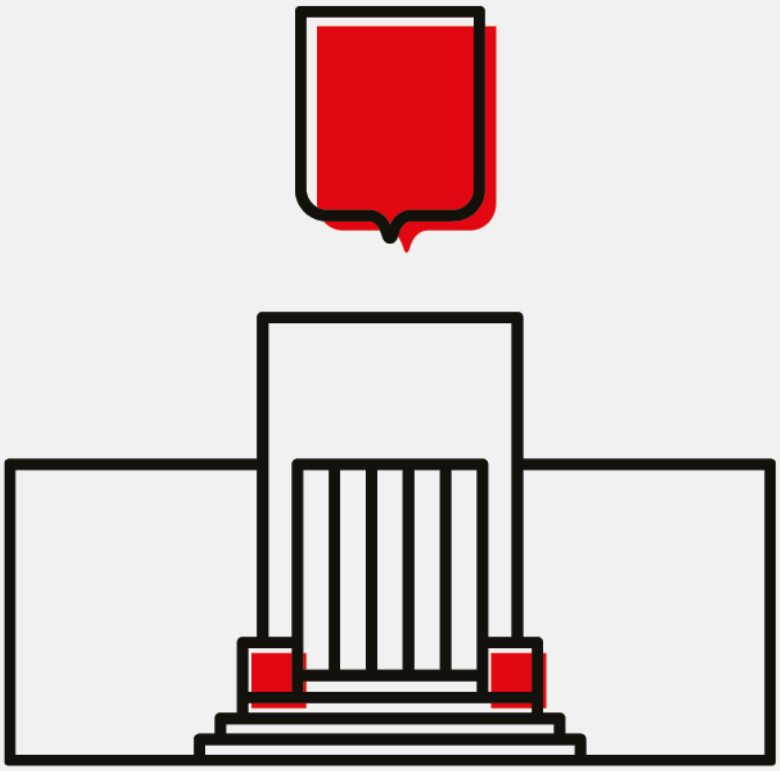
Document de travail 2021-015

A BRANCH-AND-PRICE ALGORITHM FOR THE MULTIPLE KNAPSACK PROBLEM



Olivier LALONDE
Jean-François CÔTÉ
Bernard GENDRON

Décembre 2021

Publié par :

Faculté des sciences de l'administration
2325, rue de la Terrasse
Pavillon Palasis-Prince, Université Laval
Québec (Québec) Canada G1V 0A6

Voir tous les documents de travail

Document de travail également publié par le Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, sous le numéro CIRRELT-2021-49

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2021
Bibliothèque et Archives Canada, 2021
ISBN 978-2-89524-524-7 (PDF)

Faculté des sciences de l'administration

UNIVERSITÉ LAVAL

A BRANCH-AND-PRICE ALGORITHM FOR THE MULTIPLE KNAPSACK PROBLEM

Olivier Lalonde[1,2], Jean-François Côté[1,3,*], Bernard Gendron[1,2]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Department of Computer Science and Operations Research, Université de Montréal

[3] Department of Operations and Decision Systems, 2325, rue de la Terrasse, Université Laval, Québec, Canada, G1V 0A6

*Corresponding author: jean-francois.cote@cirrelt.ca

ABSTRACT

The Multiple Knapsack Problem is a well-studied combinatorial optimization problem with several practical and theoretical applications. It consists of packing some subset of *n* items into *m* knapsacks such that the total profit of the chosen items is maximum. A novel Lagrangian relaxation based on a reformulation of the problem is presented, and it is proven that it dominates all commonly used relaxations for this problem. A branch-and-price algorithm is then derived from it, which takes advantage of the fact that the novel Lagrangian relaxation makes it possible to effectively control whether an item is included in some knapsack or not. An improved algorithm for solving the resulting packing subproblems is also introduced. Computational experiments then show that the new approach achieves state-of-the-art results.

Keywords: Multiple Knapsack Problem, branch-and-price, Lagrangian relaxation.

Acknowledgments: Financial support for this work was provided by the Natural Sciences and Engineering Council of Canada (NSERC) under grants 2017-06054 and 2021-04037. This support is gratefully acknowledged. We also thank CIRRELT for providing access to their computing facilities.

1 Introduction

Given n items, each with a weight $w_j \in \mathbb{Z}^+$ and a profit $p_j \in \mathbb{Z}^+$, and m knapsacks, each with a capacity $c_i \in \mathbb{Z}^+$, the Multiple Knapsack Problem (MKP) consists in finding an assignment of a subset of items to the knapsacks such that the total profit of the assigned items is maximal. This problem may be modeled using the following binary integer programming model:

$$\text{(MKP)} \qquad \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{1}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_{ij} \leq c_i \qquad \text{for } i = 1, ..., m \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad \text{for } j = 1, ..., n \tag{3}$$

$$x_{ij} \in \{0, 1\} \quad \text{for } i = 1, ..., m \;\; j = 1, ..., n \tag{4}$$

Here, the binary decision variable x_{ij} corresponds to whether or not item j is assigned to knapsack i. The objective (1) is to maximize the total profit of the selected items. The first set of constraints (2) enforces that the total weight of all items associated to a given knapsack i may not exceed its capacity, and the second set of constraints (3) ensures that every item may be assigned to at most one knapsack.

Several heuristic and exact algorithms for the MKP have been proposed. Heuristic algorithms were proposed by Hung and Fisk (1979), Laaloui and M'Hallah (2016), Laaloui (2013), Lalami et al. (2012) and Martello and Toth (1981b). In terms of exact algorithms, which is what this paper is concerned with, the problem was first tackled in Ingargiola and Korsh (1975), in which an enumerative algorithm was presented. Basic branch-and-bound algorithms, which select an item and branch on the knapsack to which it is assigned as well as it being excluded from the solution, were proposed in Hung and Fisk (1978) and Hung and Fisk (1979). In Martello and Toth (1981a), a new branch-and-bound framework was suggested, which only assigns items to the knapsack of least remaining capacity, and a mechanism, called bound-and-bound, is used, in which a heuristic algorithm is used to produce good partial solutions in the aim of reducing the size of the search tree. The resulting algorithm, called **MTM**, turned out to be much more efficient than the previous algorithms. Later on, Pisinger (1999) suggested an improved version of MTM, **MULKNAP**, which includes a basic item reduction procedure, capacity lifting as well as a splitting heuristic, which made it possible for the first time to solve instances with large $\frac{n}{m}$ ratios to optimality. Fukunaga and Korf (2005) and Fukunaga (2011) improved on MULKNAP by using a different branching scheme, in which the algorithm branches directly on assignments to the knapsack of smallest remaining capacity instead of branching on whether a given item is included in it or excluded from it. A dominance criterion for eliminating assignments and various symmetry-breaking procedures are used to reduce the size

of the search tree, leading to an algorithm called **2D/PS+B** that outperformed MULKNAP on instances with a small $\frac{n}{m}$ ratio.

More recently, in Dell'Amico et al. (2019), various reformulations of the problem based on an arc-flow reformulation of Valério de Carvalho (1999) are explored, and two decomposition algorithms, the so-called *Knapsack-based decomposition* and *Reflect-based decomposition*, were presented. The general idea is to solve a relaxation of the MKP that selects a set of promising items as master problem, and then solve a subproblem to verify if it is possible to assign the selected items to the bins. If a feasible assignment is possible, then the problem is solved, otherwise, the master problem is solved again with an additional constraint that excludes the set of items that was previously selected. The master problem of the Knapsack-based decomposition is basically a classical knapsack problem where the capacity is equal to the sum of all the bin capacities. The Reflect-based decomposition uses a more sophisticated formulation to select promising items. As for subproblems, they are solved using an arc-flow model. These algorithms turned out to be much more efficient than all previously known methods for large-size instances, being the first to solve problems with a large number of knapsacks and with a small $\frac{n}{m}$ ratio.

While being considerably more effective than all previously known algorithms on large instances, the decomposition algorithms of Dell'Amico et al. (2019) nevertheless have the flaw that they often need to solve many packing problems until they can find a set of items that can feasibly be packed into the bins, which tend to be very hard to solve for large instances. In this paper, we suggest an alternative approach that also aims to reduce the size of the search space by having the decision variables to control whether or not an item is included in any knapsack. This approach mostly relies on enumeration rather than solving packing problems to identify the subset of items that is part of the optimal solution. This new approach is based on a novel Lagrangian relaxation for the MKP, which not only turns out to yield tighter bounds than all other known relaxations for the MKP but also allows one to effectively control whether a given item is included in any knapsack, as opposed to only being able to effectively control whether an item is included in some specific knapsack or not, as all previous enumerative algorithms based their branching procedure upon. This novel Lagrangian relaxation is then integrated in a branch-and-price algorithm. As this approach also requires solving packing subproblems, which we refer to as variable-sized bin packing satisfiability problems (VSBPP-SAT), a new algorithm to tackle these subproblems is also presented.

The remainder of the paper is organized as follows. In Section 2, we discuss the most common relaxations for the MKP, we present a novel Lagrangian relaxation and we show the domination relations among these relaxation. Section 3 describes a branch-and-price algorithm, called **BP-MKP**, to solve the MKP. Our algorithm to solve the VSBPP-SAT subproblems are presented in 4. The results of computational experiments on benchmark instance sets are presented in Section 5. They show that the new approach achieves state-of-the-art results.

2 Relaxations for the MKP

In this section, we first present the two relaxations that have been used for the MKP, namely the surrogate relaxation (SMKP) and the Lagrangian relaxation (L_1), which both dominate the linear relaxation (see Martello and Toth, 1990). We then present our improved Lagrangian relaxation (L_2) and show that it dominates both the surrogate relaxation and the usual Lagrangian relaxation L_1.

2.1 Surrogate relaxation

Given a vector of multipliers $\pi \in \mathbb{R}_+^m$, the surrogate relaxation $SMKP(\pi)$ of the MKP, with optimal value $z_{SMKP(\pi)}$, is defined to be the result of aggregating the m knapsack constraints, each weighted with the multiplier π_i. The resulting model is the following:

$$\text{(SMKP}(\pi)\text{)} \qquad \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{5}$$

$$\text{s.t.} \sum_{i=1}^{m} \pi_i \sum_{j=1}^{n} w_j x_{ij} \leq \sum_{i=1}^{m} \pi_i c_i \tag{6}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad \text{for } j = 1, ..., n \tag{7}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for } i = 1, ..., m \;\; j = 1, ..., n \tag{8}$$

In Martello and Toth (1981a), the authors proved that the choice multipliers that provides the tightest upper bound is $\pi_i = k$ for some positive constant k. This reduces to solving a single knapsack problem on the n items whereby the capacity of the single knapsack is set to be the sum of the capacities of the original knapsacks:

$$\text{(SMKP)} \qquad \max \sum_{j=1}^{n} p_j t_j \tag{9}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j t_j \leq \sum_{i=1}^{m} c_i \tag{10}$$

$$t_j \in \{0, 1\} \qquad \text{for } j = 1, ..., n \tag{11}$$

This is what will be referred to from now on as the surrogate relaxation SMKP of the MKP, with optimal value z_{SMKP}.

While the knapsack problem is NP-hard, it can be solved very efficiently in practice using algorithms such as *combo* from Martello et al. (1999), which is why it was usually preferred to the Lagrangian relaxation L_1 in the literature when designing branch-and-bound algorithms for the MKP. Another advantage of this relaxation is the possibility of attempting to split the items in the solution of the SMKP into the m knapsacks, thereby producing a feasible solution to the initial problem, as was first suggested by Pisinger (1999). In the context of a branch-and-and-bound algorithm, this

makes it possible to find good lower bounds on the optimal value quickly, and thus to prune larger parts of the search tree. Furthermore, it is known that if $\frac{n}{m}$ is large (say, 10 or greater), it almost always holds that the bound given by the surrogate relaxation is compact, and it is very often possible to assign all the items chosen into the m knapsacks at the root node, and thus to find the optimal solution without performing any search, making such problems very easy. This technique was integrated in many successful algorithms for the MKP, most notably MULKNAP Pisinger (1999) and 2D/PS+B Fukunaga (2011). The model (SMKP) is also the MIP that is used in the knapsack-based decomposition of Dell'Amico et al. (2019).

2.2 Lagrangian relaxation (L_1)

Given a vector $\lambda \in \mathbb{R}^n_+$, whose elements are referred to as Lagrange multipliers, the Lagrangian relaxation $L_1(\lambda)$ of the MKP, with optimal value $z_{L_1(\lambda)}$, is obtained by relaxing constraints (3) in a Lagrangian way:

$$(L_1(\lambda)) \qquad \max \sum_{j=1}^{n} \left((p_j - \lambda_j) \sum_{i=1}^{m} x_{ij} \right) + \sum_{j=1}^{n} \lambda_j \tag{12}$$
$$\text{s.t. (2) and (4)}$$

The Lagrangian relaxation L_1, with optimal value z_{L_1}, is defined to be $\min\{z_{L_1(\lambda)} | \lambda \geq 0\}$. For a given choice of λ, computing $z_{L_1(\lambda)}$ reduces to computing m individual knapsack problems.

In contrast to the surrogate relaxation, there is no known analytical expression for the optimal choice of multipliers λ_j. This relaxation has been rarely used in state-of-the-art algorithms because of the very high computational cost associated to it: while solving the surrogate relaxation requires solving only one knapsack problem, typical methods for solving Lagrangian relaxations, such as subgradient optimization or column generation, might require solving thousands of knapsack problems, and are therefore much more computationally expensive. The Lagrangian relaxation L_1 was most notably used in the algorithms of Hung and Fisk (1979) and Martello and Toth (1980), although no attempt was made to solve it to optimality in either of these algorithms. Furthermore, there is no dominance relation between the Lagrangian relaxation L_1 and the surrogate relaxation SMKP, although L_1 does seem to provide tighter bounds on average, as shown in Section 5.1.

2.3 Improved Lagrangian relaxation (L_2)

The new Lagrangian relaxation which we are proposing is based on the following reformulation of the problem:

$$\max \sum_{j=1}^{n} p_j t_j \tag{13}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_{ij} \leq c_i \qquad \text{for } i = 1, ..., m \tag{14}$$

$$t_j \leq \sum_{i=1}^{m} x_{ij} \qquad \text{for } j = 1, ..., n \tag{15}$$

$$\sum_{j=1}^{n} w_j t_j \leq \sum_{i=1}^{m} c_i \tag{16}$$

$$t_j \in \{0, 1\} \qquad \text{for } j = 1, ..., n \tag{17}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for } i = 1, ..., m \;\; j = 1, ..., n \tag{18}$$

Here, the binary variable x_{ij} once again corresponds to whether or not item j is assigned to knapsack i, and the binary variable t_j equals one if item j is assigned to some knapsack and zero otherwise. As in model (1)-(4), the objective function (13) represents the total profit of the chosen items, which is to be maximized. Constraints (14) impose that the sum of the weights of the items that are assigned to knapsack i to be less than the capacity of that knapsack. Constraints (15) ensure that the binary variable t_j can be equal to one only if one of the corresponding x_{ij} for $i \in \{1, ..., m\}$ is one. Note that we no longer require that an item must be assigned to at most one knapsack, since the x_{ij} are not part of the objective function. Constraint (16) follows from aggregating the m constraints (14) and replacing $\sum_{i=1}^{m} x_{ij}$ with t_j for every j, which is valid on account of constraints (15) . Constraint (16) is redundant in model (13)-(18), but it will not be in the Lagrangian subproblem that we now introduce.

For a vector μ of n nonnegative Lagrangian multipliers, the Lagrangian relaxation L_2, with optimal value $z_{L_2(\mu)}$, is then obtained from model (13)-(18) by relaxing constraints (15) in a Lagrangian way:

$$(L_2(\mu)) \qquad \max \sum_{j=1}^{n} (p_j - \mu_j) t_j + \sum_{j=1}^{n} \mu_j \left(\sum_{i=1}^{m} x_{ij} \right) \tag{19}$$

$$\text{s.t. } (14), (16) - (18)$$

Once again, the Lagrangian relaxation L_2 with optimal value z_{L_2} is defined to be $\min\{z_{L_2(\mu)} | \mu \geq 0\}$. For a given choice of μ, computing $z_{L_2(\mu)}$ reduces to computing $m+1$ individual knapsack problems, one for every knapsack as well as one for the t_j.

2.4 Dominance relations

While there exists no dominance relation between the surrogate relaxation SMKP and the Lagrangian relaxation L_1, the Lagrangian relaxation L_2 dominates them both. Proving this for the surrogate relaxation is straightforward:

Proposition 2.1. *$z_{L_2} \leq z_{SMKP}$, with equality if and only if $\mu_j = 0$ is an optimal choice of multipliers for L_2.*

Proof. $L_2(0)$ reduces to the surrogate relaxation. Since z_{L_2} is defined to be $\min\{z_{L_2(\mu)} | \mu \geq 0\}$, the statement of the proposition follows. □

In order to prove that L_2 dominates L_1, the following lemma is required:

Lemma 2.2. *Let λ be an optimal choice of multipliers for L_1. Then, for all j, $\lambda_j \leq p_j$.*

Proof. We prove the contrapositive. Let λ be any choice of nonnegative multipliers such that, for some k, $\lambda_k > p_k$. Let $\bar{\lambda}$ be defined as:

$$\bar{\lambda}_j = \begin{cases} \lambda_j & j \neq k \\ p_j & j = k \end{cases}$$

In the case of $L_1(\lambda)$, we have that $x_{ik} = 0$ in the optimal solution for bin $i = 1, ..., m$, because their contribution in the objective value is negative, and thus they may be removed from the problem. In the case of $L_1(\bar{\lambda})$, the x_{ik} do not contribute to the objective, and thus they may be removed from the problem as well. Then, we have that:

$$z_{L_1(\bar{\lambda})} = z_{L_1(\lambda)} - \lambda_k + p_k < z_{L_1(\lambda)}$$

In other words, λ was not an optimal choice of multipliers for L_1. □

We are now in a position to show the following:

Proposition 2.3. *$z_{L_2} \leq z_{L_1}$. Furthermore, if, for some choice of optimal multipliers λ for L_1, it holds that $\sum_{j=1}^{n} \delta_j w_j > \sum_{i=1}^{m} c_i$, where δ_j is defined to be 1 if $\lambda_j > 0$ and 0 otherwise, then this inequality is strict.*

Proof. Let λ be an optimal choice of multipliers for L_1. Define μ by:

$$\mu_j = p_j - \lambda_j \geq 0$$

Then, $L_2(\mu)$ is the following problem:

$$\begin{aligned} \max \ & \sum_{j=1}^{n} \lambda_j t_j + \sum_{j=1}^{n} (p_j - \lambda_j) \sum_{i=1}^{m} x_{ij} \\ \text{s.t. } & (14), (16) - (18) \end{aligned} \tag{20}$$

By dropping the constraint on the t_j, we get back $L_1(\lambda)$, which shows that:

$$z_{L_2} \leq z_{L_2(\mu)} \leq z_{L_1(\lambda)} = z_{L_1}$$

Moreover, the equality $z_{L_1} = z_{L_2}$ can only hold if the second inequality is an equality, which can only happen if the optimal value of the following problem equals $\sum_{j=1}^{n} \lambda_j$:

$$\max \sum_{j=1}^{n} \lambda_j t_j \tag{21}$$
$$(16) \text{ and } (18)$$

This can only happen if it is possible to set t_j to one for all j's such that $\lambda_j > 0$. The statement of the theorem follows. □

The proof of the previous proposition also shows why L_2 provides much tighter bounds than L_1 in practice. Even if μ as defined in the proof of the theorem were an optimal choice of multipliers for L_2, if most λ's are nonnegative and large enough, the gap between $\sum_{j=1}^{n} \lambda_j$ and the optimal value of problem (21) could turn out to be quite large as well.
The extent to which the bounds provided by the new relaxation L_2 are tighter than those provided by the surrogate relaxation, the Lagrangian relaxation L_1 and some other possible relaxations for the MKP will be discussed in Section 5.1, where empirical results on benchmark instances will be supplied.

3 The Branch-and-Price algorithm

In this section, we present how the MKP is solved by a branch-and-price algorithm using the improved Lagrangian relaxation L_2.

3.1 Dantzig-Wolfe reformulation

Our algorithm uses the Lagrangian relaxation (L_2) to solve the MKP to optimality by using a column generation procedure. We begin by applying Dantzig-Wolfe decomposition to the model (13)-(18), where constraints (14) and (15) are part of the subproblems and constraints (16) are kept in the master problem. In this context, a pattern of items a (also referred to as a column) is an element of $\{0,1\}^n$, where a_j equals one if item j is part of the pattern and zero otherwise. To the i'th constraint (15) is associated a class of patterns $P^i \subseteq \{0,1\}^n$ which encodes the subset of items that may be assigned to the i'th knapsack without violating the corresponding capacity constraint:

$$P^i = \{a \in \{0,1\}^n | \sum_{j=1}^{n} w_j a_j \leq c_i\} \tag{22}$$

Additionally, there is a class of patterns associated with the aggregated capacity constraint (16), denoted P^0, which encodes a subset of items that may be selected in a solution without violating the aggregated capacity constraint (16):

$$P^0 = \{a \in \{0,1\}^n | \sum_{j=1}^{n} w_j a_j \leq \sum_{i=1}^{m} c_i\} \tag{23}$$

To each pattern a is then associated a binary variable y_a, which encodes whether the pattern is chosen or not. Rewriting model (13)-(18) according to the correspondence $x_{ij} \equiv \sum_{a \in P^i} a_j y_a$ and $t_j \equiv \sum_{a \in P^0} a_j y_a$ and adding the constraint that exactly one pattern must be chosen for every class of patterns gives the following equivalent model:

$$\max \sum_{j=1}^{n} p_j \left(\sum_{a \in P^0} a_j y_a \right) \tag{24}$$

$$\text{s.t.} \sum_{a \in P^0} a_j y_a \leq \sum_{i=1}^{m} \left(\sum_{a \in P^i} a_j y_a \right) \qquad \text{for } j = 1, ..., n \tag{25}$$

$$\sum_{a \in P^i} y_a = 1 \qquad \text{for } i = 0, ..., m \tag{26}$$

$$y_a \in \{0,1\} \qquad \text{for } a \in P^i, \text{ for } i \in 0, ..., m \tag{27}$$

The objective function (24) maximizes the profits obtained from the items in patterns P^0. Constraints (25) are the *item constraints*, which correspond to the constraints (15) in the model (13)-(18). Constraints (26) ensure that there is at most one pattern selected for each bin and the aggregated bin capacity constraint.

When dropping the integrality constraints (27) on the y_a, requiring only that they be nonnegative, we thereby obtain a linear programming model whose optimal value is equal to z_{L_2}, by the theory of Lagrangian duality.

3.2 Solution approach

Our approach to solve the MKP shares several characteristics with many classical branch-and-bound methods for the Knapsack Problem(Martello and Toth, 1990). These methods solve a relaxation of the KP at each node of the tree. If the solution of the relaxation has at least one fractional t_j^* variable, then one fractional t_j^* is selected and two new problems are created and solved recursively: one where the item is taken ($t_j = 1$) and one where it is excluded ($t_j = 0$). On the contrary, if all t_j^* variables are integer, then, a feasible solution was found. Once all nodes are explored, the feasible solution having the highest total profits is the optimal solution.

For the MKP, we explore a branch-and-bound tree as well, but it works differently. At each node, we solve the linear relaxation of the model (13)-(18). This helps in finding promising items that

should be part of the optimal solution. If at least one t_j^* variable is fractional ($t_j^* = \sum_{a \in P^0} a_j y_a^*$), we branch on one of them. We do not branch on the pattern variables y_a, nor on the bin assignation x_{ij} variables ($x_{ij}^* = \sum_{a \in P^i} a_j y_a^*$). If the t_j^* variables are integer, we check if the bin assignation variables x_{ij}^* are integer, and if they are, we found a feasible solution of the MKP and the best solution is updated accordingly. In the other case, if only the t_j^* variables are integer, we solve the VSBPP-SAT subproblem to find a feasible assignment of the items with $t_j^* = 1$ into the m knapsacks. If a feasible assignment is found, then we have a feasible solution of the MKP, and we update the best known solution. In that case, there is no need to pursue the enumeration of that branch and we fathom the node. In the other case, we have proved that no feasible assignment can be found and a constraint is added to the model (13)-(18) to forbid the set of items with $t_j^* = 1$ to be selected by the model. Let $S = \{j | t_j^* = 1\}$ be the set of items that were selected by the relaxation. Then, to prevent the set S of items from being selected again, we add the following constraint:

$$\sum_{j \in S} \sum_{a \in P^0} a_j y_a \leq |S| - 1. \tag{28}$$

The constraint states that at most $|S| - 1$ items of S can be present in any solution of (13)-(18). This type of constraints are also known as *no-good cuts*. Let $\mathcal{S}$ be the set of all infeasible S that were found so far. The set is empty at the root node of the tree and each time a new infeasible S is found, a no-good cut (28) is added to the model and S is added to $\mathcal{S}$. This constraint is similar to the classical *cover inequality* constraints for the knapsack problem and it can be strengthen easily by considering the extension of $E(S) = S \cup \{l = 1, ..., n | l \notin S \text{ and } w_l \geq w_{max}\}$ where $w_{max} = \max_{j \in S}\{w_j\}$. The following is an improved no-good cut:

$$\sum_{j \in E(S)} \sum_{a \in P^0} a_j y_a \leq |S| - 1. \tag{29}$$

Once a constraint (29) is added, the relaxation is solved again and the resolution process continues.

We thereafter define as *master problem* the model (13)-(18) with possibly some constraints (29). Column generation is employed to solve its linear relaxation. Each step of the column generation consists in solving *pricing subproblems* to identify columns of positive reduced cost. If some can be found, they are added to the master problem, and it is solved again. Otherwise, the VSBPP-SAT subproblem is solved if the t_j^* are integer, or a branching operation occurs if some t_j^* are fractional.

The next sections detail each step of our solution approach. Section 3.3 presents how the column generation is performed and how the pricing subproblems are solved. Next, Section 3.4 describes an alternative method to generate columns that reduces convergence problems happening for some instances. Section 3.5 presents the outline of the branch-and-price algorithm. Section 3.6 details how variables are selected for branching and Section 3.7 shows how variable filtering is used to

reduce the size of the search tree. The last details of our approach are found in Section 4 that is devoted to the solution of the VSBPP-SAT subproblem.

3.3 Column generation

We rely on column generation to solve the master problem as the sets P^i for $i = 0, ..., m$ are too large to be enumerated. At the root node, we solve a *restricted master problem* that includes only a tractable subset of the patterns P^i for $i = 0, ..., m$. Once the linear relaxation of the restricted master problem is solved, we search for patterns that were not included in the tractable subset and that have a positive reduced cost. Let $\mu_j \geq 0$ be the dual variable associated with the constraint (25) for item j, π_i be dual variable associated with the constraint (26) for bin i, and $\theta_S \geq 0$ the dual variables associated with the no-good cuts (29) of the set S. Then, the reduced cost ζ_a^i of pattern $a \in P^i$ for bin $i = 1, ..., m$ is computed as:

$$\zeta_a^i = \sum_{j=1}^{n} \mu_j a_j - \pi_i \tag{30}$$

For bin $i = 0$, let $\theta_j = \sum_{S \in \mathcal{S}, j \in S} \theta_S$, and the reduced cost ζ_a^0 of pattern $a \in P^0$ is be computed as:

$$\zeta_a^0 = \sum_{j=1}^{n} (p_j - \theta_j - \mu_j) a_j - \pi_0 \tag{31}$$

To find patterns with a positive reduced cost, we solve a knapsack problem for each bin capacity, and one for the aggregated bin capacity constraint. Let z_j be a binary variable equal to 1 if item j is being part of a pattern. Then, the pricing subproblem for bin $i = 1, ..., m$ is the following:

$$\max \sum_{j=1}^{n} \mu_j z_j - \pi_i \tag{32}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j z_j \leq c_i \tag{33}$$

$$z_j \in \{0, 1\} \qquad \text{for } j = 1, ..., n \tag{34}$$

The objective function (32) maximizes the reduced cost of the pattern. Constraints (33) ensure the capacity of the bin i is respected. The pricing subproblem for bin 0 uses the same binary variables z_j and is as follows:

$$\max \sum_{j=1}^{n}(p_j - \theta_j - \mu_j)z_j - \pi_0 \tag{35}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j z_j \leq \sum_{i=1}^{m} c_i \tag{36}$$

$$z_j \in \{0,1\} \qquad \text{for } j = 1, ..., n \tag{37}$$

The objective function (35) maximizes the reduced cost of the pattern, and constraints (36) ensure that the aggregated bin capacity is respected.

If at least one of the above problem could find a pattern with a positive reduced cost, we add those patterns to the master problem and the column generation procedure is repeated, otherwise, it is stopped. We also compute the dual bound to stop it earlier. Let ζ^i be the reduced cost of the pattern found when solving the knapsack problem of bin $i = 0, ..., m$. Then, the dual bound $\bar{z}$ can be computed as follows:

$$\bar{z} = \sum_{i=0}^{m}(\zeta^i + \pi_i) + \sum_{S \in \mathcal{S}}(|S| - 1)\theta_S \tag{38}$$

Let $\underline{z}$ be the current value of the master problem and LB a global lower bound on the MKP. Then, if $\lfloor \bar{z} \rfloor \leq$ LB or if $\lfloor \underline{z} \rfloor \leq \bar{z} < \lceil \underline{z} \rceil$, terminate the column generation procedure.

3.4 Subgradient optimization

For many large size instances, we encounter poor convergence for solving the master problem to optimality using standard column generation. Instead, following from Klose and Görtz (2007), we generate new patterns by solving L_2 using subgradient optimization at different stages of column generation to mitigate the convergence problems. We use the subgradient optimization procedure described in Görtz and Klose (2012):

1. *At the root node*: Perform 500 subgradient steps to generate the initial patterns that are feeded to the restricted master problem. The Lagrangian multipliers are initialized as $\mu_j = 0$ for all j's, which corresponds to the surrogate relaxation of the MKP.

2. *At subsequent nodes*: Perform 50 subgradient steps to initiate the restricted master problem. Use the patterns from the parent node as well.

3. *During column generation*: If more than 5 iterations of column generation have been performed, run 10 subgradient steps with initial multipliers μ_j equal to the value of the dual variables associated with constraints (25).

4. *When solving the Lagrangian subproblems*: Solve the Lagrangian subproblem, with optimal value $L_2(\mu)$ and with optimal solution t^*, x^*. If $L_2(\mu) \leq LB$, stop and fathom the node.

5. *When adding patterns to the restricted master model*: If the subgradient steps are performed at the root node, for every $i \in \{1, ..., m\}$, add a pattern in the P^i'th class of the restricted master problem composed of the items for which $x^*_{ij} = 1$, and add a pattern in the P^0'th class of the restricted master problem composed of the items for which $t^*_j = 1$. If the subgradient procedure is being performed at the other nodes, add only the patterns with positive reduced cost.

6. *Updating* μ: Set $g_j = \sum_{i=1}^{n} {x_{ij}}^* - t^*_j$ and $\mu_j = \mu_j - \theta g_j$, where θ, which denotes the step length, is given by the following formula:

$$\theta = \alpha \frac{(L_2(\mu) - LB)}{||g||^2}$$

7. *Stopping criteria*: 1) stop if 10 iterations were performed, or 2) let $\alpha = 2$ and each time the dual bound $L_2(\mu)$ is not smaller than those obtained in the last 5 iterations, divide α by 2, and stop if $\alpha \leq 10^{-5}$. The second criterion is helpful at reducing the number of subgradient steps performed at the root node (500 iterations) or at each subsequent node (50 iterations). These stopping criteria differs slightly from those proposed by Görtz and Klose (2012).

3.5 Outline of the algorithm

This section presents the outline of the branch-and-price algorithm.

1. Perform the initialization phase of Section 3.8. A lower and an upper bound are computed, and if they are equal, the algorithm stops. Otherwise, perform a preprocessing to reduce the size of the instance.

2. Solve the root node by performing the steps 5.a to 5.e. Create a list $\mathcal{L}$ of active nodes and add the root node to it.

3. Remove the node in $\mathcal{L}$ with the highest upper bound and let t^*_j and x^*_{ij} be its optimal solution. If $\mathcal{L}$ is empty, stop.

4. Select a fractional t^*_j to branch on according to the branching strategy of Section 3.6, and create two child nodes where we set $t_j = 0$ in one and $t_j = 1$ in the other.

5. Perform the following for each node:

   (a) Solve the node using column generation and subgradient optimization.

   (b) If its solution value is not better than LB, go back to Step 3.

   (c) If the t^*_j and x^*_{ij} variables are integer, update the global lower bound LB. Remove all nodes in $\mathcal{L}$ with a worse value than LB.

(d) If only the t_j^* variables are integer, solve the *packing subproblem*. If it is feasible, update the global lower bound LB, and remove all nodes in $\mathcal{L}$ with a worse value than LB. Otherwise, it is infeasible, add a no-good cut (29) and go back to Step 5.a.

(e) Add the node to $\mathcal{L}$ if the t_j^* are fractional.

6. Go back to Step 3.

3.6 Branching strategy

As is done in Klose and Görtz (2007), the branching variable is chosen among a candidate set $\mathcal{C} = \{j \in \{1, ..., n\} : t_j \text{ is free and } l \leq t_j^* \leq u\}$, where $l = 0.75\text{max}\{t_j^* | 0 < t_j^* \leq 0.5\}$ and where $u = 0.75\text{max}\{t_j^* | 0.5 \leq t_j^* < 1\}$. If there are no j's such that $0 < t_j^* \leq 0.5$, l is set to 0.5, and if there are no j's such that $0.5 \leq t_j^* < 1$, u is set to 0.5. The following branching rules were tested:

1. Branch on the $j \in \mathcal{C}$ with the largest value of $\frac{p_j}{w_j}$.
2. Branch on the $j \in \mathcal{C}$ with the smallest value of $L_2(\mu)$ with t_j forced to $1 - \lfloor t_j^* + 0.5 \rfloor$.
3. Branch on the $j \in \mathcal{C}$ such that setting $t_j = 1 - \lfloor t_j^* + 0.5 \rfloor$ maximizes the number of patterns from P^0 with associated $y_a > 0$ that become infeasible.

Branching rule 3 is inspired by the fifth branching rule that is mentioned in Klose and Görtz (2007). While it would appear that branching rule 1 is the most efficient one for small-sized problems, its performance is rather poor for larger problems. Branching rule 2 aims to branch on the variable that is most "important", i.e., so that the two resulting subtrees will be as balanced as possible. Branching rules 2 and 3 are much more resilient to size increase and appear to have a comparable performance, although preliminary tests have shown branching rule 3 to be more efficient, and is therefore the one that is used in the branch-and-price algorithm.

3.7 Filtering

We perform the following two variable filtering at each node of the branch-and-node tree:

Item dominance: It was proposed by Dell'Amico et al. (2019) for the MKP. Given two items j_1 and j_2, j_1 is said to dominate j_2 if $w_{j_1} \leq w_{j_2}$ and $p_{j_1} > p_{j_2}$. If, at any point in the search, j_1 is excluded from the solution, i.e., t_{j_1} is forced to 0, then j_2 may also be excluded from the solution, for any solution that contains j_2 and does not contain j_1, a better solution may be obtained by replacing j_2 with j_1. Similarly, if, at any point in the search, j_2 is included in the solution, i.e. t_{j_2} is forced to 1, then j_1 may be included in the solution as well.

More precisely, for every item j, we associate two sets of items, namely the dominated set D_{1j}, which contains all items dominated by j, and the dominating set D_{2j}, which contains all items which dominate j:

$$D_{1j} = \{k | w_j \leq w_k \text{ and } p_j > p_k\}$$

$$D_{2j} = \{k | w_j \geq w_k \text{ and } p_j < p_k\}$$

Whenever j is included in the solution, all $k \in D_{2j}$ are included in the solution as well, and whenever j is excluded from the solution, all $k \in D_{1j}$ are excluded as well.

Lagrangian probing: A basic Lagrangian probing, as described in Görtz and Klose (2012), is applied at each node after the column generation ran to reduce the number of free t_j. For every j such that t_j is free, t_j is tentatively set to $1 - \lfloor t_j^* + 0.5 \rfloor$, and the optimal value of the Lagrangian relaxation $L_2(\mu)$ is computed. If this value is smaller than the global lower bound LB, then t_j may be set to $\lfloor t_j^* + 0.5 \rfloor$.

3.8 Initialization

We perform the following two steps before running the branch-and-price algorithm. First, we run the MULKNAP algorithm for one second to compute an initial lower and upper bound. If they are equal, the problem is solved and we stop. Next, we perform the following instance reduction algorithm that was proposed by Dell'Amico et al. (2019). Let I be a subset of bins and $J = \{j | w_j \leq \max_{i \in I}\{c_i\}\}$ be the set of items that can be packed inside I. If a feasible packing of J inside the bins I can be found, then we remove both I and J from the instance. We start with $I = 1$ and iteratively add the next smallest bin to I. For each I, we try to solve the VSBPP-SAT with a time limit of 10 seconds if $\sum_{j \in J} w_j \leq \sum_{i \in I} c_i$.

4 Solving the packing subproblems

This section decribes how to solve the variable-sized bin packing satisfiability problem (VSBPP-SAT). In the following, we consider that bins and items can be duplicated, meaning that some items might have the same weight and some bins might have the same capacity. Let $I = \{1, ..., m'\}$ be the set of bins, where d_i is the number of bins of capacity c_i. Similarly, let $J = \{1, ..., n'\}$ be the set of items, w_i its weight and b_i the number of duplicates. The objective problem is to determine whether there exists a feasible packing of the items J inside the bins. This problem is similar to a satisfiability variant of the variable-sized bin packing problem (VSBPP) where bins have a usage cost and availabilities. The VSBPP has been extensively studied in the literature by numerous authors. The most efficient methods for solving the VSBPP include branch-and-price algorithms from Belov and Scheithauer (2002) and Alves and Valério de Carvalho (2008), and also pseudo-polynomial MIP formulations, also known as arc-flow formulations, from Valério de Carvalho (2002) and Delorme and Iori (2020).

The problem is transformed into a VSBPP where the cost of bins is 0. The objective of VSBPP-SAT is to decide if a feasible solution exists or not. To solve it, we developed a solver with three methods, namely a heuristic, a set-packing based procedure, and an arc-flow based procedure. Each method is called in order until a feasible packing is found, or it is proven that none exists, or a timeout has occurred. It should be noted that only the last method can prove infeasibility.

4.1 Heuristic

The problem is first tackled using a quick heuristic that performs a two-stage approach in which the first stage is a greedy heuristic and the second is an improvement procedure based on simulated annealing that calls the greedy heuristic. First, items are sorted by non-increasing weight and bins by non-decreasing capacity. Then, items are stored in an array where the sequence is important. The heuristic starts with the first bin and at each iteration it searches for the first item in the array with a weight equal to the residual capacity of the bin. If there is one, it adds the item to the bin and removes the item from the array. If there is none, it adds the first item in the array that fits inside the residual capacity. If there are no items that fits inside the bin, the heuristic moves to the next bin. The heuristic iterates until all items have been added, and in such case, it returns 0. If there are no bins left, it returns the sum of the weights of the items that are left.

The simulated annealing is executed for at most 2500 iterations to keep the computation time low. At each iteration, it selects two items randomly in the array, swaps them, and calls the greedy heuristic. The new sequence is accepted if the returned value is lower than the one of the previous sequence or it passes the classical simulated annealing criterion. If the sequence is rejected, the algorithm reverts to the previous sequence.

4.2 Set packing formulation

Experiments have shown that if the problem is feasible, the number of items per bin in the optimal solution is very often between 1 and 5. Also, many sets J coming from the master problem have a near perfect fit, meaning that the sum of weights of the items in J is very close to the sum of the capacities of the bins. Our second method exploits these observations by solving the VSBPP-SAT using a formulation similar to the classical formulation of the set packing problem with a reduced subset of patterns. Let $f = \sum_{i=1}^{m'} d_i c_i - \sum_{j=1}^{n'} b_j w_j$ be the free space. Our method tries to find a feasible solution of the VSBPP-SAT by generating patterns, where each pattern has at most f free space and at most α items.

If a feasible solution is found, the procedure is terminated: otherwise, we reiterate with $\alpha = \alpha+1$ or until an upper value is reached. Experiments have shown that this method works best with an initial value of $\alpha = 3$ and iterating up to $\alpha = 5$ (as larger values of α create millions of patterns that the MIP solver cannot handle).

At each iteration, the patterns of each bin i are defined as $\bar{P_\alpha^i} = \{a \in \mathbb{Z}^{n'} | c_i - f \leq \sum_{j=1}^{n'} a_j w_j \leq c_i, \sum_{j=1}^{n'} a_j \leq \alpha \text{ and } a_j \leq b_j, j = 1, ..., n'\}$. Each pattern is required to have at most f of free space, it has to respect the bin capacity, and the maximum number of items allowed at the iteration. The model contains binary variables $\bar{y_a}$ indicating if pattern a is chosen or not. The formulation is as

follows :

$$\max \sum_{j=1}^{m'} \sum_{a \in \bar{P}_\alpha^i} \sum_{i=1}^{n'} a_j \bar{y_a} \tag{39}$$

$$\text{s.t.} \sum_{a \in \bar{P}_\alpha^i} \bar{y_a} \leq d_i \qquad i = 1, ..., m' \tag{40}$$

$$\sum_{i=1}^{n'} \sum_{a \in \bar{P}_\alpha^i} a_j \bar{y_a} \leq b_j \qquad j = 1, ..., n' \tag{41}$$

$$\bar{y_a} \in \{0, 1\} \qquad a \in \bar{P}_\alpha^i,\ i = 1, ..., m' \tag{42}$$

Objective function (39) maximizes the number of taken items. Constraints (40) and (41) ensure that at most one pattern is chosen per bin and per item. Finally, constraints (42) define the nature and domain of the variables. If the optimal value is equal to $\sum_{j=1}^{n'} b_j$, meaning that all items have been assigned, a feasible solution has been found and we stop. Otherwise, we increment α and try again. If $\alpha \geq 6$, we stop and move to the next method. A time limit of 400 CPU seconds was given to this method.

4.3 Arc-flow formulation

The third method consists of solving one of two possible arc-flow formulations of the VSBPP developed. These formulations were first proposed for the classical bin packing problem (BPP) by Valério de Carvalho (1999). The idea is to construct a graph of pseudo-polynomial size where the nodes represent the possible bin fillings and the arcs represent the packing of an item. It is common for these graphs to grow very large in size when the bin capacity or the number of items is large. New types of graphs were proposed by Côté and Iori (2018) and Delorme and Iori (2020) to reduce their size and hence facilitate their resolution. These formulations are among the most performant methods to date for solving the BPP and its variants(Loti de Lima et al., 2022). This third method solves the VSBPP-SAT using the formulation from Valério de Carvalho (2002). Let $G = (V, A)$ be a directed graph where $V = \{0, 1, ..., c^*\}$ is the set of vertices and $c^* = \max_{i=1,...,m'}\{c_i\}$ is the capacity of the largest bin. Each node $p \in V$ represents a partial packing of a set of items having a sum of weights lesser or equal to p. Each arc $(p, q) \in A$ represents either 1) the packing of an item of weight $q - p$ added to the partial packing p and giving the partial packing q or 2) an empty space, also called *loss arc* between p and q. Let $\delta^-(q)$ and $\delta^+(q)$ define the set of arcs entering and leaving arcs of node q. Let x_{pq} be an integer variable indicating the number of times the arc $(p, q) \in A$ is

taken. Also, let r_i be an integer representing the number of times the bin i is used.

$$\min \sum_{i=1}^{m'} r_i \tag{43}$$

$$\text{s.t.} \sum_{(q,p)\in\delta^+(q)} y_{qp} - \sum_{(p,q)\in\delta^-(q)} y_{pq} = \begin{cases} \sum_{i=1}^{m'} r_i & \text{if } q = 0 \\ -r_i & \text{if } q = w_i, i = 1, ..., m' \\ 0 & \text{otherwise} \end{cases} \tag{44}$$

$$\sum_{(p,p+w_j)\in A} y_{p,p+w_j} \geq b_j \qquad j = 1, ..., n' \tag{45}$$

$$y_{pq} \geq 0 \text{ and integer} \qquad (p,q) \in A \tag{46}$$

$$0 \leq r_i \leq d_i \text{ and integer} \qquad i = 1, ..., m' \tag{47}$$

Objective (43) minimizes the number of used bins, whereas constraints (44) ensure flow conservation. Constraints (45) impose that at least b_j units of flow goes through the arcs of each item j. Graph G can be built using dynamic programming (details can be found in Côté and Iori (2018)). The basic idea is to start from a graph empty of arcs and to consider each item j iteratively in non-increasing order of weight. An arc $(p, p + w_j)$ can be placed in p only if $p = 0$ or if there exists a path from 0 to p that does not go through any arcs of item j. Arcs are added progressively until all items have been looked at.

We also consider the *Reflect* formulation proposed by Delorme and Iori (2020) and used by Dell'Amico et al. (2019) for the MKP. It is another type of arc-flow formulation using a different graph representation that only uses half the number of nodes. In many instances, Reflect is able to reduce significantly the number of arcs, but on many hard instances, however, this number can also be several times bigger than that of the classical arc-flow. Our method chooses the graph representation (arc-flow or Reflect) that uses the smallest amount of arcs.

We also propose the following improvement to remove more arcs. At the time of placing arc $(p, p + w_j)$ for item j, it is possible to calculate the maximal bin usage that this arc will lead to. If the maximal bin usage leads to an empty space of at least f units in any bin, then, the arc can be discarded. Let f' be the minimal empty space that can be generated using the remaining items if arc $(p, p + w_j)$ is taken. It is defined as follows :

$$f' = \max_{i=1,...,m'} \left\{ c_i - p - w_j - \left\{ \max \sum_{l=j+1}^{n'} \bar{z}_l w_l \middle| \sum_{l=j+1}^{n'} \bar{z}_l w_l \leq c_i - p - w_j, \bar{z}_l \in \{0, ..., b_l\}, l = j+1, ..., n' \right\} \right\} \tag{48}$$

If $f' > f$, then arc $(p, p+w_j)$ can be discarded. We calculate f' by dynamic programming while building the graph. Experiments have shown that this procedure can remove several thousands of arcs on hard instances. Both arc-flow and Reflect models benefit from this preprocessing.

5 Computational experiments

All algorithms were coded in C++ and compiled using gcc 4.8.5 with -O3. The detailed results and our codes can be found at `https://sites.google.com/view/jfcote/`. We ran our tests on a machine running Linux Oracle Server 7.7 with an Intel i7-6700x CPU at 3.50 GHz and 125 GB of RAM. The single knapsack problems were solved using combo of Martello et al. (1999), and the linear programming master problems as well as the set packing and arc-flow models were solved using CPLEX 12.10. We also used MULKNAP of Pisinger (1999), suitably adapted to halt and yield the current incumbent solution if the time elapsed exceeds the specified time limit.

Benchmark instances

We performed our tests on the benchmark instances proposed in Dell'Amico et al. (2019), which can be obtained at `http://or.dei.unibo.it/library`. These comprise of five instance sets, namely SMALL, FK_1, FK_2, FK_3 and FK_4. SMALL was adapted from an instance set suggested by Kataoka and Yamada (2014) for the closely related multiple knapsack assignment problem, while the FK_i were generated using a classical procedure for generating benchmark instances for the MKP. An instance in a given set is characterized by two parameters, namely:

1. The dimension of the instance (in the form of the values of n and m).
2. The so-called correlation class of the profits p_j.

10 different instances were generated for every choice of parameters for SMALL and 20 were for each FK_i. In each of the FK_i, there is exactly one choice of (n, m) corresponding to each of the $\frac{n}{m}$ ratios $\{2, 3, 4, 5, 6, 10\}$. We will refer to the set of all instances from a given instance set that were generated according to a given choice of parameters as a subset. The possible values of said parameters are given in Tables 1 and 2. Since there are 6 possible choices of n, m and 3 different correlation classes for SMALL, it contains $3 \times 6 = 18$ instance subsets, for a total of $18 \times 10 = 180$ instances, and since there are 6 possible choices of n, m and 4 different correlation classes for each FK_i, each contains $4 \times 6 = 24$ instance subsets, for a total of $24 \times 20 = 480$ instances.

For both sets, the item weights w_j were generated before generating the profits and the knapsack capacities, and were uniformly generated in an interval of the form $[\alpha, 1000]$, where α is a set-dependent parameter, equal to 1 for SMALL and to 10 for the FK_i. The correlation class of a given instance controls the way in which the components of the pairs (p_j, w_j) for $j = 1, ..., n$ are related to each other, ranging from being independent to being the same. The possibilities are:

1. Uncorrelated: The p_j are uniformly distributed in $[\alpha, 1000]$.
2. Weakly correlated: The p_j are uniformly distributed in $[0.6w_j + 1, 0.6w_j + 400]$ for SMALL and in $[\max(1, w_j - 100), w_j + 100]$ for the FK_i.
3. Strongly correlated: The p_j are set to $w_j + 200$ for SMALL and to $w_j + 10$ for the FK_i.

4. Subset-sum: (only for the FK_i) $p_j = w_j$.

Set $W = \sum_{j=1}^{n} w_j$. For SMALL, the knapsack capacities c_i were generated dissimilarly, according to the rule $c_i = \lfloor 0.5\lambda_i W \rfloor$, where λ was uniformly generated with the constraint $\sum_{i=1}^{m} \lambda_i = 1$ and $\lambda_i \geq 0$. For the FK_i, the knapsack capacities c_i were generated similarly, with c_i being uniformly generated in $[\frac{0.4W}{m}, \frac{0.6W}{m}]$ for $1 \leq i \leq m-1$ and c_m being set to $0.5W - \sum_{i=1}^{m-1} c_i$. All instances with $w_j > \min_i c_i$ for all j's (so that some knapsacks are redundant), $\max_j w_j > c_i$ for all i's (so that some items are redundant) or $W \leq \max_i c_i$ (so that all items may be packed in a single knapsack, rendering the problem trivial) were rejected and generated again.

Table 1: Values of n/m used for generating the instance sets

Set	n/m
SMALL	(20,10),(40/10),(60,10),(20,20),(40,20),(60,20)
FK_1	(60,30),(45,15),(48,12),(75,15),(60,10),(100,10)
FK_2	(120,60),(90,30),(96,24),(150,30),(120,20),(200,20)
FK_3	(180,90),(135,45),(144,36),(225,45),(180,30),(300,30)
FK_4	(300,150),(225,75),(240,60),(375,75),(300,50),(500,50)

Table 2: Characteristics of SMALL and the FK_i

Set	Instances per group	Correlation classes	α	Capacities	Number of Instances
SMALL	10	Uncorrelated, weakly, strongly	1	Dissimilar	180
FK_i	20	Uncorrelated, weakly, strongly, subset-sum	10	Similar	480

5.1 Empirical results regarding various relaxations

Experiments were performed on the benchmark instances $SMALL$ and FK_1 to compare the performance of the various existing relaxations for the MKP with the new relaxation. No preprocessing of any kind was performed on the instances. Both Lagrangian relaxations were solved using column generation. When running the initial subgradient algorithm, the initial choice of multipliers for computing L_1 that was used is the one that is described in Hung and Fisk (1979): let the items be ordered in decreasing order of density $\frac{p_j}{w_j}$, and let $l \in \{1, ..., n\}$ be the break item of the continuous relaxation, i.e. the smallest l such that $\sum_{j=1}^{l} w_j > \sum_{i=1}^{m} c_i$. Then, we set $\lambda_j = p_j - \frac{w_j p_l}{w_l}$ if $j < l$ and 0 otherwise. The initial choice of multipliers chosen in the case of L_2 was simply $\mu_j = 0$.

Table 3: Characteristics of various relaxations for SMALL (180)

Relaxation	Value	Time	Gap	Max gap	Instances closed
Linear relaxation	16734.1	0.0 (0.0)	18.3	154.1	0
Surrogate relaxation (SMKP)	16625.1	0.0	17.0	147.3	32
Arc-flow	15829	2.8	0.3	3.6	64
Reflect model + Priority + $n_{\max}$	15790.1	0.7	0.2	2.2	75
Lagrangian relaxation (L_1)	15828.3	0.0 (0.0)	0.3	3.6	65
Lagrangian relaxation (L_2)	**15787.1**	0.0 (0.1)	0.2	3.6	**106**
Average optimal value	15765.2				

Table 4: Characteristics of various relaxations for FK_1 (480)

Relaxation	Value	Time	Gap	Max gap	Instances closed
Linear relaxation	19149.9	0.0	5.6	69.7	77
Surrogate relaxation (SMKP)	19126.1	0.0	5.4	69.5	264
Arc-flow	18420.8	6.1	0.1	0.9	155
Reflect model + Priority + $n_{\max}$	18420.1	1.7	0.1	0.9	220
Lagrangian relaxation (L_1)	18420.6	0.0 (0.1)	0.1	0.9	156
Lagrangian relaxation (L_2)	**18404.9**	0.0 (0.1)	0.0	0.9	**347**
Average optimal value	18400.8				

Tables 3 and 4 report some statistics concerning various possible relaxations for the MKP. The surrogate relaxation is the relaxation described in Section 2.1, and was solved using combo. "Arc-flow" and "Reflect model + Priority + $n_{\max}$" correspond to the linear relaxations of arc-flow models for the MKP, as described in Dell'Amico et al. (2019), which is where these figures come from, with "Arc-flow" corresponding to a basic model and "Reflect model + Priority + $n_{\max}$" corresponding to a more sophisticated model where some supplementary constraints were added, and is the tightest arc-flow based relaxation reported. Column "Value" corresponds to the average upper bound, to be compared with the average solution, which is provided under the table; column "Time" reports the average time taken to compute the relaxation; column "Gap" and "Max gap" respectively report the average gap and the maximum gap, which is computed as $100\frac{\text{UB}-z^*}{z^*}$, whereby UB stands for the upper bound provided by the relaxation and z^* corresponds to the optimal solution of the problem; and "Instances closed" reports the total number of instances for which UB was equal to z^*.

We see that the new Lagrangian relaxation L_2 is clearly the strongest relaxation, providing tighter bounds on average than all other methods and providing bounds that are greatly tighter than those provided by the Lagrangian relaxation L_1. It could also be computed reasonably quickly. These results clearly demonstrate that the new relaxation L_2 has independent interest, as it could for example be used in a traditional branch-and-bound algorithm to prune large parts of the search tree, or be used within a variable fixing scheme such as the one presented in Section 3.7 to reduce the size of the problem without too much computational effort, provided that a good feasible solution is known.

It is worth pointing out that there appears to be a connection between the Lagrangian relaxation

L_1 and the linear relaxations of various arc-flow models for the MKP, as there is in the case of the bin-packing problem, where these are equal(Valério de Carvalho, 1999). For all 180 SMALL instances, it always held that $z_{L_1} \leq z_{\text{Arc-flow}}$, with the difference being smaller than 1 for 162 problems. Interestingly, it turns out that the Lagrangian relaxation L_1 is very tight for instances with a small $\frac{n}{m}$ ratio and generally closes instances with $\frac{n}{m} = 2$, but gives increasingly weaker bounds as $\frac{n}{m}$ grows, while the opposite holds for the surrogate relaxation SMKP, which provides very weak bounds for instances with a small $\frac{n}{m}$ ratio but gets progressively tighter as $\frac{n}{m}$ increases, generally closing instances with $\frac{n}{m} \geq 6$. This might give a theoretical explanation to the remark in Dell'Amico et al. (2019) that the Reflect-based decomposition, which uses an arc-flow based MIP and thus may behave similarly to the Lagrangian relaxation L_1, works best for instances with an $\frac{n}{m}$ ratio of 3 or 4, while the knapsack-based decomposition, which uses a MIP based on the surrogate relaxation, works bests for instances with an $\frac{n}{m}$ ratio of 4, 5 or 6.

5.2 Empirical results regarding the VSBPP-SAT solver

Table 5 compares the performance of the VSBPP-SAT solver described in Section 4 and that of the solver suggested by Dell'Amico et al. (2019) on the subproblems encountered by BP-MKP. The strategy used by Dell'Amico et al. (2019), which we will refer to as CP + Reflect, goes as follows: first, a constraint programming approach was tried for one second (using CPLEX's IloPack constraint). If this failed to find a feasible packing and to prove that none existed, their original Reflect's code was ran. To compare the performance of the two solvers, while running BP-MKP on the test instances, we stored the VSBPP-SAT subproblems encountered in memory, along with the time used and the result. We then ran the CP + Reflect procedure on the subproblems with a time limit of 10 CPU seconds for problems that were encountered during preprocessing and 1200 CPU seconds otherwise. Problems with fewer than 3 knapsacks or with fewer than 5 items were discarded from the statistics. Also, as problems encountered during instance reduction tend to be much easier than those encountered during the course of the main algorithm, they are excluded from the set-specific columns and are reported in a separate column.

The first four entries in the "Algorithm" columns correspond to the various algorithms of the solver, as described in Section 4. The last entry corresponds to the performance of the overall solver. The number in parentheses next to the name of a set/phase corresponds to the total number of VSBPP-SAT problems that were encountered when solving problems from the set/phase. Each entry of the "Opt" subcolumns report both the number of problems that the corresponding algorithm succeeded in solving and the number of problems that the algorithm was run on, and each entry of the "Time" subcolumns report the average CPU time the algorithm took on the problems it was run on, in seconds. In the case of "Arc-flow" and "Reflect', the second number corresponds to the number of problems on which the method was ran. We see that the new solver was very effective overall, solving all problems encountered when solving instances in SMALL, FK_1, FK_2 and FK_3 and solving most problems coming from instances in FK_4 and most problems encountered during preprocessing (for which it had a time limit of 10 seconds). The heuristic did not perform

particularily well on problems in the first five categories, but it solved more than half of the preprocessing problems extremely quickly. We also see that the set packing's performance was more than satisfactory, solving more than half of the instances in the first five categories rather quickly, while the majority of the remaining instances were solved by the arc-flow and Reflect. It is also interesting to see that, for large-sized problems, it was almost always found that arc-flow produced a smaller model than Reflect, with this being the case for all 78 problems from FK_4 that were not solved by the heuristic or by the Set Packing.

Although our VSBPP-SAT solver did much better than the solver suggested in Dell'Amico et al. (2019), being more than three times faster than it on all instance sets and solving more instances, we nevertheless see that it had some difficulty with solving the subproblems coming from the larger instances, as it took 120 seconds on average in the case of FK_4, which corresponds to 10% of the time limit and 21.9% of the adjusted time limit (which is 550 seconds). This implies that very few VSBPP-SAT subproblems could usually be solved within the time limit.

Table 5: Respective performances of the two VSBPP-SAT solvers

Algorithm	SMALL (70)		FK_1 (216)		FK_2 (371)		FK_3 (310)		FK_4 (239)		Preprocessing (458)	
	Opt	Time	Opt	Time	Opt	Time	Opt	Time	Opt	Time	Opt	time
CP + Reflect	70	5.7	216	3.9	371	19.2	307	86.4	198	423.0	304	8.1
New Solver	70	1.4	216	0.3	371	5.0	310	18.8	226	120.0	431	1.3
-Heuristic	1	0.0	3	0.0	0	0.0	0	0.1	1	0.5	253	0.1
-Set packing	46	0.7	173	0.1	219	1.2	204	9.8	156	53.5	74	2.4
-Arc-flow	5/5	1.3	8/8	2.3	94/94	13.3	94/94	29.1	69/78	202.5	99/102	0.7
-Reflect	18/18	2.4	32/32	0.5	58/58	2.4	12/12	2.3	0/0	0.0	5/5	0.1

5.3 Comparison of solution methods

Our algorithm was given a time limit of 1200 CPU seconds on every test instance. For a given instance and for a given method, we define the gap as be given by the formula $100\frac{\text{LB}^*-\text{LB}}{\text{LB}^*}$, whereby LB^* stands for the best known solution and LB stands for the best found solution by the method within the time limit. We compare our results to two other exact methods for solving the MKP: Pisinger's MULKNAP algorithm as well as Dell'Amico et al. (2019)'s best-performing algorithm, namely Hy-MKP. The results given for MULKNAP and Hy-MKP in Tables 6, 8, 9 and 10 come from the computational experiments of Dell'Amico et al. (2019), whose authors graciously accepted to share their detailed results with us. It is worth mentioning that the average times reported for MULKNAP in the Table 4 of Dell'Amico et al. (2019) are erroneous, and the results given here in Table 6 are the correct values.

To account for the difference in computing power between our machine and that of Dell'Amico et al. (2019), which cpubenchmark.com estimates to be a ratio of $\frac{2489}{1142} \approx 2.18$ in processing speed, whenever comparing our results with theirs, we specify in parentheses what every important result would have been if our tests had been run on a machine of comparable processing speed and with the same time limit as them, namely 1200 CPU seconds: in this respect, the adjusted number of solved instances given corresponds to the number of instances that were solved in less than $\frac{1200}{2.18} \approx 550$ CPU

Table 6: Overall results of various exact methods on the benchmark instances

	SMALL (180)		FK_1 (480)		FK_2 (480)	
Method	Opt	Time	Opt	Time	Opt	Time
MULKNAP	150	230.7	353	378.5	290	482.1
Hy-MKP	**180**	11.5	**480**	10.3	469	91.9
BP-MKP	**180 (180)**	3.3 (7.3)	**480 (480)**	1.8 (3.9)	**478 (473)**	29.3 (53.5)
	FK_3 (480)		FK_4 (480)		All (2100)	
Method	Opt	Time	Opt	Time	Opt	Time
MULKNAP	311	427.6	313	421.2	1417	410.6
Hy-MKP	461	146.3	398	286.9	1988	123.4
BP-MKP	**477 (470)**	36.1 (60.5)	**452 (443)**	98.9 (122.0)	**2067 (2046)**	38.2 (55.5)

seconds, and the adjusted average CPU time given corresponds to the average of $2.18\min(550, t)$, where t stands for the actual CPU time that was spent by BP-MKP on a given instance.

Table 6 showcases the overall results on every instance set and on all benchmark instances of the exact algorithms under scrutiny. For every instance set, the number in parentheses next to the name of the set corresponds to the number of instances in the set, while column "Opt" corresponds to the number of instances that were solved to proven optimality by the method and column "Time" corresponds to the average CPU time that it spent on every instance, in seconds. We see that even when taking the difference in computing power into account, BP-MKP significantly outperforms the best algorithm of Dell'Amico et al. (2019), being more than twice as fast on average on hard instances and solving to proven optimality 58 more instances than it.

Table 7: Details on instances that were not solved by the preprocessing

	SMALL (180)	FK_1 (480)	FK_2 (480)	FK_3 (480)	FK_4 (480)
Instances	101	291	280	251	247
Opt	101	291	278	248	219
Time	5.9	3.0	50.2	69.0	192.2
VSBPP-SAT time	1.0	0.2	6.9	23.2	116.1
VSBPP-SAT calls	0.7	0.8	1.5	1.2	1.0
Relaxation time / node	0.0	0.0	0.6	1.1	4.6
Nodes	66.4	35.2	167.9	77.5	42.4
Patterns	21,117.3	12,045.6	77,263.8	118,848.1	100,761.1
Average gap	0.00	0.00	0.00	0.00	0.06
Max gap	0.00	0.00	0.08	0.10	1.82

Table 7 provides more detailed information on BP-MKP's performance on the instances sets. Instances that were solved by the MULKNAP phase or during preprocessing were discarded from these statistics, as both of these strategies are also used by Hy-MKP. For every instance set, the number next to the name of the instance corresponds to the total number instances, and row "Instances" corresponds to the total number of instances that were not solved by MULKNAP or during preprocessing. Once again, row "Opt" corresponds to the number of instances that were solved to proven optimality, and row "Time" corresponds to the average CPU time spent, in seconds. Row "VSBPP-SAT time" corresponds to the average CPU time spent on solving VSBPP-SAT problems in seconds, "VSBPP-SAT calls" corresponds to the average number of calls to the VSBPP-SAT solver (both excluding preprocessing), row "Relaxation time / node" reports the average of the average time spent running column generation per node for every instance, with the time spent on the VSBPP-SAT solver excluded, row "nodes" reports the average number of nodes processed

Table 8: Detailed results on the FK_2 benchmark instances

Instances		MULKNAP			Hy-MKP				BP-MKP						
n/m	Type	Opt	Gap	Time	Opt	Gap	Time	Iter	Opt	Gap	Time	Nodes	Root	%Pack	Pack
120/60	Uncorrelated	20	-	3.1	20	-	3.0	0.0	20 (20)	-	1.4 (3.2)	1.0	20	0.0	0.0
	Weakly	20	-	3.2	20	-	3.2	0.0	20 (20)	-	1.2 (2.7)	1.0	20	0.0	0.0
	Strongly	20	-	6.5	20	-	3.2	0.6	20 (20)	-	1.6 (3.5)	1.0	20	0.0	0.0
	Subset-sum	20	-	3.6	20	-	3.2	0.6	20 (20)	-	1.6 (3.5)	1.0	20	0.0	0.0
90/30	Uncorrelated	0	1.96	t.l.	20	-	24.9	11.2	20 (20)	-	13.8 (30.1)	14.5	5	22.3	1.5
	Weakly	0	1.68	t.l.	20	-	27.7	18.8	20 (20)	-	37.8 (82.5)	191.8	1	11.2	1.5
	Strongly	0	0.92	t.l.	16	0.00	684.8	31.0	20 (18)	-	187.8 (385.8)	590.6	0	26.1	6.1
	Subset-sum	0	0.37	t.l.	13	0.00	781.7	23.4	18 (15)	0.00	375.9 (595.4)	1177.9	0	8.3	3.7
96/24	Uncorrelated	0	1.76	t.l.	20	-	26.9	1.2	20 (20)	-	2.7 (5.9)	1.9	17	15.6	1.2
	Weakly	0	1.76	t.l.	20	-	119.4	16.9	20 (20)	-	30.3 (66.1)	117.1	1	9.8	2.8
	Strongly	1	0.40	1144.5	20	-	46.8	1.0	20 (20)	-	2.7 (5.9)	3.1	12	6.6	1.0
	Subset-sum	19	0.00	148.2	20	-	115.1	10.0	20 (20)	-	31.5 (68.6)	208.9	9	0.2	0.6
150/30	Uncorrelated	7	0.25	784.4	20	-	126.2	0.7	20 (20)	-	7.5 (16.4)	0.7	20	48.0	0.7
	Weakly	0	0.58	t.l.	20	-	151.1	1.1	20 (20)	-	4.9 (10.7)	1.2	18	12.3	1.1
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
120/20	Uncorrelated	19	0.00	60.3	20	-	6.1	0.1	20 (20)	-	0.2 (0.4)	0.05	20	3.0	0.1
	Weakly	4	0.12	970.1	20	-	82.2	0.9	20 (20)	-	1.5 (3.3)	0.8	20	16.5	0.9
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
200/20	Uncorrelated	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Weakly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0

per instance, row "Patterns" reports the average number of patterns added to a master model per instance, and rows "Average gap" and "max gap" respectively report the average and max gap over all instances, including the ones that were solved to optimality. The values given for "Nodes" explain why BP-MKP exhibits a much better performance on hard instances than all previous branch-and-bound algorithms for the MKP, such as Pisinger's MULKNAP and Fukunaga's 2D/PS+B: while BP-MKP requires much more computational effort per node than either of these algorithms, the strength of the relaxation L_2 as well as the fact that the search space grows much more slowly with respect to instance size (as there are only n variables to branch on in total as opposed to nm) make it so that BP-MKP has to process very few nodes, no more than a few hundred on average, and the number of nodes processed does not grow out of hand as the problems increase in size. In fact, interestingly, the node count was actually smaller for larger problems, on average. We also see that the reported values for "VSBPP-SAT calls" are always very small, never exceeding two, which shows that when the t_j^* turn out to be integer, they very often correspond to the subset of items that are included in the optimal solution of the problem. Also, 79% of the packing problems encountered (excluding those encountered during preprocessing) could be proven to be feasible by our algorithm.

Tables 8, 9 and 10 report the performance of all three considered methods on each specific subset (i.e. choice of (n,m) and choice of correlation class) for FK_2, FK_3 and FK_4, respectively. We refrained from presenting the corresponding tables for SMALL and FK_1 as all instances from both sets were solved rather easily to proven optimality by both Hy-MKP and BP-MKP and are thus not very interesting. For every method, a dash in an entry of the "Gap" subcolumn indicates that all 20 problems were solved to proven optimality by the method, with the exception of the

Table 9: Detailed results on the FK_3 benchmark instances

Instances		MULKNAP			Hy-MKP				BP-MKP						
n/m	Type	Opt	Gap	Time	Opt	Gap	Time	Iter	Opt	Gap	Time	Nodes	Root	%Pack	Pack
180/90	Uncorrelated	20	-	6.3	20	-	6.4	0.0	20 (20)	-	2.5 (5.4)	1.0	20	0.0	0.0
	Weakly	20	-	9.2	20	-	6.4	0.6	20 (20)	-	2.8 (6.1)	1.0	20	0.0	0.0
	Strongly	20	-	6.3	20	-	6.6	0.6	20 (20)	-	2.7 (5.8)	1.0	20	0.0	0.0
	Subset-sum	20	-	6.2	20	-	6.6	0.6	20 (20)	-	2.7 (5.8)	1.0	20	0.0	0.0
135/45	Uncorrelated	0	N/A	t.l.	20	-	154.6	12.2	20 (19)	-	70.6 (150.2)	18.3	7	23.4	1.9
	Weakly	0	N/A	t.l.	20	-	170.0	20.6	17 (12)	0.01	520.2 (755.9)	728.1	0	16.7	4.0
	Strongly	0	N/A	t.l.	16	20.0	457.5	3.4	20 (20)	-	14.6 (31.9)	4.0	10	28.4	1.1
	Subset-sum	0	N/A	t.l.	12	0.00	928.8	21.3	20 (20)	-	6.0 (13.0)	14.9	0	17.0	1.0
144/36	Uncorrelated	0	N/A	t.l.	20	-	106.6	1.0	20 (20)	-	2.5 (5.5)	1.0	20	26.5	1.0
	Weakly	0	N/A	t.l.	17	4.89	622.6	12.0	20 (19)	-	184.4 (350.5)	163.7	2	32.9	4.1
	Strongly	4	N/A	994.1	20	-	279.6	0.9	20 (20)	-	3.1 (6.8)	1.5	15	16.5	0.9
	Subset-sum	20	-	4.0	19	0.00	60.1	1.1	20 (20)	-	7.7 (16.8)	35.9	19	0.0	0.1
225/45	Uncorrelated	16	N/A	240.8	20	-	86.0	0.2	20 (20)	-	32.9 (71.8)	0.2	20	16.5	0.2
	Weakly	0	N/A	t.l.	19	2.78	359.8	1.1	20 (20)	-	10.0 (21.8)	1.1	19	25.1	1.0
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
180/30	Uncorrelated	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Weakly	11	N/A	543.4	18	5.46	254.0	0.6	20 (20)	-	2.6 (5.6)	0.7	18	25.3	0.5
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
300/30	Uncorrelated	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Weakly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0

Table 10: Detailed results on the FK_4 benchmark instances

Instances		MULKNAP			Hy-MKP				BP-MKP						
n/m	Type	Opt	Gap	Time	Opt	Gap	Time	Iter	Opt	Gap	Time	Nodes	Root	%Pack	Pack
300/150	Uncorrelated	20	-	11.9	20	-	11.7	0.0	20 (20)	-	6.4 (14.0)	1.0	20	0.0	0.0
	Weakly	20	-	11.8	20	-	11.7	0.0	20 (20)	-	11.4 (24.8)	1.0	20	0.0	0.0
	Strongly	20	-	12.7	20	-	11.9	0.6	20 (20)	-	9.3 (20.2)	1.0	20	0.0	0.0
	Subset-sum	20	-	11.4	20	-	11.9	0.6	20 (20)	-	9.3 (20.3)	1.0	20	0.0	0.0
225/75	Uncorrelated	0	1.87	t.l.	13	15.00	847.6	14.7	17 (14)	0.24	358.0 (487.4)	16.2	14	68.9	1.8
	Weakly	0	1.35	t.l.	7	0.03	1074.6	20.4	5 (1)	0.33	1102.2 (1186.0)	484.3	0	27.9	3.9
	Strongly	0	0.51	t.l.	8	60.00	873.0	6.2	20 (20)	-	26.3 (57.4)	2.0	13	59.6	1.0
	Subset-sum	0	0.23	t.l.	0	0.05	t.l.	17.8	20 (20)	-	32.9 (71.7)	9.7	0	40.3	1.0
240/60	Uncorrelated	0	1.06	t.l.	17	14.55	429.1	1.2	20 (20)	-	12.7 (27.7)	1.0	20	61.0	1.0
	Weakly	0	1.39	t.l.	8	39.49	907.0	2.0	18 (17)	0.11	187.5 (242.3)	2.2	17	51.8	1.1
	Strongly	3	0.36	1022.4	16	20.00	507.1	1.5	18 (18)	0.01	133.2 (148.5)	2.4	13	52.8	0.9
	Subset-sum	20	-	0.0	20	-	0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
375/75	Uncorrelated	18	0.01	120.2	19	5.00	114.5	0.2	18 (18)	0.00	120.3 (120.0)	0.1	20	9.8	0.1
	Weakly	0	0.21	t.l.	11	39.20	663.3	3.1	16 (16)	0.00	283.4 (333.5)	1.1	19	69.1	1.0
	Strongly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
300/50	Uncorrelated	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Weakly	12	0.01	483.0	19	3.33	220.7	0.5	20 (19)	-	80.5 (175.2)	0.4	20	33.6	0.4
	Strongly	20	-	0.2	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
500/50	Uncorrelated	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Weakly	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Strongly	20	-	1.2	20	-	1.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0
	Subset-sum	20	-	0.0	20	-	0.0	0.0	20 (20)	-	0.0 (0.0)	0.0	20	0.0	0.0

entries for MULKNAP in Table 9, as the lower bounds found by MULKNAP for problems in FK_3 were not conserved by Dell'Amico et al. (2019) and are thus unknown. In the "Time" subcolumn, which, as before, reports the average CPU time spent by the method on the instances of the given group in seconds, "t.l.', indicates that the method exceeded the time limit on all 20 instances. The "Iter" subcolumn of Hy-MKP reports the average total number of iterations of either decomposition performed by Hy-MKP, which corresponds to the number of VSBPP-SAT subproblems solved or one less than that number if a timeout occurred while solving the decomposition MIP. The "Nodes" subcolumn of BP-MKP reports the average number of nodes processed by BP-MKP while solving instances from the given group, the "Root" subcolumn corresponds to the total number of instances that were solved at the root node, and the "%Pack" subcolumn contains the average proportion of the time spent in the VSBPP-SAT solver: if, for a given group instance i, where $i \in \{1, ..., 20\}$, BP-MKP spent t_i CPU seconds in total on the instance and spent v_i CPU seconds in the VSBPP-SAT solver, then the value reported is $\frac{100}{20} \sum_{i=1}^{20} \frac{v_i}{t_i}$. The "Pack" column reports the average number of calls to the VSBPP-SAT solver.

We can make the following observations:

1. As noted in Dell'Amico et al. (2019), all algorithms are very efficient at solving instances with an $\frac{n}{m}$ ratio equal to 2 or 10 (or more generally greater or equal to 10). In the case of $\frac{n}{m} = 2$, this is due to the fact that the instance reduction procedure described in Section 3.8 almost always succeeds in packing nearly all items, and in the case of $\frac{n}{m} \geq 10$, this is due to the fact that the MULKNAP component solves these problems very quickly thanks to its splitting heuristic, which, as mentioned in Section 2.1, almost always succeeds in inserting the items chosen in the optimal solution of the surrogate relaxation of the problem at the root node, which generally takes very little time.

2. It would seem that the only instances that require much work to identify the subset of items that is part of the optimal solution are the ones with an $\frac{n}{m}$ ratio of 3 or 4, as both Hy-MKP and BP-MKP used very few iterations/nodes on most instances with an $\frac{n}{m}$ ratio on 5 or 6 in all instance sets (and BP-MKP used a considerable proportion of its time in the VSBPP-SAT solver in the case of the ones that it could not solve), so that solving such problems may be considered more or less as hard as solving VSBPP-SAT problems. This also seems to hold for very large instances with an $\frac{n}{m}$ ratio of 4, so that the only very large instances that require much searching may be those with an $\frac{n}{m}$ ratio close to 3. In this respect, it should be noted that the improved VSBPP-SAT solver made it possible to solve many problems with a moderately large $\frac{n}{m}$ ratio very quickly. For example, all 20 weakly correlated problems with $n = 180$ and $m = 30$ in FK_3 could be solved by BP-MKP in 5.6 adjusted CPU seconds on average while Hy-MKP could only solve 18 and took 254 CPU seconds on average.

3. More generally, we see that the proportion of the total time spent by BP-MKP that was spent in the VSBPP-SAT solver steadily increases as the instances get larger for all choices of correlation class and $\frac{n}{m}$ ratio, ranging from around 15% on average in the case of FK_2 to

around 50% on average in the case of FK_4, while the average node count and the number of instances that required any searching, on the contrary, both steadily decrease. This outlines the importance of having an efficient algorithm for solving the packing subproblems, as these appear to be a major roadblock to solving many very large instances for both Hy-MKP and BP-MKP.

4. The efficiency of BP-MKP does not seem to reside solely in the superior performance of our VSBPP-SAT solver, but also in the fact that it requires solving far fewer VSBPP-SAT subproblems than Hy-MKP, which, as noted in the preceding section, appear to be extremely difficult to solve in the case of large instances. This difference is most pronounced in the case of strongly correlated and subset-sum instances in FK_3 and FK_4: while Hy-MKP had considerable difficulty on these instances, solving only 36 out of 80, BP-MKP could solve all 80 instances in 43.5 normalized CPU seconds on average, and in all 80 instances but two, it needed to solve only one packing subproblem, while Hy-MKP performed 12.2 iterations of its decomposition algorithms on average. Also, BP-MKP's node count was very small on such instances, being 7.7 on average, so that very little search was usually required. In general, Hy-MKP relies exclusively on solving successive packing subproblems to explore the search space, whereas BP-MKP relies mainly on enumeration, which required solving far fewer packing subproblems in all cases. This seems preferable given that, in the case of FK_4, solving a packing problem took 26 times more computing time than processing a node did, on average (though it can be assumed that most of the packing problems encountered by Hy-MKP were easily proven to be infeasible, considering how many iterations of its decomposition algorithms it could perform). We may expect this difference to be even more drastic in the case of even larger instances. However, Hy-MKP performed comparably on large uncorrelated instances with an $\frac{n}{m}$ ratio of 3, and slightly better on large weakly correlated instances with an $\frac{n}{m}$ ratio of 3, where BP-MKP too had to solve many VSBPP-SAT problems.

6 Conclusion

In this paper, a branch-and-price algorithm for the Multiple Knapsack Problem was presented. This algorithm is based on a new Lagrangian relaxation based on a reformulation of the problem which dominates all known upper-bounding techniques. It works by controlling whether an item is included in the solution or not, thereby reducing greatly the search space of the algorithm. Computational experiments have shown that our algorithm shows a better performance than the previous state-of-the-art algorithm for this problem on benchmark instances.

7 Acknowledgments

Financial support for this work was provided by the Canadian Natural Sciences and Engineering Research Council (NSERC) under grants 2017-06054 and 2021-04037. This support is gratefully

acknowledged. We also thank CIRRELT for providing access to their computing facilities.

References

Alves, C. and Valério de Carvalho, J. M. (2008). A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research*, 35:1315–1328.

Belov, G. and Scheithauer, G. (2002). A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths. *European Journal of Operational Research*, 141(2):274–294.

Côté, J.-F. and Iori, M. (2018). The meet-in-the-middle principle for cutting and packing problems. *INFORMS Journal on Computing*, 30(4):646–661.

Dell'Amico, M., Delorme, M., Iori, M., and Martello, S. (2019). Mathematical models and decomposition methods for the multiple knapsack problem. *European Journal of Operational Research*, 274(3):886–899.

Delorme, M. and Iori, M. (2020). Enhanced pseudo-polynomial formulations for bin packing and cutting stock problems. *INFORMS Journal on Computing*, 32(1):101–119.

Fukunaga, A. S. (2011). A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, 184:97–119.

Fukunaga, A. S. and Korf, R. E. (2005). Bin-completion algorithms for multicontainer packing and covering problems. *Journal of Artificial Intelligence Research*, 28:393–429.

Görtz, S. and Klose, A. (2012). A simple but usually fast branch-and-bound algorithm for the capacitated facility location problem. *INFORMS Journal on Computing*, 24(4):597–610.

Hung, M. S. and Fisk, J. C. (1978). An algorithm for 0-1 multiple-knapsack problems. *Naval Research Logistics Quarterly*, 25(3):571–579.

Hung, M. S. and Fisk, J. C. (1979). A heuristic routine for solving large loading problems. *Naval Research Logistics Quarterly*, 26(4):643–650.

Ingargiola, G. and Korsh, J. F. (1975). An algorithm for the solution of 0-1 loading problems. *Operations Research*, 23(6):1110–1119.

Klose, A. and Görtz, S. (2007). A branch-and-price algorithm for the capacitated facility location problem. *European Journal of Operational Research*, 179(3):1109–1125.

Laaloui, Y. (2013). Improved swap heuristic for the multiple knapsack problem. In Rojas I., Joya G., G. J., editor, *Advances in Computational Intelligence*, volume 7902 of *Lecture Notes in Computer Science*, pages 547–555. Springer Berlin / Heidelberg.

Laaloui, Y. and M'Hallah, R. (2016). A binary multiple knapsack model for single machine scheduling with machine unavailability. *Computers & Operations Research*, 72:71–82.

Lalami, M. E., Elkihel, M., El Baz, D., and Boyer, V. (2012). A procedure-based heuristic for the 0-1 multiple knapsack problems. *International Journal of Operational Research*, 4(3):214–224.

Loti de Lima, V., Alves, C., Clautiaux, F., Iori, M., and Valério de Carvalho, J. M. (2022). Arc flow formulations based on dynamic programming: Theoretical foundations and applications. *European Journal of Operational Research*, 296(1):3–21.

Martello, S., Pisinger, D., and Toth, P. (1999). Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424.

Martello, S. and Toth, P. (1980). Solution of the zero-one multiple knapsack problem. *European Journal of Operational Research*, 4(4):276–283.

Martello, S. and Toth, P. (1981a). A bound and bound algorithm for the zero-one multiple knapsack problem. *Discrete Applied Mathematics*, 3(4):275–288.

Martello, S. and Toth, P. (1981b). Heuristic algorithms for the multiple knapsack problem. *Computing*, 27:93–112.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley.

Pisinger, D. (1999). An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114(3):528–541.

Valério de Carvalho, J. M. (1999). Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659.

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