European Journal of Operational Research 277 (2019) 84–99

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

An empirical analysis of exact algorithms for the unbounded knapsack problem

Henrique Becker*, Luciana S. Buriol*

Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

ARTICLE INFO

Article history:
Received 24 May 2018
Accepted 4 February 2019
Available online 12 February 2019

Keywords:
Combinatorial optimization
Unbounded knapsack problem
Dynamic programming
Integer programming
Branch and bound

ABSTRACT

This work presents an empirical analysis of exact algorithms for the unbounded knapsack problem, which includes seven algorithms from the literature, two commercial solvers, and more than ten thousand instances. The terminating step-off, a dynamic programming algorithm from 1966, presented the lowest mean time to solve the most recent benchmark from the literature. The threshold and collective dominances are properties of the unbounded knapsack problem first discussed in 1998, and exploited by the current state-of-the-art algorithms. The terminating step-off algorithm did not exploit such dominances, but has an alternative mechanism for dealing with dominances which does not explicitly exploits collective and threshold dominances. Also, the pricing subproblems found when solving hard cutting stock problems with column generation can cause branch-and-bound algorithms to display worst-case times. The authors present a new class of instances which favors the branch-and-bound approach over the dynamic programming approach but do not have high amounts of simple, multiple and collective dominated items. This behaviour illustrates how the definition of hard instances changes among algorithm approaches. The codes used for solving the unbounded knapsack problem and for instance generation are all available online.

1. Introduction

The objective of this work is to provide an extensive comparison of the exact algorithms for solving the Unbounded Knapsack Problem (UKP). Given the weight capacity of a knapsack and a collection of items (each with a weight and a profit value), the UKP consists in choosing how many copies of each item will be packed in the knapsack to maximize the profit carried by it while respecting its weight capacity. The UKP is similar to the Bounded Knapsack Problem (BKP) and the 0–1 Knapsack Problem (0–1 KP). The only difference between the UKP and the BKP (or the 0–1 KP) is that the UKP has an unlimited quantity of each item available. The UKP is a weakly NP-Hard problem, as are the BKP and the 0–1 KP.

Next the authors present some selected papers and a summary of their relevance for the UKP.

Summary of the prior work

Gilmore and Gomory (1961) The column generation approach for the CSP linear programming relaxation is proposed; the UKP is the pricing problem.
Gilmore and Gomory (1966) The ordered and the terminating step-off algorithms (dynamic programming algorithms to solve the UKP) are proposed.
Martello and Toth (1977) The MTU1 (branch-and-bound algorithm) is proposed, and then compared with the previous algorithms over artificial instances up to a hundred items, obtaining slightly better results
Martello and Toth (1990) Datasets of instances with up to 250,000 items (but rich in simple and multiple dominances) are proposed. MTU2 is proposed as an improvement of MTU1 for such dataset.
Andonov, Poirriez, and Rajopadhye (2000); Poirriez and Andonov (1998) EDUK (a dynamic programming algorithm) is proposed. Threshold dominance is proposed. EDUK is the first algorithm to exploit collective and threshold dominances. Old datasets receive some criticism for their high percentage of dominated items. New artificial datasets without the same flaws of the previous datasets are proposed. EDUK is compared to MTU2.
Kellerer, Pferschy, and Pisinger (2004) A book on knapsack problems cite EDUK as state-of-the-art dynamic programming for the UKP.
Poirriez, Yanev, and Andonov (2009) EDUK2 is proposed. It consists in EDUK hybridized with B&B. The datasets are updated to be 'harder'. MTU2 is used in some comparisons, but not in all because it has the risk of integer overflow. EDUK2 is compared to EDUK.
Becker and Buriol (2016) The terminating step-off is reinvented (with the name of UKP5) and outperforms the hybrid method in the updated datasets.

The UKP is the pricing subproblem generated by solving the linear programming relaxation of the set covering formulation for

* Corresponding authors.
E-mail addresses: hbecker@inf.ufrgs.br (H. Becker), buriol@inf.ufrgs.br (L.S. Buriol).

https://doi.org/10.1016/j.ejor.2019.02.011
0377-2217/

the unidimensional Bin Packing Problem (BPP) and Cutting Stock Problem (CSP) using the column generation approach (Gilmore & Gomory, 1961; 1963). The BPP and the CSP are classical optimization problems in the area of operations research (Delorme, Iori, & Martello, 2016). The best lower bounds known for the BPP and CSP are their linear programming relaxations of the set covering formulation. This is the tightest formulation for these problems but it has an exponential number of columns, and thus is solved by using the column generation approach (Gilmore & Gomory, 1961). However, recently Delorme and Iori (2017) proved that a pseudo-polynomial formulation (dynamic programming-flow formulation) is equivalent to the set covering formulation and, therefore, provides the same lower bounds for the problem.

The main contributions made by this paper follow:

1. A comprehensive empiric analysis including several algorithms, solvers, and datasets, in a single self-contained paper – in Section 5;
2. An updated pseudocode of the ordered step-off (Gilmore & Gomory, 1966), using the current nomenclature and a new tiebreaker – in Section 2;
3. The concept of *solution dominance* which generalizes all previously proposed dominances – in Section 2;
4. The evidence that *partial solution dominance* is a competitive alternative to the state-of-the-art application of simple, multiple, collective, and threshold dominances – in Section 5;
5. The discussion on how tighter bounds for periodicity do not help to improve the performance of state-of-the-art algorithms – in Section 3;
6. A new item distribution and how it favors a solving approach in relation to other (B&B over DP) – in Sections 4.2.3 and 5.3;
7. A study of the item distribution evolution in CSP/BPP pricing problems and some of its implications (how it biases against some UKP-solving approaches, how it is affected by 'hard' CSP/BPP instances) – in Section 5.4;

In the remainder of this section we discuss the mathematical formulation and well-known properties of the problem.

1.1. Formulation and notation

An instance of the UKP is a pair of a capacity c and a list of n items. Each item i can be referenced by its index in the item list $i \in \{1, \ldots, n\}$, and 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, of a solution s is denoted by w_s, or p_s, and is referred to as the weight, or profit, of the solution.

The mathematical formulation of UKP is

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

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq c, \tag{2}$$

$$x_i \in \mathbb{N}_0. \tag{3}$$

The quantities of each item i in a solution are denoted by x_i, and are restricted to the non-negative integers, as Eq. (3) indicates. We assume that the capacity c, the quantity of items n and the weights of the items w_i are positive integers, while the profit values of the items p_i are positive real numbers.

The efficiency of an item i is its profit-to-weight ratio (p_i/w_i). We use w_{min}, or w_{max}, to denote the lowest weight among all items, or the highest weight among all items, within an instance of the UKP. We refer to the item with the greatest efficiency among all items of a specific instance as the *best item* (or b). If more than one item shares the greatest efficiency, then the item with the lowest weight among them is considered the best item type. If more than an item has both previously stated characteristics (i.e., they are equal), then the first item with both characteristics in the items list is the best item. The authors make such distinction because the instance generators do not guarantee unique items, and is often faster to let the algorithms themselves deal with the replicas than running a preprocessing phase to remove duplicated items.

1.2. Properties of the UKP

Algorithms that solve the UKP often exploit two properties to reduce the problem size: *dominance* and *periodicity*. Dominance relations are exploited to reduce n, and periodicity is exploited to reduce c.

1.2.1. Simple, multiple, collective, and threshold dominances

Any item j that does not appear in all optimal solutions of an instance can be excluded without affecting our capability of solving such instance. Given two items i and $j : j \neq i$, if $w_i \leq w_j$ and $p_i \geq p_j$, then j cannot be in all optimal solutions, since for any optimal solution that contains j there will exist another optimal solution with i in place of j. Consequently, j can be ignored when solving an instance that contains both i and j. Such relationship between i and j is called *simple dominance*, more specifically we can say that i simple dominates j (or that j is simple dominated by i). For an example, (5, 5) simple dominates (6, 1), as shown in Fig. 1.

Given a positive integer α, if $\alpha \times w_i \leq w_j$ and $\alpha \times p_i \geq p_j$, then α copies of i can replace one of j, and j can be ignored. Such relationship is called *multiple dominance*, and it generalizes simple dominance in which $\alpha = 1$. For an example, $\alpha = 2$ copies of (5, 5) multiple dominates (12, 9).

Given a valid solution s, if $w_s \leq w_j$ and $p_s \geq p_j$, then the items that compose s can replace j, and j can be ignored. Such relationship is called *collective dominance* (Poirriez & Andonov, 1998), and it generalizes multiple dominance in which s consists of α copies of i. For an example, solution {(5, 5), (5, 5), (3, 2)} collective dominates (14, 11).

If $w_s \leq \beta \times w_j$ and $p_s \geq \beta \times p_j$, then the items that compose s can replace β copies of j, and solutions including β or more copies of j can be ignored. Such relationship is called *threshold dominance*, and it generalizes collective dominance in which $\beta = 1$. Threshold dominance with $\beta > 1$ does not allow to exclude an item j as a preprocessing phase, but it reduces the search space by allowing the algorithm to ignore all solutions in which $x_j \geq \beta$. For an example, (5, 5) threshold dominates $\beta = 2$ copies of (3, 2).

1.2.2. Periodicity

The *periodicity* property shows the existence of a capacity y^+ such that, for every capacity $y' : y' > y^+$, there exists an optimal solution for capacity y' that is the same as an optimal solution for capacity $y' - w_b$ except it includes one more copy of the best item b (Gilmore and Gomory, 1966, p. 10). If $y^+ < c$, then the UKP can be solved for capacity $y^* = c - \lceil (c - y^+)/w_b \rceil w_b$, and the gap between y^* and c filled with exactly $(c - y^*)/w_b$ copies of the best item b, effectively reducing the knapsack size from c to y^* (see Fig. 2). However, the effort necessary to compute the exact value of y^+ (and y^*) is about the same as solving the UKP for all capacities $y \leq y^+$ while checking for threshold dominance. Consequently, y^+ is implicitly determined by some dynamic programming algorithms (as EDUK) while solving instances in which $y^+ < c$. An upper bound on y^+ is less valuable than y^+ itself, but it can be computed in polynomial time, as a preprocessing phase. The authors discuss the usefulness of computing

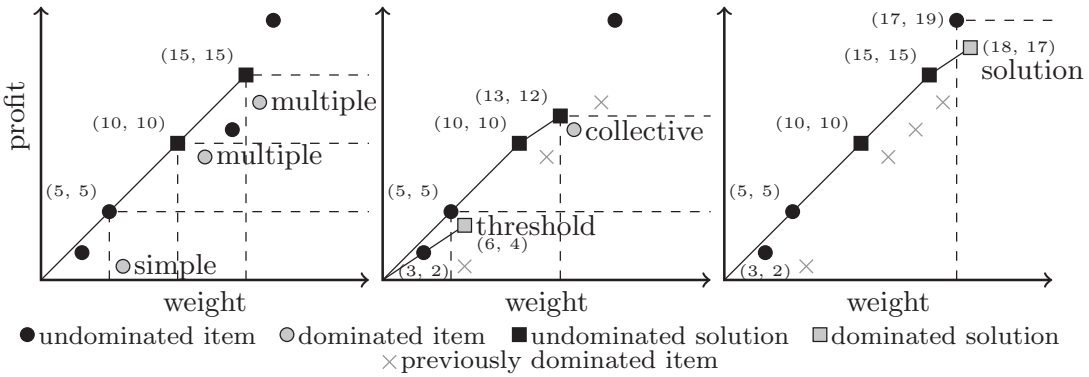

Fig. 1. Examples of the simple, multiple, collective, threshold and solution dominances. The solution dominance is proposed by the authors in Section 2. The item set considered in the three graphs is: (3, 2), (5, 5), (6, 1), (12, 9), (14, 11), (16, 13), (17, 19). An item or solution dominates any item or solution that have the same weight or more and the same profit or less (dashed lines). The dominated items and solutions are labeled with the type(s) of dominance they are subject to. Source: primary.

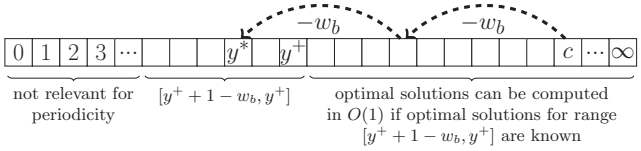

Fig. 2. The knapsack capacities and their connection with periodicity in an instance where $y^+ < c$. Source: primary.

or bounding y^+ in Section 3. The periodicity property is a direct consequence of the threshold dominance. Except by the best item b, every item j: $j \neq b$ is threshold dominated for some α_j. For example, given that y is the lowest common multiple of w_b and w_j, $\beta_j = y/w_b$ and $\alpha_j = y/w_j$, then $\beta_j \times w_b \leq \alpha_j \times w_j \equiv y \leq y$ and $\beta_j \times p_b \geq \alpha_j \times p_j$. Consequently, the threshold dominance defines a constraint $x_j < \alpha_j$ on every non-best item j. All solutions that do not break such constraints and do not make use of the best item weight less than $y'' = \sum \alpha_j \times w_j$. Consequently, an optimal solution for any capacity greater than y'' can be obtained by the procedure described in the previous paragraph. The capacity y'' is an upper bound on y^+ (Kellerer et al., 2004, p. 215).

2. The revisited ordered step-off and (partial) solution dominance

The ordered step-off (OSO) is a dynamic programming algorithm for the UKP (Gilmore & Gomory, 1966). The OSO is described in Algorithm 1, a complementary overview follows. The arrays g and d store, respectively, the profit value of generated solutions and the index of the most efficient item present in such solutions. Together they can be seen as a solution pool indexed by solution weight. A summary of the algorithm follows: the solution pool is initialized with all single-item solutions; the solution pool is iterated by weight order; for each solution, the solution pool is expanded with new solutions, each new solution is a copy of the current solution plus an extra item. This process enumerates all undominated solutions, and it returns the optimal profit value. The algorithm skips the creation of new solutions from old solutions already known to be dominated (lines 11 to 13), discards some dominated solutions created (e.g., if two or more solutions share the same weight, the algorithm keeps only one of these which is tied for the highest profit), and avoids considering symmetric solutions (by restricting the loops up to $d[y]$, the algorithm only considers the permutation of the solution in which the items are added in order of efficiency). When the algorithm finishes executing, *opt* contains the optimal profit value and for every $g[y] > 0$ there is a solution with: weight y, profit value $g[y]$, and the index of the last (and most efficient) item added $d[y]$.

Algorithm 1 The revisited ordered step-off (R-OSO).

```
1:  procedure oso(n, c, w, p)
2:      Sort w and p by non-increasing item efficiency*
3:      Find the values of w_min and w_max
4:      g ← array of profit values, size c + 1, zeroed
5:      d ← array of item indexes, size c + 1, uninitialized
6:      for i ← n to 1 do                    ▷ Stores one-item solutions
7:          g[w_i] ← p_i
8:          d[w_i] ← i
9:      end for
10:     opt ← 0
11:     for y ← w_min to c do
12:         if g[y] ≤ opt then               ▷ Prunes dominated solutions
13:             continue                     ▷ Skips current iteration
14:         end if
15:         opt ← g[y]
16:         for i = 1 to d[y] do
17:             if y + w_i > c then          ▷ Skips invalid solutions
18:                 continue
19:             end if
20:             if g[y + w_i] < g[y] + p_i then
21:                 g[y + w_i] ← g[y] + p_i
22:                 d[y + w_i] ← i
23:             else if g[y + w_i] = g[y] + p_i and i < d[y + w_i] then
24:                 d[y + w_i] ← i
25:             end if
26:         end for
27:     end for
28:     return opt
29: end procedure
```

*Items that share the same efficiency are sorted by non-decreasing weight.

The authors revisited OSO and added the *else if* found in lines 22 to 24 of Algorithm 1 (to distinguish this version from the original one, it will be called Revisited OSO, or R-OSO). When two or more solutions generated have the same weight and profit (a special case of solution dominance), the original algorithm arbitrarily keeps the first solution generated and ignores the others. In the same situation, the revisited algorithm keeps the solution that will generate less descendents (smallest $d[y]$), therefore minimizing the computational effort. This change of tiebreaker reduced the run times of the algorithm over subset-sum and strongly correlated instances by orders of magnitude. Subset-sum instances are UKP instances with items respecting $\forall i.\ p_i = w_i$, while strongly correlated

instances have items respecting $\forall i.\ p_i = w_i + \alpha$ (where α is a small positive integer value which is the same for the whole instance). Therefore, the performance gain can be explained by the fact that, in subset-sum instances, all solutions with the same weight have the same profit and, in strongly correlated instances, all solutions with the same weight and the same number of items have the same profit.

Given two valid solutions s and t, if $w_s \leq w_t$ and $p_s \geq p_t$, then the items of solution s can replace the items of solution t, and any solution that is a superset of t can be ignored (see Fig. 1). The authors call this relationship *solution dominance*, and it generalizes threshold dominance in which solution t can only consist of copies of the same item. As far the authors know, the concept of solution dominance was not formalized before. The authors believe such overlook happened because: (1) the original concept of dominance focused on discarding single items from further consideration; (2) an algorithm that fully exploits such dominance relation has not been proposed.

The authors also propose the term *partial solution dominance* to describe the mechanism by which (R-)OSO avoids the generation of some dominated solutions (but not all of them) with no overhead. The mechanism consists in skipping the generation of up to n child solutions (and all their descendents) because the original solution is already known to be dominated at the time they will be generated (lines 11 to 13), combined with the fact no solution is enumerated two times (no symmetric solution thanks to the use of $d[y]$). Our experiments show that, while *partial solution dominance* lacks the same guarantees that applying simple, multiple, collective, and threshold dominances as soon as possible (as in EDUK), its performance is state of the art.

A formal description of which dominated solutions are guaranteed to not be generated and which ones may yet be generated follows. The notation $min_{ix}(s)$ refer to the index of the item of lowest index present in solution s; the notation $max_{ix}(s)$ has the analogue meaning. If a solution t is skipped because solution s dominates t (lines 11 to 13), only the solutions u where $t \subset u$ and $max_{ix}(u - t) \leq min_{ix}(t)$ are guaranteed to not be generated anymore. However, any solutions $t \cup \{i\} : i > min_{ix}(t)$ yet generated are dominated by $s \cup \{i\}$ (or by a solution that dominates $s \cup \{i\}$). Consequently, such solutions are guaranteed to be skipped and the amount of supersets of t generated is further reduced. Unfortunately, after skipping those solutions, the solutions in the set $t \cup \{i, j\}$: $j > i > min_{ix}(t)$ may be yet generated, and so on.

For an example, if {3, 2} is dominated, then {3, 2, 2} and {3, 2, 1} are not generated by OSO and, consequently, {3, 2, 2, 2}, {3, 2, 2, 1} and {3, 2, 1, 1} are also not generated, and so on. However, given {4, 3} is not dominated, it will be used to generate {4, 3, 2} (which is a superset of {3, 2}, {4, 2} and {4, 3}), and is dominated as {3, 2} is dominated. Ideally, {4, 3, 2} should not be generated, but: (1) if the weight of {4, 3} is lower than {3, 2}, then {3, 2} is not even known to be dominated at the time {4, 3, 2} is generated; (2) if the weight of {4, 3} is greater than {3, 2}, yet we would need to check all subsets of {4, 3, 2} with one less item, this is a $O(n)$ overhead, greater than letting the dominated solution be generated, and skipping it after.

The (R-)OSO partial solution dominance mechanism is rendered useless if the instance has some properties. However, such properties seem to not be found in instances of the literature, nor seem to arise naturally in the real-world or in current artificial item distributions. The properties are: the same item has the lowest weight and the lowest efficiency; the second lowest weight is orders of magnitude larger than the lowest weight. In the most extreme cases, this is, when the least efficient item has one unit of weight and the second lowest weight is close to c, the (R-)OSO will display its worst-case behavior, reverting to the performance of the naïve DP algorithm (nc steps). This happens because: there will be a solution for each capacity (i.e., no sparsity); the least efficient item has index n and each solution visited before the second lowest weight will, therefore, generate n new solutions (many possibly discarded for being larger than the knapsack).

3. A critique of periodicity checks and bounds

State-of-the-art dynamic programming algorithms like EDUK (Andonov et al., 2000) and the terminating step-off (Gilmore & Gomory, 1966) finish the execution early if they detect that only the best item will be used to generate new solutions. This mechanism is called *periodicity check*. If a periodicity check finishes the execution at capacity y, it saves the effort of iterating the last $c - y$ capacities. However, as only the best item is yet used, the algorithm would execute only $\Theta(1)$ operations for each capacity in the range $[y + 1, c]$. Periodicity checks often demand about n and up to $y \leq c$ extra operations to save up to $c - y$ operations. Consequently, periodicity checks save little effort if any, in comparison to the application of dominances which they depend on to work.

If periodicity checks save little effort, then the computation of upper bounds on y^+ save up even less effort. When the global item list of EDUK is reduced to the best item, the current capacity is probably a far better upper bound on y^+ than any upper bound computed by a polynomial algorithm. While branch-and-bound algorithms do not have periodicity checking, they also do not obtain any considerable benefit from periodicity bounds, as their bound computation already supersedes them. Consequently, periodicity checks and bounds are of little relevance for the performance of state-of-the-art algorithms for the UKP.

Poirriez et al. (2009) states that if $w_{max} \leq c/2$ then "computing all optimal states $(y, f(N, y))$ with $y \leq c/2$ is enough[1], since any knapsack with capacity $y \in]c/2, c]$ can be solved by completing the solution of $UKP^{y-c/2}_{\mathbf{w,b}}$ with the one of $UKP^{c/2}_{\mathbf{w,b}}$". If this claim was correct and $w_{max} \leq c/2^x : x \geq 1$, then a dynamic programming algorithm for the UKP would never need to solve for a capacity greater than or equal to $2 \times w_{max}$. It would suffice to solve the instance up to capacity $c/2^x$ and then multiply the number of items of each type in an optimal solution by 2^x. Unfortunately, the claim is incorrect. A counterexample is presented below. Consider the following instance: $c = 6,\ n = 2,\ w_1 = p_1 = 1,\ w_2 = 2,\ p_2 = 10$. The optimal solution for capacity $c/2 = 3$ is 11 (one copy of each item), but the optimal solution for c is 30 (three copies of item two) not 22 (two copies of each item).

4. Methods

This section describes the algorithms, their implementations, the instance datasets and the computer setup used. The authors also present a rationale for not including some algorithms and datasets from the literature.

4.1. Algorithms and their implementations

The usual dynamic programming (DP) algorithms for the UKP have a worst-case time complexity of $O(nc)$ (pseudo-polynomial), and worst-case space complexity of $O(n + c)$. The branch-and-bound (B&B) algorithms have an exponential worst-case time complexity (all item combinations that fit the knapsack), but the worst-case space complexity of the B&B algorithms used in the experiments is $O(n)$.

MTU1 is a B&B algorithm for the UKP (Martello & Toth, 1977). In the enumeration tree of MTU1, each node at depth $0 \leq d < n$ has $(c'/w_{d+1}) + 1$ children (where c' is the remaining capacity at

[1] This test was not implemented in EDUK.

the current node), with each child representing a valid amount of copies of the item $d+1$ being packed in the knapsack (including zero copies). In practice, since it follows a depth-first search, MTU1 has no explicit tree, but instead, make changes directly over the current solution, which represents the current branch from the root to a leaf. Items are added to and removed from the solution to simulate the tree traversal. The exploration order favors the siblings representing higher amounts before lower amounts. As the items are sorted by efficiency, this means the first node after the root will represent the maximum amount of copies of the best item, and the path to the first visited leaf will be the solution given by the classic greedy heuristic which packs the most efficient item that yet fits the knapsack until there is no item that fits the knapsack. Consequently, the algorithm already begins with this lower bound. The upper bound computed by MTU1 in each node is the continuous relaxation of the problem with the already set variables fixed, which is solved by multiplying the efficiency of the most efficient item not yet fixed by the remaining capacity.

The MTU2 algorithm calls MTU1 over the $x\%$ most efficient items, and if this is not sufficient to obtain a solution with proved optimality, it repeats the process with $x\%$ increased (Martello & Toth, 1990). The Fortran implementation of MTU1 and MTU2 used in our experiments was the original implementation by Martello and Toth but with all 32 bits integers or float variables/parameters replaced by their 64 bits counterparts. This version is publicly available in the authors' code repository[2]. The C++ implementations of both algorithms were written by the authors. The C++ and Fortran implementations of the MTU1 algorithm have no significant differences.

The MTU2 implementations differed in the algorithm used to partially sort the items and the exact ordering. Martello and Toth (1990) does not specify the exact method for performing the partial sorting. The original Fortran implementation uses a complex algorithm developed by one of the authors of MTU2 in Fischetti and Martello (1988) to find the kth most efficient item in an unsorted array, and then select and sort only the items that have the same or a greater efficiency. The C++ implementation uses the `std::partial_sort` procedure of the standard C++ library `algorithm`. The Fortran implementation only sorts the items by nonincreasing efficiency; the C++ implementation breaks efficiency ties sorting by nondecreasing weight.

A description of the recursion, states, and stages in which the DPs for the UKP are based follow. Given $opt(y)$ denotes the optimal solution value for capacity y, the recursion for the UKP can be written as $opt(y) = max\{0, p_1 + opt(y - w_1), p_2 + opt(y - w_2), \ldots, p_n + opt(y - w_n)\}$ (where $\forall\ y < 0.\ opt(y) = -\infty$). The UKP has a single state, that is the remaining knapsack capacity y. Different from 0–1 KP and BKP, the UKP has no need to take into account which items were already used at each decision as there is an unlimited amount of copies available for each item. This difference allows UKP to need only $O(c)$ memory, instead of $O(nc)$ as in 0–1 KP and BKP. For each capacity $y = c - w_s$ (where s is a valid solution), there is a decision point; consequently, the number of stages is not exact but can go up to c (as in the case of $w_{min} = 1$).

The ordered step-off (OSO) is a DP algorithm proposed in (Gilmore & Gomory, 1966, p. 15). The authors already presented and discussed a revisited version of OSO in Section 2. The terminating step-off (TSO) is the same as OSO but it includes periodicity checking. Greenberg and Feldman (1980) proposes another variant of OSO, referred in this paper as GFDP (Greenberg and Feldman's Dynamic Programming). The GFDP does not use the best item b, but interrupts the DP at each w_b capacity positions to verify if the DP can stop, and the remaining capacity filled with copies of b. If two or more items share the greatest efficiency, GFDP verification does not work, and it is the same as OSO.

The original implementations of the step-offs and GFDP were not publicly available, so the authors wrote their own implementations in C++. The authors' implementations of TSO and GFDP include the same tiebreaking improvement added by the authors to OSO and described in Section 2. All C++ implementations written by the authors are available at the first author's code repository[3].

EDUK (Efficient Dynamic programming for the Unbounded Knapsack problem) was the first DP algorithm to explicitly check for threshold dominance (a concept proposed together with the algorithm) and collective dominance (that was independently discovered in Pisinger, 1994), it also features a sparse representation of the iteration domain (Andonov et al., 2000; Kellerer et al., 2004; Poirriez and Andonov, 1998, p. 223). EDUK seems to be based on ideas first discussed in Andonov and Rajopadhye (1994). Before EDUK2 was proposed, it was said that "[...] EDUK [...] seems to be the most efficient dynamic programming based method available at the moment" (Kellerer et al., 2004, p. 223).

EDUK2 is a hybrid DP/B&B algorithm which improves EDUK with a B&B preprocessing phase (Poirriez et al., 2009). If B&B can solve the instance using less than a parameterized number of nodes, then EDUK is not 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 implementation of EDUK and EDUK2 used in the experiments was PYAsUKP (PYAsUKP: Yet Another solver for the Unbounded Knapsack Problem), which was written in OCaml. Vincent Poirriez gave access to this code to the authors, by email, in January 11th, 2016[4].

Finally, the authors also implemented the UKP formulation (i.e., Eqs. (1)–(3)) using the C++ interface of both Gurobi 8.0.1 (Gurobi Optimization, 2018) and CPLEX 12.8 (IBM, 2018) to solve single UKP instances. To make comparison to other methods fair and the results reproducible, the solvers were configured to: execute in single thread and deterministically (i.e., random seed fixed to zero); finish before time limit only with a 0% gap between the upper and lower bounds (i.e., search for a proven optimal solution); have the smallest tolerance possible to variable values deviating from integrality (without this about 9% of the results were slightly below or above the optimal value)[5].

4.1.1. Algorithms deliberately ignored

The naïve DP algorithm for the UKP (Hu, 1969, p. 311), an improved version of it presented in (Garfinkel & Nemhauser, 1972, p. 221) and the OSO (Gilmore & Gomory, 1966, p. 15) are all $O(nc)$ DP algorithms similar to each other. However, OSO does not need to execute n operations for each distinct c value and, in practice, will iterate only a small fraction of n (or even an empty list) for most c values of most instances. The other two algorithms *always* execute nc operations regardless of any instance properties. Preliminary tests confirmed that OSO dominated the other two

[2] The MTU1 and MTU2 adapted Fortran code used in the experiments is available at https://github.com/henriquebecker91/masters/tree/136c1c1fbeb6ef7baa7ab6bcc8f48cb0bb68b697/codes.

[3] The C++ implementations of MTU1, MTU2, the revisited ordered/terminating step-offs, and R-GFDP are available at https://github.com/henriquebecker91/masters/tree/136c1c1fbeb6ef7baa7ab6bcc8f48cb0bb68b697/codes/cpp.

[4] The code is available at Henrique Becker master's thesis code repository (https://github.com/henriquebecker91/masters/blob/f5bbabf47d6852816615315c8839d3f74013af5f/codes/ocaml/pyasukp_mail.tgz).

[5] The exact parameters used for each solver were: GRB_IntParam_Threads (Gurobi) and Threads (CPLEX); GRB_IntParam_Seed (Gurobi) and RandomSeed (CPLEX); GRB_DoubleParam_MIPGap (Gurobi) and MIP::Tolerances::MIPGap (CPLEX); GRB_DoubleParam_IntFeasTol (Gurobi) and MIP::Tolerances::Integrality (CPLEX). All CPLEX parameters are prefixed by IloCplex::Param. The codes are available in: https://github.com/henriquebecker91/masters/tree/efea8a981a72237edd17fedb4742ac568ded831c/codes/cpp/lib (files *cplex_ukp_model.hpp* and *gurobi_ukp_model.hpp*).

algorithms and, consequently, both were not included in our experiments.

The UKP5 algorithm proposed in Becker and Buriol (2016) was found to be very similar to TSO[6] and therefore only TSO was included.

The authors' implementation of the first algorithm proposed in Greenberg (1986) exceeded the time limits we used in the experiments, while the second algorithm does not work for all UKP instances[7]. Both weren't included in the experiments. The Sage-3D algorithm from Landa (2004) cited in Hu, Landa, and Shing (2009) needs $O(nw_b p_b)$ memory and time, which is prohibitive for many instances considered and, therefore, was also not included. The algorithm holds a considerable theoretical value, and its complexity is justified by the fact Sage-3D does not solve the UKP for a specific knapsack capacity, but instead, builds a data structure which allows querying the solution for a specific capacity in $O(log(p_b))$.

In Martello and Toth (1977), the B&B algorithm proposed in Gilmore and Gomory (1963) is said to be two times slower than the algorithm proposed in Cabot (1970), which was found to be dominated by MTU1; also, the algorithm in Gilmore and Gomory (1963) seems to have been abandoned by its authors in favor of OSO. Thus, the B&B algorithms proposed in Gilmore and Gomory (1963) and Cabot (1970) were not included in the experiments.

In Greenberg and Feldman (1980) it is implied that GFDP is an improved version of the algorithm proposed in Shapiro and Wagner (1967), so only GFDP was included. The authors could not obtain the code of the algorithm proposed in Babayev, Glover, and Ryan (1997), and they chose to not reimplement it to not risk misrepresenting it, as it was not trivial to implement.

UKP-specific algorithms perform better than applying the BKP or the 0–1 KP algorithms over converted UKP instances (Martello and Toth, 1977), so BKP and 0–1 KP algorithms were not considered.

4.2. Instance datasets

The datasets used in the experiments include: artificial datasets from the literature that focus on being hard-to-solve (PYAsUKP and realistic random datasets); a dataset proposed by the authors in order to prove an hypothesis (BREQ dataset); a dataset based on solving of CSP/BPP instances with the column generation technique (CSP dataset). The reasoning for not including *uncorrelated* and *weakly correlated* instances is presented in the end of the section.

4.2.1. PYAsUKP dataset

The PYAsUKP dataset is described in Becker and Buriol (2016), and comprises 4540 instances from five smaller datasets. The PYAsUKP dataset was heavily based on the datasets presented in Poirriez et al. (2009), which were used to compare EDUK2 to other UKP solving algorithms. The instance generator used to generate the 4540 instances share the code with EDUK/EDUK2 implementations (PYAsUKP), which is the reason we call this dataset the *PYAsUKP dataset*. The PYAsUKP dataset comprises: 400 subset-sum instances ($10^3 \leq n \leq 10^4$); 240 strongly correlated instances ($5 \times 10^3 \leq n \leq n = 10^4$); 800 instances with postponed periodicity ($2 \times 10^4 \leq n \leq 5 \times 10^4$); 2000 instances without collective dominance ($5 \times 10^3 \leq n \leq 5 \times 10^4$); 1100 SAW instances ($10^4 \leq n \leq 10^5$). The PYAsUKP dataset has multiple-of-ten amounts of instances generated with different random seeds for each combination of the remaining generation parameters (n, w_{min},...). The authors selected the first one-tenth of the instances for each parameter combination[8] (in total 454) and will refer to it as the *reduced PYAsUKP dataset*.

4.2.2. Realistic random dataset

A dataset of *realistic random* instances was used in the experiments. The generation procedure, based on Andonov et al. (2000), is summarized as follows: generate two lists of n unique random integers uniformly distributed in $[min, max]$ and sort them by increasing value; combine both lists in an item list, by pairing up the ith of one list to the ith element of the other; randomly shuffle the item list; generate a random capacity $c \in [c_{min}, c_{max}]$ (uniform distribution). Simple dominance cannot occur in such instances; other dominances may be present. Our dataset comprises ten instances generated with distinct random seeds for each one of eight n values ($2^{n'}$, where $n' \in \{10, 11, \ldots, 17\}$), totalling 80 instances. The values of the remaining parameters come from n: $max = n \times 2^{10}$, $min = max/2^4$, $c_{min} = 2 \times max$, and $c_{max} = c_{min} + min$.

4.2.3. BREQ 128–16 Standard Benchmark

The Bottom Right Ellipse Quadrant (BREQ) is an items distribution proposed in Becker (2017). The items of an instance follow the BREQ distribution iff the profits and weights respect $p_i = p_{max} - \lfloor \sqrt{p_{max}^2 - w_i^2 \times (p_{max}/w_{max})^2} \rfloor$, where w_{max} (p_{max}) is an upper bound on the items weight (profit). The distribution name comes from the fact that the formula describes the bottom right quarter of an ellipse.

The purpose of this items distribution is to illustrate the authors' point that artificial distributions can be developed to favor one solving approach over another. In the case of the BREQ distribution, it favors B&B over DP. Distributions with the opposite property (favor DP over B&B) are common in the recent literature.

The optimal solution of BREQ instances is often in the first fraction of the search space examined by B&B algorithms. Moreover, the lower bounds from good solutions allow B&B methods to skip a large fraction of the search space and promptly prove optimality. In BREQ instances, the presence of simple, multiple and collective dominance is minimal[9], but threshold dominance is widespread: an optimal solution will never include the item i two or more times if there is an item j such as that $\sqrt{2} \times w_i \leq w_j \leq 2 \times w_i$. Such characteristics lead to optimal solutions comprised of the largest weight items, which do not reuse optimal solutions for lower capacities. This means that solving the UKP for lower capacities as DP algorithms do is mostly a wasted effort.

The authors named the BREQ dataset used in the experiments of *BREQ 128–16 Standard Benchmark*. This dataset comprises 100 instances generated from all combinations of ten random seeds and ten distinct n values defined as $n = 2^{n'}$, where $n' \in \{11, 12, \ldots, 20\}$. The values of the remaining parameters are: $p_{min} = w_{min} = 1$, $c = 128 \times n$, $w_{max} = c$ and $p_{max} = 16 \times w_{max}$. The items generation procedure follows: generate n unique random integer weights uniformly distributed in $[w_{min}, w_{max}]$; for each item weight, the corresponding profit is calculated by the formula presented in the first paragraph of this section.

[6] The authors of this article reinvented an algorithm from Gilmore and Gomory (1966) and published a paper calling it UKP5 while believing it was novel Becker and Buriol (2016). The authors would like to apologize to the scientific community for such disregard. The only improvement of UKP5 over TSO is the tiebreaker change described in Section 2, which the authors included in all algorithms it was applicable.

[7] The authors' implementations of both algorithms were made available at https://github.com/henriquebecker91/masters/blob/e2ff269998576cb69b8d6fb1de59fa5d3ce02852/codes/cpp/lib/greendp.hpp.

[8] The entire PYAsUKP dataset is available at https://drive.google.com/open?id=0B30vAxj_5eaFSUNHQk53NmFXbkE. The instances with the same parameter combination are numbered.

[9] If the BREQ formula did not include the rounding, the profit of the item would be a strictly monotonically increasing function of the items weight. Any item distribution with this property cannot present simple, multiple, or collective dominance.

4.2.4. CSP pricing subproblem dataset

An often mentioned UKP application is to solve the pricing subproblems generated by the linear programming relaxation of the Set Covering Formulation (SCF) for the Bin Packing Problem (BPP) and Cutting Stock Problem (CSP) using the column generation approach (Gilmore & Gomory, 1961; Kellerer et al., 2004, p. 455–459). To analyze the performance of the algorithms in the context of this application, the authors have written a C++ program that uses the CPLEX Solver to solve the SCF and feed the pricing problems generated to a custom UKP solving algorithm.

The experiments included eight datasets of BPP/CSP instances. The datasets are: Falkenauer (160 instances), Scholl (1210 instances), Wäscher (17 instances), Schwerin (200 instances), Hard28 (28 instances), Randomly Generated Instances (3840 instances), Augmented Non IRUP and Augmented IRUP Instances (ANI&AI, 500 instances), and Gschwind and Irnich instances (GI instances, 240). These datasets amount to 6195 instances, all made available in Delorme, Iori, and Martello (2018). The first seven datasets are described in Delorme et al. (2016), the last one (GI Instances) comes from Gschwind and Irnich (2016). The code used to solve the SCF relaxation can be found in the first author's repository[10].

4.2.5. Datasets deliberately ignored

The *uncorrelated* and *weakly correlated* item distributions were commonly used in the literature (Andonov et al., 2000; Babayev et al., 1997; Martello & Toth, 1977; 1990), but the authors decided to not include them in the experiments. The literature has already questioned the suitability of *uncorrelated* item distributions datasets for the analysis of the UKP (Poirriez & Andonov, 1998; Zhu & Broughan, 1997). Uncorrelated instances often exhibit a vast amount of simple and multiple dominated items, and polynomial algorithms can reduce the number of items in such instances by orders of magnitude.

The *weakly correlated* item distribution can be seen as a *strongly correlated* item distribution with more dominated items. The authors found redundant to present weakly correlated datasets in addition to the strongly correlated datasets, as preliminary results suggested that the time spent solving weakly correlated datasets was similar to the time spent solving strongly correlated datasets of smaller size.

4.3. Computer setup

All experiments were run using a computer with the following characteristics: the CPU was an Intel® Core™ i5-4690 CPU @ 3.50 gigahertz; there were 8 gibibytes RAM available (DIMM DDR3 Synchronous 1600 megahertz) and three levels of cache (256 kibibytes, 1 mebibyte, and 6 mebibytes, with the cores sharing only the last one). The operating system used was GNU/Linux 4.7.0-1-ARCH × 86_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, as the authors found that parallel executions effected the run times, even if each isolated core only hosted one run at each time (Becker, 2017, p. 87).

The OCaml code (PYAsUKP/EDUK/EDUK2) was compiled with ocamlopt and the flags suggested by the authors of the code for maximum performance (*-unsafe -inline 2048*). The Fortran code (original MTU1/MTU2) was compiled with gcc-fortran and *-O3 -std=f2008* flags enabled. The C++ code (all remaining implementations) was compiled with gcc and the *-O3 -std=c++11* flags enabled.

5. Results and analyses

The experiments are split into five subsections. Each section addresses one dataset and the results of running some selected algorithms over them. To keep the results close to its discussion, each experiment section brings both the results and their immediate analysis. Only implementations including the tiebreaker improvement were used in the experiments, thus, for simplicity, the Revisited Ordered Step-Off, Revisited Terminating Step-Off and Revisited GFDP will be referred as OSO, TSO and GFDP in this section.

5.1. Results on the PYAsUKP dataset

The experiment presented in this section is an updated version of the experiment first presented in Becker and Buriol (2016). In this version GFDP is considered, UKP5 is replaced by TSO, and all runs were executed serially. The same 4540 instance files were used.

In Fig. 3, the instances (x-axis) are sorted by the time EDUK2 spent to solve them. EDUK2 B&B phase allows EDUK2 to solve some instances (distributed between all instance sizes) faster than TSO/GFDP. This behavior shows that EDUK2 times for a specific instance cannot be predicted based on the number of items and distribution of the instance. Nonetheless, when the B&B phase has little effect, EDUK2 DP phase (basically EDUK) spend considerably more time than TSO/GFDP to solve the instance.

TSO and GFDP run times form plateaus in the figure. For each class of instances, the plateaus aggregate runs over instances with a number of items of similar magnitude. This behavior shows that the specific items that constitute an instance affect TSO and GFDP less than the number of items and distribution, both which are good predictors of TSO and GFDP run times.

The run times of TSO and GFDP are similar, except for the SAW instances, in which GFDP performed considerably worse than TSO. GFDP DP phase does not generate solutions including the best item. The use of the best item allows the generation of more efficient solutions, which dominate less efficient solutions, reducing the total number of solutions generated and, consequently, the computational effort spent. The authors believe that the exclusion of the best item from the DP phase is the reason that GFDP presented run times higher than TSO over SAW instances.

When EDUK2 solves an instance in less time than TSO, the difference is often less than a second, and up to 10 seconds. When EDUK2 solves an instance in more time than TSO, the difference is often more than 5 seconds and up to 6 minutes. Such behavior harms EDUK2 mean run time in comparison to simpler DP methods.

5.1.1. MTU1 and MTU2 (C++ and Fortran)

This section compares the performance of the C++ and Fortran implementations of MTU1 and MTU2 algorithms. These four implementations were executed over the reduced PYAsUKP benchmark.

In Fig. 4, both MTU1 implementations show run times in the same order of magnitude. However, considering only the instances that both MTU1 implementations solved before the timeout, the mean run time of Fortran MTU1 was 59 seconds and the mean

[10] The C++/CPLEX code used for solving SCF relaxation is available at https://github.com/henriquebecker91/masters/tree/8367836344a2f615640757ffa49254758e99fe0a/codes/cpp. The code can be compiled by executing *make bin/cutstock* in the folder. 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).

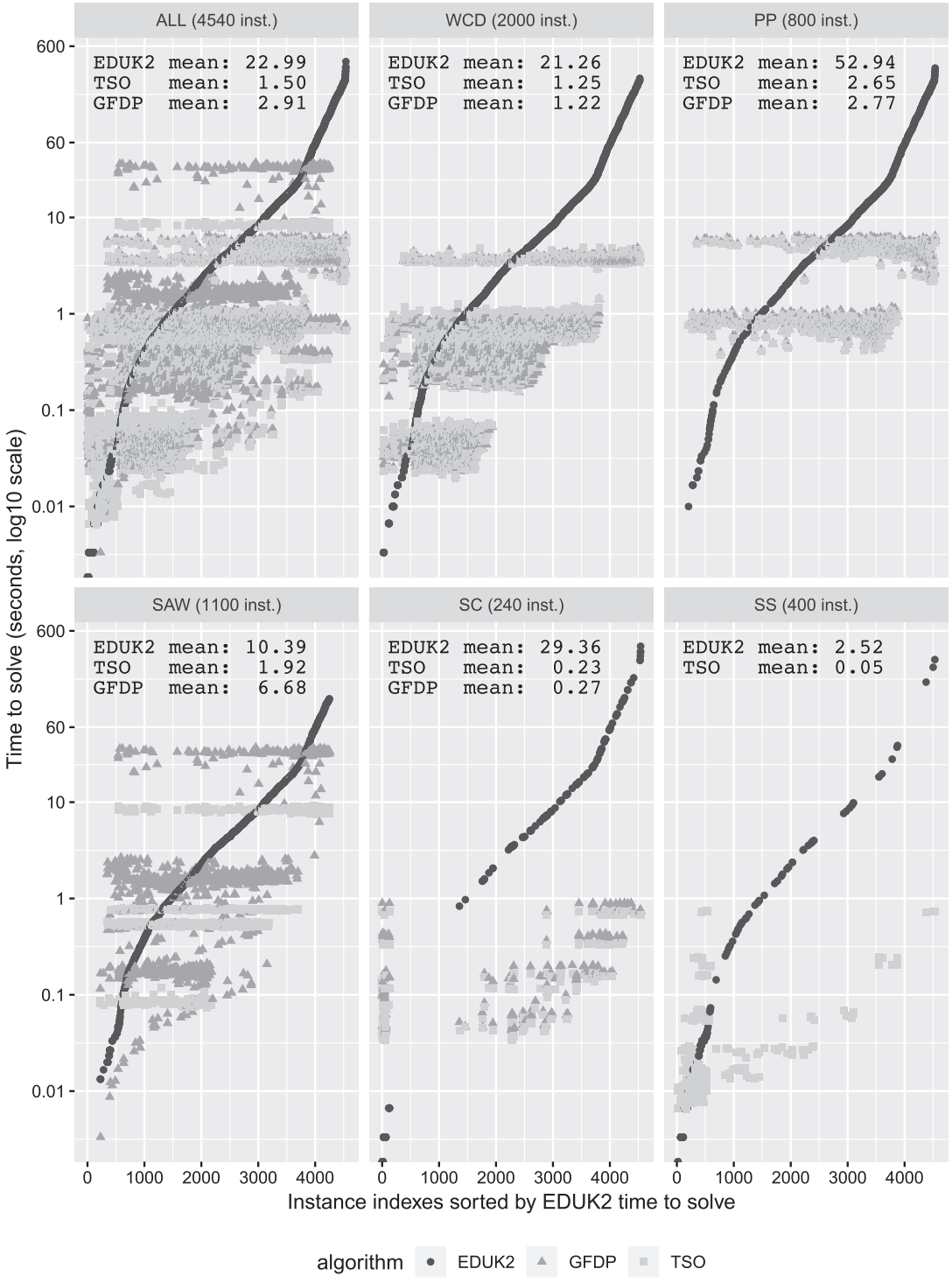

Fig. 3. Run times of TSO, GFDP and EDUK2 algorithms over the 4540 instances of the PYAsUKP dataset. There was no time limit. The instance classes acronyms stand for Postponed Periodicity (PP); Strongly Correlated (SC); Subset-Sum (SS); and Without Collective Dominance (WCD). EDUK was not included because EDUK2 supersedes EDUK. The ordered step-off run times were omitted because they were too similar to the terminating step-off run times. The GFDP run times over subset-sum instances are not displayed because in this case the two most efficient items have the same efficiency and, therefore, GFDP behaves as OSO. The mean labels inform the mean run times of each algorithm for the corresponding dataset, in seconds.

run time of C++ MTU1 was 30 seconds. The analysis of the individual run times shows that for many instances Fortran MTU1 spent about the double of the time spent by C++ MTU1 to solve the same instance.

For many instances, the run times of the MTU2 implementations differed in more than one order of magnitude. The authors believe that this divergence was caused by the difference of sorting algorithms and items ordering (C++ MTU2 order items by nondecreasing weight if they share the same efficiency). The subset-sum instances were the ones that exhibited the largest difference. The range [190, 221] of the x-axis of Fig. 4 is composed of subset-sum instances. C++ MTU2 solved all 40 subset-sum

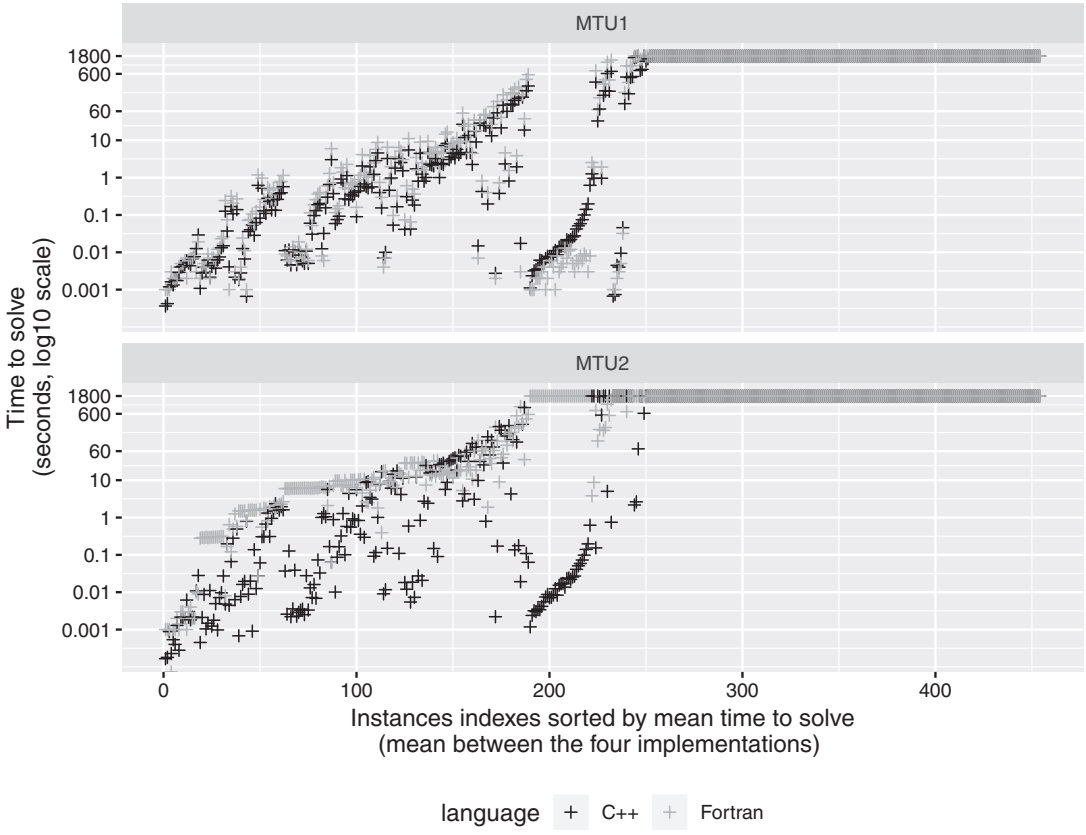

Fig. 4. Run times of MTU1 and MTU2 (C++ and Fortran) over the 454 instances of the reduced PYAsUKP dataset. The time limit was set to 30 minutes. Runs terminated by timeout are displayed as taking exactly the time limit.

instances with a mean time of 0.04 seconds while Fortran MTU2 solved only eight instances with a mean time of 155 seconds. Disregarding the subset-sum instances, the mean run time of Fortran MTU2 was 56 seconds and the mean run time of C++ MTU2 was 40.5 seconds. Only the C++ implementations are used in the rest of the experiments.

5.1.2. CPLEX and Gurobi

As in last section, this section uses the reduced PYAsUKP benchmark to gauge the relative performance of two alternatives, in this case: our Gurobi and CPLEX models.

Both Gurobi and CPLEX have an emphasis/focus setting. The authors selected the emphases they deemed more adequate, and performed this experiment with all of them. For each solver, only the results for the emphasis with the best performance are presented[11].

Gurobi returned slightly wrong results for three instances (in which it warned about integrality tolerance violation) and exhausted the available memory (more than 7 gibibytes) in another fifteen instances. CPLEX did not present the same issues. Performance-wise, Fig. 5 shows a clear advantage of CPLEX over Gurobi, not only with slightly shorter times for most instances, but there were also many cases in which Gurobi ended in timeout (or near it) and CPLEX finished in orders of magnitude less time. Even only considering the finished and correct runs of both solvers, CPLEX mean time was 57 seconds while Gurobi mean time was 94 seconds. None of the solvers can compete with TSO/GFDP/EDUK2 (as seen in Section 5.1), however their performance is superior to MTU1 and MTU2 (as seen in Section 5.1.1). Taking in account Gurobi performance and its memory and integrality issues, the authors chose to restrict the solvers presented to only CPLEX for the remaining experiments (the same emphasis was kept, unless said otherwise).

5.2. Results on the realistic random dataset

The results of the experiments over the realistic random dataset are summarized in Fig. 6.

The run times of both step-off algorithms and GFDP become almost identical as the size of the input grows, so the authors chose to present only the results of GFDP. As the instance size grow, EDUK run times get a bit worse than GFDP. EDUK2 has many run times similar to EDUK but, for some instances in each instance size, its B&B phase solves the instance or helps to reduce the instance size considerably. The B&B methods (CPLEX, MTU1 and MTU) had similar a similar spreading behaviour, and the authors chose CPLEX as their best representative. MTU1 and MTU2 had the best run times for smaller instances but, as the instance size and w_{min} grew, they had more timeouts than finished runs by $n = 32,768$ and only one finished run for the largest instance size. Despite EDUK2 solving some instances orders of magnitude faster than the other algorithms (especially in the larger instance sizes), the mean run time of EDUK2 (8.51 seconds) is higher than TSO mean run time (5.36 seconds). As already observed in Section 5.1, EDUK2 often presents a few run times that are one order of mag-

[11] The focus/emphasis parameters are GRB_IntParam_MIPFocus (Gurobi) and IloCplex::Param::Emphasis::MIP (CPLEX). The default of both solvers is balance between feasibility and optimality (code 0). Besides the default value (Gurobi best performance), the focus on optimality (code 2 on both solvers, CPLEX best performance) and the aggressive focus on optimality (code 3 on both solvers) were also tested. Other foci, as the emphasis in feasibility, were ignored as they did not seem relevant for an UKP model.

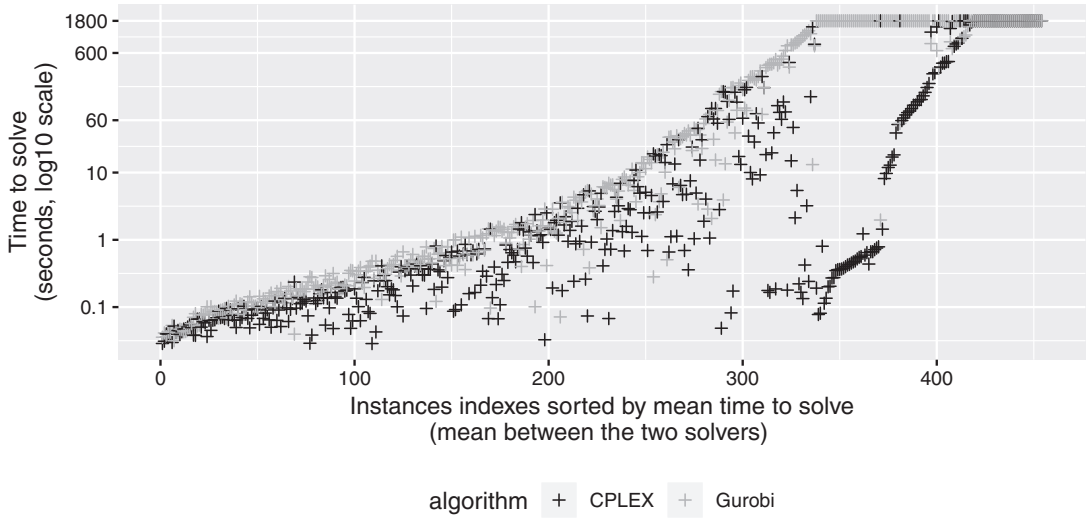

Fig. 5. Run times of CPLEX and Gurobi over the 454 instances of the reduced PYAsUKP dataset. The time limit was set to 30 minutes. Runs terminated by timeout, memory exhaustion, or with wrong results are displayed as taking exactly the time limit.

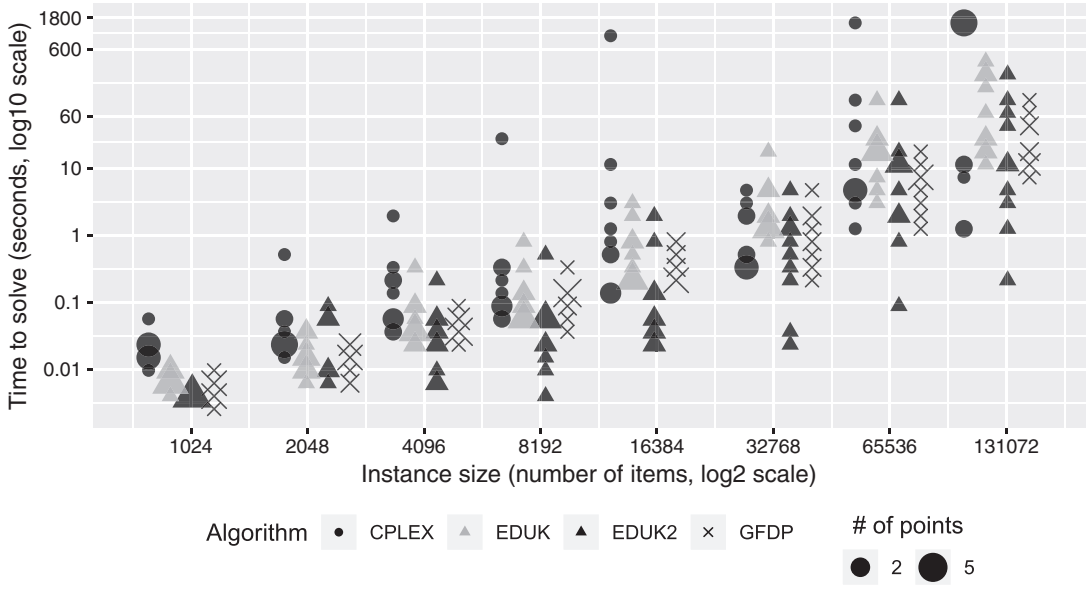

Fig. 6. Run times of the algorithms over the 80 instances of the realistic random dataset. The time limit was set to 30 minutes. Runs terminated by timeout are displayed as taking exactly the time limit. Shape and color are used to distinguish between algorithms. The shape size is used to indicate the amount of overlapping points. The horizontal position of the points was adjusted for better visualization.

nitude higher than the highest run time of TSO, what considerably increases its mean time.

5.3. Results on the BREQ 128–16 Standard Benchmark

The results for the BREQ 128–16 Standard Benchmark are summarized in Fig. 7. The run times outline two groups with distinct time growth: a steep-growth group (which hit the time limit) and a gradual-growth group (in which no run ends in timeout). The steep-growth group is mainly composed of the DP algorithms: TSO, OSO, and EDUK. The gradual-growth group is mainly composed of the B&B and hybrid algorithms: CPLEX, MTU1, MTU2, and EDUK2. MTU1 is not shown because its results are similar but dominated by MTU2, the analogue applies to TSO and OSO. CPLEX is classified as gradual-growth, it is orders of magnitude worse than its fellow group members, but yet orders of magnitude better than the steady-growth group members, and has no run ending in timeout. GFDP is the only algorithm with run times in both groups.

As EDUK2 is basically EDUK plus a B&B phase and bounds checking, seems clear that it is the B&B hybridization that allows for EDUK2 to be in the gradual-growth group (while EDUK that is pure DP is in the steep-growth group). As the difference between MTU1 and MTU2 is the core strategy, which allows MTU2 to avoid sorting the entire item list, the time spent by these algorithm to sort the items is significant when executed over BREQ instances. GFDP is an OSO variant which periodically computes bounds (similar to the ones used by the B&B approach) to verify if the DP can be stopped and any remaining capacity be filled with copies of the best item. If the bounds stop the DP early then GFDP run time falls in the gradual-growth group. Otherwise, the run time is similar to OSO run time for the same instance. CPLEX run times are explained by the fact that they are mostly spent solving the relaxation of the root node (i.e., struggling with the large number

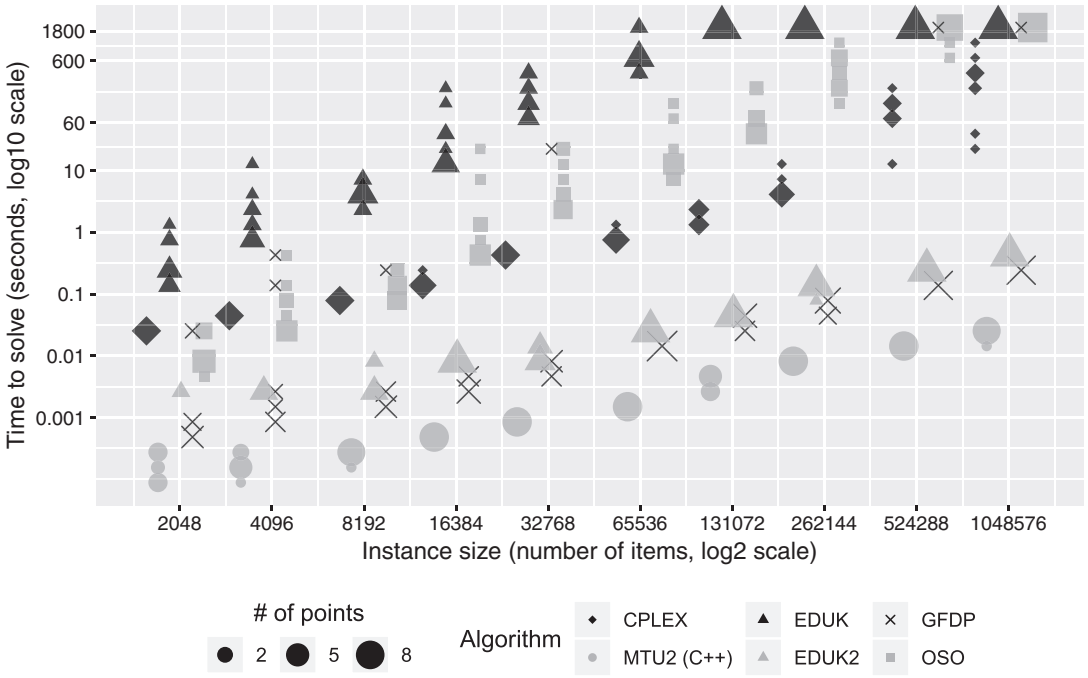

Fig. 7. Run times of the algorithms over the 100 instances of the BREQ 128–16 Standard Benchmark. The time limit was set to 30 minutes. Runs terminated by timeout are displayed as taking exactly the time limit. Shape and color are used to distinguish between algorithms. The shape size is used to indicate the amount of overlapping points. The horizontal position of the points was adjusted for better visualization.

of variables), virtually no time is spent in the B&B phase (often the B&B tree has a single node).

The results confirm the hypothesis that instances generated with this distribution would be hard to be solved by DP algorithms and easy to be solved by B&B algorithms.

5.4. Results on the pricing subproblems from BPP/CSP

The previous experiments included datasets of UKP instances and binaries that read and solved a single instance and then returned the solving time. In this experiment, the instances are BPP/CSP instances. Also, the run time for each BPP/CSP instance presented is the sum of all time spent solving the multiple pricing subproblems generated by the column generation approach applied over the linear programming relaxation of the set covering formulation. In this experiment, the CPLEX version used was 12.5, and the emphasis is the default.

Only CPLEX, OSO and MTU1 (C++) were used as pricing problem solvers in this experiment. The values of n and c in the pricing problems are sufficiently small to exist little difference between GFDP/TSO compared to OSO, or MTU2 compared to MTU1, and therefore GFDP, TSO and MTU2 were removed from the comparison. CPLEX was orders of magnitude slower than the other two and its results are presented only in the supplementary material to simplify the analysis. The authors attribute the CPLEX poor performance to the same difficulties than MTU1 (arisal of 'hard' problems) but also to a costly initialization of the solving process (each 'easy' problem costed much more to CPLEX than to MTU1). The authors did not succeed in integrating the PYAsUKP code (written in OCaml) with the C++/CPLEX code needed by this experiment.

In a pricing subproblem, the profit of the items is a real number. Adapting MTU1 for using floating point profit values is not trivial. 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} (approx. 10^{12}). In a pricing subproblem, the profit of the items can also be non-positive, which breaks the assumptions of some algorithms. Items with non-positive profits are removed from the item list before passing it to the UKP solving algorithm.

In Fig. 8, it can be seen that, except by two recent datasets (ANI&AI and GI), the mean time spent solving pricing problems in an instance is below 1 second. Such times are explained by two main factors: (1) for completeness, the authors included many classic but old datasets which nowadays are easy to solve; (2) these BPP/CSP datasets were meant to be hard to solve exactly, and solving the linear programming relaxation of the problem takes only a fraction of that time. In the majority of these easy datasets, the highest times presented are from MTU1, while OSO presented the lowest mean time.

Many instances of the ANI&AI dataset exceeded the timeout when MTU1 was used to solve the pricing subproblems. To study this behavior, next the times of individual pricing problems of the instance 1002_80000_DI_12 (ANI&AI dataset) are analyzed. The pricing problems generated by the same BPP/CSP instance always share the same n, c and items weights. The only difference between the pricing problems is the item profit values[12]. The pricing problems generated when solving this specific instance have $n = 911$ and $c = 66432$.

In Fig. 9, the times of pricing problems generated by this instance are presented. It can be seen that the MTU1 run exceed the time limit because the time spent solving the last pricing problems increases exponentially. The time taken by OSO to solve pricing problems also increases, but one or two orders of magnitude (not five or six).

In Fig. 10, it can be seen that as the profit values of the pricing problems change, their item distribution also change. The initial distribution is consequence of the set of patterns used to initialize the column generation. Given the bin size c and the n item sizes w_i $(i = 1, \ldots, n)$, the initial set of patterns consists in one pattern for each item i, with $\lfloor c/w_i \rfloor$ copies of that item.

[12] The specific code used removes items with non positive profit values before the beginning of the algorithms. Consequently, n do vary, but the results are the same.

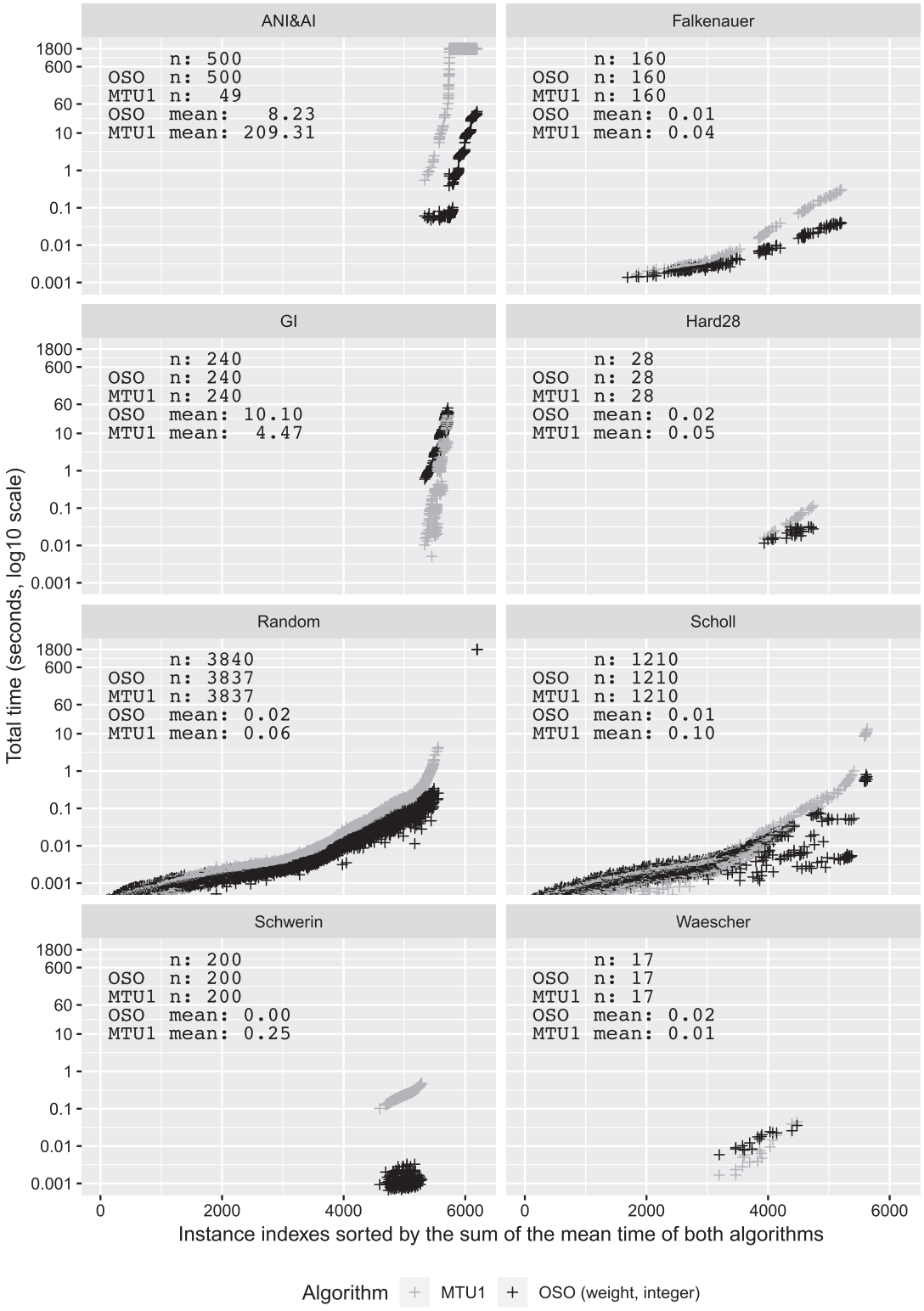

Fig. 8. Time spent solving the pricing subproblems from the 6195 CSP instances, with MTU1 and OSO (weight, integer). The time limit was set to 30 minutes (the time limit considered the total run time, not only the time spent solving pricing subproblems). If a run is terminated by timeout, the time spent solving pricing subproblems is displayed as exactly the time limit. The labels mean: n – number of instances in the dataset; OSO/MTU1 n – number of instances solved before timeout by the algorithm; OSO/MTU1 mean – mean of the algorithm run times that did not end in timeout.

Consequently, each item i with the same $q = \lfloor c/w_i \rfloor$ value has the same $1/q$ profit value in the first iteration. As patterns including items of different sizes are added, the profit of the items seems to approximate c/w_i. Instances with such distribution can be harder to solve by B&B algorithms, as the similar efficiency weaken the capability of the bounds of reducing the solution space. The authors also do not discard the possibility that using the multiplicative factor has reduced the quality of the MTU1 bounds (and its capability of finishing early) by cutting off some precision of the profit value.

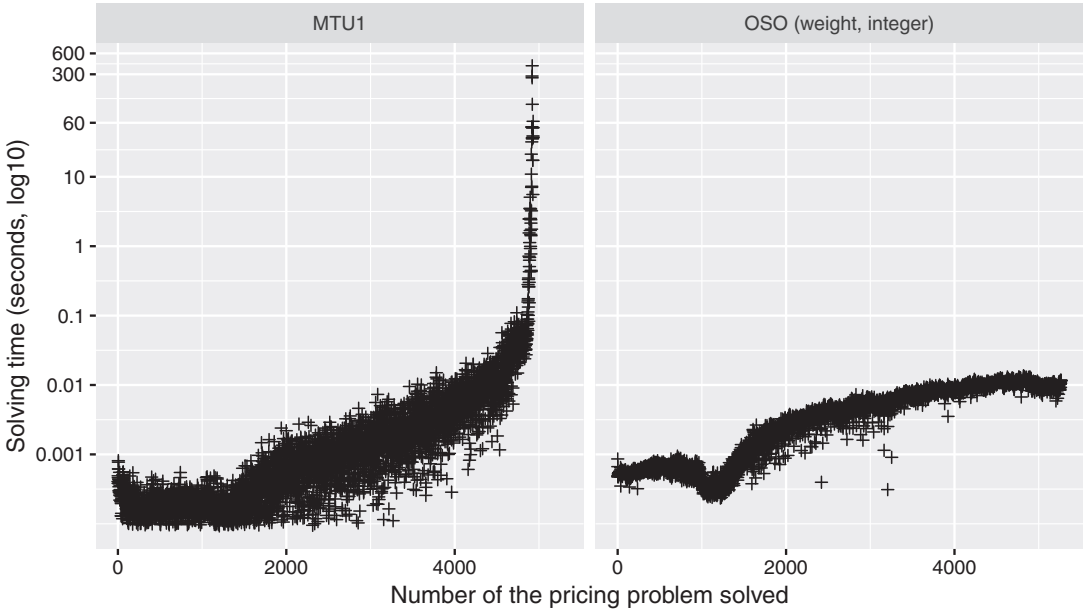

Fig. 9. Time spent solving each individual pricing problem generated by instance 1002_80000_DI_12 (ANI&AI dataset) with MTU1 and TSO. The MTU1 run was killed by timeout (30 minutes), TSO run finished normally.

To measure the impacts of the multiplicative factor, OSO variants using floating point profit values and using integer profit values were used in the experiments. The weight of an item does not change from a pricing problem to the next, consequently, the items can be kept ordered by increasing weight with no effort. The OSO algorithm sorts the items by nonincreasing efficiency but does not depend on this ordering to work. To verify if the sorting cost would pay off, variants with sorting enabled and disabled were used. Consequently, four versions of OSO were used in this experiment, for all combinations of profit type (the original floating point, or the converted integer), and sorting (sorting each new pricing problem by nonincreasing efficiency, or sorting the items by increasing weight a single time before the first pricing problem).

In the experiments with UKP instance datasets, all algorithms agreed on the value of the optimal solution, while sometimes returned different optimal solutions. In this experiment, the difference in exactly which optimal solution is returned for a pricing subproblem (because of different tiebreaking or because of floating point inaccuracy) changes the next pricing problem and, consequently, the next optimal solution, in a feedback loop. However, between MTU1 and the four OSO variants, for the same BPP/CSP instance, no solution of the linear programming relaxation (number of rolls needed) differed more than a 2^{-18} fraction of a roll. Solving pricing problems using the multiplicative factor to work over integer profits seems to be a viable approach.

The four OSO variants spent time in the same order of magnitude to solve all pricing problems of the same BPP/CSP instances. The mean time of the four variants differed significantly: 1.56 seconds (efficiency, floating point), 1.21 seconds (weight, floating point), 1.46 seconds (efficiency, integer), 1.06 seconds (weight, integer). Using integer profits show a small but consistent improvement in the times (the time used to convert the value is included), keeping the items in the natural increasing weight order shows a greater and also consistent improvement. The mean time reduction observed in 'weight' variants did not come from cutting the time spent sorting the items (less than 1% of the total pricing time), but from how the DP phase of OSO behaved with a differently sorted list. The times of OSO in all figures of this section are from the weight/integer variant.

6. General discussion and conclusions

Except for the BREQ dataset, a revisited version of a DP algorithm from 1966 (Revisited Terminating Step-Off, R-TSO) had lower mean times than the only known implementation of the current state-of-the-art algorithm (EDUK2 /PYAsUKP). Such results bring up two central questions the authors address: Why TSO was not considered in recent comparisons? How R-TSO outperformed the current state-of-the-art after five decades of study of the UKP?

The authors believe TSO was not considered in recent comparisons because: the algorithm was very similar to the naïve DP for the UKP but the authors did not emphasize it was orders of magnitude faster; B&B algorithms performed better than TSO in uncorrelated and weakly correlated instances with one hundred items. As far as the authors know, the proposal of MTU1 was the last time TSO was included in a comparison (Martello & Toth, 1977). In the experiments presented in Martello and Toth (1977), TSO 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 $n = 100$. Such instances are now too small to consider, and the relative difference between TSO and MTU1 mean times was less than one order of magnitude apart and diminishing. This trend hinted the possibility of the times taken by OSO/TSO and MTU1 converging (or even OSO/TSO taking less time than MTU1) for larger instances (e.g., OSO/TSO algorithm could have a costly initialization process but a better average-case complexity).

6.1. An algorithm is dominated by other in the context of a dataset

The comparisons often found in the literature: (1) only compared a newly proposed algorithm to the algorithm that 'won' the last comparison; (2) proposed new artificial datasets based on some definition of being 'harder to solve'. These two characteristics led to the following scenario: algorithm B dominates algorithm A in the context of dataset D1; algorithm C dominates algorithm B in the context of dataset D2; nothing guarantees that algorithm A does not dominate algorithm C in the context of dataset D2, as algorithm A was not included in the last comparison.

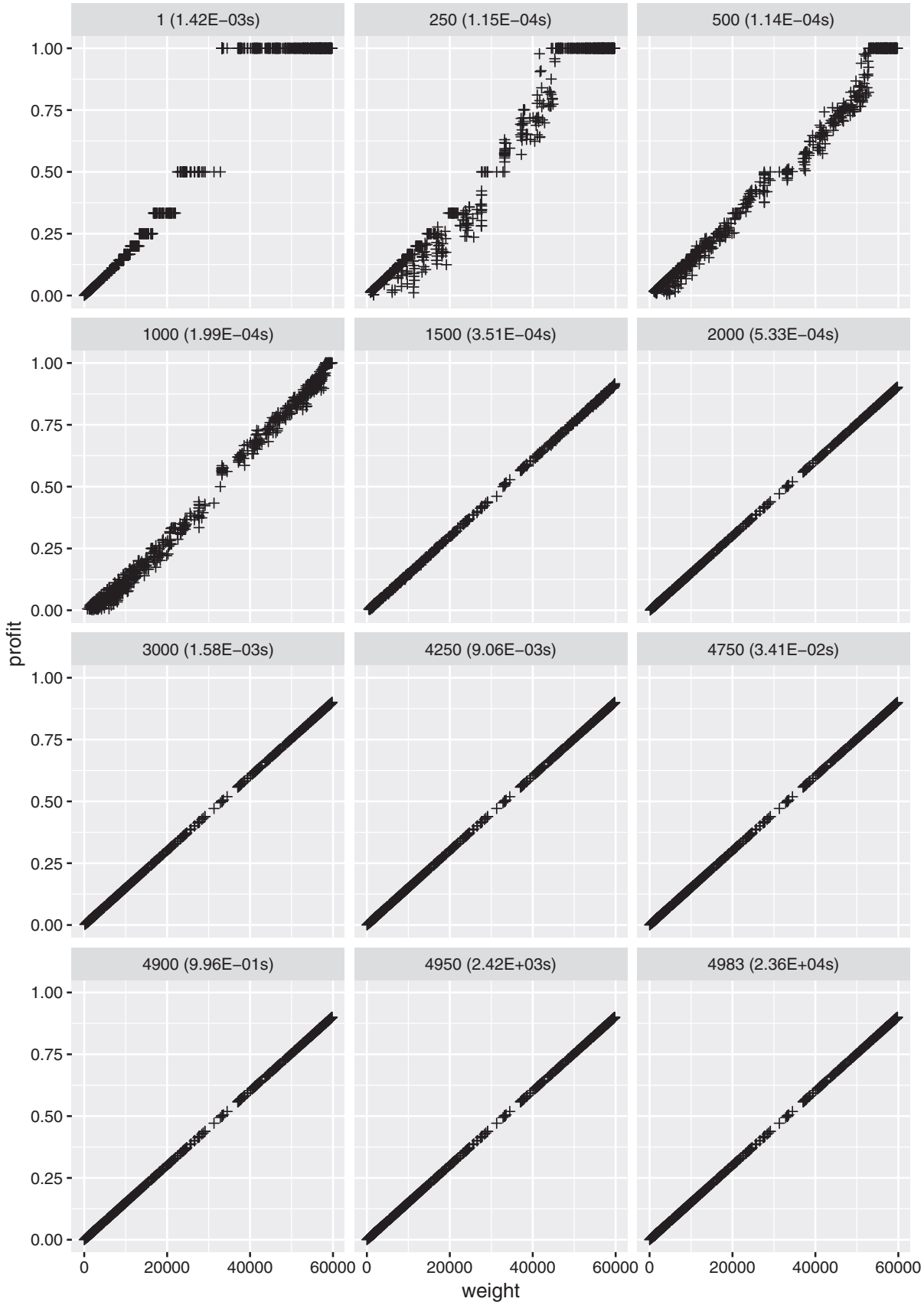

Fig. 10. Selected pricing problems generated while solving instance 1002_80000_DI_12 (ANI&AI dataset) linear programming relaxation with MTU1 as pricing problem solver. The titles of the facets follow the format *number of the pricing problem (how many seconds MTU1 spent to solve it)*. For this figure the time limit set was 90 hours.

Table 1
The number of finished runs of a method (row) over a dataset (column), and the average time (in seconds) of the finished runs (dashes mean the respective method was not used to solve the respective dataset). [a] The MTU1, MTU2, CPLEX, and Gurobi were executed only over the 454 instances of the reduced PYAsUKP dataset (not all 4540 instances as the remaining methods). [b] The times reported are for the following variants: R-TSO – integer profit, sort by efficiency; R-OSO – integer profit, sort by weight; CPLEX – custom code reusing the model and only changing the profits (coefficients of the objective function) between the iterations.

	PYAsUKP[a]		RR		BREQ		CSP[b]	
Method	fin	avg	fin	avg	fin	avg	fin	avg
R-TSO	4540	1.50	80	5.36	83	111.64	6192	1.45
R-OSO	4540	1.52	80	5.42	83	111.57	6192	1.07
R-GFDP	4140	2.91	80	5.32	98	0.33	–	–
EDUK2	4540	22.99	80	8.51	100	0.09	–	–
EDUK	–	–	80	18.97	59	164.74	–	–
MTU2	231	38.11	55	11.61	100	0.01	–	–
MTU1	249	54.11	58	20.82	100	0.02	5741	2.04
CPLEX	411	56.70	75	34.03	100	51.64	5606	33.38
Gurobi	349	94.44	–	–	–	–	–	–

A concrete example of this behavior in the UKP literature follows: MTU2 was compared only to MTU1, and in the context of a new dataset of large n and rich in simple and multiple dominated items (Martello & Toth, 1990); EDUK and EDUK2 were compared only to MTU2, and in the context of new datasets with a smaller n but with less dominated items and higher w_{min} (Andonov et al., 2000; Poirriez et al., 2009). To be fair, the objective of these papers was to show how the newly proposed algorithm was not negatively impacted by some instance characteristics as the older algorithm was. However, in such experiments, no old competitive algorithm of the same solving approach as the newly proposed algorithm (DP or B&B) was included.

The purpose of the BREQ item distribution is to further illustrate how artificial datasets that favor one approach over another can be created. The BREQ instances are hard to solve by DP algorithms and easy to solve by B&B algorithms. If they were the only instances considered, then MTU2 would be considered the best algorithm (as it can be seen in Table 1). However, considering all remaining datasets, MTU2 often presents the worst time performance. The BREQ instances do not suffer from the same richness of simple, multiple and collective dominated items that led uncorrelated instances to be criticized and abandoned. However, threshold dominance is widespread in BREQ instances and, as far the authors know, no real-world instances follow the BREQ distribution.

The effects of artificial instances in shaping what is considered the best algorithms for UKP is not limited to the instances proposed for the UKP. Gschwind and Irnich (2016) created the GI instances to be harder to solve by their column generation implementation: "[...] generated new and harder CS instances. These are characterized by huge values for the capacity (to complicate the subproblems) and larger numbers of items with distinct lengths". Their implementation used a DP algorithm to solve pricing problems. The characteristics of these newly proposed BPP/CSP instances made the pricing problems harder to solve by a DP algorithm, but not necessarily by a B&B algorithm, which is less affected by parameters like the knapsack capacity. To evaluate which is the best UKP algorithm to solve pricing subproblems, the BPP/CSP instances had also to be representative of real-world instances; otherwise, another layer of bias is laid. The ANI&AI instances, which MTU1 had difficulties to solve the pricing problems, are also instances created with the purpose to be hard to solve (Delorme et al., 2016). The definition of hard to solve is different between the GI and ANI&AI instances, as in ANI&AI instances the objective is to make B&B algorithms for the BPP/CSP struggle to prove the optimality of a solution for a BPP/CSP instance. The way such characteristic makes the pricing problems from ANI&AI instances harder to solve by MTU1 is not so clear as in the case of GI instances and DP algorithms.

6.2. Comments on R-TSO and EDUK2 times gap

The authors believe many factors allowed R-TSO to outperform EDUK2 PYAsUKP implementation. Some of them are: the weak solution dominance implicitly applied by (R-)TSO seems to be as effective as the explicit simple, multiple, collective and threshold dominance applied by EDUK2; R-TSO has better space locality; R-TSO solution backtrack trades memory for time (while EDUK does the opposite).

EDUK2 PYAsUKP implementation uses lazy lists to store solutions, while R-TSO uses an array. A strongly correlated instance ($\alpha = -5,\ n = 10,000,\ w_{min} = 110,000,\ c = 9,008,057$) of the PYAsUKP dataset was the one EDUK2 spent more time to solve (416 seconds, R-TSO spent about 1 second to solve the same instance). By the use of the *perf* profiler, it was possible to verify that EDUK2 PYAsUKP implementation executed about 1122 instructions per cache miss, while R-TSO executed about 288, 653 instructions per cache miss. The performance gain for using an array-based implementation was also observed in (Gschwind & Irnich, 2016, p. 19), which tried to follow the approach suggested by EDUK in their pricing problem solver: "In contrast, for UKP we found that a straightforward array-based implementation of the DP approach is faster than the list-based approach. We suspect that on a modern CPU, the smaller state graph of UKP can be accessed much faster (due to caching techniques) so that the solution of the UKP subproblems as they occur in the BP benchmark instances is possible in almost no (measurable) time".

6.3. Conclusions

In conclusion, (1) the choice of artificial instance datasets had an important role defining which algorithms were considered the best by the literature; (2) the simple, multiple, collective, and threshold dominance relations can be generalized to solution dominance, and the application of a weak version of it shows similar efficiency; (3) there is evidence that the items distribution of pricing problems can converge to a strongly correlated distribution (which can take exponential time to solve by B&B); (4) there is evidence that converting the profit of pricing problems to large integers do not cause significant loss to the master problem objetive value; (5) the development of new and tighter periodicity bounds is of little use to the improvement of the state-of-the-art algorithms for UKP; (6) CPLEX has better performance than B&B algorithms for UKP in instances with few items but 'hard' to solve by B&B approach, but it is worse when the instances are large but 'easy' (many variables) or too many small instances (costly initialization).

6.4. Future work

Many questions raised during the development of this paper ended up unanswered:

- 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 favor some approaches over others?
- If a B&B phase was added to the terminating step-off (as in EDUK2), and a C++ and array-based implementation of EDUK2 was written, would they have a similar performance?
- The difference in performance of MTU1/MTU2 and CPLEX/Gurobi over the PYAsUKP dataset comes from the fact MTU1/MTU2 are depth-first? What would be the performance of a best-bound B&B made for the UKP?

Acknowledgments

The authors are thankful to the CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for the PROEX 0487 Bolsa País 1653154 and FAPERGS (Fundação de Amparo á pesquisa do Estado do Rio Grande do Sul) for the Edital 02/2017 - PqG 27720.414.17857.23062017. The authors are also thankful to Vincent Poirriez for his help with PYAsUKP.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2019.02.011.

References

Andonov, R., Poirriez, V., & Rajopadhye, S. (2000). Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research, 123*(2), 394–407. doi:10.1016/S0377-2217(99)00265-9.

Andonov, R., & Rajopadhye, S. (1994). A sparse knapsack algo-tech-cuit and its synthesis. In *Proceedings of the international conference on application specific array processors, 1994.* (pp. 302–313). IEEE. doi:10.1109/ASAP.1994.331794.

Babayev, D. A., Glover, F., & Ryan, J. (1997). A new knapsack solution approach by integer equivalent aggregation and consistency determination. *INFORMS Journal on Computing, 9*(1), 43–50. doi:10.1287/ijoc.9.1.43.

Becker, H. (2017). *The unbounded knapsack problem: A critical review*. Federal University of Rio Grande do Sul (Master's thesis).. URL: http://www.hdl.handle.net/10183/163413.

Becker, H., & Buriol, L. S. (2016). UKP5: A new algorithm for the unbounded knapsack problem. In *Proceedings of the international symposium on experimental algorithms* (pp. 50–62). Springer. doi:10.1007/978-3-319-38851-9_4.

Cabot, A. V. (1970). An enumeration algorithm for knapsack problems. *Operations Research, 18*(2), 306–311. doi:10.1287/opre.18.2.306.

Delorme, M., & Iori, M. (2017). Enhanced Pseudo-Polynomial Formulations for Bin Packing and Cutting Stock Problems. *Technical Report*. Optimization Online. URL: http://www.optimization-online.org/DB_HTML/2017/10/6270.html.

Delorme, M., Iori, M., & Martello, S. (2016). Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research, 255*(1), 1–20. doi:10.1016/j.ejor.2016.04.030.

Delorme, M., Iori, M., & Martello, S. (2018). BPPLIB: A library for bin packing and cutting stock problems. *Optimization Letters, 12*(2), 235–250. doi:10.1007/s11590-017-1192-z.

Fischetti, M., & Martello, S. (1988). A hybrid algorithm for finding the kth smallest of n elements in o(n) time. *Annals of Operations Research, 13*(1), 399–419. doi:10.1007/BF02288326.

Garfinkel, R. S., & Nemhauser, G. L. (1972). *Integer programming*: 4. New York: Wiley.

Gilmore, P. C., & Gomory, R. E. (1961). A linear programming approach to the cutting-stock problem. *Operations Research, 9*(6), 849–859. doi:10.1287/opre.9.6.849.

Gilmore, P. C., & Gomory, R. E. (1963). A linear programming approach to the cutting stock problem – part II. *Operations Research, 11*(6), 863–888. doi:10.1287/opre.11.6.863.

Gilmore, P. C., & Gomory, R. E. (1966). The theory and computation of knapsack functions. *Operations Research, 14*(6), 1045–1074. doi:10.1287/opre.14.6.1045.

Greenberg, H. (1986). On equivalent knapsack problems. *Discrete Applied Mathematics, 14*(3), 263–268. doi:10.1016/0166-218X(86)90030-2.

Greenberg, H., & Feldman, I. (1980). A better step-off algorithm for the knapsack problem. *Discrete Applied Mathematics, 2*(1), 21–25. doi:10.1016/0166-218X(80)90051-7.

Gschwind, T., & Irnich, S. (2016). Dual inequalities for stabilized column generation revisited. *INFORMS Journal on Computing, 28*(1), 175–194. doi:10.1287/ijoc.2015.0670.

Gurobi Optimization, L. (2018). *Gurobi optimizer reference manual*. URL: http://www.gurobi.com.

Hu, T. C. (1969). Integer programming and network flows. *Technical Report*. DTIC. DTIC Document.

Hu, T. C., Landa, L., & Shing, M.-T. (2009). The unbounded knapsack problem. In W. Cook, L. Lovász, & J. Vygen (Eds.), *Research trends in combinatorial optimization* (pp. 201–217). Berlin, Heidelberg: Springer. doi:10.1007/978-3-540-76796-1_10.

IBM (2018). *CPLEX user's manual*. URL: https://www.ibm.com/analytics/cplex-optimizer.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. Berlin: Spinger-Verlag. doi:10.1007/978-3-540-24777-7.

Landa, L. (2004). Sage Algorithms for Knapsack Problem. *Technical Report, CS2004-0794*. San Diego 9500 Gilman Drive, La Jolla, CA 92093-0021: University of California. URL: http://csetechrep.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2004-0794.

Martello, S., & Toth, P. (1977). Branch-and-bound algorithms for the solution of the general unidimensional knapsack problem. In *Advances in operations research* (pp. 295–301). North-Holland, Amsterdam.

Martello, S., & Toth, P. (1990). An exact algorithm for large unbounded knapsack problems. *Operations Research Letters, 9*(1), 15–20. doi:10.1016/0167-6377(90)90035-4.

Pisinger, D. (1994). *Dominance relations in unbounded knapsack problems. DIKU report 94/33*. DIKU.

Poirriez, V., & Andonov, R. (1998). Unbounded knapsack problem: New results. In R. Battiti, & A. A. Bertossi (Eds.), *Proceedings of the workshop on algorithms and experiments, Alex98* (pp. 103–111).

Poirriez, V., Yanev, N., & Andonov, R. (2009). A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization, 6*(1), 110–124. doi:10.1016/j.disopt.2008.09.004.

Shapiro, J. F., & Wagner, H. M. (1967). A finite renewal algorithm for the knapsack and turnpike models. *Operations Research, 15*(2), 319–341. doi:10.1287/opre.15.2.319.

Zhu, N., & Broughan, K. (1997). On dominated terms in the general knapsack problem. *Operations Research Letters, 21*(1), 31–37. doi:10.1016/S0167-6377(97)00018-7.