 1961).

As the BPP and the CSP are very similar, and instances of one problem can be converted to instances of the other (similarly to 0-1 KP and BKP), we will explain the relationship only in terms of the CSP.

An instance of the CSP problem consists of n distinct sheet sizes; each sheet size $i = \{1, \ldots, n\}$ has an unidimensional size w_i and a demand $d_i > 0$, that needs to be satisfied; to satisfy the demand it is necessary to cut sheets of the desired size from master rolls of size c, where c is bigger than any sheet size. It is assumed that there is a sufficient amount of master rolls to satisfy all demands and, as such, the instance does not define a number of master rolls available. The objective of the problem is to find a way to fill all demands while using the smallest possible number of master rolls. If one or more sheets are cut from a master roll, that master roll is considered used, and remaining space is considered waste.

The previously mentioned Set Covering Formulation (SCF) for BPP and CSP is a tight formulation proposed in (GILMORE; GOMORY, 1961). The SCF eliminated the problems of the classic formulation that was loose and had too many symmetric solutions. However, as a consequence, the SCF needs to compute all cutting patterns, i.e. all combinations of sheet sizes that can be cut from a single master roll. As the cutting patterns are combinations, the amount of cutting patterns can be superexponential in the number of sheet sizes. The exact number of cutting patterns is affected by the sheet sizes. The best case happens when $\forall i.\ w_i > \frac{c}{2}$, in this case the number of cutting patterns is n. The worst case happens when all n sheet sizes have almost the same size (let us call this size w^*), and $n > k = \frac{c}{w^*}$, in this case the number of cutting patterns is given by the binomial coefficient $\binom{n}{k}$ (that computes $n!$, which is superexponential).

The SCF follows:

$$\text{minimize} \sum_{j=1}^{m} x_j \tag{3.1}$$

$$\text{subject to} \sum_{j=1}^{m} a_{ij} x_j \geq d_i, \quad \forall i \in \{1, ..., n\}, \tag{3.2}$$

$$x_j \in \mathbb{N}_0, \quad \forall j \in \{1, ..., m\}. \tag{3.3}$$

All cutting patterns $j = \{1, \ldots, m\}$ can be represented by a matrix a_{ij} that

stores the amount of sheets of size i obtained when the cutting pattern j is used. If we know all cutting patterns, a solution for the CSP can be represented by a variable x_j that stores the amount of master rolls which were cut using a specific cutting pattern j.

It is important to remember that, in this work, our objective is not to solve CSP but its continuous relaxation.

The column generation approach consists in avoiding the enumeration of all m cutting patterns. The SCF relaxation is initialized with a small set of cutting patterns that can be computed in polynomial time and in which each sheet size appears at least in one of the patterns. This reduced problem is called the *master problem*. It is solved by using the simplex method, as it is a linear programming problem. A by-product of this solving process are the dual variables of the master problem model. Those variables are used as input for a *pricing subproblem*. The solution of this pricing subproblem is the cutting pattern that, if added to the master problem, will give the greatest improvement to master problem optimal solution.

The pricing subproblem for the column generation of the BPP/CSP is the UKP. An instance of the UKP created by the procedure described above will have the following structure:

$$\text{maximize} \sum_{i=1}^{n} y_i x_i \tag{3.4}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq c, \tag{3.5}$$

$$x_i \in \mathbb{N}_0, \quad \forall i \in \{1, ..., n\}. \tag{3.6}$$

The formulation above is clearly equivalent to the formulation presented in Section 1.2. The sheet sizes $i = \{1, \ldots, n\}$ are the items $i = \{1, \ldots, n\}$. The size of the sheets w_i is the weight of the items w_i. The value of the dual variable associated to a specific sheet size y_i is the profit value p_i. The size of the master roll c is the knapsack capacity c. The new cutting pattern described by x_i is an optimal solution for UKP x_i (items inside the knapsack are equivalent to sheets cut from the master roll).

The solving process alternates between solving the master problem and the pricing subproblem, until all cutting patterns that could improve the solution of the

master problem are generated and added to the master problem. The profit values of the pricing subproblem (dual variables) are real numbers close to *one*. If the value of the optimal solution for the pricing subproblem is *one* or less, we have a guarantee that the master problem cannot be improved by adding any new cutting patterns to it. The computation could be stopped and the optimal solution for the master problem is the exact optimal solution for the continuous relaxation of the CSP instance. However, floating point arithmetic is imprecise. In the real-world, an implementation of the pricing subproblem can return a value slight above *one*, when it should have returned *one* or slight less than *one*. In this case, adding the newly generated cutting pattern will not improve the master problem solution, and will re-generate the same pricing subproblem with the same optimal solution value incorrectly above *one* (infinite loop). Taking this into account, a better method for stopping the computation is verifying if the current pricing subproblem is equal to the one from last iteration, or if the solution of the pricing subproblem is equal to the one from the last iteration.

The method described above can generate thousands of UKP instances for one single instance of the CSP. For the same instance of the CSP, the number of UKP instances generated, and their exact profit values, can vary based on the choice of optimal solution made by the UKP solver (for the same pricing subproblem many cutting patterns can be optimal, but only one among them is added to the master problem). Consequently, such dataset is hard to describe (has a large and variable number of instances with variable profit values). The best way found by the author to ensure that the results are reproducible is making available the exact codes used in the experiment, together with the list of CSP instances from the literature used in the experiment. The codes are deterministic, and consequently will produce the same results if executed many times over the same CSP instance.

A recent survey on BPP and CSP gathered the instances from the literature, and also proposed new ones (DELORME; IORI; MARTELLO, 2014). The total number of instances in all datasets presented in the survey is 5692. The author of this thesis chose ten percent of those instances for the experiments presented at Section 5.4. This fraction of the instances was randomly selected among instances within the same dataset or, in the larger datasets, the same generation parameters. The address of a repository containing the data and code used in the above mentioned experiments, and the instructions to compile the code, can be found in A.1.

(GILMORE; GOMORY, 1961; GILMORE; GOMORY, 1963; GILMORE; GOMORY, 1966) present some optimizations to the master problem solver that we have not implemented. The author of this thesis believes that these optimizations do not considerably affect the structure of the pricing subproblem. Also, this work has no intention of providing a state-of-the-art algorithm for solving the CSP continuous relaxation, but only to study algorithms for the UKP in the context of a pricing subproblem and independently.

A list of implementation details follows: cutting patterns that are not used by the last solution for the master problem could be removed, however they can end up being valuable again in the future and re-generated by the pricing subproblem, so the author chose not to remove them; the classic polynomial method for creating the initial set of cutting patterns was used: it consists in creating n patterns, each one with only one sheet size that is cut as many times as possible, state-of-the-art solvers often begin using cutting patterns generated by a more sophisticated heuristic; the sheet sizes can be divided in two groups, half with higher demand, and half with lower demand; and the pricing subproblem can be restricted to the high demand group in some conditions – this also has not been done in our experiments.

Before ending this section, it is important to correct a false claim published in (BECKER; BURIOL, 2016). The article stated that "The currently fastest known solver for BPP/CSP[2, 3] uses a column generation technique (introduced in [5]) that needs to solve an UKP instance as the pricing subproblem at each iteration of a column generation approach.". The mentioned solver is the one proposed in (BELOV; SCHEITHAUER, 2006), and it was found to be the fastest by the experiments conducted in (DELORME; IORI; MARTELLO, 2014). It makes use of the column generation approach, however the pricing subproblem is not exactly the UKP, as consequence of the way the algorithm adds cuts to the continuous relaxation.

3.4 Bottom Right Ellipse Quadrant instances

The Bottom Right Ellipse Quadrant Instances ('BREQ instances', for short) is a new UKP instance item distribution proposed by the author of this thesis[5].

[5] The author is aware of the existence of the *circle instances*, which are described in (KELLERER; PFERSCHY; PISINGER, 2004, p. 158). In this book, the formula presented for the circle instances is $p_i = d\sqrt{4R^2 - (w_i - 2R)^2}$ – in which d is an arbitrary positive constant ($\frac{2}{3}$) and R is w_{max}. This formula describes the *upper left* quadrant of an ellipse, which is different from

This instance distribution was created to illustrate that different item distributions favors different solution approaches and, therefore, the choice of instances (or specifically, their item distribution) defines what is considered the *best algorithm*. These instances will be used in the experiments presented in Section 5.3.

Distributions that are easy to solve by the DP approach and hard to solve by the B&B approach are common in the recent literature. This distribution has the opposite characteristic, it is hard to solve by DP and easy to solve by B&B. The BREQ distribution does not model any real-world instances that the author is aware of.

The name given to this distribution is derived from the fact that, when plotted on a graph, the items show the form of a quarter of ellipse (specifically, the bottom right quadrant). All items with such distribution respect the following equation: $p_i = p_{max} - \lfloor\sqrt{p_{max}^2 - (w_i^2 \times \frac{p_{max}}{w_{max}}^2)}\rfloor$. In this context, w_{max} and p_{max} define the quadrant top right corner, i.e. the possible maximum value for an item weight and profit, an item with those exact values does not need to exist in a BREQ instance. The rounding down in the formula can be dropped if the profit is a real number and not an integer.

A natural consequence of this distribution shape is that the item efficiency grows with the item weight. This leads to the inexistence of simple, multiple and collective dominance[6]. In other words, for any solution s composed of two or more items, and for any single item i, if $w_s \leq w_i$ then $p_s < p_i$.

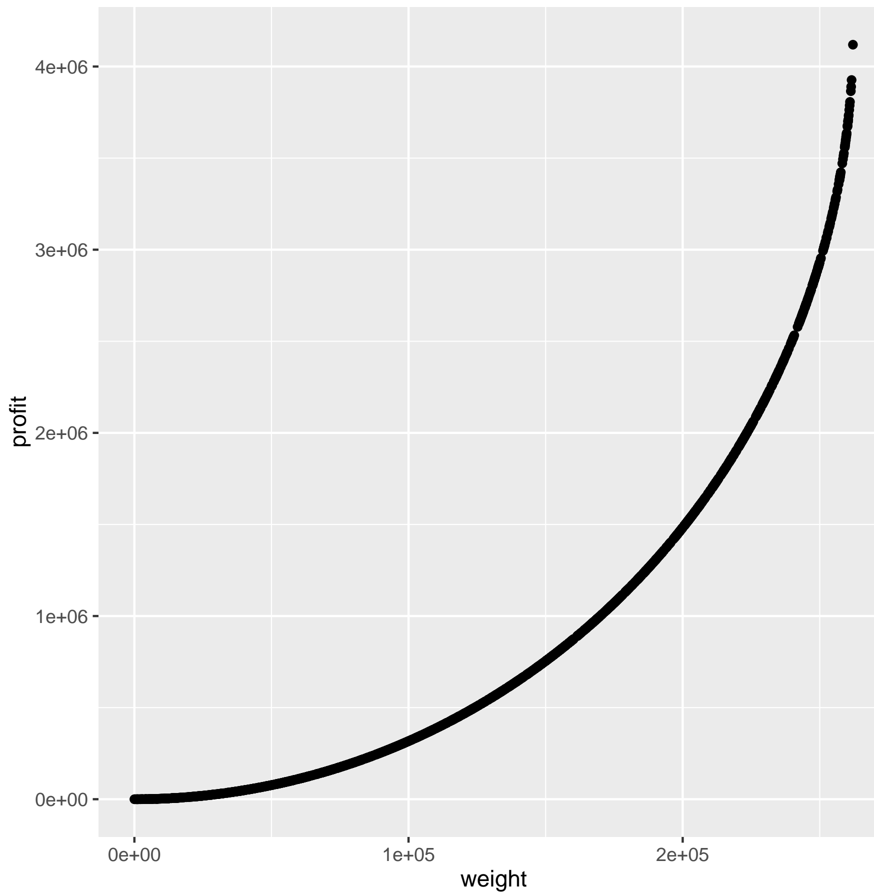
On the other hand, threshold dominance is very common in this instance type. Except for the best item, any item of any instance (of any distribution, not only BREQ instances) will always be threshold dominated at some capacity. In many UKP instances, however, the knapsack capacity is smaller than those threshold values and therefore the threshold dominance is not applied or relevant. In BREQ instances, also as a consequence of the efficiency growth, an optimal solution will never include the item i two or more times if there is an item j such as that $\sqrt{2} \times w_i \leq w_j \leq 2 \times w_i$.

The solutions of BREQ instances will often contain the maximum number of

the BREQ formula that describes the *bottom right* quadrant of an ellipse. Consequently, circle instances and BREQ instances have completely different properties. Also, in the referred book, circle instances are solved as they were instances of the 0-1 KP (not of the UKP).

[6]If the profit is integer, some small items can display those three dominance relations because of the profit precision loss caused by the rounding. The author of this thesis believe that this exception is of little relevance and can be safely ignored for most purposes. If profit is an infinite precision real, the statement has no exceptions.

Figure 3.2: A 128-16 BREQ Standard instance with $n = 2048$



Source: the author.

copies of the largest item (that is also the most profitable, and the most efficient) allowed by the instance capacity. Any gap left will probably be filled with the heaviest item that fits the gap. The process should be repeated until no item fits the gap left (or there is no gap). The classical greedy heuristic procedure that follows those steps would probably yield an optimal solution. However, this is not always the case[7].

The reasons that make BREQ instances favor B&B over DP can be understood by examining the two approaches behaviour. The B&B approach begins by creating a solution using some sort of greedy heuristic similar to the one described in the last paragraph. This solution will be close to optimal, if not optimal, and pro-

[7]A counter-example follows: consider an instance with $n = 4$, $c = 512$, $w_1 = 384$, $p_1 = 2774$, $w_2 = 383$, $p_2 = 2756$, $w_3 = 129$, $p_3 = 265$, $w_4 = 32$ and $p_4 = 17$; the optimal solution does not use the best item (w_1, p_1); the best solution when using the best item has a profit value of $2842 = 2774 + 4 \times 17$ (weight $512 = 384 + 4 \times 32$) while the best solution when using the second most efficient item has the optimal profit value of $3021 = 2756 + 265$ (weight $512 = 383 + 129$). In this case, between two solutions with the same weight, the one with the best item is not the best one. The weight and profit values of this example follow a BREQ distribution with $w_{max} = 512$ and $p_{max} = 8192$.

vides a good lower bound. With this lower bound, the search space will be greatly reduced, making the algorithm end almost instantly. On the other side, the DP approach is based on solving subproblems. Subproblems which will be solved for increasing capacity values yielding optimal solutions to be reused (combined with other items). However, solutions with small weight and/or solutions composed of items with small weights are often less efficient than solutions composed of items with a greater weight and, consequently, solutions for lesser capacities are unlikely to be reused.

The objective is not to explore this distribution and its behaviour with different parameters, we only intend to show that it favors the B&B approach over the DP. The author of this thesis proposes a subset of the BREQ instances which the only parameters are n and the seed for the PRNG, with the other instance parameters being computed over the value of n. This instance distribution will be referred to as the BREQ 128-16 Standard: a BREQ distribution in which $c = 128 \times n$, $p_{min} = w_{min} = 1$, $w_{max} = c$ and $p_{max} = 16 \times w_{max}$. The PRNG seed is used to create n unique random weight values between w_{min} and w_{max} (the random number distribution is uniform); the profit values for each weight are computed using the first formula presented at this section (and the w_{max} and p_{max} values for that n).

The reasoning for the choices made when defining the BREQ 128-16 Standard follows. There was no reason to restrict the w_{min} to w_{max} interval to be smaller than c (there are c distinct valid weight values). The constant 128 used to compute the capacity c for a n value was chosen as the first power of two higher than a hundred. Consequently, less than 1% of all possible items will appear in an instance, making instances generated with different seeds significantly different. The p_{min} to p_{max} value interval was chosen to be sixteen times bigger than the w_{min} to w_{max} interval to alleviate the effects of rounding the profit value to an integer value (it would not need to be done if the profit was a floating point number with reasonable precision). The efficiency of the items in a BREQ distribution will vary between zero and $\frac{p_{max}}{w_{max}}$, so if $p_{max} = w_{min}$ the efficiency would vary between zero and one, giving more relevance to the rounding. Finally, for the biggest n value that we were interested in testing (2^{20}), the highest possible value of an item profit is $2^{31} = 2^{20} \times 128 \times 16$, what keeps the values smaller than 32 bits.

The BREQ 128-16 Standard allows us to create a simple benchmark dataset,

in which we only need to worry about varying n and the seed. We propose a benchmark with a hundred instances, with all combinations of $n = 2^{11} = 2048,\ 2^{12},\ ...,\ 2^{20} \approx 10^6$ (ten n values), and ten different seed values. We will refer to it as the BREQ 128-16 Standard Benchmark (or BREQ 128-16 SB).

3.5 Other distributions

While the artificial item distributions presented in (POIRRIEZ; YANEV; ANDONOV, 2009) are used in the experiments of this work, some other artificial distributions are ignored. The reasons for not using the uncorrelated distribution were extensively discussed in this chapter. However, no reason was presented for not using the *weakly correlated* distribution, presented in (MARTELLO; TOTH, 1977), (MARTELLO; TOTH, 1990a), (BABAYEV; GLOVER; RYAN, 1997) and (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000), or the *realistic random* distribution, presented in (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000).

The main reason for using the artificial distributions described in (POIRRIEZ; YANEV; ANDONOV, 2009) was that questions about the performance of the algorithms in the most recent benchmark dataset for the UKP would most certainly arise. Moreover, these experiments answer these questions preemptively. Not using the dataset would raise unfounded suspicion about this choice. The BREQ instances are a special case, as their purpose is exactly showing that it is easy to design a items distribution that favors one approach over another.

4 APPROACHES AND ALGORITHMS

In this chapter, some approaches and algorithms for solving the UKP will be discussed. The objectives of this chapter are: to present relevant details that did not fit in the literature review; to further develop what was said in the last section, about some approaches favoring some item distributions; to give readers some base to understand the results of the experiments (Section 5); and to explain the concept of solution dominance, mentioned in Section 1.3.1. The objective is not to present an exhaustive list of approaches and algorithms.

4.1 Conversion to other KP variants

In Section 1, it was pointed out that the UKP can be seen as a special case of BKP where, for each item type i, there are at least $\lfloor \frac{c}{w_i} \rfloor$ copies of that item type available in an instance. Consequently, it is possible to convert any instance of the UKP in an instance of the BKP, and to solve it with an algorithm designed to solve the BKP. In this work, this approach will not be used or thoroughly studied. The rationale for this choice is that such approach cannot yield competitive performance results, for reasons that are explained next paragraph.

An algorithm designed to solve the BKP needs a mechanism to prevent solutions from exceeding the available amount of each item. An algorithm designed for the UKP does not have this overhead. An algorithm designed for the UKP needs to keep track of the items used in the solutions, but does not need to frequently access this information (as to check if it can add an additional copy of one item to a solution). Also, an algorithm designed for the UKP can fully exploit the properties described in Section 1.3.

In (MARTELLO; TOTH, 1977), experiments converting instances of the UKP into instances of the BKP were realized. The experiments yielded the expected result (i.e. the BKP algorithms performed poorly in comparison to UKP-specific algorithms). The conclusions derived from the experiment are fragile, because only small instances were used. However, based on the rationale exposed in the last paragraph, the author of this thesis believes it is safe to assume that, for the same instance of the UKP, and the same solving approach (DP, B&B, ...), a state-of-the-art algorithm for the UKP will outperform a state-of-the-art algorithm for the

BKP.

4.2 Dynamic Programming

The Dynamic Programming (DP) approach is the oldest one found in the literature review (Section 2). Its worst-case time complexity is $O(nc)$ (pseudo-polynomial). The DP approach can be considered stable, or predictable, compared to other approaches. Stable in the sense that its run time variation when solving many instances with the same characteristics (i.e. n, c and items distribution) can be lower than other approaches. Predictable in the sense that it is easier to predict a reasonable time interval for solving an instance based in the characteristics just mentioned, than it is with other approaches.

The DP worst-case space complexity is $O(n + c)$, which can be considerably greater than other approaches that do not allocate memory linear to c. However, the space needed can be reduced by many optimizations. Some of these optimizations are: using a periodicity bound as explained in Section 1.3.2; using modular arithmetic to reduce c to w_{max} in at least one array, see (GILMORE; GOMORY, 1966, p. 17); using binary heaps instead of arrays, as the heap can use less memory than an array of c positions if w_{min} is sufficiently big.

The DP approach often gives an optimal solution for each capacity smaller than c. However, some space optimizations can remove such feature.

4.2.1 The naïve algorithm

In this thesis, the author sometimes mentions to the naïve (or basic) DP algorithm for the UKP. The naive algorithm is an algorithm specifically designed for the UKP, but that applies no optimizations. It is the most straightforward implementation of the recursion that describes the problem. This algorithm will always execute about $n \times c$ operations (which is the worst case performance for other DP algorithms). It does not implement any dominance relation[1], does not use periodicity or prunes symmetric solutions.

The pseudo-code of the naive DP algorithm can be seen in Algorithm 1. The

[1] The only exception is that when two or more distinct solutions have the same weight only one of them is kept.

Algorithm 1 Naive DP algorithm

```
1: procedure NaiveDP(n, c, w, p, w_min)
2:     g ← array of c + 1 positions, range [0, w_min) initialized to 0
3:     d ← array of c + 1 positions, range [0, w_min) initialized to 0
4:     for y ← w_min, c do
5:         g[y] ← 0
6:         for i ← 1, n do
7:             if w_i > y then
8:                 end inner loop
9:             end if
10:            if g[y − w_i] + p_i > g[y] then
11:                g[y] ← g[y − w_i] + p_i
12:                d[y] ← i
13:            end if
14:        end for
15:    end for
16:    return g[c]
17: end procedure
```

notation is the one introduced in Section 1.2. The letter y will be used in this and other algorithms to denote an arbitrary capacity value. This implementation of the algorithm considers that the items $i \in \{1, \ ..., \ n\}$ are ordered by non-decreasing weight (i.e. $w_1 \leq w_2 \leq ... \leq w_n$). The arrays indices will always be base zero, and the items list indices base one. The procedure to recover the items that constitute the optimal solution will not be given for this and the remaining DP algorithms because, in general, it is the same procedure explained in UKP5 (see Section 4.2.4).

4.2.2 The algorithm of Garfinkel and Nemhauser

The algorithm given in (GARFINKEL; NEMHAUSER, 1972, p. 221) can be seen as variation of the naive DP algorithm (see Algorithm 2). While it does not seem to be much of an improvement, the use of the $i \leq d[y - w_i]$ test eliminates solution symmetry. This test, together with the items being ordered by non-increasing efficiency ($\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$), can considerably improve the running times. The condition can also be seen as: add a new item to a pre-existing solution if, and only if, the new item is the most efficient item already in the solution, or an item even more efficient.

Algorithm 2 Garfinkel's DP algorithm

```
1: procedure GarDP(n, c, w, p, w_min)
2:     g ← array of c + 1 positions, range [0, w_min) initialized to 0
3:     d ← array of c + 1 positions, range [0, w_min) initialized to n
4:     for y ← w_min, c do
5:         g[y] ← 0
6:         for i ← 1, n do
7:             if w_i ≤ y and i ≤ d[y − w_i] and g[y − w_i] + p_i > g[y] then
8:                 g[y] ← g[y − w_i] + p_i
9:                 d[y] ← i
10:            end if
11:        end for
12:    end for
13:    return g[c]
14: end procedure
```

4.2.3 The step-off algorithms of Gilmore and Gomory

Four DP algorithms for the UKP are described in (GILMORE; GOMORY, 1966, p. 14 to 17). With the exception of the first algorithm, each one of the three remaining algorithms is an improvement of the previous one. The second and third algorithms (respectively, the 'ordered step-off' and the 'terminating step-off') are very similar to the UKP5. The second algorithm is basically UKP5 without a periodicity check, and the third algorithm is an UKP5 with a different periodicity check. To avoid repetition, we will ignore small implementation differences, and present only UKP5, in the next section. The first and the fourth DP algorithms can be ignored because the first is dominated by the second/third versions; and the fourth algorithm reduce memory usage at cost of a little extra processing (not an interesting trade-off in the context of this work).

As already mentioned, the author of this thesis proposed UKP5 in (BECKER; BURIOL, 2016), believing it was novel. The UKP5 algorithm was thought as an improvement of the Garfinkel's DP Algorithm presented in last section. The (GILMORE; GOMORY, 1966) paper was not checked at time because it was cited in (GARFINKEL; NEMHAUSER, 1972), and we did not expect the book to provide a worsened version of the algorithm on purpose. In Section 6.4 of that book, a DP algorithm was presented as the last of a series of improvements over the naive DP algorithm. However, if we check the chapter notes, there is the comment: "6.4: The recursion of this section is based on Gilmore and Gomory (1966). See Exercise 21 for a variation

that will be more efficient for some data sets.". The algorithm presented in Section 6.4 was a version of the algorithm in (GILMORE; GOMORY, 1966) with *many* of its relevant optimizations removed, and in exercise 21 it is expected of the reader to recreate *one* of these optimizations based on hints given at the exercise. The author of this thesis believes that this fact went unnoticed by previous authors that cited the book. The book did not provide the answers for the exercises.

4.2.4 UKP5

UKP5 was inspired by the DP algorithm described by Garfinkel (GARFINKEL; NEMHAUSER, 1972, p. 221). The name "UKP5" is due to five improvements applied over that algorithm:

1. **Symmetry pruning**: symmetric solutions are pruned in a more efficient fashion than in (GARFINKEL; NEMHAUSER, 1972);
2. **Sparsity**: not every position of the optimal solutions value array has to be computed;
3. **Dominated solutions pruning**: dominated solutions are pruned;
4. **Time/memory trade-off**: the test $w_i \leq y$ from the algorithm in (GARFINKEL; NEMHAUSER, 1972) was removed, the trade-off was using $O(w_{max})$ memory;
5. **Periodicity**: the periodicity check suggested in (GARFINKEL; NEMHAUSER, 1972), but not implemented there, was adapted and implemented.

As already pointed out, UKP5 is very similar to the ordered step-off algorithm from (GILMORE; GOMORY, 1966). Aside for minor adaptations, this section is the same as the one presented in (BECKER; BURIOL, 2016), written when the author was not yet aware of those similarities. The discussion about the similarities between UKP5 and the DP algorithms from Gilmore and Gomory is restricted to Section 4.2.3.

A pseudocode of UKP5 is presented in Algorithm 3. We have two main data structures, the arrays g and d, both with dimension $c + w_{max} - w_{min}$. g is a sparse array where we store solutions profit. If $g[y] > 0$ then there exists a non-empty solution s with $w_s = y$ and $p_s = g[y]$. The d array stores the index of the last item used on a solution. If $g[y] > 0 \land d[y] = i$ then the solution s with $w_s = y$ and $p_s = g[y]$ has at least one copy of item i. This array makes it trivial to recover

Algorithm 3 UKP5 – Computation of *opt*

1: **procedure** UKP5($n, c, w, p, w_{min}, w_{max}$)
2: $g \leftarrow$ array of $c + w_{max} - w_{min}$ positions each one initialized with 0
3: $d \leftarrow$ array of $c + w_{max} - w_{min}$ positions, not initialized
4: **for** $i \leftarrow 1, n$ **do** ▷ Stores one-item solutions
5: **if** $g[w_i] < p_i$ **then**
6: $g[w_i] \leftarrow p_i$
7: $d[w_i] \leftarrow i$
8: **end if**
9: **end for**
10: $opt \leftarrow 0$
11: **for** $y \leftarrow w_{min}, c - w_{min}$ **do** ▷ Can end earlier because of periodicity check
12: **if** $g[y] \leq opt$ **then** ▷ Handles sparsity and prunes dominated solutions
13: **continue** ▷ Ends current iteration and begins the next
14: **end if**
15: $opt \leftarrow g[y]$
16: **for** $i = 1, d[y]$ **do** ▷ Creates new solutions (never symmetric)
17: **if** $g[y + w_i] < g[y] + p_i$ **then**
18: $g[y + w_i] \leftarrow g[y] + p_i$
19: $d[y + w_i] \leftarrow i$
20: **end if**
21: **end for**
22: **end for**
23: **for** $y \leftarrow c - w_{min} + 1, c$ **do**
24: **if** $g[y] > opt$ **then**
25: $opt \leftarrow g[y]$
26: **end if**
27: **end for**
28: **return** *opt*
29: **end procedure**

the optimal solution, but its main use is to prune solution symmetry.

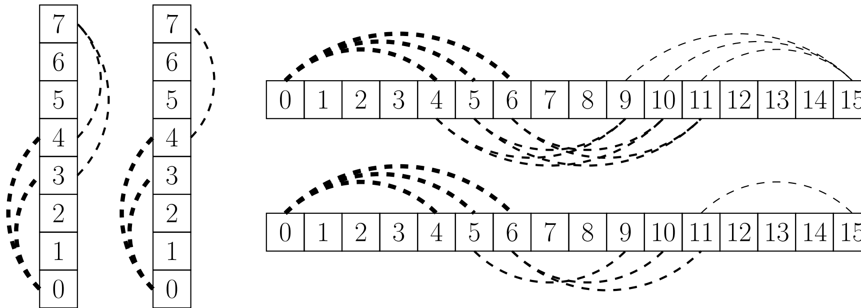
Our first loop (lines 4 to 9) simply stores all single item solutions in the arrays g and d. For a moment, let us ignore lines 12 to 14, and replace $d[y]$ (at line 16) by n. With these changes, the second loop (between lines 11 and 22) iterates g and when it finds a stored solution ($g[y] > 0$) it tests n new solutions (the combinations of the current solution with every item). The new solutions are stored at g and d, replacing solutions already stored if the new solution has the same weight but a greater profit value.

When we add the lines 12 to 14 to the algorithm, it stops creating new solutions from dominated solutions. If a solution s with a smaller weight ($w_s < y$) has a bigger profit ($p_s = opt > p_t$, where $w_t = y \wedge p_t = g[y]$), then s dominates t. If a solution s dominates t then, for any item i, the $s \cap \{i\}$ solution will dominate

the $t \cap \{i\}$ solution. This way, new solutions created from t are guaranteed to be dominated by the solutions created from s. A whole superset of t can be discarded without loss to solution optimality.

The change from n to $d[y]$ is based on the algorithm from (GARFINKEL; NEMHAUSER, 1972) and it prunes symmetric solutions. In the naive DP algorithm, if the item multiset $\{1, 2, 3\}$ is a valid solution, then every permutation of it is reached in different ways, wasting processing time. To avoid computing symmetric solutions, we enforce non-increasing order of the items index. Any item inserted in a solution s has an index that is equal to or lower than the index of the last item inserted on s. This way, solutions like $\{1, 2, 3\}$ or $\{2, 1, 3\}$ cannot be reached. However, this is not a problem because these solutions are equivalent to $\{3, 2, 1\}$, and this solution can be reached.

Figure 4.1: The solutions $\{1, 2\}$ and $\{2, 1\}$ (with $w_1 = 3$ and $w_2 = 4$) are equivalent. The solution $\{3, 2, 1\}$ can be also constructed in many ways (with $w_1 = 4$, $w_2 = 5$ and $w_3 = 6$). The figure shows the solution generation with and without symmetry pruning.



When the two changes are combined and the items are sorted by non-increasing efficiency, UKP5 gains in performance. The UKP5 iterates the item list only when it finds a non-dominated solution, i.e, $g[y] > opt$ (line 12). Undominated solutions are more efficient (larger ratio of profit by weight) than the skipped dominated solutions. Therefore, the UKP5 inner loop (lines 16 to 21) often iterates up to a low $d[y]$ value. Experimental results show that, after some threshold capacity, the UKP5 inner loop consistently iterates over only for a small fraction of the item list.

The algorithm ends with the optimal solution stored in opt. The solution assembly phase is not described in Algorithm 3, but it is similar to the one described in (GARFINKEL; NEMHAUSER, 1972, p. 221, Steps 6-8), and can be used for the already described Algorithms 1 and 2. Let y_{opt} be a capacity where $g[y_{opt}] = opt$. We add a copy of item $i = d[y_{opt}]$ to the solution, then we add a copy of item $j = d[y_{opt} - w_i]$, and so on, until $d[0]$ is reached. This phase has a $O(c)$ time complexity,

as a solution can be composed of c copies of an item i with $w_i = 1$.

4.2.4.1 A note about UKP5 performance

In the computational results section we will show that UKP5 outperforms PYAsUKP (EDUK2 original implementation) by a considerable amount of time. We grant the majority of the algorithm performance to the ability of applying sparsity, solution dominance and symmetry pruning with almost no overhead. At each iteration of capacity y, sparsity and solution dominance are integrated in a single constant time test (line 12). This test, when combined with an item list sorted by non-increasing efficiency, also helps to avoid propagating big index values for the next positions of d, benefiting the performance of the solution generation with symmetry pruning (the use of $d[y]$ on line 16).

4.2.4.2 Weak solution dominance

In this section we will give a more detailed explanation of the workings of the previously cited weak solution dominance. We use the notation $min_{ix}(s)$ to refer to the lowest index among the items that compose the solution s. The notation $max_{ix}(s)$ has analogue meaning.

When a solution t is pruned because s dominates t (lines 12 to 14), some solutions u, where $t \subset u$, are not generated. If s dominates t, and $t \subset u$, and $max_{ix}(u - t) \leq min_{ix}(t)$, then u is not generated by UKP5. For example, if $\{3, 2\}$ is dominated, then $\{3, 2, 2\}$ and $\{3, 2, 1\}$ will never be generated by UKP5, but $\{3, 2, 3\}$ or $\{3, 2, 5\}$ could yet be generated (note that, in reality, it is the equivalent $[3, 3, 2]$ and $[5, 3, 2]$ that could yet be generated). Ideally, any u where $t \subset u$ should not be generated as it will be dominated by a solution u' where $s \subset u'$ anyway. It is interesting to note that this happens eventually, as any $t \cap \{i\}$ where $i > min_{ix}(t)$ will be dominated by $s \cap \{i\}$ (or by a solution that dominates $s \cap \{i\}$), and at some point no solution that is a superset of t will be generated anymore.

4.2.4.3 Implementation details

With the purpose of making the initial explanation simpler, we have omitted some steps that are relevant to the algorithm performance, but not essential for assessing its correctness. A complete overview of the omitted steps is presented in

this section.

All the items are sorted by non-increasing efficiency and, among items with the same efficiency, by increasing weight. This speeds up the algorithm but does not affect its correctness.

A periodicity bound is computed as in (GARFINKEL; NEMHAUSER, 1972, p. 223) and used to reduce the c value. We further proposed an UKP5-specific periodicity check that was successfully applied. This periodicity check is not used to reduce the c capacity before starting UKP5, as y^*. The periodicity check is a stopping condition inside UKP5 main loop (lines 11 to 22). Let y be the value of the variable y in line 11, and let y' be the biggest capacity where $g[y'] \neq 0 \wedge d[y'] > 1$. If at some moment $y > y'$ then we can stop the computation and fill the remaining capacity with copies of the first item (that has index 1). This periodicity check works only if the first item is the best item. If this assumption is false, then the described condition will never happen, and the algorithm will iterate until $y = c - w_{min}$ as usual. The algorithm correctness is not affected.

There is an *else if* test at line 20. If $g[y + w_i] = g[y] + p_i$ and $i < d[y + w_i]$ then $d[y] \leftarrow i$. This may seem unnecessary, as appears to be an optimization of a rare case, where two distinct item multisets have the same weight and profit. Nonetheless, without this test, UKP5 was about 1800 (one thousand and eight hundreds) times slower on some subset-sum instance datasets.

4.2.5 EDUK

The EDUK (Efficient Dynamic programming for the Unbounded Knapsack problem) is a complex DP algorithm for the UKP, first mentioned in (POIRRIEZ; ANDONOV, 1998). However, only in (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000) the algorithm essentials were described for the first time. The author of this thesis, however, is partial to the algorithm description to be found in (KELLERER; PFERSCHY; PISINGER, 2004, p. 223). Some basic ideas used by EDUK were already exposed by a simple and functional-oriented algorithm proposed in (ANDONOV; RAJOPADHYE, 1994)[2]. Before EDUK2 was proposed, EDUK was con-

[2]The author of this thesis tried to implement this simple and functional-oriented algorithm in Haskell and in C++. Both codes had a very poor performance and were not even considered for the experiments. The author of this thesis admits that the reason of the poor performance could be the poor quality of his implementations. The C++ code can be accessed

sidered by some the state-of-the-art DP algorithm for the UKP. An example is the comment in (KELLERER; PFERSCHY; PISINGER, 2004, p.): "[...] EDUK [...] seems to be the most efficient dynamic programming based method available at the moment.".

A version of the original code of the EDUK and EDUK2 algorithms is available here[3]. Unfortunately, this version is not stable and has some bugs. Consequently, the author of this thesis recommends the use of the version available here[4]. We were given access to the latter version by Vincent Poirriez in January 11th, 2016.

It is important to admit that we do not have full understanding of the EDUK algorithm inner workings. The original code is written in OCaml (a functional language), and we had difficulties in our attempts to analyze it. Furthermore, EDUK is a more complex algorithm than any other algorithm described in this chapter (with the obvious exception of EDUK2). A basic overview of the EDUK algorithm essentials is given here, in which we recommend the sources mentioned in the first paragraph of this section for the reader who is interested in a deeper analysis.

The authors of EDUK cite threshold dominance (that generalizes collective, multiple and simple dominances), a sparse representation of the iteration domain and the periodicity property to explain the efficiency of the algorithm. In (KELLERER; PFERSCHY; PISINGER, 2004, p. 223 to 227), the reasons given are "the detection of various notions of dominance not only as a preprocessing step but also during the dynamic programming computation" and "the test for collective dominance of an item type by a previously computed entry of the dynamic programming array". The EDUK algorithm sorts the item list in increasing weight order, differently from the majority of the algorithms for the UKP that use the non-increasing efficiency order.

The sparse representation of the iteration domain is achieved by using lazy lists (a functional programming concept) instead of an array of size c (or more) to store the solutions. Consequently, the memory use is less dependent of c and w_{max} than other DP algorithms. In (ANDONOV; RAJOPADHYE, 1994), where the sparse representation idea was first presented, the solutions are represented as pairs

in <https://github.com/henriquebecker91/masters/blob/663324e4f071b5bca22ab5301e29273b9db88a41/codes/cpp/lib/eduk.hpp>, and the Haskell code can be accessed in <https://github.com/henriquebecker91/masters/blob/f5bbabf47d6852816615315c8839d3f74013af5f/codes/hs/ukp.hs>.

[3] PYAsUKP official site: <http://download.gna.org/pyasukp/pyasukpsrc.html>

[4] The repository of this master's thesis: <https://github.com/henriquebecker91/masters/blob/f5bbabf47d6852816615315c8839d3f74013af5f/codes/ocaml/pyasukp_mail.tgz>.

of weight and profit value, as the solution was a pseudo-item (i.e. a set of items that can be treated as it was a single item). For example, a solution s consisting of the items i and j is represented by the following pair: $(w_i + w_j, p_i + p_j)$. Adding an item to a solution is equivalent to adding the weight and profit values of a pair to another. For some instances, especially the ones with big w_{min} and c values, this sparse representation allows for saving time and memory. In UKP5, for example, the algorithm allocates memory, initializes, and iterates over many capacities y that are accessed and then skipped immediately because a solution s with $w_s = y$ does not exist. In EDUK, such skipped capacities are never explicitly enumerated to begin with, and no memory is allocated for them, or time used iterating over them. A similar effect could be obtained in UKP5 by using a `std::map` instead of an `std::vector` for the data structures g and d .

Both the item list and the knapsack capacities are broken down and evaluated in slices. Each slice of the item list, beginning with the ones with smallest items, is then processed. The items inside the current slice are combined with each other to generate solutions with weight up to the slice w_{max}. The solutions are used to test collective dominance between the items inside the slice and in the next slices. A global list of the undominated items is kept, and after an item is dominated, it is not used again. After evaluating all slices of the item list, and if $c > w_{max}$, EDUK begins to evaluate slices of the capacity values, using the items that were not eliminated by the collective dominance tests. After each one of those capacity slices, the EDUK algorithm tests the items for threshold dominance, potentially removing some of them from the global list of undominated items. If this list ends up consisting only of the best item (that can never be threshold dominated), then the EDUK stops and fills the remaining capacity with copies of the best item. Otherwise, EDUK solves the UKP up until capacity c.

4.3 Branch-and-Bound

The B&B approach was established in the seventies as the most efficient approach for solving the UKP, what is greatly a consequence of the datasets, items distributions, and generation parameters used at the time. The author of this thesis believes that this claim was first made in (MARTELLO; TOTH, 1990a), and then other papers as (BABAYEV; GLOVER; RYAN, 1997) began repeating it. The fact

that only the code for MTU1 and MTU2 was readily available also did not help the situation, as some began to claim that MTU2 was the *de facto* standard algorithm for the UKP, see (POIRRIEZ; ANDONOV, 1998).

The author does not intend to give a complete introduction to the B&B approach, but he believes a quick overview is in order. The B&B approach can be seen as an improvement of the brute-force approach. The brute-force approach consists in exhaustively checking all solutions in the search space. A B&B algorithm will keep track of the best solution found so far. In the case of a maximization problem like the UKP, this solution is called a lower bound on the optimal solution (in the sense that 'the optimal solution is at least this good'). A B&B algorithm will divide the search space in two or more (often exclusive) subdivisions. In the case of the UKP, an example would be 'all solutions with 4 or less copies of item i' and 'all solutions with 5 or more copies of item i'. An optimistic guess for the best value to be found in each subdivision of the search space is computed: these are called upper bounds. An upper bound is a value that is guaranteed to be equal to or greater than the value of the best solution to be found in the correspondent subdivision of the search space. The subdivisions are recursively and systematically divided in smaller subdivisions. If the upper bound of any subdivision is smaller than or equal to the global lower bound, then the solutions of that subdivision of the search space do not need to be examined (i.e. the best solution known at the moment is already equal to or better than any solution that can be found in that subdivision of the search space). If, by the use of the lower and upper bounds, the B&B algorithm obtains proof that no solution in all the search space can be better than the best solution found so far, the B&B algorithm stops.

Regarding the explanation above, one thing should be clear: the quality of a B&B algorithm is directly correlated with the quality of its bounds. Also, the time taken to solve an instance will vary based on how much of the search space can be safely skipped/ignored by the use of the bounds. The time taken by a B&B algorithm over an instance of the UKP can be hard to predict, and is not very dependent on the magnitude of n and c, but more dependent on the item distribution. In the worst case, a B&B algorithm cannot eliminate a significant portion of the search space by the use of bounds and then, consequently, it needs to examine all search space. In the case of the UKP, the search space is clearly combinatorial (all possible items combinations that fit the knapsack), so the worst-case of an B&B approach

for the UKP can be exponential.

The memory use of a B&B algorithm can follow its worst case and be exponential too, as for many times a tree is used to keep the enumeration of the subdivisions. However, some optimized algorithms can avoid enumerating the tree explicitly, and keep a constant memory use linear in n (even in the worst case). The B&B algorithms for the UKP often are not affected by the magnitude of c, however they can be affect by how close c is from a capacity that can be entirely filled by copies of the best item. The B&B algorithms for the UKP will often solve the problem instantly if $c \ mod \ w_b$ is small, because the greedy heuristic lower bound will probably be optimal, and will exclude the remaining search space easily.

The fact that this approach is not significantly affected by huge values of n and c, and more by the distribution used, makes it clear why it was considered the best approach in the seventies. The datasets used back then had large n and c values, and items distributions that made easy to exclude large portions of the search space with the greedy lower bound solution (the uncorrelated distribution is the perfect example).

4.3.1 MTU1

The MTU1 algorithm is a B&B algorithm for the UKP that avoids the explicit unfolding of the typical B&B tree (MARTELLO; TOTH, 1977). The implicit tree used by MTU1 is described in what follows, as this makes the algorithm easier to visualize and understand. The MTU1 sorts the items in non-increasing efficiency order before beginning. Such ordering is needed to assure the correctness of the bounds and, consequently, of the algorithm itself. In the algorithms description, it is to be assumed that the items are ordered in the mentioned order

The implicit enumeration tree of MTU1 has $n + 1$ levels. The root node represents all the search space, for convenience the author will consider it level zero. The first level of the tree contains $\lfloor \frac{c}{w_1} \rfloor + 1$ nodes. Each of those nodes represents a subdivision of the search space where the solution has a specific number of copies of the first item (from zero to $\lfloor \frac{c}{w_1} \rfloor$ copies). The nodes of the second level subdivide the search space by the amount of copies of the second item in a solution, and so on (until the last item, in level n). From the second level on, the levels have a variable number of nodes. There are $\lfloor \frac{c}{w_2} \rfloor + 1$ nodes in the second level that are children of

the first level node that represents the solutions with zero copies of the first item, this because as all of the knapsack capacity c is empty. Consequently, there are only $\lfloor \frac{c \ mod \ w_1}{w_2} \rfloor + 1$ nodes in the second level that are children of the first level node that represents the solutions with $\lfloor \frac{c}{w_1} \rfloor$ copies of the first item.

The MTU1 algorithm can be seen as the application of a modified depth-first search over the implicit tree described above. In each level, the first node to be visited will always be the one representing the use of the greatest amount of copies of the current level item type, and the last node to be visited will be the one representing zero copies. This visiting order, together with the items non-increasing efficiency order, result in an intuitive behaviour: the first solutions tried will be the ones with the greatest amount of copies of the most efficient items.

As it is common in B&B algorithms, at each visited node MTU1 computes an upper bound for the tree below the current node and, if this upper bound is equal to or lower than the global lower bound, MTU1 will skip the subtree and backtrack to the parent node. These upper bounds consist of a solution with the amount of items already described by the path between the root node and the current node, and a pseudo-item with weight equal to the capacity gap and efficiency equal to the efficiency of the next level item type.

When visiting a leaf node, MTU1 will check if the solution described by the path from root to the leaf node is better than the lower bound, and update the lower bound if this is the case.

For convenience and to reduce the size of the tree, if the capacity gap left by a node is smaller than w_{min}, then that node is a leaf node (no need for a list of nodes indicating zero copies of the remaining item types). Consequently, every node (leaf or not) represents a unique solution (denoted by the path from the root node to it). As the nodes/solutions are visited in a systematic order, for any given node/solution, it is possible to know what part of the 'search space'/tree was already visited or skipped, and what part has not yet been explored. Consequently, the tree does not need to be enumerated explicitly, the current node/solution is sufficient to know which solutions should be tried next.

4.3.2 MTU2

The MTU2 algorithm was first proposed in (MARTELLO; TOTH, 1990a). The objective of MTU2 is to improve MTU1 run time when it is used in very large instances (e.g. up to 250,000 items). MTU2 calls MTU1 internally to solve the UKP: it can be seen as a wrapper around MTU1 that avoids unnecessary computational effort. The two main factors that motivated the creation of MTU2 were: 1) for the majority of the instances used in the period[5], an optimal solution is composed of the most efficient items; 2) for some of those instances, sorting the entire items list was more expensive than solving the instance with MTU1.

The explanation of the inner workings of the MTU1 (Section 4.3.1) should make it easier to understand how solving the UKP with MTU1 can require less time than the sorting phase, for instances with the characteristic above mentioned (i.e. only the most efficient items are present in an optimal solution). MTU1 first investigates the regions of the search space with the biggest amount of the most efficient items. If instances with a large amount of items have optimal solutions among the first ones tested by MTU1, then the implicit enumeration tree will never be explored in depth, and the vast majority of the items will be ignored. The time spent sorting any items other than the most efficient ones was unnecessary.

To address this waste of computational effort, and to solve even larger instances, MTU2 was proposed. MTU2 is based on the concept of 'core problem' that was already introduced in other works of the period, such as (BALAS; ZEMEL, 1980). Informally, the core problem would be a knapsack instance sharing an optimal solution, the knapsack capacity, and a small fraction of the items list with the original instance.

The size of the core problem cannot be defined a priori. Consequently, the idea is to guess a value k, where $k \leq n$, find and sort only the k most efficient items, and then solve this tentative core problem (in the specific case of MTU2, the solver used is MTU1). If the optimal solution value of the tentative core problem is equal to an upper bound for the full instance, then the algorithm has found an optimal solution of the full instance and can stop. Otherwise, the algorithm uses the solution found as a lower bound to remove items outside of the tentative core problem. For

[5]For an example, one of the datasets of the paper that introduced MTU2 was analyzed in Section 3.1.2.

each item j that is not in the core problem, an upper bound is computed over the solutions with one single copy of item j. If this upper bound is equal to or smaller than the value of the lower bound, then the item can be discarded without loss to the optimal solution value. If all items not in the tentative core problem are discarded by this procedure, the algorithm also stops. Otherwise, more items from outside the core problem are added to it, and the process restart with a larger core problem, and the reduced item list outside of it.

4.3.3 Other B&B algorithms

Some B&B algorithms were not implemented or deeply studied by the author of this thesis, but are cited here for completeness. In (CABOT, 1970), a B&B algorithm for the UKP is presented. In that period, the B&B algorithms were often referred to as 'enumeration algorithms'. As already said in Section 2, Cabot's algorithm was indirectly compared with MTU1 in (MARTELLO; TOTH, 1977), and MTU1 has shown better times. However, it would be interesting to see a comparison with more extensive and recent datasets.

Another B&B algorithm is proposed in (GILMORE; GOMORY, 1963). The author is not aware of any work where the proposed algorithm was compared to any other algorithms. Three years after proposing this algorithm, the same authors wrote (GILMORE; GOMORY, 1966), which focused on the one-dimensional and two-dimensional knapsack problems. This last paper presented four variations of a DP algorithm for the UKP, but does not appear to have mentioned the old B&B algorithm.

4.4 Hybrid (DP and B&B)

As expected, some algorithms try to combine the best of two most popular approaches (DP and B&B) for better results.

4.4.1 GREENDP

The algorithm presented in (GREENBERG; FELDMAN, 1980) is an improvement on the ordered step-off from (GILMORE; GOMORY, 1966). It is very similar to UKP5. The author does not know if it could be defined as a hybrid, but a good definition for it would be a 'DP algorithm with bounds'. The algorithm was not named in the paper and will be called GREENDP for the rest of the thesis. The implementation of the GREENDP made by the author of this thesis, and used in the experiments (Section 5), will be called MGREENDP (Modernized GREENDP, in the sense that the algorithm now uses loops instead of the `goto` directive).

The GREENDP algorithm consists in solving the UKP by using the ordered step-off algorithm, but without using the best item in the DP, and with interruptions at each w_b capacity positions, for checking if the DP can be stopped and the remaining capacity filled with copies of the best item. In those interruptions, two bounds are computed. A lower bound for solutions using the best item is computed by combining the current best solution of the DP with as many copies of the best item as possible. An upper bound for solutions not using the best item is computed by combining the current best solution of the DP with a pseudo-item that would fill the entire capacity gap and has the same efficiency as the second best item (it could also be seen as solving a continuous relaxation of the UKP without the best item, and only for the remaining capacity). If the algorithm discovers that the lower bound with the best item is better than the upper bound without the best item, then the lower bound solution is optimal and the DP can be stopped.

This approach using bounds is fundamentally different from the periodicity check used by UKP5 (or the periodicity check used by the 'terminating step-off'). For example, the use of bounds save computational time of GREENDP when it is used to solve BREQ instances, the periodicity check do not save computational time of UKP5 when it is used to solve the same instances (see experiments of the Section 5.3). However this seems to have an impact on other families of instances (see experiments of the Section 5.2).

A weakness of this bounds calculation is that it fails if the two most efficient items have the same efficiency. In this case, the algorithm would be the same as running UKP5 with a little overhead.

4.4.2 EDUK2

The EDUK2 algorithm was proposed in (POIRRIEZ; YANEV; ANDONOV, 2009), and it is an hybridization of EDUK (a DP algorithm) and MTU2 (a B&B algorithm). The author of this thesis gives here a quick overview of the hybridization, but more details can be found in the paper above mentioned. Just as with EDUK, the author does not claim to fully comprehend the EDUK2 internals, and only summarizes what is said in the original paper. The author recommends reading Sections 4.2.5 (EDUK) and 4.3.2 (MTU2) before the explanation below, as it is strongly based in both algorithms. The comments made about EDUK code in its own section also apply to EDUK2.

The description of the changes in EDUK caused by the hybridization follows. The $k = min(n, max(100, \frac{n}{100}))$ most efficient items are gathered in a tentative core problem. A B&B algorithm "similar to the one in MTU1"[6] tries to solve the tentative core problem. This B&B algorithm has the possibility of choosing among three bound formulas, and stops after exploring $B = 10,000$ nodes (of the implicit enumeration tree). If the B&B algorithm returns a solution with value equal to an upper bound for the whole instance, then the DP algorithm never executes. Otherwise, the solution given by the B&B algorithm is used as a global lower bound in the hybridized EDUK algorithm. The hybridized EDUK algorithm works like EDUK would do, with the addition of an extra phase between the slices. The extra phase uses the lower bound to eliminate items *and solutions for lesser capacities* from the algorithm. This phase is very similar to a phase of MTU2: an upper bound is computed for solutions using one copy of the respective item (a solution s can be treated as a pseudo-item (w_s, p_s)). If this upper bound is equal to or smaller than the global lower bound, then the item (or solution) is abandoned by the algorithm. A new lower bound is computed for each solution that was not removed by the process described above. The lower bound consists in filling the remaining capacity with a greedy algorithm. If this new lower bound is better than the global lower bound, it replaces it.

[6]Quoted from (POIRRIEZ; YANEV; ANDONOV, 2009).

4.5 Consistency Approach

The Consistency Approach (CA) consists in combining both the objective function (i.e. *maximize* $p_i x_i$) and the UKP only constraint (i.e. $w_i x_i \leq c$) in a single Diophantine equation[7] ($\alpha_i x_i = \beta$, where α_i are coefficients computed for each item, and β is the analogue of an optimal solution upper bound). The combination procedure preserves the set of valid solutions, consequently, an optimal solution for the UKP can be sought by testing values for the equation variables/unknowns until the equation holds true. Such variables are often the quantity of each item in a solution (x_i) (on one side of the equation) and a tentative value derived from the optimal solution upper bound (β) (on the other side of the equation).

4.5.1 GREENDP1

The only algorithm implemented by the author of this thesis that uses the consistency approach is the first algorithm described in (GREENBERG, 1986). The algorithm, that was not named in the original paper, will be called GREENDP1 for the rest of the thesis (because it is the first of the two algorithms from the same paper, and because GREENDP is an older algorithm by the same author). The implementation of GREENDP1 made by the author of this thesis, and used in the experiments section, will be called MGREENDP1 (Modernized GREENDP1, in the sense that the algorithm now use loops instead of the `goto` directive). The algorithm was meant to be a theoretical experiment, and did not have performance as a priority.

The basic idea of the GREENDP1 algorithm consists in encoding both profit and weight in a single integer value. Let us call the value given by this encoding for a weight and profit pair, the *coefficient* of such pair. If the coefficient of two items is summed, the result is a coefficient that encodes the weight and profit value of the solution composed of the two items. The algorithm then enumerates all valid solutions by adding the items coefficients to each other, and to the coefficients of the solutions it creates[8]. The enumeration process is very similar to a basic DP algorithm

[7] "A Diophantine equation is an equation in which only integer solutions are allowed." (WEISSTEIN, 2016). In other words, an equation where the values of the variables/unknowns are restricted to the integer numbers.

[8] It is important to note here, for future research in the subject, that in (GREENBERG, 1986),

with solution symmetry pruning, and dispensable extra arrays. After the process of enumerating all valid solutions, the algorithm has to find the optimal solution. The algorithm will start with an upper bound on the optimal solution value, and will check if some solution has this profit value (in $O(1)$), if not, it will decrease the bound in one unity, and repeat the process. A DP algorithm with solution symmetry pruning would probably outperform MGREENDP1 in most instances.

4.5.2 Babayev's algorithm

In (BABAYEV; GLOVER; RYAN, 1997), a CA algorithm for the UKP designed with performance on mind was proposed. Unfortunately, the author did not obtain access to the code, and the algorithm demanded considerable time and effort to implement[9]. The algorithm is similar to GREENDP1, but presents some improvements: the encoding tries to create the smallest coefficients as possible; the enumeration of the solutions (or 'consistency check') can be done using the Dijkstra's algorithm (shortest path, $O(n+\alpha^2)$), or a method for solving group problems developed in (GLOVER, 1969) ($O(n \times \alpha)$). As the algorithm can choose between the method with the best complexity for a given instance, its overall complexity is $O(min(n+\alpha^2, n \times \alpha))$, where α is the value of the smallest coefficient for an item of the instance.

4.6 Other approaches

The UKP is an optimization problem that have one single constraint and, consequently, is simple to model. Probably, many approaches could be applied to it, and many will, even if with only theoretical interest. The list presented in this chapter has no intent of being exhaustive, but only to give the readers a base to

---

the algorithms description is incomplete. The author believes that on step 2d of the first algorithm, the assignment 'D(z) = k' should be executed together with the other assignments; and on step 2d of the second algorithm, the assignment 'D(x) = k' should also be executed together with the other assignments. If these statements are not included, then the algorithm cannot backtrack the items that form an optimal solution.

[9]The author started to implement the algorithm, but because of time restraints had to abandon the implementation. The current state of the implementation can be found at <https://github.com/henriquebecker91/masters/blob/2623472fad711bac10bf4d34c437b24b3fd7f30f/codes/cpp/lib/babayev.hpp>.

understand most of the discussion carried here.

Examples of approaches that were not covered in this chapter are the *shortest path formulations* and *group knapsack formulations*. Such approaches are discussed in (GARFINKEL; NEMHAUSER, 1972, p. 239), and (KELLERER; PFERSCHY; PISINGER, 2004), and internally used by the algorithm described in (BABAYEV; GLOVER; RYAN, 1997), mentioned last section. The author finds both approaches to be similar to DP, in the sense that they are more like ways to interpret the problem and then solve it by a correspondent DP algorithm, and less like B&B, which is a completely different approach when compared to DP.

5 EXPERIMENTS AND ANALYSES

In this chapter, five experiments are presented and their results analyzed. The first experiment is an updated version of the experiment first presented in (BECKER; BURIOL, 2016), which consisted in the execution of UKP5 and EDUK2 over the PYAsUKP dataset (see Section 3.2). The experiment presented in this thesis includes one extra algorithm (GREENDP), and does not use parallel execution. The dataset used is exactly the same.

The second experiment consists in the execution of MTU1 (Fortran), MTU1 (C++), MTU2 (Fortran), and MTU2 (C++) over the reduced PYAsUKP dataset (see Section 3.2.6).

The third experiment consists in the execution of all algorithms presented in Section 4 (except by the naïve DP and the Garfinkel's algorithm) over the instances of the BREQ 128-16 Standard Benchmark (see Section 3.4).

The fourth experiment execute MTU1 (C++), CPLEX, and four variants of the UKP5 to solve the pricing subproblem described in Section 3.3. The fifth and last experiment addresses some concerns raised by the comments of an anonymous referee in the peer review of (BECKER; BURIOL, 2016). The concerns are related to the effects of the shared memory cache in the comparison of the algorithms run times, when more than one run is executed at the same time. The results of this experiment justify the setup used in the four previous experiments (i.e. serial runs).

5.1 Setup of the first four experiments

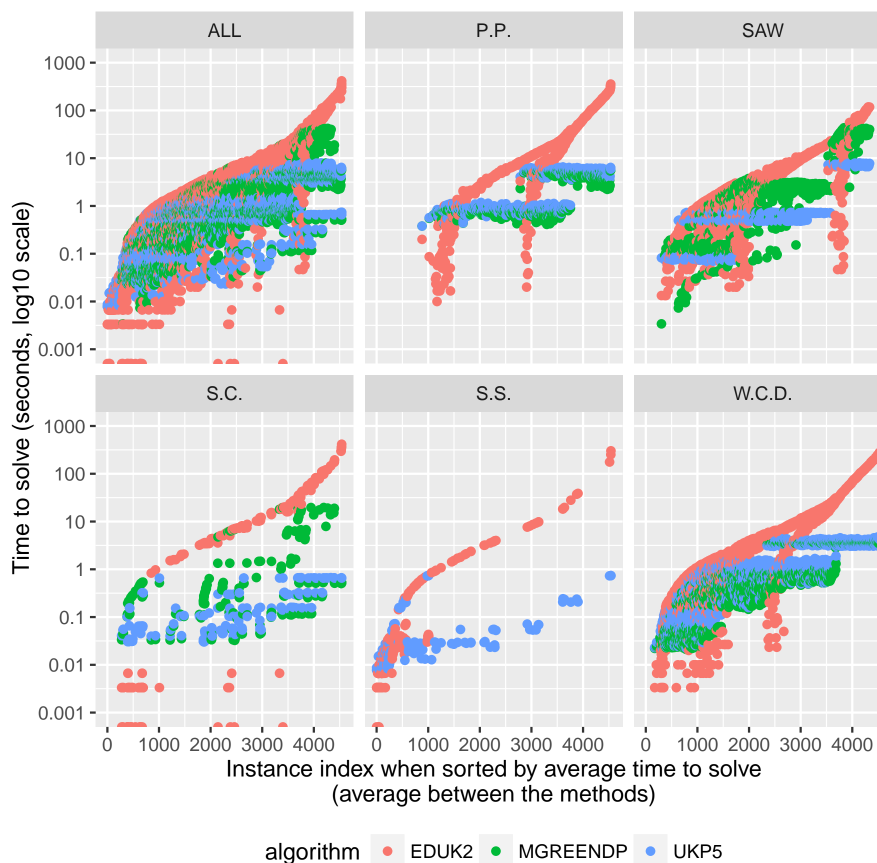
The experiments were run using a computer with the following characteristics: the CPU was an Intel® Core™ i5-4690 CPU @ 3.50GHz; there were 8GiB RAM available (DIMM DDR3 Synchronous 1600 MHz) and three levels of cache (256KiB, 1MiB and 6MiB, where only L3 is shared between cores). The operating system used was GNU/Linux 4.7.0-1-ARCH x86_64 (i.e. Arch linux). Three of the four cores were isolated using the *isolcpus* kernel flag (the non-isolated core was left to run the operating system). The *taskset* utility was used to execute runs in one of the isolated cores. All runs were executed in serial order (i.e. two algorithm executions did not coexisted), this choice was made for reasons explained in Section 5.5. C++ code was compiled with GCC and the *-O3 -std=c++11* flags enabled. The OCaml

code (PYAsUKP/EDUK/EDUK2) was compiled with the flags suggested by the authors of the code for maximum performance.

5.2 Solving the PYAsUKP dataset

The first experiment is an updated version of the experiment first presented in (BECKER; BURIOL, 2016), which consisted in the execution of UKP5 and EDUK2 over the PYAsUKP dataset. The experiment presented in this thesis includes one extra algorithm (GREENDP), and does not execute any runs in parallel. The dataset used in this experiment is the same used in (BECKER; BURIOL, 2016), which is presented in Section 3.2).

Figure 5.1: Benchmark using PYAsUKP dataset; the UKP5, GREENDP and EDUK2 algorithms; and no timeout.



Source: the author.

In Figure 5.1), the subset-sum (S.S.) plot does not show the run times of the MGREENDP algorithm. This happens because it fails automatically over instances where the two most efficient items have the same efficiency (what always

happens at subset-sum instances). We could fix this problem, but this would disable MGREENDP bounds check (which it is its main feature) and make the algorithm even more similar to UKP5. So we opted for not running MGREENDP over those instances.

Figure 5.1 shows that, except for Strong Correlated (S.C.) and S.S., the run times of the EDKU2 runs often show two main trends. The first trend starts at the left of the x axis and covers UKP5 and MGREENDP (i.e. no run of UKP5 or MGREENDP has a run time above this trend). The second trend begins at middle/right of the x axis and is composed of run times significantly smaller than UKP5 and MGREENDP. Both trends merge in a single trend that points to the top right corner of the chart. UKP5 and MGREENDP present similar run times that form plateaus. Those plateaus aggregate instances that take about the same time to solve by UKP5/MGREENDP.

The author believes that the second EDUK2 trend is formed by instances where the B&B preprocessing phase had considerable success (stopped the computation or used bounds to remove items before executing the DP). This would also explain the lack of this trend at the S.C. and S.S. instance classes, where such bounds and tests have less effect (the S.S. instances do not have different efficiency between items and, consequently, are little affected by bounds). When EDUK2 B&B phase do not solve the instance or help to greatly reduce the amount of states, the run times of the EDUK2 DP phase are greater than the UKP5 and MGREENDP times.

The plateaus formed by the UKP5 and MGREENDP runs show very little variation of the run times among some groups of instances. A closer examination of the data reveals that the instance groups (plateaus) aggregate instances with the same number of items (for the same distribution), or instances with different number of items that are a magnitude smaller (or greater) than the numbers of items of the plateau(s) above (or below). This behaviour shows that UKP5 and MGREENDP1 are little affected by the specific items that constitute an instance, and that the instance size and distribution are good predictors of the UKP5/MGREENDP run time.

EDUK2 seems to have a much greater variation between the run times for instances with similar number of items of the same distribution. We can see a line with a slope of about 45 degrees close to the top right corner of the charts. This line is over a logarithmic y axis, so it is a huge variation compared to the UKP5

plateau below it, which is solving the same instances (these instances, as we have seen above, have similar number of items).

The run times of UKP5 and MGREENDP are very similar, especially in the datasets Without Collective Dominance (W.C.D) and Postponed Periodicity (P.P.), in which MGREENDP has a small advantage. This should come at no surprise as they are similar algorithms. What is surprising is its behaviour at SAW and S.C. instance classes, in which the situation is reversed, and MGREENDP takes considerably more time than UKP5 in many cases. The behaviour at those two classes can be explained by three main factors.

The first factor is a characteristic of the optimal solutions from the SAW and S.C. instances. The optimal solutions of these distributions are composed of about $\frac{c}{w_b}$ copies of the best item and a single copy of another item. This happens because, in such distributions, the smallest item is the best item (as already pointed out in Sections 3.2.2 and 3.2.5). This characteristic gives a big importance to the best item in those instances, and together with the next two factors, it explains the algorithms behavior.

The second factor is that MGREENDP solves the pricing subproblems without the best item. The algorithm does that to allow for a mechanism that periodically checks if it can stop the computation and fill the remaining knapsack capacity with copies of the best item. Solving the DP subproblems without the best item weakens the effect of the solution dominance applied by MGREENDP (also used by UKP5). Some solutions that would be never be generated in UKP5 will be generated in MGREENDP, since better solutions (using the best item) do not exist to dominate those inferior solutions.

The third and last factor is the periodicity check applied by UKP5. As with the MGREENDP mechanism discussed above, if it finds that the remaining capacity can be filled with copies of the best item, then UKP5 is stopped. As this check does not remove the best item from the item list, it results in less overhead than the MGREENDP test for those instances. The UKP5 periodicity check benefited 222 of the 240 S.C. instances and 972 of the 1100 SAW instances.

In summary, for the instances of the PYAsUKP dataset, UKP5 and MGREENDP often need less time than EDUK2 to solve an instance. PYAsUKP solves some instances in less time than UKP5 and MGREENDP, but it takes much more time than UKP5/MGREENDP to solve the greatest instances. For some distributions, UKP5

takes slight more time than MGREENDP; for other distributions, MGREENDP takes considerably more time than UKP5. It should also be noted that MGREENDP needs a workaround to work with distributions that allow the highest efficiency to be shared by many items.

5.2.1 MTU1 and MTU2 (C++ and Fortran)

This subsection serves two purposes. The first purpose is to show that the implementations of MTU1 and MTU2 written by the author of this thesis, in C++, are on par with the original ones, written in Fortran77, and therefore the C++ implementations can be used in the remaining experiments. The second purpose is to complement the comparison presented in last section by showing that both algorithms (independent of the implementation) are not competitive with UKP5, MGREENDP, and EDUK2 when executed over the PYAsUKP dataset.

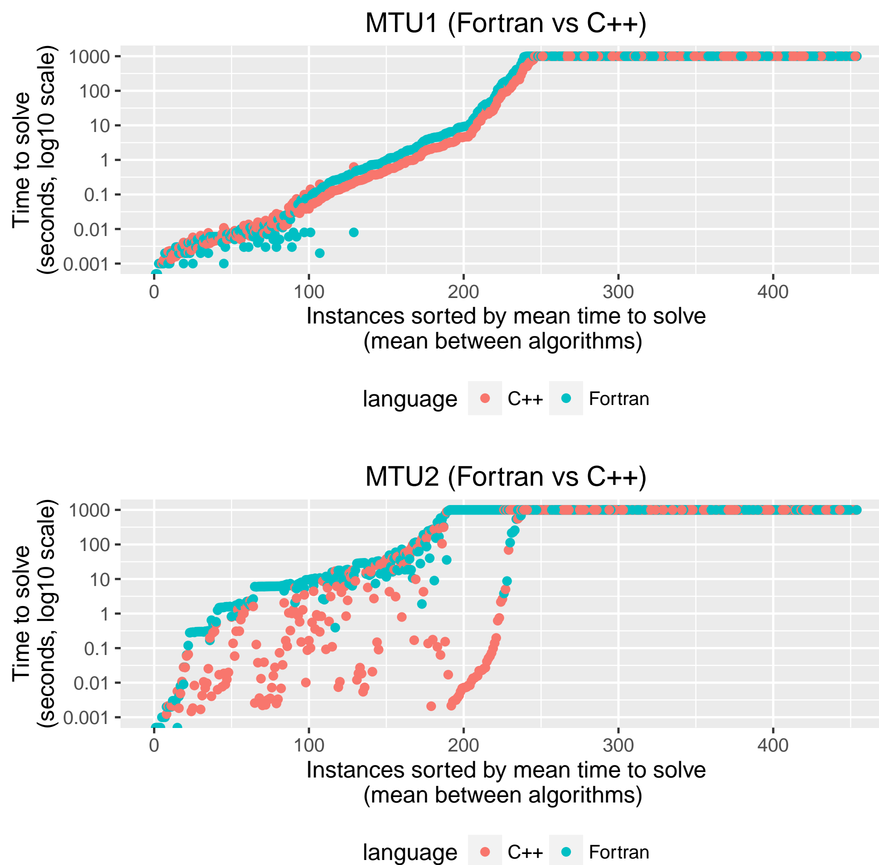
The Fortran codes used were not exactly the original MTU codes. The only difference to the original code was that any 32 bits integers or float variables/parameters were replaced by their 64 bits counterparts.

The two implementations of the MTU1 algorithm have no significant differences (besides the programming language). The only significant difference between the MTU2 implementations was the algorithm used to partially sort the items array. The original algorithm described in (MARTELLO; TOTH, 1990a) did not specify the exact method for performing this partial sorting. The original implementation used a complex algorithm developed by one of the authors of MTU2 in (FISCHETTI; MARTELLO, 1988) to find the kth most efficient item in an unsorted array (and then sort). Our implementation uses the `std::partial_sort` procedure of the standard C++ library `algorithm`. Our implementation also checks if the entire vector is sorted before starting to execute the partial sorts, and it sorts by non-increasing efficiency and if tied by non-decreasing weight. The author of this thesis has made available the exact codes used in the experiment and the version of the compiler used[1].

[1] The exact version of the code used for compilation is available in <https://github.com/henriquebecker91/masters/tree/42ecda29905c0ab56c03b7254b52bb06e67ab8d7>. The code was compiled using the available Makefiles, and both the gcc and gcc-fortran versions were the 6.1.1 (2016-06-02). The Makefile pass the *-O3* flag for both gcc and gcc-fortran. The Fortran implementation reads the same instances from a simplified format, which preserves the order of the instance item list. The binary ukp2sukp.out (codes/cpp) was used to convert from one format to the other.

The test was performed with the reduced PYAsUKP benchmark (see Section 3.2.6). The author tried to execute the four B&B algorithms over the entire PYAsUKP dataset, but many runs were ended by timeout (run time greater than 1000 seconds), and executing the four B&B codes over the 4540 instances would take time from more relevant experiments. Executing the four B&B codes over the reduced PYAsUKP dataset suffices to show that they cannot compete with UKP5, MGREENDP, and EDUK2 (compared in Section 5.2) if they were executed over the entire PYAsUKP dataset.

Figure 5.2: Comparison between MTU1 and MTU2, C++ and Fortran implementations.



Source: the author.

We can see that the MTU1 implementations had a very small variation, with the C++ version being slightly faster. The MTU2 implementation had a bigger variation. We do believe that this variation is caused by the small difference in ordering and the difference of sorting algorithms.

There is a trend between the values 180 and 230 of the x axis (CPP-MTU2). This trend is composed by the subset-sum instances. The subset-sum instances are

always naturally sorted by efficiency as their efficiency is the same for all items (the weight and profit are equal). The instance when generated by PYAsUKP is also sorted by non-decreasing item weight, what makes it perfectly sorted for our C++ implementation. The Fortran implementation does not seem to work well with this characteristic of the subset-sum instances. We have chosen to use the C++ implementation in our experiments. The reasons are: the C++ implementation can read the same format used by PYAsUKP (and the other algorithms); with the exception of PYAsUKP (which uses OCaml) all other methods use C++ and the same structures; choosing the Fortran implementation would considerably harm MTU2 run times for something that seems to be a minor implementation detail.

5.2.2 Algorithms implemented but not used

Harold Greenberg wrote a paper with two other methods for solving the UKP, (GREENBERG, 1986), and it was published after (GREENBERG; FELDMAN, 1980) (who proposed the algorithm GREENDP). The author of this thesis found interesting to implement those algorithms too, as they used different approaches[2], and also because they were from the same author that already designed an efficient algorithm. However, the first algorithm proved itself very time- and memory-consuming, and the second algorithm did not work for all cases (could not be considered an exact method without using a backup solver to solve instances where the method failed). After some tests it became clear that the intent of the paper was a theoretical exploration of new approaches, not proposing efficient algorithms, and the algorithms were discarded from experimentation over this benchmark instances. Readers who are interested in those methods can check implementations of them, made available at Github[3].

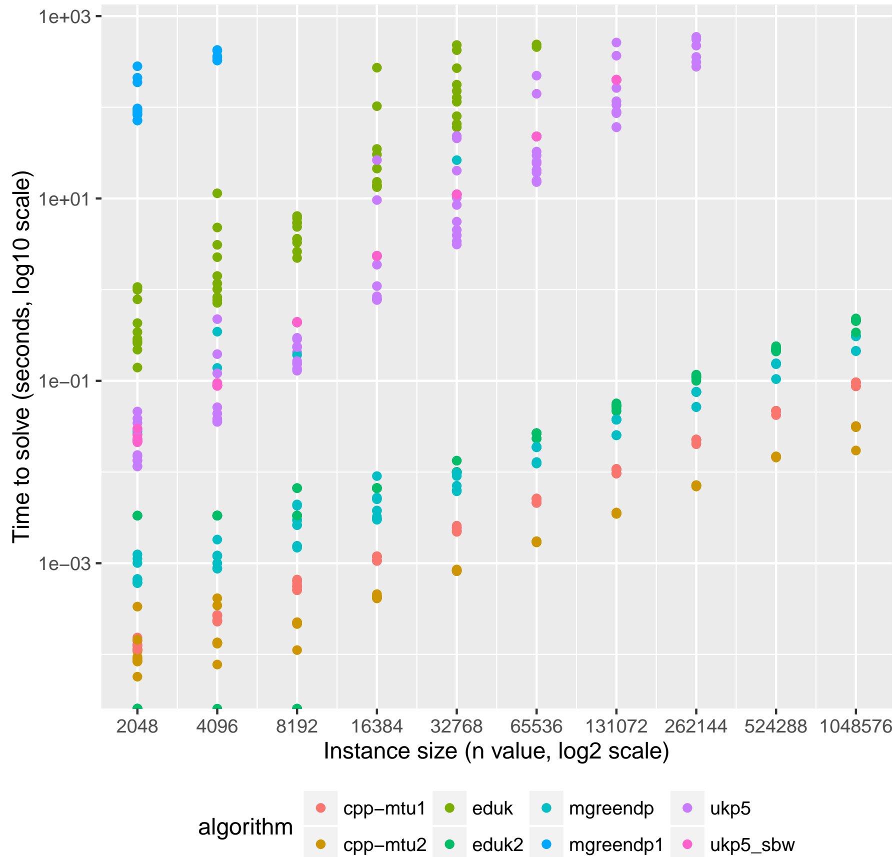
[2]Those approaches are described as "following the corner polyhedron approach of integer programming" and "the approach of solving (K) as a group knapsack problem.".

[3]The C++ code implementing both methods can be found at <https://github.com/henriquebecker91/masters/blob/e2ff269998576cb69b8d6fb1de59fa5d3ce02852/codes/cpp/lib/greendp.hpp>.

5.3 Solving the BREQ 128-16 Standard Benchmark

We have run eight algorithms over the BREQ 128-16 Standard Benchmark (proposed in Section 3.4). The results confirm our hypothesis that this distribution would be hard to solve by DP algorithms and easy to solve by B&B algorithms.

Figure 5.3: Benchmark with the 128-16 Standard BREQ instances.



Source: the author.

Let us examine Figure 5.3. The MGREENDP1 algorithm (a CA algorithm) is clearly dominated, and after the first two instance sizes, all of its runs end in timeout. So we will exclude it from the rest of the analysis.

The rest of the methods form two lines with different slopes, one line with a steep slope and a line with a more gradual slope. The steep slope line shows algorithms whose time grows very fast relative to the instance size growth. This group is mainly composed by the DP methods: UKP5, UKP5_SBW (i.e. UKP5 Sorted By Weight), and the EDUK algorithm. The second group, which forms a more gradual slope, has algorithms whose time grows much slower with the instance growth. This group is mainly composed by B&B and hybrid methods, as: MTU1,

MTU2, EDUK2 and MGREENDP.

Examining only MTU1 and MTU2, we can clearly see that for small instances their times overlap, but with the instance size growth the core problem strategy of MTU2 (that tries to avoid sorting and examining all the items) begins to pay off, making it the *best algorithm* to solve BREQ instances.

The behavior of EDUK2 shows that the default B&B phase (executed before defaulting to EDUK) solves the BREQ instances in all cases. If it did not, some EDUK2 points would be together with the EDUK points for the same instance size. Among the pure DP algorithms, EDUK was the one with the worst run times, being clearly dominated by our two UKP5 versions.

The UKP5 algorithm sorted the items by non-increasing efficiency, and had the y^* bound and periodicity checking enabled. These last two optimizations benefited none of the one hundred runs. No knapsack capacity from an instance was reduced by the use of the y^* bound; all instances had only overhead from the use of the periodicity checking. The UKP5_SBW sorted the items by increasing weight and had these two optimizations disabled. The benchmark instance files had the items in random order, so both algorithms used a small and similar time ordering the items[4].

The UKP5_SBW times had a much smaller variation than the UKP5 for the same instance size, which can be attributed to the change in ordering (as the two previously cited optimizations had only wasted time with overhead). The decreasing efficiency ordering helped UKP5 to be faster than UKP5_SBW in some cases, and turned it slower in others, what does not give us a clear winner.

The MGREENDP is a modern implementation in C++, made by the author, of an algorithm made by Harold Greenberg. The algorithm of Harold Greenberg (that was not named in the original paper) was an adaptation of the *ordered step-off* algorithm from Gilmore and Gomory. This algorithm periodically computes bounds (similar to the ones used by the B&B approach) to check if it can stop the DP computation and fill any remaining capacity with copies of the best item. In the majority of the runs, the bound computation allowed the algorithm to stop the computation at the beginning, having results very similar to EDUK2 (the hybrid B&B-DP algorithm). However, six of the MGREENDP executions had times in the steep slope line (the bound failed to stop the computation). Without the bound computation,

[4]It is interesting to note that, except for the small items that have profit rounding problems, in BREQ instances, the increasing efficiency order is the increasing weight order.

MGREENDP is basically the *ordered step-off* from Gilmore and Gomory (which is very similar to UKP5, as already pointed out); consequently, those six outlier runs have times that would be expected from UKP5 for the respective instance size.

A run from the greatest instance size and one from the second greatest instance size were both ended by timeout. The bound failed to stop the computation and the DP algorithm was terminated by timeout.

While the simple, multiple and collective dominances are rare in a BREQ distribution with integer profits, the solution dominance used by UKP5 works to some extent. The UKP5 combines optimal solutions for small capacities with single items and generate solutions that, if optimal for some capacity, will be used to generate more solutions after (recursively). In a BREQ instance, solutions composed of many small items rarely are optimal and, consequently, often discarded, wasting the time used to generate them. The weak solution dominance used by UKP5 does not completely avoid this problem, but helps to generate less of the useless subproblem solutions.

5.4 Solving pricing subproblems from BPP/CSP

The experiment described in this section is different from the previous experiments. All the other experiments consisted of instances of the UKP with specific distributions saved in files, and executables that read those instances, solved them and returned the solving time. In this experiment, the instances are not UKP instances but Bin Packing Problem (BPP) instances and Cutting Stock Problem (CSP) instances. The times presented are the sum of the times used to solve all the pricing subproblems (and/or master problems) generated while solving the continuous relaxation of those BPP/CSP instances.

In a pricing subproblem, the profit of the items is a real number. Adapting MTU1 for using floating point profit values proved to be difficult, as the bound computation procedure is based on the assumption that both weight and profit values were integer. The solution found was to multiply the items profit values by a multiplicative factor, round them down and treat them as integer profit values. The multiplicative factor chosen was 2^{40}. The inner working of floating point numbers favored the choice of a potency of two. This way, an integer profit value is exactly the first 40 bits of the mantissa from the respective floating point profit value.

Also, $2^{40} \approx 10^{12}$, so any value discarded by the rounding down is smaller than one part in one trillion. The code using MTU1 as solver for the pricing subproblem will be referred to as MTU1_CUTSTOCK.

To measure the impact of the conversion described above, the author used two versions of the UKP5, one using floating point profit values, and the other using integer profit values and the same conversion described above. The UKP5 by default sorts the items by non-increasing efficiency, but it works with any ordering. The items from a pricing subproblem are always naturally sorted by increasing weight[5]. The author executed the two UKP5 variants with sorting enabled and disabled, to verify if the sorting cost would pay off. Consequently, four versions of UKP5 were used in the tests, for all combinations of profit type (the original floating point, or the converted integer), and sorting (sorting by non-increasing efficiency, or not sorting, which is the same that having the items sorted by increasing weight). The codes using each one of the four versions of the UKP5 described above to solve pricing subproblems will be referred to as: UKP5_FP_CUTSTOCK (floating point, sort by efficiency), UKP5_INT_CUTSTOCK (integers, sort by efficiency), UKP5_FP_NS_CUTSTOCK (floating point, no sort) and UKP5_INT_NS_CUTSTOCK (integers, no sort).

The CPLEX was already being used for solving the master problem, and was used to solve the pricing subproblem too. The code using MTU1 as solver for the pricing subproblem will be referred to as MTU1_CUTSTOCK. For reasons explained in Section 5.4.4, only UKP5, CPLEX and MTU1 were used in this experiment.

In (GILMORE; GOMORY, 1961), it is suggested that instead of solving the pricing subproblem exactly, an heuristic for the UKP could be used, and only if the cutting pattern found by the heuristic did not improve the solution, the exact method would be used. In (GILMORE; GOMORY, 1963, p. 17), the experiments show that there is gain in always solving the pricing subproblems exactly. In the experiment described in this section, the pricing subproblem is always solved exactly.

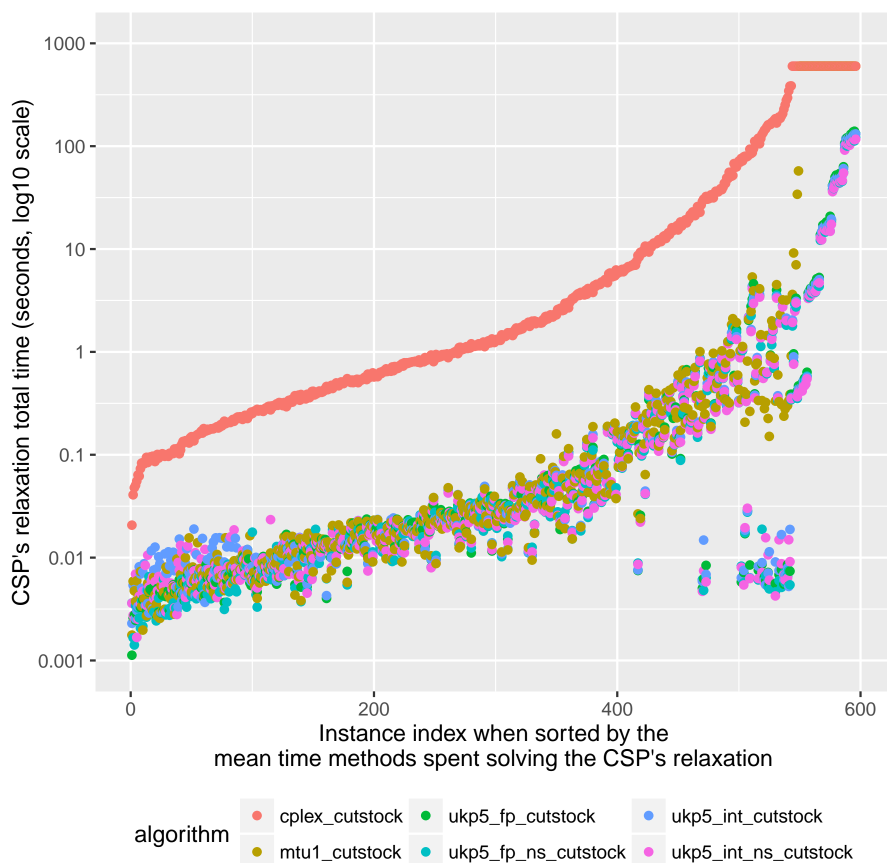
The timeout used in those experiments was of 600 seconds. This choice was based in the number of CSP instances, and that each one of them had the potential for generating thousands of pricing subproblems.

The dual values of the CSP master model can be negative or zero (non-positive). Such non-positive values can break codes optimized with the assumption

[5]The term 'increasing' is used because two items never share the same weight in a pricing subproblem.

that all items have a positive profit. Consequently, in the implementations of this experiment, those items are removed before solving the pricing subproblem with a non-CPLEX solver (CPLEX has no problem with such non-positive profits). The indices of the items in the solutions of the pricing subproblems are remapped to their original values before returning the solution to the master problem. The same process of remapping is already done for algorithm that change the items order. The time taken by sorting, conversion and remapping procedures is accounted in the times taken by the UKP algorithms to solve the pricing subproblems.

Figure 5.4: Total time used solving the continuous relaxation of the BPP/CSP.



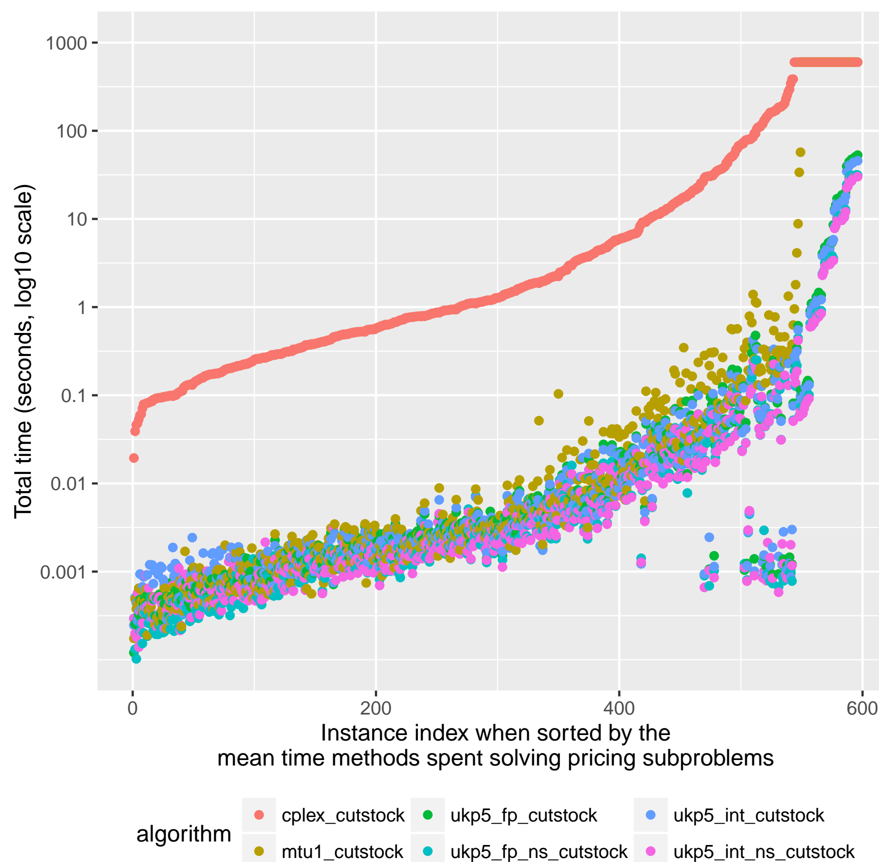
Source: the author.

Let us examine Figure 5.4. To simplify visualization, in the instances that CPLEX and MTU1 ended in timeout, these two methods are plotted as having used exactly 600 seconds. Two facts should be immediately obvious: 1st) using CPLEX to solve the pricing subproblems is not really competitive; 2nd) for the majority of the CSP instances in the benchmark, solving their continuous relaxation with a non-CPLEX method is very fast (takes less than one second).

The second fact is not so surprising considering that the original dataset

from (DELORME; IORI; MARTELLO, 2014)[6] included many old datasets, as it tried to be extensive; also, solving the continuous relaxation of a CSP instance takes considerably less effort than solving the problem itself. To solve a CSP instance, a B&B algorithm needs to solve the instance continuous relaxation multiple times. If each relaxation takes more than couple of seconds to solve, then solving the original CSP instance can become impracticable.

Figure 5.5: Total time used solving pricing subproblems (UKP) in the continuous relaxation of the BPP/CSP.



Source: the author.

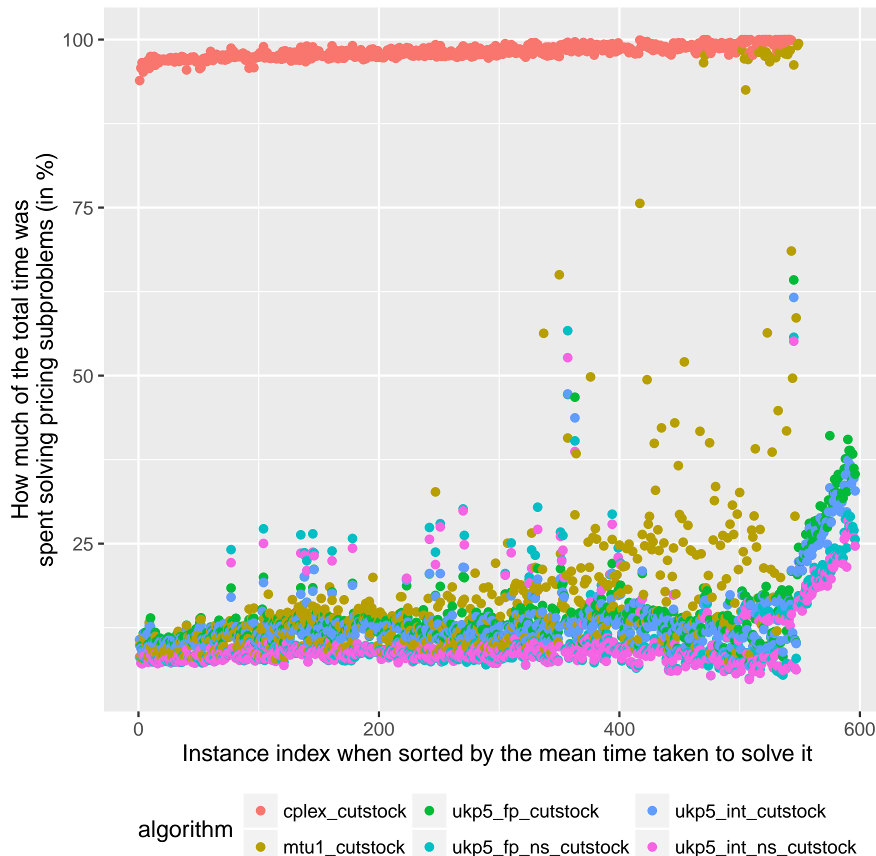
There is a small cluster of points where $450 < x < 550$ and $y < 0.1$. This cluster highlights the instances whose pricing subproblems were quickly solved by the UKP5 variants but slowly solved by the CPLEX or MTU1. The same cluster can be seen in Figure 5.5 (for $y < 0.001$), which only shows the time taken by solving the pricing subproblems.

The author will focus on the runs that spent more than a second solving

[6]The dataset used in this section was described in Section 3.3. It is composed of 10% of the instances used in (DELORME; IORI; MARTELLO, 2014), some gathered from datasets of the literature and some proposed in the just cited work.

pricing subproblems.First, it becomes clear that the B&B approach is not viable for larger/harder instances, as its exponential worst-case times make the problem untractable. Second, there seems to be an advantage in not sorting the items, and this difference is not caused by the time taken by the sorting procedure. In the runs that lasted more than a second, the time used to sort the items was between 3% and 0.5% of the time used to solve the pricing subproblems. Such small difference does not explain the gap displayed in the graph. When the items were kept sorted by increasing weight, UKP5 was up to two times faster than when the items were sorted by non-increasing efficiency. Keeping the items sorted by increasing weight seems to be the best choice for such instances. Third, it seems that executing all computation using integers is slight faster than using floating point profit values, but the difference is barely noticeable.

Figure 5.6: Percentage of time taken by solving pricing subproblems (UKP) in the continuous relaxation of the BPP/CSP.



Source: the author.

Figures 5.4 and 5.5 can give the erroneous impression that solving the pricing subproblems was where the majority of the solving time was spent. This is true in the

case of CPLEX, partially true for MTU1, and not true at all for UKP5. Figure 5.6 allows a better visualization of what was just said. When CPLEX is used to solve the pricing subproblems, almost all total time is spent solving the pricing subproblems. However, the time taken by non-CPLEX methods to solve the pricing subproblem remained mostly below 25% of the total time (or even less). Also, it is clear that when MTU1_CUTSTOCK used large amounts of time, it was consequence of the MTU1 method taking too much time to solve the pricing subproblems.

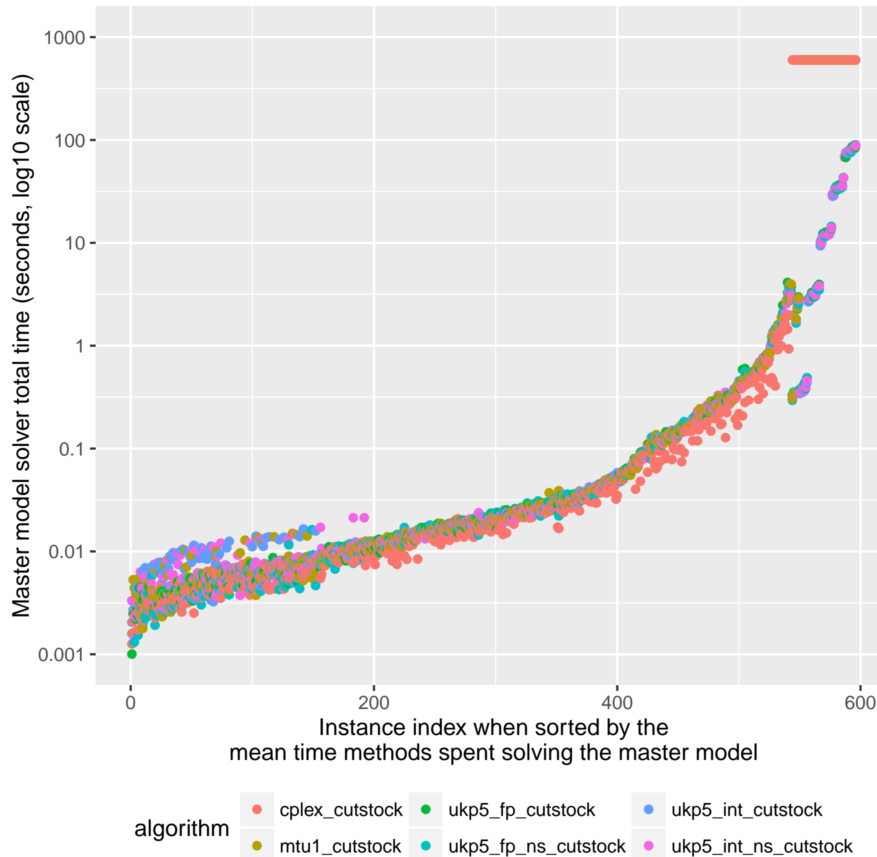
In the case of CUTSTOCK_UKP5_*, the most time-consuming instances spent no more than 40% of the time solving the pricing subproblems (often considerably less). This is not to say that, in such time-consuming instances, the pricing subproblems were not harder. In Figure 5.6, where $x > 550$, we can see that there is a rise of the relative time taken to solve the pricing subproblem instances with UKP5. However, the reason for the growth in the total time spent to solve the most time-consuming instances was that many cutting patterns had to be added to the master problem before it could not be improved anymore; what results in more iterations and, consequently, more instances of the pricing subproblem and master problem solved.

Finally, Figure 5.7 shows that the time taken by solving the master model is dependent on the instance itself, and not so much on the method used to solve the pricing subproblems. Yet, we can see that using CPLEX to solve the pricing subproblems seems to make solving the master problem slightly easier. Such behaviour seems to be related to the differences in the number of iterations (or master/pricing subproblems) solved. This is a topic that will be explored in the following sections.

5.4.1 The differences in the number of pricing subproblems solved

In Figure 5.8, it is possible to see that the methods solved different numbers of pricing subproblems for the same instance. The author used a constant for the CPLEX seeds of the master problem solver and the method that used CPLEX as the knapsack solver (cplex_cutstock). All methods are deterministic, they will give the same pricing subproblems if executed many times over the same instance. However, for the same instance, the amount of pricing subproblems (and their profit values) is affected by the algorithm chosen to solve them. This happens because the pricing subproblems can have many optimal solutions, and different methods break this tie

Figure 5.7: Total time used solving the master problem in the continuous relaxation of the BPP/CSP.

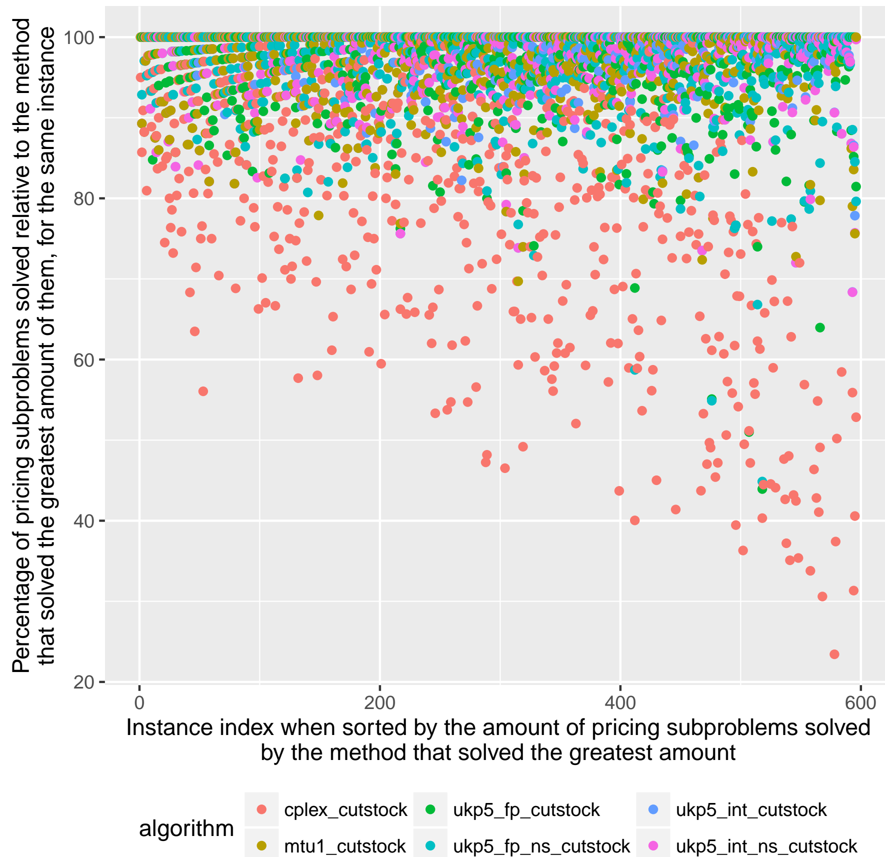


Source: the author.

in different ways. The choice of optimal solution will affect the master problem, which will generate a slightly different pricing subproblem. This effect cascades and can change the profit values of all next pricing subproblems, and the number of pricing subproblems that are needed to solve (which is the same as the number of iterations, and the number of master problems solved).

CPLEX_CUTSTOCK stands out by requiring many of the smallest number of iterations. The author cannot explain what property the pricing subproblem solutions returned by CPLEX have that creates such a difference. MTU1 does not exhibit the same effect, therefore this property does not seem to be associated with the B&B approach. We can only explain the differences in iteration count between the UKP5 variants (further discussed in Section 5.4.3). The UKP5 variants always return an optimal solution with the smallest weight possible. This translates to generating patterns with the greatest possible waste (the gap between optimal solution weight and the knapsack capacity equals to the waste in a cutting pattern).

Figure 5.8: Quantity of pricing subproblems each method solved, relative to the method that solved the greatest amount for the same instance (in %).



Source: the author.

The author's hypothesis is that this idiosyncrasy of the UKP5 can affect negatively the number of iterations needed to find a set of cutting patterns that cannot be improved (loop end condition).

5.4.2 The only outlier

In every experiment presented in this thesis, the author verified if the optimal solution value (the value of the objective function) was the same for all methods. Given the innacurate nature of floating point arithmetic, in this experiment, the optimal solution values differed from method to method. However, except for one outlier, no method had an absolute difference from the mean optimal value greater than 2^{-20}. In other words, for each instance of the CSP/BPP, the optimal value in rolls of any method, subtracted by the mean of the optimal value in rolls for all

methods, was smaller than 2^{-20} and greater than $-(2^{-20})$.

The outlier was the run of UKP5_INT_CUTSTOCK over N4C1W1_O.csp, which resulted in 257.7500 rolls, while the other methods resulted in 257.5833 rolls for the same instance (a 0.1667 roll difference). Hoping to find the origin of the outlier, the author tracked what differed between UKP5_INT_CUTSTOCK and UKP5_FP_CUTSTOCK, that are the same algorithm with the same item ordering, but one uses integer profits and the other floating point profits.

The solutions given by the two methods at the third iteration differed. The solution given by the floating point variant is not the same optimal solution given by the integer method (and vice-versa). This happens because, when the items profit were made integer (by multiplying them by 2^{40} and rounding them down), a small value was lost with the rounding. For the floating point method, *one* solution was the optimal one. For the integer method, *many* solutions were optimal, including the one that was optimal for the floating point variant. The tie breaker of the integer method choose a different optimal solution, and started a cascade effect.

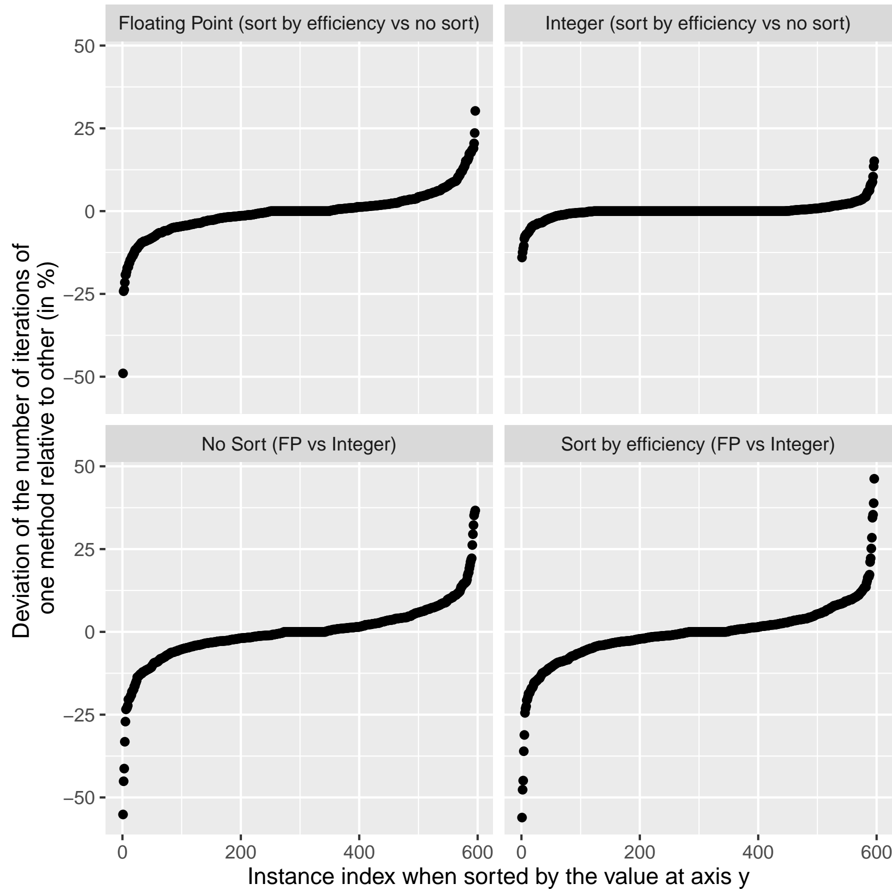
The author found that differences among knapsack solutions because of precision loss, followed by the cascade effect, are common. However, there was only one outlier, so the precision loss alone does not explain the outlier. The precision loss only explains the difference in the number of pricing subproblems solved by UKP5_INT_CUTSTOCK and UKP5_FP_CUTSTOCK. Section 5.4.3 has further discussion on how small changes to the algorithm changes the iteration count, and except by this outlier, this does not seem to affect the final result. Section 5.4.3 was written based on the analysis made while trying to (unsuccessfuly) find the origin of the outlier.

5.4.3 Similar methods generate different amounts of pricing subproblems

In Figure 5.9, we can see the relative difference in the number of pricing subproblems solved between any two versions of UKP5 that share one trait and differ in the other. The two traits are: the type used for the profit values (floating point or integer); and if the items were sorted by efficiency or not. It is interesting that such small variations of the same algorithm can yield significant differences to the numbers of pricing subproblems generated.

All four charts show that some instances were solved in the same number of

Figure 5.9: For the four UKP5 variants, the relative difference between the number of pricing subproblems solved in the same instance.



Source: the author.

iterations between the two variants compared; those instances form a horizontal line at $y = 0$. The two variants with less difference in the number of iterations are the two integer variants with different sort methods (top right chart). This was expected. If the profits are integer, the change in the items order only makes difference when a knapsack has two optimal solutions tied for the smallest weight, and the difference of order changes the optimal solution chosen (the items order acts as a tiebreaker in this case).

When the items profit is a floating point, the order the items are added influences the result (floating point addition is not associative). The difference is usually very small (least significant hex digit of a double mantissa), but the magnitude of the difference is irrelevant for the algorithm, that will choose the solution with the greatest profit. As the item order changes the order of the additions inside UKP5, which solution will be considered optimal (between solutions with very similar profit values) will change.

When we compare variants of the same sorting method, but with different profit types (two charts at bottom of the figure), the differences are not caused by optimal solutions tied for with the smallest weight, or by the lack of the associative property in the floating point addition, but by the loss of precision caused by the use of integer values. Solutions will be different because for the integer version, some items will have the same profit value, while for the floating profit version, one of those items will have a greater profit than the others (by less than 2^{-40}).

We can conclude that choosing one optimal solution over another (or choosing between two solutions with values very close to the optimal) can considerably change the number of iterations, without affecting the master problem optimal solution value (in a significant manner). A researcher studying UKP algorithms for this application has to consider the implications of this fact in his or her work.

In Section 6.4 (Future Works), we will see some questions raised by this experiment. Although there was one outlier, converting profit values to integer seems to be a valid method to test classic UKP algorithms (that work only with integer profits) for solving pricing subproblems (where the profit values are floating point values).

5.4.4 Algorithms not used in this experiment

Some algorithms were available, but were not used in this experiment Those are: GREENDP, GREENDP1, MTU2 and EDUK/EDUK2 (PYAsUKP).

GREENDP fails if the two most efficient items share the same efficiency, what often happens for at least one pricing subproblem of a CSP instance. GREENDP1 is very inefficient, as already shown by the experiment of Section 5.3. MTU2 was not used because it was designed for large UKP instances, which are the majority of the instances of this dataset. Moreover, in instances with a small number of items, MTU2 behaves almost the same way that MTU1.

The code of this experiment needs to call the UKP solving methods from inside the C++ code that also solves the CPLEX master model. The author tried to integrate EDUK and EDUK2 (which are written in OCaml) to the experiment, using the interface between C/C++ and OCaml, but the examples provided together with the PYAsUKP sources for this kind of integration were not working. The author could have saved the knapsack instances generated when solving the CSP problem

with other algorithm, and solved them using the PYAsUKP executable. This was not done because many knapsacks are small and solved very fast, and in these cases the PYAsUKP timing reports zero or inaccurate values. Also, the solutions returned by PYAsUKP should affect the next knapsacks generated (what cannot happen in such setting), so the comparison would be unfair.

5.5 The effects of parallel execution

After the publication of (BECKER; BURIOL, 2016), the author realized that run times were affected by the execution of multiple runs in parallel, *even if each run was being executed in an exclusive/isolated core*. The author credits this effect to the fact that the L3 memory cache is often shared between cores. Some experiments were performed to discern the magnitude of this effect.

For the rest of this section, the author will differ between runs that executed in parallel and runs that executed one-at-time (serially). It is important to note that the times compared are the wall-clock time of the algorithm execution (without the instance reading, or output printing) in: 1) a run that executes in an isolated core (with no other process in the same core), with one of the other cores running the operating system, and two of the remaining cores isolated and executing one run too (i.e. parallel runs); 2) a run that executes in an isolated core (with no other process in the same core), with one of the other cores running the operating system, and the remaining cores isolated and free (i.e. serial runs).

5.5.1 Setup

The setup of this experiment is different than the previous experiments. The experiment is repeated in two distinct computers; this was necessary to observe the impact of different CPUs (what include different amounts of shared cache). Only the UKP5 and the EDUK2 were used; both algorithms were present in the original comparison (BECKER; BURIOL, 2016), and both algorithms are relatively fast (a practical concern).

The two computer settings will be referred to as `notebook` and `desktop`. The `notebook` setting is the same used in (BECKER; BURIOL, 2016). The `notebook`

configurations included: Intel® Core™ i7-4700HQ CPU @ 2.40GHz; there were 8GiB RAM available (2× DIMM DDR3 Synchronous 1600 MHz) and three levels of cache (L1d: 32K and L1i: 32K, L2: 256K, L3: 6144K, only the L3 is shared between cores). The CPU had Intel® HyperThreading, where 4 physical cores simulate 8 virtual cores. This technology was disabled for the tests. The `desktop` configurations included: Intel® Core™ i7 5820k @ 3.3GHZ; there were 16GiB RAM available (2× Crucial Ballistix Sport DDR4 SDRAM 2400 MHz) and three levels of cache (L1i: 32K and L1d: 32K, L2: 256K, L3: 15MiB, only the L3 is shared between cores). This CPU also had Intel® HyperThreading, where 6 physical cores simulate 12 virtual cores. This technology was also disabled for the tests in this CPU.

The computers settings have different amounts of cores (four for `notebook` and six for `desktop`). During the experiments, in both computers, one core was left to run the operating system and three cores were isolated. In the parallel runs, the three isolated cores were used and, in the serial runs, only one isolated core was used. The author believes this choice made the comparison fairer.

To allow the computation of the standard deviation, the experiments consist in ten runs for each combination of solver (UKP5 and EDUK2), computer (`notebook` and `desktop`), instance (the 454 of the reduced PYAsUKP benchmark, see Section 3.2.6), and mode (parallel or serial). For the same computer, all runs of one algorithm were finished before starting the first run of the other algorithm. This is especially important for the parallel runs, as this means that each algorithm only competed for cache with other runs of the same algorithm. For the same computer, algorithm, and mode, the order of the instances was randomly chosen.

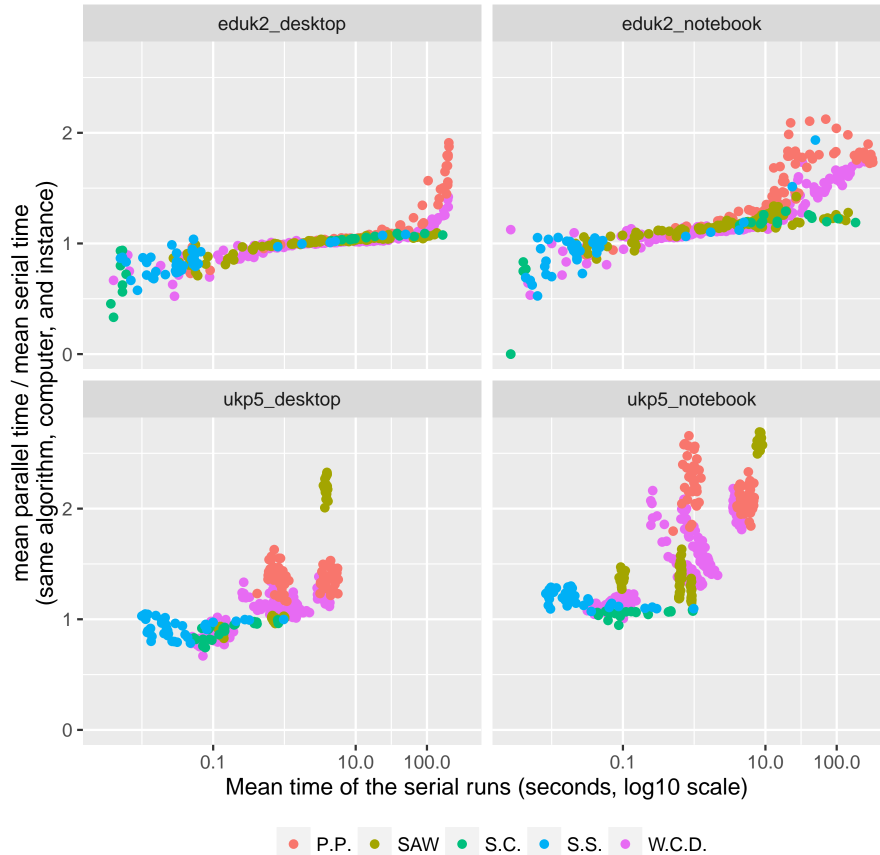
5.5.2 Experiment

The experiment has shown that the mean time of multiple runs of the same algorithm in the same computer vary considerably if they are executed in parallel or serially; how much they vary depends on the specific algorithm and computer.

The author does not intend to do an in-depth analysis, but only to show that the difference exists and can be significant. The experiments described in all the other sections of this chapter were executed serially to avoid this noise. The already mentioned referee of (BECKER; BURIOL, 2016), questioned if the faster memory access could have benefited UKP5 over EDUK2. This experiment is also

an answer to that question. The experiment setting in (BECKER; BURIOL, 2016) corresponds to parallel runs over the `notebook` computer. The author believes that the point made in (BECKER; BURIOL, 2016) still stands, as the effect found over the algorithms run times would not render a different analysis.

Figure 5.10: Comparison of the mean times of UKP5 and EDUK2 when executed in two diferent computers, in parallel and in serial mode.



Source: the author.

Let us examine Figure 5.10. The execution of runs in parallel seems to affect UKP5 far more than EDUK2, as it has much more points distant from $y \approx 1$ (where serial and parallel runs have similar mean times) than EDUK2, and only UKP5 has a significative share of points over $y = 2$ (where the mean times of the parallel runs are more than two times the serial mean time). The author believes that the reasons for UKP5 being more affected by cache sharing are the greater use of memory (about $2 \times (c - w_{min} + w_{max})$); the initialization of such arrays; and accessing many diferent memory regions in sequence (at each iteration of the outer loop, up to n array positions in a range of size up to w_{max} are accessed in an arbitrary order).

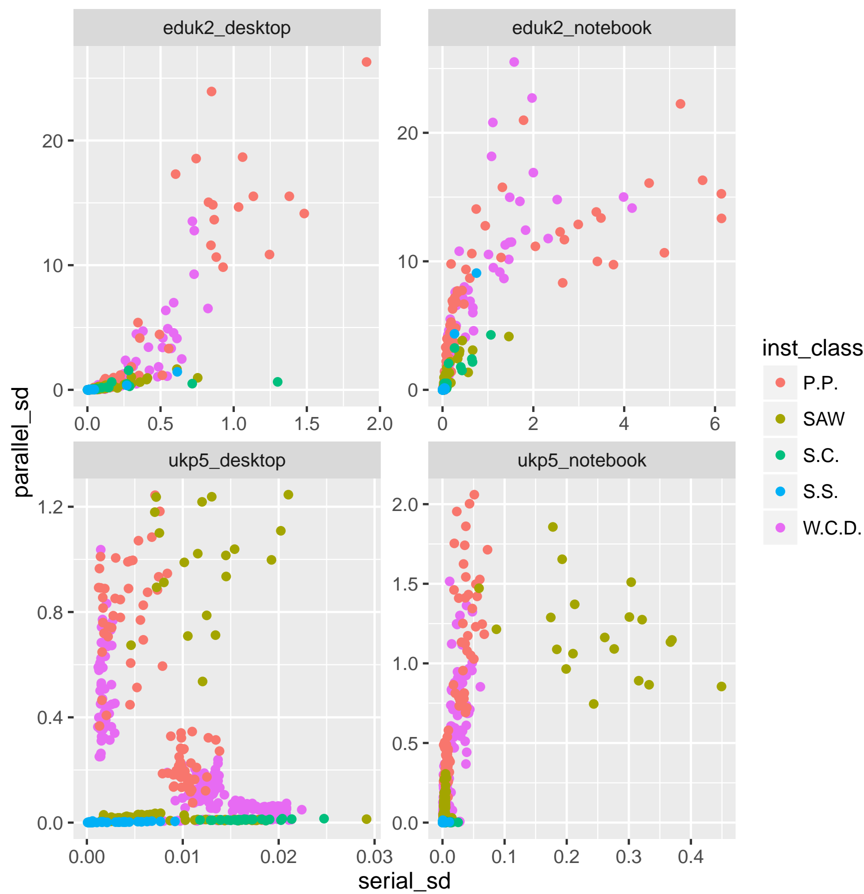
While UKP5 seems to be more affected by running in parallel, one could

point out that all UKP5 runs use little time (less than 10 seconds when serially and, consequently, no more than thirty when in parallel), where many of the EDUK2 runs have times between 10 and 100 seconds (serially), with some up to 300 seconds. Besides that, the EDUK2 runs that take more time are also the more affected by the parallel execution. The author hypothesizes that these runs take more time because the instance is harder, and use more memory for the same reason. This way, these runs end up using more than the L1/L2 cache and being more strongly affected by this effect (the instance 'hardness' acts as a confounding variable, the same effect happens to UKP5). The result would be that more time is added to the EDUK2 total time because of this effect than time is added to the UKP5 total time (in absolute numbers), making the comparison unfair to EDUK2.

Let us closely examine the values for the `notebook` setting, as this was the one used at the (BECKER; BURIOL, 2016). We will refer to UKP5 (or EDUK2) Parallel (or Serial) Total time (or TT, for short) as the sum of the run times of all the runs of that algorithm in that mode and, for now, only in the `notebook` setting. The UKP5 Parallel TT is 2.03 times greater than UKP5 Serial TT, while EDUK2 Parallel TT is only 1.60 times greater than the EDUK2 Serial TT. However, this means about 8252 extra seconds for UKP5 Parallel TT (compared to the serial time) and about 74180 extra seconds for EDUK2 Parallel TT (compared to the serial time). We will see that this considerable absolute difference does not end up benefiting UKP5 in the analysis, as the EDUK2 Serial TT is about 15.33 times greater than the UKP5 Serial TT, but the EDUK2 Parallel TT is about 12.10 times bigger than UKP5 Parallel TT. In fact, the parallel execution seems to benefit the EDUK2 numbers against UKP5 while not by much. If we focused in the `desktop` setting, we would see that EDUK2 was benefited too, while to a lesser degree. In the `desktop` setting, the EDUK2 Parallel TT is 15.62 times greater than the UKP5 Parallel TT; and the EDUK2 Serial TT is 16.50 times greater than the UKP5 Serial TT.

Figure 5.11 presents the Standard Deviation (SD) values computed over each ten runs for the same algorithm, computer, mode and instance. The author will not draw many conclusions from this data, as SD values with different means are not directly comparable. However, it is interesting to point out that, in general, the runs in the serial mode, and/or the `desktop` setting, have a much smaller SD than the ones in parallel mode, and/or the `notebook` setting (either because of the

Figure 5.11: Parallel and Serial Runs Standard Deviation



Source: the author.

serial/`desktop` run times are smaller, or because the algorithms do not compete for cache).

The author belives this superficial analysis is sufficient to convince the reader that there is a significant difference in running such experiments serially or in parallel. If some real-world application of the UKP needs to solve multiple knapsacks in parallel, algorithms that use less memory (or with better locality of reference) can be prefered not only to avoid swapping, but because they will affect each others times less. As this is not the case here, and as the CSP (the real-world application of the UKP covered in this chapter) specifically solves multiple knapsacks serially, all experiments of this chapter consisted of serially executed runs, with only one run over each combination of algorithm and instance.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The author believes that the major contribution of this thesis is the critical review of the UKP literature, which is summarized in the following subsection. The review expresses and discusses statements that can be generalized to all the experimental research in computer science. Besides the review, the more objective and UKP-specific knowledge contributions and technical contributions are listed in the subsequent subsections.

6.1.1 A critical review

An algorithm is dominated by other in the context of a dataset. An example presented in this thesis is MTU2. MTU2 is the best algorithm between eight algorithms for one dataset (see Section 5.3), and is not competitive for other five datasets (see Section 5.2). The literature review, and the further discussion of instance datasets and solving approaches, have shown *how the choice of datasets defined the best algorithm through the last fifty years.* The unfolding of the events can be summarized as:

(GILMORE; GOMORY, 1961) A real-world application of the UKP (pricing problem for BPP and CSP) is proposed.

(GILMORE; GOMORY, 1966) Some dynamic programming algorithms for such application are proposed.

(MARTELLO; TOTH, 1977) A B&B method is proposed, and then compared with the DP algorithms over small artificial instances, obtaining marginally better results

(MARTELLO; TOTH, 1990a) The goal changes to solving larger instances faster. Artificial datasets are used for this. For such datasets, DP methods are clearly dominated by B&B methods.

(ZHU; BROUGHAN, 1997) The large artificial instances are shown to have only a little amount of relevant/undominated items and are discredited.

(POIRRIEZ; ANDONOV, 1998) A new DP method is proposed, together with

new artificial datasets (without the same flaws of the previous datasets).

(ANDONOV; POIRRIEZ; RAJOPADHYE, 2000) The new DP method only compares to B&B and naive DP, the old non-naive DP algorithms were forgotten or excluded because of previous experiments.

(KELLERER; PFERSCHY; PISINGER, 2004) The new DP method is considered the state-of-the-art.

(POIRRIEZ; YANEV; ANDONOV, 2009) The new DP method is hybridized with B&B, and the datasets are updated to be 'harder'. Such datasets are hard for B&B, and the hybrid method is only compared to B&B. The hybrid method is the new state-of-the-art.

(BECKER; BURIOL, 2016) An old DP method is reinvented and outperforms the hybrid method in the updated datasets.

(this thesis) Old algorithms are revised, reimplemented and tested. The influence of the datasets and the historical context becomes apparent.

It is worth mentioning that, taking into account the historical context, the choices made by previous researchers were sound and reasonable. Despite this, an efficient DP algorithm was ignored for decades when it was relevant for the comparisons and experiments realized.

This leads to another conclusion: *the bibliographical research is important, and should be followed by a reevaluation of the evidence chain.* The author first reinvented an old DP algorithm, published a paper about how it surpassed the state-of-the-art, and then discovered that his bibliographical research was faulty. While the author does not intend to justify this overlook, it is important to note that, in the context, reading papers about algorithms from fifty years ago, and implementing them to compare with a recent state-of-the-art algorithm did not seem to be as a good use of time, as they were already compared and discarded by works of the same period. If the author did not end up accidentally reinventing the algorithm, and then recognizing it in an old paper, it is possible that the *terminating/ordered step-off* would remain ignored. A possibility that can only be considered more worrying by the fact that the algorithm pseudocode is available in (GILMORE; GOMORY, 1966), a paper that is cited by many of the works mentioned above. Implementing (or obtaining) all algorithms already proposed for a problem clearly is not a viable strategy. However, this work shows that the context of one experiment

that concludes that an algorithm is dominated by other has to be critically evaluated.

Also, *papers and experiments do not always provide all relevant context.* One example of context that is not always provided, or given much attention, is if the runs were executed in parallel, or not. The experiments from Section 5.5 show a significant difference between performing an experiment with runs in parallel, or serially. The magnitude of the difference vary with the specific algorithm, computer and dataset. In one of the computer settings, the average run times of UKP5 when executed in parallel were about the double of the average serial run times. In an indirect comparison between results presented by different papers, this detail could lead to a method being considered significantly faster than other only because one author executed serial runs and the other parallel runs.

6.2 UKP-specific knowledge contributions

An outline of the UKP-specific knowledge contributions follows. It is ordered by the level of importance the author gives to them.

1. The knowledge that an old DP algorithm outperforms a state-of-the-art hybrid algorithm, in the most recent benchmark dataset.
2. The concept of solution dominance, and its implications for periodicity bounds and the four previously established dominance relations.
3. Evidence that the B&B approach worst-case can arise when solving pricing subproblems of BPP/CSP.
4. Evidence that variations in the optimal solution returned by the pricing problem solver can have a strong effect in the number of pricing problems generated.
5. Evidence that UKP algorithms that are memory intensive should be executed serially for comparison.
6. Evidence that converting the pricing problem profit values to large integers do not cause significant loss.

The author believes that the first contribution is already well exposed, either in Section 5.2, or in (BECKER; BURIOL, 2016).The technical details of the second contribution (i.e. the concept of solution dominance) were already discussed (see Section 4.2.4.2), but not how it impacts the previous techniques described in the literature. The weak solution dominance reduces the further improvement that can

be found by applying the four dominance relations and/or periodicity bounds in the same algorithm. The weak solution dominance and the four dominance relations are two different ways of dealing with the same task. The first involves keeping an index in each solution to mark which items can still be added to the solution. The second involves keeping a global list of undominated items,

The approach used by EDUK gives a strong guarantee that any dominated item will be discarded and never used again. However, the weak solution dominance described in Section 4.2.4 is implemented with almost no overhead, and seems to have a very similar impact in the removal of dominated items/solutions. One could argue that EDUK can be at disadvantage for being implemented in a functional language, or that better implementations of the algorithm could be written, the author can not refute such claims. Maybe new implementations of the EDUK approach can show the superiority of applying the four dominances in the EDUK fashion. However, for the tested datasets, the weak solution dominance approach seems to be the most efficient one.

The periodicity check exists both in algorithms like EDUK/EDUK2 and the old terminating step-off. In EDUK/EDUK2 it is a consequence of applying all four dominances repeatedly, and in the terminating step-off it is a consequence of applying weak solution dominance. A periodicity check can save effort by stopping the computation at a capacity $y < c$. However, in all algorithms that implement the periodicity check, when this early termination happens, it is because the only item that could possibly be used is the best item. Consequently, in each one of these last positions (between y and c), the algorithm would not execute $O(n)$ steps anymore, but only $O(1)$ steps. The periodicity check only saves the effort of iterating these last $c - y$ positions. It is a minor improvement over the application of weak solution dominance, or the application of the four item dominances.

The periodicity check (and, by consequence, the dominances) also reduces the utility of periodicity bounds. If an upper bound on y^+ could be used to stop the computation before it reaches capacity c, then the periodicity check would stop the computation even before the capacity predicted by the upper bound (with slightly more overhead). In an algorithm with periodicity checking, the utility of upper bounds on the periodicity capacity y^+ is restricted to saving memory and saving the time spent initializing such saved memory. Note that some algorithms would not even have such benefits, as they do not allocate or initialize the memory in advance.

The evidence that constitutes the third knowledge contribution can be found in Section 5.4. In the majority of the instances, the time spent by both the B&B and DP approaches when solving pricing problems was significantly smaller than the times solving the master problems of the same instance. Nevertheless, for some instances, the B&B approach has presented its worst-case behavior and, in these instances, more time was spent solving pricing subproblems then solving the master problem.

For a single example, we will focus on instance 201_2500_DI_11.txt, for which MTU1_CUTSTOCK ended in timeout. The pricing problems generated when solving such instance have $n < 200$ and $c = 2472$. Before timeout, MTU1 solved about 700 of those pricing problems in much less than a millisecond each. However, there was also a few pricing problems that took ten seconds or more to solve; run times like: 10, 12, 13, 13, 26, 32, 49, 54, and 351 seconds. All instances shared the same n, c and the same items weights, the only difference between them is the profit values. Such behaviour corroborates with what was said about B&B algorithms being strongly affected by the items distribution, and less by n and c.

It is worth mentioning that almost all instances that ended in timeout are artificial instances created to be hard to solve. Such instances were proposed in (DELORME; IORI; MARTELLO, 2014). Unfortunately, the author of this thesis did not have the time to gather CSP problems from industrial sources, and the experimentation had to stop at the literature instances. Consequently, we cannot state that B&B algorithms are not viable for solving pricing problems, only that there is evidence that the B&B worst-case can arise in such circumstances.

While the following quote was written in the context of the 0-1 KP, the author found it relevant to complement what was just said: “Dynamic programming is one of our best approaches for solving difficult (KP), since this is the only solution method which gives us a worst-case guarantee on the running time, independently on whether the upper bounding tests will work well.” (PISINGER, 2005, p. 13).

The last three contributions are thoroughly discussed in the context of the experiments that originated them (for the fourth and sixth contributions see Section 5.4, and for the fifth, Section 5.5). The fourth contribution is specially important to a researcher testing an algorithm that needs to solve unbounded knapsack pricing problems. Minor changes in the operation of the UKP solver, as the order it receives the items, can change the exact optimal solution returned, and consequently the

number of pricing problems generated and the whole algorithm run time. Such chaotic behaviour can lull an experimenter into believing that irrelevant changes have some deeper meaning, or add noise to relevant changes. The same can be said about executing memory intensive algorithms in serial, or parallel (fifth contribution). The sixth and last contribution, while corroborated by the vast majority of the runs, maybe has contributed to our only outlier, and needs further validation.

6.3 Technological UKP-specific contributions

An outline of the UKP-specific technological contributions follows.

- The only known implementations of the GREENDP and GREENDP1 algorithms, modernized to use loops.
- The UKP5 implementation, that can be seen as a variant of the terminating step-off.
- New implementations for MTU1 and MTU2. Such implementations are coded in C++11 (instead of Fortran77) and use generic templates. Both implementations are slight faster than the respective Fortran77 implementations, and the MTU2 C++ implementation does not have the problem with the subset-sum instances presented by the Fortran77 implementation.
- A copy[1] of the exact PYAsUKP benchmark used, and scripts[2] to generate it (based on PYAsUKP).
- The BREQ 128-16 Standard Benchmark, which is a new dataset.

The author will not discuss such technical contributions as they are only a byproduct of the research. The author does not suggests that the BREQ 128-16 Standard Benchmark should be used for new experiments comparing algorithms for the UKP. The dataset can yet be used to study the behaviour of new solving approaches, maybe giving some insight about the approach inner workings.

[1] Available at: <https://drive.google.com/open?id=0B30vAxj_5eaFSUNHQk53NmFXbkE>

[2] Available at: <https://github.com/henriquebecker91/masters/tree/f5bbabf47d68528166153 15c8839d3f74013af5f/codes/sh/pyasukp_paper_bench>. Note that a PYAsUKP binary compiled from the version in <https://github.com/henriquebecker91/masters/blob/f5bbabf47d685281661 5315c8839d3f74013af5f/codes/ocaml/pyasukp_mail.tgz> has to be in the executable path to the scripts work.

6.4 Future work

Unfortunately, many questions could not be answered in the scope of this thesis. The author selected and listed some of such questions in this section.

- Which other real-world problems have the UKP as a pricing subproblem? Are there real-world problems that can be modelled as the UKP? Would instances of those problems be easy for the current state-of-the-art algorithm?
- How similar are the datasets of the UKP and the BPP/CSP presented in the literature to the ones existent in the real-world?
- Do the instances found in the real-world benefit some approaches over others?
- Would a hybrid algorithm based on the UKP5/'terminating step-off' and MTU2 would present the same level of improvement that EDUK2 has over EDUK?
- Regarding the many times mentioned but never reimplemented CA algorithm from (BABAYEV; GLOVER; RYAN, 1997): what would be its performance in real-world instances? Or, at least, in the most recent benchmark datasets?
- How do the traits of the optimal solution for the pricing subproblem affect the master problem? Does always returning an optimal solution with minimal weight has a negative effect? What about adding all patterns that improve the master problem solution, and not only the best pattern (i.e. optimal solution)?
- How often do pricing subproblems have multiple optimal solutions? How many of these optimal solutions are cut down because of dominance?
- How would an optimized C++ implementation of EDUK(2) perform? How would EDUK(2) perform over real-world instances?
- Are the profit values (and, consequently, the items distributions) of the pricing subproblems uniform between similar BPP/CSP instances, and/or the same BPP/CSP instance? Is it possible that they converge to a specific distribution at each iteration of the column generation?

REFERENCES

ANDONOV, R.; POIRRIEZ, V.; RAJOPADHYE, S. Unbounded knapsack problem: Dynamic programming revisited. **European Journal of Operational Research**, Elsevier, v. 123, n. 2, p. 394–407, 2000.

ANDONOV, R.; RAJOPADHYE, S. A sparse knapsack algo-tech-cuit and its synthesis. In: IEEE. **Application Specific Array Processors, 1994. Proceedings. International Conference on**. [S.l.], 1994. p. 302–313.

BABAYEV, D. A.; GLOVER, F.; RYAN, J. A new knapsack solution approach by integer equivalent aggregation and consistency determination. **INFORMS Journal on Computing**, INFORMS, v. 9, n. 1, p. 43–50, 1997.

BALAS, E.; ZEMEL, E. An algorithm for large zero-one knapsack problems. **operations Research**, INFORMS, v. 28, n. 5, p. 1130–1154, 1980.

BECKER, H.; BURIOL, L. S. UKP5: A new algorithm for the unbounded knapsack problem. In: SPRINGER. **International Symposium on Experimental Algorithms**. [S.l.], 2016. p. 50–62.

BELOV, G.; SCHEITHAUER, G. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. **European journal of operational research**, Elsevier, v. 171, n. 1, p. 85–106, 2006.

CABOT, A. V. An enumeration algorithm for knapsack problems. **Operations Research**, INFORMS, v. 18, n. 2, p. 306–311, 1970.

CHUNG, C.-S.; HUNG, M. S.; ROM, W. O. A hard knapsack problem. **Naval Research Logistics (NRL)**, Wiley Online Library, v. 35, n. 1, p. 85–98, 1988.

DELORME, M.; IORI, M.; MARTELLO, S. Bin packing and cutting stock problems: Mathematical models and exact algorithms. In: **DECISION MODELS for SMARTER CITIES**. [S.l.: s.n.], 2014.

FISCHETTI, M.; MARTELLO, S. A hybrid algorithm for finding the kth smallest of n elements in o(n) time. **Annals of Operations Research**, Springer, v. 13, n. 1, p. 399–419, 1988.

GARFINKEL, R. S.; NEMHAUSER, G. L. **Integer programming**. [S.l.]: Wiley New York, 1972. v. 4.

GILMORE, P. C.; GOMORY, R. E. A linear programming approach to the cutting-stock problem. **Operations research**, INFORMS, v. 9, n. 6, p. 849–859, 1961.

GILMORE, P. C.; GOMORY, R. E. A linear programming approach to the cutting stock problem-part ii. **Operations research**, INFORMS, v. 11, n. 6, p. 863–888, 1963.

GILMORE, P. C.; GOMORY, R. E. The theory and computation of knapsack functions. **Operations Research**, INFORMS, v. 14, n. 6, p. 1045–1074, 1966.

GLOVER, F. Integer programming over a finite additive group. **SIAM Journal on control**, SIAM, v. 7, n. 2, p. 213–231, 1969.

GREENBERG, H. On equivalent knapsack problems. **Discrete applied mathematics**, Elsevier, v. 14, n. 3, p. 263–268, 1986.

GREENBERG, H.; FELDMAN, I. A better step-off algorithm for the knapsack problem. **Discrete Applied Mathematics**, Elsevier, v. 2, n. 1, p. 21–25, 1980.

HU, T. C. **Integer programming and network flows**. [S.l.], 1969.

HUANG, P. H.; TANG, K. A constructive periodicity bound for the unbounded knapsack problem. **Operations Research Letters**, Elsevier, v. 40, n. 5, p. 329–331, 2012.

IIDA, H. Two topics in dominance relations for the unbounded knapsack problem. **The Open Applied Mathematics Journal**, v. 2, n. 1, p. 16–19, 2008.

KELLERER, H.; PFERSCHY, U.; PISINGER, D. **Knapsack problems**. [S.l.]: Spinger-Verlag, Berlin, 2004.

MARTELLO, S.; TOTH, P. Branch-and-bound algorithms for the solution of the general unidimensional knapsack problem. **Advances in Operations Research, North-Holland, Amsterdam**, p. 295–301, 1977.

MARTELLO, S.; TOTH, P. An exact algorithm for large unbounded knapsack problems. **Operations research letters**, Elsevier, v. 9, n. 1, p. 15–20, 1990.

MARTELLO, S.; TOTH, P. **Knapsack problems: algorithms and computer implementations**. [S.l.]: John Wiley & Sons, Inc., 1990.

PISINGER, D. Algorithms for knapsack problems. Citeseer, 1995.

PISINGER, D. Where are the hard knapsack problems? **Computers & Operations Research**, Elsevier, v. 32, n. 9, p. 2271–2284, 2005.

POIRRIEZ, V.; ANDONOV, R. Unbounded knapsack problem: new results. In: **Workshop**. [S.l.: s.n.], 1998. p. 103–111.

POIRRIEZ, V.; YANEV, N.; ANDONOV, R. A hybrid algorithm for the unbounded knapsack problem. **Discrete Optimization**, Elsevier, v. 6, n. 1, p. 110–124, 2009.

SHAPIRO, J. F.; WAGNER, H. M. A finite renewal algorithm for the knapsack and turnpike models. **Operations Research**, INFORMS, v. 15, n. 2, p. 319–341, 1967.

WEISSTEIN, E. W. **Diophantine Equation**: From mathworld–a wolfram web resource. 2016. Visited on 15/11/2016. Disponível em: <http://mathworld.wolfram.com/DiophantineEquation.html>.

ZHU, N.; BROUGHAN, K. On dominated terms in the general knapsack problem. **Operations Research Letters**, Elsevier, v. 21, n. 1, p. 31–37, 1997.

APPENDIX A — TABLES

	algorithm	n	avg	sd	min	max	ter
1	cpp-mtu1	2048	0	0	0	0	10
2	cpp-mtu1	4096	0	0	0	0	10
3	cpp-mtu1	8192	0	0	0	0	10
4	cpp-mtu1	16384	0	0	0	0	10
5	cpp-mtu1	32768	0	0	0	0	10
6	cpp-mtu1	65536	0	0	0	0.01	10
7	cpp-mtu1	131072	0.01	0	0.01	0.01	10
8	cpp-mtu1	262144	0.02	0	0.02	0.02	10
9	cpp-mtu1	524288	0.05	0	0.04	0.05	10
10	cpp-mtu1	1048576	0.09	0	0.09	0.1	10
11	cpp-mtu2	2048	0	0	0	0	10
12	cpp-mtu2	4096	0	0	0	0	10
13	cpp-mtu2	8192	0	0	0	0	10
14	cpp-mtu2	16384	0	0	0	0	10
15	cpp-mtu2	32768	0	0	0	0	10
16	cpp-mtu2	65536	0	0	0	0	10
17	cpp-mtu2	131072	0	0	0	0	10
18	cpp-mtu2	262144	0.01	0	0.01	0.01	10
19	cpp-mtu2	524288	0.01	0	0.01	0.01	10
20	cpp-mtu2	1048576	0.03	0	0.02	0.03	10
21	eduk	2048	0.48	0.34	0.14	1.07	10
22	eduk	4096	2.75	3.31	0.72	11.4	10
23	eduk	8192	4.35	1.47	2.23	6.42	10
24	eduk	16384	53.15	81.57	13.32	271.97	10
25	eduk	32768	195.03	149.39	60.37	481.69	10
26	eduk	65536	473.28	22.09	457.66	488.9	2
27	eduk	131072	–	–	–	–	0
28	eduk	262144	–	–	–	–	0
29	eduk	524288	–	–	–	–	0
30	eduk	1048576	–	–	–	–	0
31	eduk2	2048	0	0	0	0	10

	algorithm	n	avg	sd	min	max	ter
32	eduk2	4096	0	0	0	0	10
33	eduk2	8192	0	0	0	0.01	10
34	eduk2	16384	0.01	0	0.01	0.01	10
35	eduk2	32768	0.01	0	0.01	0.01	10
36	eduk2	65536	0.03	0	0.02	0.03	10
37	eduk2	131072	0.05	0	0.05	0.06	10
38	eduk2	262144	0.11	0.01	0.1	0.12	10
39	eduk2	524288	0.22	0.01	0.21	0.24	10
40	eduk2	1048576	0.45	0.04	0.34	0.48	10
41	mgreendp	2048	0.01	0.01	0	0.03	10
42	mgreendp	4096	0.05	0.11	0	0.35	10
43	mgreendp	8192	0.02	0.06	0	0.19	10
44	mgreendp	16384	0	0	0	0.01	10
45	mgreendp	32768	2.64	8.33	0.01	26.34	10
46	mgreendp	65536	0.02	0	0.01	0.02	10
47	mgreendp	131072	0.03	0.01	0.03	0.04	10
48	mgreendp	262144	0.07	0.01	0.05	0.08	10
49	mgreendp	524288	0.14	0.02	0.1	0.16	9
50	mgreendp	1048576	0.27	0.05	0.21	0.32	9
51	mgreendp1	2048	127.31	72.93	71.72	281.86	10
52	mgreendp1	4096	368.65	43.72	326.49	425.47	6
53	mgreendp1	8192	–	–	–	–	0
54	mgreendp1	16384	–	–	–	–	0
55	mgreendp1	32768	–	–	–	–	0
56	mgreendp1	65536	–	–	–	–	0
57	mgreendp1	131072	–	–	–	–	0
58	mgreendp1	262144	–	–	–	–	0
59	mgreendp1	524288	–	–	–	–	0
60	mgreendp1	1048576	–	–	–	–	0
61	ukp5	2048	0.02	0.01	0.01	0.05	10
62	ukp5	4096	0.11	0.14	0.04	0.48	10
63	ukp5	8192	0.2	0.06	0.13	0.3	10
64	ukp5	16384	4.53	8.14	0.77	26.37	10

	algorithm	n	avg	sd	min	max	ter
65	ukp5	32768	15.36	17.47	3.13	48.49	10
66	ukp5	65536	54.58	69.86	15.12	222.43	10
67	ukp5	131072	165.19	152.01	60.26	513.26	10
68	ukp5	262144	428.72	131.43	278.56	592.34	6
69	ukp5	524288	–	–	–	–	0
70	ukp5	1048576	–	–	–	–	0
71	ukp5_sbw	2048	0.02	0	0.02	0.03	10
72	ukp5_sbw	4096	0.09	0	0.09	0.09	10
73	ukp5_sbw	8192	0.44	0	0.44	0.44	10
74	ukp5_sbw	16384	2.34	0.01	2.34	2.36	10
75	ukp5_sbw	32768	11.07	0.03	11.03	11.14	10
76	ukp5_sbw	65536	47.91	0.07	47.8	48	10
77	ukp5_sbw	131072	200.11	0.54	199.67	201.15	10
78	ukp5_sbw	262144	–	–	–	–	0
79	ukp5_sbw	524288	–	–	–	–	0
80	ukp5_sbw	1048576	–	–	–	–	0

Table A.3: Results of the BREQ 128-16 Standard Benchmark (see Section 5.3). For each row, there is a set T comprised by the run times that **algorithm** spent solving instances of size **n**. We do not count the run time of runs that ended in timeout. The meaning of the columns **avg**, **sd**, **min**, **max** and **ter** are, respectively, the arithmetic mean of T, the standard deviation of T, the minimal value in T, the maximal value in T, and the cardinality of T (i.e. the number of runs that did not end in timeout). The time unit of the table values is seconds.

Table A.1: Results for the PYAsUKP 4540 Instances (see Section 5.2). Columns **n** and w_{min} values must be multiplied by 10^3 to obtain their true value. Let T be the set of run times reported by UKP5, MGREENDP1 or EDUK2 for the instance dataset described by a row. The meaning of the columns **avg**, **sd** and **max**, is, respectively, the arithmetic mean of T, the standard deviation of T, the maximum value of T. The time unit of the table values is seconds.

Instance desc.			UKP5			MGREENDP			PYAsUKP		
400 inst. per line			Subset-sum. Random c between $[5 \times 10^6; 10^7]$								
	n	w_{min}	**avg**	**sd**	**max**	**avg**	**sd**	**max**	**avg**	**sd**	**max**
See section 3.2.1			0.05	0.12	0.74	–	–	–	2.52	21.75	302.51
20 inst. per line			Strong correlation. Random c between $[20\overline{n}; 100\overline{n}]$								
α	**n**	w_{min}	**avg**	**sd**	**max**	**avg**	**sd**	**max**	**avg**	**sd**	**max**
5	5	10	0.03	0.00	0.03	0.24	0.03	0.29	1.57	1.78	3.62
		15	0.04	0.00	0.05	0.43	0.06	0.53	3.85	1.53	5.13
		50	0.13	0.00	0.16	1.01	0.52	1.70	12.12	8.17	28.84
5	10	10	0.06	0.00	0.06	0.50	0.04	0.54	0.00	0.00	0.01
		50	0.29	0.00	0.30	5.93	0.82	6.79	22.43	17.85	45.19
		110	0.66	0.00	0.66	16.05	3.36	19.68	76.53	62.54	175.61
-5	5	10	0.04	0.00	0.05	0.04	0.00	0.04	4.02	2.72	7.12
		15	0.05	0.00	0.05	0.05	0.00	0.05	6.76	4.22	12.24
		50	0.14	0.00	0.15	0.11	0.02	0.12	24.76	19.41	66.23
-5	10	10	0.10	0.00	0.10	0.11	0.01	0.13	6.74	6.28	15.38
		50	0.32	0.00	0.32	0.28	0.01	0.29	48.70	42.53	111.61
		110	0.65	0.00	0.66	0.52	0.01	0.53	144.87	143.53	416.41
200 inst. per line			Postponed periodicity. Random c between $[w_{max}; 2 \times 10^6]$								
	n	w_{min}	**avg**	**sd**	**max**	**avg**	**sd**	**max**	**avg**	**sd**	**max**
	20	20	0.79	0.10	0.97	0.74	0.11	0.96	8.65	7.74	28.63
	50	20	5.70	0.37	6.54	5.12	0.65	6.13	78.34	82.46	356.67
	20	50	0.89	0.12	1.19	0.75	0.14	1.09	11.57	8.20	39.20
	50	50	4.72	0.69	6.27	3.97	0.75	5.30	113.21	87.16	267.10
500 inst. per line			No collective dominance. Random c between $[w_{max}; 1000\overline{n}]$								
	n	w_{min}	**avg**	**sd**	**max**	**avg**	**sd**	**max**	**avg**	**sd**	**max**
	5	n	0.07	0.03	0.14	0.04	0.01	0.07	0.59	0.44	2.03
	10	n	0.65	0.31	1.30	0.33	0.10	0.60	2.34	1.86	8.44
	20	n	1.04	0.32	1.91	0.72	0.12	1.31	8.62	7.64	31.22
	50	n	3.64	0.36	4.74	3.56	0.20	4.46	73.49	72.26	279.01
qtd inst. per line			SAW. Random c between $[w_{max}; 10\overline{n}]$								
qtd	**n**	w_{min}	**avg**	**sd**	**max**	**avg**	**sd**	**max**	**avg**	**sd**	**max**
200	10	10	0.08	0.00	0.09	0.14	0.02	0.21	1.32	0.85	3.01
500	50	5	0.50	0.01	0.53	2.09	1.00	3.75	3.36	2.86	11.16
200	50	10	0.72	0.01	0.74	2.15	0.85	3.65	6.99	5.81	23.04
200	100	10	7.34	0.32	8.09	33.93	6.94	43.40	40.43	35.13	118.28

Table A.2: Results for the MTU implementatios over the reduced PYAsUKP's dataset (see Section 5.2.1). Columns **n** and w_{min} values must be multiplied by 10^3 to obtain their true value. Let T be the set of run times reported by CPP-MTU1, CPP-MTU2, F77-MTU1 and F77-MTU2, for the instance dataset described by a row (in this case, we don't count runs that ended in timeout). The meaning of the columns **avg** and **ter**, is, respectively, the arithmetic mean of T and the cardinality of T (i.e. the number of runs that didn't end in timeout). The time unit of the table values is seconds.

Instance desc.			CPP-MTU1		CPP-MTU2		F77-MTU1		F77-MTU2	
40 inst. per line			Subset-sum. Random c between $[5 \times 10^6; 10^7]$							
	n	w_{min}	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**
See section 3.2.1			0.04	40	0.04	40	0.00	40	154.97	8
2 inst. per line			Strong correlation. Random c between $[20\overline{n}; 100\overline{n}]$							
α	**n**	w_{min}	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**
5	5	10	0.00	1	–	0	0.00	1	–	0
		15	–	0	–	0	–	0	–	0
		50	–	0	–	0	–	0	–	0
5	10	10	0.00	1	–	0	0.00	1	–	0
		50	0.04	1	–	0	0.03	1	–	0
		110	0.01	1	–	0	0.00	1	–	0
-5	5	10	–	0	–	0	–	0	–	0
		15	–	0	–	0	–	0	–	0
		50	–	0	–	0	–	0	–	0
-5	10	10	0.00	1	–	0	0.00	1	–	0
		50	0.00	1	0.79	1	0.00	1	0.83	1
		110	0.00	1	–	0	0.00	1	–	0
20 inst. per line			Postponed periodicity. Random c between $[w_{max}; 2 \times 10^6]$							
	n	w_{min}	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**
	20	20	67.17	19	67.17	19	18.05	17	15.12	17
	50	20	101.93	18	2.15	20	134.09	17	143.47	17
	20	50	3.15	20	1.74	20	6.33	20	7.83	20
	50	50	2.22	20	21.13	20	4.45	20	13.81	20
50 inst. per line			No collective dominance. Random c between $[w_{max}; 1000\overline{n}]$							
	n	w_{min}	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**
	5	n	16.54	9	37.01	9	19.85	9	37.29	9
	10	n	147.08	6	5.84	5	34.41	5	10.09	5
	20	n	17.95	3	19.23	3	27.45	3	27.78	3
	50	n	13.36	2	1.40	2	26.73	2	2.64	2
qtd inst. per line			SAW. Random c between $[w_{max}; 10\overline{n}]$							
qtd	**n**	w_{min}	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**	**avg**	**ter**
20	10	10	1.33	20	12.92	20	2.54	20	2.87	20
50	50	5	43.08	46	43.97	19	59.14	38	38.63	16
20	50	10	55.14	45	87.68	19	47.05	41	85.97	19
20	100	10	10.10	16	38.41	17	20.23	16	44.41	16

A.1 Data and code related to CSP pricing subproblem dataset

The 596 selected instances of CSP are available in <https://github.com/henriquebecker91/masters/tree/8367836344a2f615640757ffa49254758e99fe0a/data/selected_csp_inst>, and the code used to solve the SCF relaxation can be found in the same repository (<https://github.com/henriquebecker91/masters/tree/8367836344a2f615640757ffa49254758e99fe0a/codes/cpp>). The code can be compiled by executing *make bin/cutstock* in the folder. Unfortunately, the code has external dependencies, and the user will need to install them before having success in the compilation. The dependencies are the Boost C++ library (see: <http://www.boost.org/>), and IBM ILOG CPLEX Studio 12.5 (see: <https://www.ibm.com/developerworks/community/blogs/jfp/entry/cplex_studio_in_ibm_academic_initiative?lang=en>). The library paths are configured inside the Makefile. The binaries generated inside the automatically created *bin* subfolder will have the same names as the ones used to identify them in Section 5.4.

A.2 Capítulo de resumo em português

A.2.1 Introdução

O Problema da Mochila com Repetições (PMR) é uma variante dos conhecidos Problema da Mochila com Limite de Itens (PMLI) e Problema da Mochila 0-1 (PM 0-1). A única diferença entre o PMR e essas outras duas variantes é que, no PMR, cada item possui uma quantidade de cópias ilimitada disponível. O PMR é NP-difícil e, portanto, não existe um algoritmo que possa resolvê-lo em tempo polinomial.

Instâncias do PMR são compostas pelo limite de peso c suportado pela mochila e um conjunto de n itens. Cada item tem dois valores associados: seu peso e o seu lucro. O objetivo do PMR é maximizar o lucro dos itens colocados dentro da mochila, enquanto respeitando o limite de peso da mochila.

A aplicação prática do PMR discutida nessa dissertação é: resolver os problemas de avaliação (*pricing subproblem*) que são gerados quando se resolve a relaxação contínua de uma instância do *Bin Packing Problem* (BPP) ou do *Cutting Stock Problem* (CSP) usando a formulação de cobertura de conjuntos e a abordagem de

geração de colunas. O BPP e o CSP são problemas clássicos da área de pesquisa operacional e importantes para a indústria, vide (DELORME; IORI; MARTELLO, 2014) e (GILMORE; GOMORY, 1961; GILMORE; GOMORY, 1963). Os melhores limites inferiores para a solução ótima desses dois problemas são os valores das soluções ótimas de suas relaxações contínuas. A formulação do BPP/CSP com menos simetrias tem um número exponencial de colunas e, por causa disso, é resolvida usando a abordagem de geração de colunas (GILMORE; GOMORY, 1961). Os problemas de avaliação gerados pela abordagem de geração de colunas (quando a mesma é aplicada ao BPP/CSP) são instâncias do PMR.

A.2.2 Trabalhos relacionados

Nesta seção é apresentada uma revisão bibliográfica sobre o PMR em ordem cronológica.

(GILMORE; GOMORY, 1961) Uma aplicação prática do PMR é proposta. Essa aplicação é a resolução dos 'problemas de avaliação' que surgem na resolução da relaxação contínua do BPP/CSP.

(GILMORE; GOMORY, 1966) São propostos algoritmos de programação dinâmica para o PMR no contexto dessa aplicação prática.

(MARTELLO; TOTH, 1977) Um algoritmo da abordagem *branch-and-bound* é proposto, e comparado com algoritmos de programação dinâmica usando pequenas instâncias artificiais, onde obtem os melhores resultados, por uma pequena margem.

(MARTELLO; TOTH, 1990a) O foco muda para solucionar instâncias grandes em pouco tempo. Instâncias artificiais são usadas com esse propósito. No contexto dessas instâncias, os algoritmos de programação dinâmica são claramente dominados pelos algoritmos de *branch-and-bound*.

(ZHU; BROUGHAN, 1997) É demonstrado que alguns desses conjuntos de instâncias artificiais possuem somente uma pequena quantidade de itens relevantes, esses conjuntos de instâncias são desacreditados.

(POIRRIEZ; ANDONOV, 1998) Um novo algoritmo de programação dinâmica é proposto, assim como novas instâncias artificiais.

(ANDONOV; POIRRIEZ; RAJOPADHYE, 2000) O novo algoritmo com-

para somente com algoritmos de *branch-and-bound* e o algoritmo de programação dinâmica ingênuo, os antigos algoritmos de programação dinâmica não ingênuos foram esquecidos ou excluídos devido a experimentos anteriores.

(KELLERER; PFERSCHY; PISINGER, 2004) O novo algoritmo é considerado o estado da arte.

(POIRRIEZ; YANEV; ANDONOV, 2009) O novo algoritmo de programação dinâmica é hibridizado com *branch-and-bound*, e os conjuntos de dados são atualizados para serem mais difíceis. Estes conjuntos de dados são difíceis para *branch-and-bound*, e o algoritmo híbrido compara somente com *branch-and-bound*. O método híbrido é o novo estado da arte.

(BECKER; BURIOL, 2016) Um algoritmo de programação dinâmica antigo é redescoberto e tem um desempenho melhor que o estado da arte quando executado sobre o conjunto de instâncias mais recente proposto.

(essa dissertação) Algoritmos antigos são revistos, reimplementados e testados. A influência dos conjuntos de instâncias e o contexto histórico se torna aparente.

A.2.3 Classes de instâncias

Esta seção trata das classes de instâncias usadas nos experimentos apresentados neste trabalho. Essas instâncias são diferenciadas principalmente pela distribuição dos seus itens, ou seja qual a relação entre o peso e o lucro de cada item. Neste trabalho não foram utilizadas instâncias com uma distribuição de itens sem correlação (peso e lucro de cada item escolhidos aleatoriamente), pois estas foram desacreditadas em (ZHU; BROUGHAN, 1997).

A.2.3.1 Instâncias do PYAsUKP

O conjunto de instâncias proposto em (POIRRIEZ; YANEV; ANDONOV, 2009), e reutilizado na comparação apresentada em (BECKER; BURIOL, 2016), será referenciado como *conjunto de instâncias do PYAsUKP*. Estas são instâncias geradas artificialmente com o propósito de serem "difíceis de resolver". O conjunto de instâncias usado neste trabalho é similar ao usado em (POIRRIEZ; YANEV; ANDONOV, 2009). A mesma ferramenta (PYAsUKP) foi usada para gerar os conjuntos de dados, e os mesmos parâmetros foram usados, com exceção das instâncias

subset-sum, que foram ampliadas multiplicando os seus parâmetros por dez. Algumas instâncias fazem uso de sementes aleatórias que não foram publicadas, então as exatas instâncias usadas em (POIRRIEZ; YANEV; ANDONOV, 2009) podem ser diferentes. As instâncias usadas aqui são exatamente as mesmas que foram usadas em (BECKER; BURIOL, 2016).

O conjunto de instâncias do PYAsUKP é composto 4540 instâncias provenientes de cinco conjuntos de instâncias menores. Os cinco conjuntos menores são: *subset-sum*, com $10^3 \leq n \leq 10^4$ e $5 \times 10^6 \leq c \leq 10^7$, totalizando 400 instâncias; *strongly correlated*, com $n = 10^4$ e $2010000 \leq c \leq 10010000$, totalizando 240 instâncias; *postponed periodicity*, com $20000 \leq n \leq 50000$ e $1020000 \leq c \leq 2 \times 10^6$, totalizando 800 instâncias; *without collective dominance*, com $5000 \leq n \leq 50000$ e $105000 \leq c \leq 100050000$, totalizando 2000 instâncias; *SAW*, com $10^4 \leq n \leq 10^5$ e $11000 \leq c \leq 10100000$, totalizando 1100 instâncias.

Em cada um desses cinco conjuntos menores de instâncias, para mesma combinação de parâmetros, o número de instâncias gerado é sempre perfeitamente divisível por dez. Dessa forma, é possível definir um conjunto de instâncias composto de um décimo das instâncias do PYAsUKP (ou seja, 454 instâncias), e que possui pelo menos uma instância para cada combinação distinta de parâmetros usada. Essa distribuição reduzida será referenciada como: *o conjunto de instâncias reduzido do PYAsUKP*.

A.2.3.2 Problemas de avaliação gerados a partir do BPP/CSP

O *Bin Packing Problem* (BPP) e o *Cutting Stock Problem* (CSP) são problemas clássicos da área de pesquisa operacional. Quando a relaxação contínua de uma instância do BPP/CSP é resolvida usando a formulação de cobertura de conjuntos e a abordagem de geração de colunas, são gerados de dezenas até milhares de 'problemas de avaliação' (*pricing subproblems*). Cada um desses problemas de avaliação é uma instância do PMR.

Para uma mesma instância do BPP/CSP, o número de instâncias do PMR geradas, e o lucro dos itens em uma dada instância, pode variar baseado na escolha da solução ótima. Uma instância do PMR pode ter múltiplas soluções ótimas, entretanto, somente uma delas é usada pela abordagem de geração de colunas, que define os próximos problemas de avaliação gerados. Consequentemente, este conjunto de instâncias do PMR é difícil de descrever, ele possue um número grande e variável

de instâncias com valores de lucro dos itens também variáveis. A melhor forma encontrada pelo autor para garantir que os resultados são reproduzíveis é disponibilizando publicamente os códigos usados nos experimentos, conjuntamente com o conjunto de instâncias do BPP/CSP da literatura usado nos experimentos. Os códigos são determinísticos e, consequentemente, irão produzir os mesmos resultados se executados várias vezes sobre a mesma instância.

Um *survey* recente sobre o BPP e o CSP reuniu instâncias da literatura, e também propôs novas (DELORME; IORI; MARTELLO, 2014). O número total de instâncias em todos conjuntos de instâncias apresentados no *survey* é de 5692. O autor dessa dissertação escolheu 10% dessas instâncias para os experimentos realizados nesse trabalho. Essa fração das instâncias foi escolhida aleatoriamente dentre instâncias do mesmo conjunto ou, em conjuntos maiores, dentre os mesmos parâmetros de geração. As 596 instâncias do BPP/CSP selecionadas estão disponíveis no repositório GitHub do autor[1].

A.2.3.3 Instâncias BREQ

O *Bottom Right Ellipse Quadrant* (abreviado como BREQ, e que pode ser traduzido como ‘Quadrante de Elipse Inferior Direito’) é um tipo de distribuição de itens proposto pelo autor dessa dissertação. O nome dessa distribuição é derivado do fato que, quando plotado em um gráfico (com o lucro e o peso como eixos x e y), os itens formam um quarto de elipse (especificamente, o quadrante inferior direito). Essa distribuição foi criada para ilustrar que diferentes distribuições de itens favorecem diferentes abordagens e, consequentemente, a escolha de conjuntos de instâncias (ou especificamente, a distribuição dos itens) define o que é considerado o melhor algoritmo.

As instâncias BREQ favorecem algoritmos fazem uso de limites, como aqueles que fazem uso da abordagem *branch-and-bound* (que pode ser traduzida literalmente

[1]As instâncias podem ser encontradas em <https://github.com/henriquebecker91/masters/tree/8367836344a2f615640757ffa49254758e99fe0a/data/selected_csp_inst>, e o código usado para resolver a relaxação pode ser encontrado no mesmo repositório (<https://github.com/henriquebecker91/masters/tree/8367836344a2f615640757ffa49254758e99fe0a/codes/cpp>). O código pode ser compilado executando *make bin/cutstock* no diretório. Infelizmente, o código pode ter dependências externas, e o usuário precisará instalar eles antes de ter sucesso na compilação. As dependências são a biblioteca Boost C++ (veja: <http://www.boost.org/>), e o IBM ILOG CPLEX Studio 12.5 (veja: <https://www.ibm.com/developerworks/community/blogs/jfp/entry/cplex_studio_in_ibm_academic_initiative?lang=en>). Os caminhos da biblioteca são configurados dentro do Makefile.

como 'ramificar-e-limitar'). Dessa forma, caso um conjunto de instâncias baseado no BREQ seja utilizado em uma comparação entre algoritmos de programação dinâmica e algoritmos de *branch-and-bound*, os algoritmos de B&B serão favorecidos. Como veremos na seção de experimentos, muitas das instâncias do PYAsUKP levam muito mais tempo para serem solucionadas por algoritmos de B&B do que por algoritmos de programação dinâmica. Consequentemente, as conclusões sobre qual é o melhor algoritmo para o PMR seriam contrárias caso fossem somente usadas as instâncias do BREQ, do que se fossem usadas somente as instâncias do PYAsUKP.

O conjunto de instâncias utilizado na seção de experimentos consiste em um total de cem instâncias, dez instâncias para cada um dos dez valores de n utilizados: 10^{11}, 10^{12}, 10^{13}, 10^{14}, 10^{15}, 10^{16}, 10^{17}, 10^{18}, 10^{19} e 10^{20}. A capacidade da mochila para uma determinada instância é $128 \times n$.

A.2.4 Abordagens e algoritmos

As abordagens mais comuns usadas para solucionar o PMR são a programação dinâmica e o *branch-and-bound* (B&B). A conversão de instâncias do PMR em instâncias de outras variantes do problema da mochila (PM 0-1 e PMLI) não é usada pois costuma resultar em perda de desempenho.

Algoritmos de programação dinâmica para o PMR tem um pior caso pseudo-polinomial de $O(nc)$ (onde n é o número de itens, e c é a capacidade da mochila), e um uso de memória de $O(n + c)$. Os algoritmos de programação dinâmica (PD) que são usados nos experimentos são: UKP5 e EDUK.

O UKP5 foi proposto pelo autor dessa dissertação em (BECKER; BURIOL, 2016). Após a publicação, o autor percebeu que o UKP5 era basicamente o mesmo algoritmo que o *ordered step-off* apresentado em (GILMORE; GOMORY, 1966). O pseudo-código do UKP5/'*ordered step-off*' pode ser encontrado no Algoritmo 3 da Seção 4.2.4.

Algoritmos de B&B para o PMR tem um pior caso exponencial, e um uso de memória de $O(n)$. Os algoritmos de B&B que são usados nos experimentos são: MTU1 e MTU2. Alguns algoritmos combinam ambas abordagens (PD e B&B). Os algoritmos híbridos que são usados nos experimentos são: EDUK2 e GREENDP.

Os seguintes algoritmos foram implementados em C++ pelo autor desse trabalho para realização de experimentos: UKP5, MTU1, MTU2 e GREENDP.

Os seguintes algoritmos já possuiam uma implementação disponível publicamente: EDUK (OCaml), EDUK2 (OCaml), MTU1 (Fortran77) e MTU2 (Fortran77).

A.2.5 Experimentos e análises

Cinco experimentos são apresentados nessa seção. O primeiro experimento consistiu na execução de UKP5, EDUK2, e GREENDP sobre o conjunto de instâncias do PYAsUKP. O maior tempo que o UKP5 dispendeu em uma única instância foi 20 segundos. O GREENDP teve resultados similares ao UKP5, usando significativamente mais tempo em algumas instâncias e marginalmente menos tempo em outras. O maior tempo que o MGREENDP dispendeu em uma única instância foi 43 segundos. O PYAsUKP dispendeu consideravelmente mais tempo do que o UKP5 e o MGREENDP em cada conjunto de instâncias. Na maioria dos conjuntos de instâncias, o PYAsUKP dispendeu pelo menos dez vezes mais tempo que o UKP5 ou o MGREENDP. O maior tempo que o PYAsUKP dispendeu em uma única instância foi 416 segundos. Para mais informações, consultar a tabela A.1 que se encontra nesse apêndice.

O segundo experimento consistiu na execução de MTU1 (C++), MTU2 (C++), MTU1 (Fortran77) e MTU2 (Fortran77) sobre o conjunto de instâncias reduzido do PYAsUKP (454 instâncias). A implementação original em Fortran foi comparada com a implementação do autor dessa dissertação em C++. Considerando o conjunto de instâncias utilizado, ambos algoritmos (em ambas implementações) não são competitivos com os algoritmos de PD e híbridos previamente testados (UKP5, MGREENDP e PYAsUKP) em relação ao tempo de execução. Todas as quatro implementações falharam em resolver cerca de metade das instâncias antes do limite de tempo de 1000 segundos por instância. Entre as implementações do MTU1 (C++ e Fortran) não há diferença significativa. Entre as implementações do MTU2, pela forma como foi implementada a ordenação dos itens, a implementação em C++ usa menos tempo, em especial para instâncias do tipo *subset-sum*. Para mais informações, consultar a tabela A.2 que se encontra nesse apêndice.

O terceiro experimento consiste na execução de todos os algoritmos mencionados na Seção A.2.4 sobre as cem instâncias do *BREQ 128-16 Standard Benchmark*. Os resultados demonstram que algoritmos de B&B, ou que possuem cálculo de limites (*bounds*), tem ampla vantagem sobre algoritmos que utilizam unicamente

programação dinâmica, considerando somente esse conjunto de instâncias. Isso corrobora o argumento de que a escolha de instâncias define o que é considerado o melhor algoritmo. Os resultados desse experimento podem ser visualizados na figura 5.3. Para mais informações, consultar a tabela A.3 que se encontra nesse apêndice.

O quarto experimento consiste na execução do UKP5, MTU1 e o CPLEX para solucionar os problemas de avaliação gerados a partir de instâncias do BPP/CSP. Os resultados mostram que o CPLEX não é competitivo para solução dos problemas de avaliação. Quando o MTU1 e o UKP5 são usados para solucionar os problemas de avaliação, para a maioria das instâncias, solucionar todos os problemas de avaliação de uma determinada instância do BPP/CSP toma menos de um segundo. Cerca de cinquenta instâncias do conjunto de instâncias do BPP/CSP exigem mais do que um segundo para que o UKP5 solucione todos os problemas de avaliação gerados pela instância (até um máximo de 50 segundos em uma das instâncias). Quando o CPLEX e o MTU1 são usados para solucionar os problemas de avaliação dessas cinquenta instâncias mais difíceis, cada uma dessas instâncias excede o limite de tempo de dez minutos por instância do BPP/CSP.

O quinto experimento consiste na execução do UKP5 e do PYAsUKP (EDUK2) em dois computadores com diferentes quantidades de cache compartilhada, de forma serial e paralela, sobre a versão reduzida do conjunto de instâncias do PYAsUKP (454 instâncias). O paralelismo aqui citado se refere a execuções independentes (i.e. sobre diferentes instâncias), em cores isolados, com o propósito de reduzir o tempo necessário para realizar um *benchmark*. Os resultados mostram que as execuções em paralelo tomam mais tempo que as execuções seriais (comparando cada execução paralela com a serial correspondente). Isto indica que mesmo em *cores* isolados, a execução concorrente afeta os tempos de execução. O UKP5 é mais afetado por esse efeito que o PYAsUKP, o que atribuímos ao maior uso de memória e, consequentemente, maior disputa pela cache compartilhada. Todos os experimentos descritos anteriormente foram executados de forma serial, para evitar esse ruído.

A.2.6 Conclusões

A análise crítica da literatura, conjuntamente com o resultado dos experimentos apresentados nesse trabalho, leva a crer que a escolha de instâncias e algoritmos usados nos experimentos de trabalhos anteriores permitiu que um algoritmo antigo,

porém eficiente, fosse esquecido pela comunidade científica. Este algoritmo é o *ordered step-off* que é implementado, com pequenas alterações, pelo UKP5.

Outras contribuições são o conceito de dominância de solução (fraca e forte), que é utilizado pelo *ordered step-off* mas não havia sido conceitualizado. Além da evidência trazidas pelos experimentos de que: o pior caso do B&B pode ocorrer em problemas de avaliação (*pricing subproblems*); a escolha entre soluções ótimas para um problema de avaliação altera consideravelmente o número de problemas de avaliação gerados subsequentemente (para uma mesma instância do BPP/CSP); algoritmos do PMR que fazem uso intensivo da memória apresentam alteração nos tempos quando executados concorrentemente porém em *cores* isolados; a conversão do lucro dos problemas de avaliação de ponto flutuante para inteiro não causa perda significativa no valor da solução ótima da instância do BPP/CSP subjacente.