

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Algorithmic approaches to the multiple knapsack assignment problem

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

Martello S., Monaci M. (2020). Algorithmic approaches to the multiple knapsack assignment problem. OMEGA, 90, 1-11 [10.1016/j.omega.2018.11.013].

Availability:

This version is available at: <https://hdl.handle.net/11585/899477> since: 2024-02-28

Published:

DOI: <http://doi.org/10.1016/j.omega.2018.11.013>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

Algorithmic Approaches to the Multiple Knapsack Assignment Problem

Silvano Martello, Michele Monaci

¹*DEI “Guglielmo Marconi”, Università di Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy*

{silvano.martello, michele.monaci}@unibo.it

Abstract

We consider a variant of the multiple knapsack problem in which some assignment-type side constraints have to be satisfied. The problem finds applications in logistics sectors related, e.g., to transportation and maritime shipping. We derive upper bounds from Lagrangian and surrogate relaxations of a mathematical model of the problem. We introduce a constructive heuristic and a metaheuristic refinement. We study the computational complexity of the proposed methods and evaluate their practical performance through extensive computational experiments on benchmarks from the literature and on new sets of randomly generated instances.

Key words: multiple knapsack problem, assignment problem, relaxations, heuristic algorithms, computational experiments.

1. Introduction

We consider a combinatorial optimization problem that can be seen as a variant of the multiple knapsack problem in which some assignment-type side constraints have to be satisfied.

The *0-1 Multiple Knapsack Problem* (MKP) has been intensively studied in the combinatorial optimization literature. We are given n items, each having an associated *profit* p_j and *weight* w_j ($j = 1, \dots, n$), and m knapsacks, each having a *capacity* c_i ($i = 1, \dots, m$). The problem is to select m disjoint subsets of items such that the total profit of the selected items is a maximum, and each subset can be packed into a different knapsack having a capacity at least equal to the total weight of the items in the subset. The MKP has been shown to be strongly \mathcal{NP} -hard by transformation from 3-partition (see, e.g., Martello and Toth [18], Section 1.3). For extensive reviews on properties and algorithms for the MKP and its variants, we refer the reader to the corresponding chapters in the books by Martello and Toth [18] and Kellerer, Pferschy, and Pisinger [12]. Additional contributions to the MKP, that appeared after the publication of [12], have been presented by Chekuri and Khanna [2], Fukunaga and Korf [9], Fukunaga [7, 8], Jansen [10], Yamada and Takeoka [20], Lalami, Elkihel, Baz, and Boyer [14], and Balbal, Laalaoui, and Benyettou [1]. When $m = 1$, this problem reduces to the famous (single) *Knapsack Problem* (KP).

The MKP variant considered in this paper has been recently introduced by Kataoka and Yamada [11]. In the *Multiple Knapsack Assignment Problem* (MKAP) the input still consists of m knapsacks and n items, but the items are partitioned into r subsets (*classes*) S_k ($k = 1, \dots, r$): an additional constraint imposes that a knapsack can only contain items

of the same class. The problem is thus to assign a set of knapsacks to each class S_k , in such a way that the sum of the solution values of the r resulting MKPs is a maximum. Note that it is possible that a knapsack is not assigned to any class (i.e., it is not used in the solution) or, conversely, that a class has no knapsack assigned (i.e., none of its items belongs to the solution). In [11] the authors proposed upper and lower bounds for the MKAP. The upper bound is based on a Lagrangian relaxation, that is shown to be equivalent to the continuous relaxation of a mathematical model of the problem, while the lower bound is obtained from a greedy heuristic combined with local search. The approach was computationally tested on benchmark sets of small and large randomly generated instances. More recently, Lalla-Ruiz and Voß [15] proposed a genetic algorithm for the MKAP and showed, through computational experiments, that it obtains a better approximation on a set of small difficult instances extracted from the benchmarks introduced in [11].

As observed in [11], the special case of the MKAP in which $r = 1$ coincides with the MKP: it follows that the MKAP is strongly \mathcal{NP} -hard.

This paper presents algorithms for the computation of upper and lower bounds for the MKAP. In the next section we discuss how real-life problems have been modeled as an MKAP or similar MKP variants. In Section 1.2 we examine a mathematical formulation of the MKAP. Section 2 presents preprocessing and reduction techniques. Section 3 deals with Lagrangian and surrogate relaxations. Heuristic algorithms are then introduced in Section 4. In Section 5 we show, through extensive computational experiments, that the proposed algorithms provide better or faster (or both) solutions than state-of-the-art heuristics for the MKAP. Conclusions follow in Section 6.

1.1 Real-world applications

Problem MKAP has managerial applications in transportation logistics. Indeed it models problems arising when m vehicles of given capacities c_i , located at the same depot, have to be used to serve r customers. Each customer has an associated set of goods, each having a profit and a weight: the problem is to decide which goods have to be transported by each vehicle so that each vehicle visits at most one customer and the total profit of the selected goods is maximized. Note that a customer can be fully or partially served, or not visited at all.

The MKAP is the core problem in the maritime shipping application recently addressed by Zhen, Wang, Wang, and Qu [21]. They considered a hinterland barge transport system in which barges are not self-propelled and need to be assigned to tugs. The specific problem, originated by an application at the Port of Shanghai, included additional side constraints coming from the connection of the river’s inland ports to the worldwide maritime transportation network. The authors developed a mixed-integer programming model that optimizes the assignment of barges to tugs and the time schedule of the tugs departures.

Special cases of the MKAP arising in emergency relocation problems have been investigated by Dimitrov, Solow, Szmerekovsky, and Guo [6]. They considered a problem arising, in an emergency situation, when people at different locations must be moved to a safe place. In such context, a group of persons at the same location defines a class, and all items (the individuals) are identical, i.e., they share the same profit and weight. The problem calls for maximizing the number of persons that are moved to a safe location, using a given set of

trucks (the knapsacks). The need for a quick intervention imposes that any truck can only pick up people at a single location. It is shown in [6] that the problem can be solved in polynomial time if all trucks are identical, while a heuristic is proposed for the case of an heterogeneous fleet of vehicles.

Other combinations of the MKP with assignment-type constraints have been addressed in the literature. Dawande, Kalagnanam, Keskinocak, Salman, and Ravi [4] considered a problem coming from applications in the steel industry: Given a set of orders, production planning requires to assign them to a given set of production units in such a way that the throughput is maximized and the compatibility of orders and production units is taken into account. In this case the profit and weight of each item (order) is the same. A generalization of the problem, in which profits and weights are different, was studied by Dahl and Foldnes [3] in the context of wireless telecommunication: Each of a given set of mobile phone users must be assigned to a base station covering her location so that the total communication flow is maximized and the capacity of no base station is exceeded. Production planning problems with machine failures have also been modeled as MKPs with additional assignment-like constraints (see, e.g., Laalaoui and M'Hallah [13] and Diaz, Handl, and Xu [5]).

1.2 Problem formulation

In the following we will assume that all item weights are positive. It follows that we can also assume, without loss of generality, that all input data are positive integers. In addition, it can be assumed that $n \geq m$ as otherwise the smallest $m - n$ knapsacks could be eliminated.

In order to formally define the problem at hand, we report in the following an *Integer Linear Programming* (ILP) model from the literature. Let $N = \{1, 2, \dots, n\}$ denote the set of items, $M = \{1, 2, \dots, m\}$ the set of knapsacks, and $K = \{1, 2, \dots, r\}$ the set of item classes. For each knapsack $i \in M$ and item $j \in N$, let x_{ij} be a binary variable taking the value one if and only if item j is inserted (packed) into knapsack i . Similarly, for each knapsack $i \in M$ and class $k \in K$, let y_{ik} be a binary variable taking the value one if and only if knapsack i is assigned to class k . The MKAP can then be formally described by the ILP (see [11])

$$\max \sum_{j \in N} p_j \sum_{i \in M} x_{ij} \tag{1}$$

$$\sum_{i \in M} x_{ij} \leq 1 \quad j \in N \tag{2}$$

$$\sum_{k \in K} y_{ik} \leq 1 \quad i \in M \tag{3}$$

$$\sum_{j \in S_k} w_j x_{ij} \leq c_i y_{ik} \quad i \in M, k \in K \tag{4}$$

$$x_{ij} \in \{0, 1\} \quad i \in M, j \in N \tag{5}$$

$$y_{ik} \in \{0, 1\} \quad i \in M, k \in K. \tag{6}$$

The objective function (1) maximizes the sum of the profits of the selected items. Constraints (2) impose that each item is packed into at most one knapsack, while constraints (3) impose that each knapsack is assigned to at most one class. For each knapsack i , the associated

constraints (4) impose that: (i) only items of the class assigned to knapsack i (if any) can be inserted into it, and (ii) the capacity of the knapsack is not exceeded. Model (1)-(6) has $mn + mr$ binary variables and $n + m + mr$ constraints, i.e., $O(mn)$ variables and constraints.

Observe that, if an optimal solution has some knapsack i with no class assigned, we can define an equivalent solution by assigning i to an arbitrarily chosen class. It follows that, in (3) we could impose equality instead of inequality.

Also note that, when $K = \{1\}$, i.e., $S_1 \equiv N$, constraints (3) become redundant and y_{i1} can be set to 1 for all $i \in M$: the resulting ILP simplifies then to the classical MKP model (see [18]). If in addition $m = 1$, constraints (2) become redundant too, and the model further simplifies to a KP.

2. Preprocessing and reduction procedures

Some immediate considerations, frequently used for reducing packing problems (see, e.g., [18]), can be adapted to the MKAP:

- remove all knapsacks $i \in M$ for which $c_i < \min_{j \in N} \{w_j\}$;
- remove all items $j \in N$ for which $w_j > \max_{i \in M} \{c_i\}$.

This simple reduction clearly takes linear time $O(n)$.

2.1 Linear time reduction

A stronger reduction can be obtained, with a small increase of the computational effort, by determining the contents of a subset of knapsacks in an optimal solution:

Procedure Reduction:

sort the elements of knapsack set M by increasing capacity, and let $OM(i)$ be the i -th element in the ordering;

sort the elements of item set N by increasing class number, breaking ties by decreasing weight, and let $ON(j)$ be the j -th element in the ordering;

for $d := 1$ **to** m **do**

$J := \{j \in N : w_j \leq c_{OM(d)}\}$ and let OJ be the corresponding ordering;

execute procedure **NFpack**;

if **NFpack** returns a feasible packing of all items of J in the smallest d knapsacks

then pack such items into such knapsacks and remove items and knapsacks

end for.

Observation. Due to preliminary sorting, for a given value of d , the items of J are the only ones that can be packed in the smallest d knapsacks. Hence, if there exists a feasible packing of all the items of J into such knapsacks, this is optimal for the original instance: items and knapsacks can then be removed from the instance, and the process can be iterated.

The core of **Reduction** is procedure **NFpack**, i.e., the search for a feasible packing, which can be seen as a feasibility check for a variant of the *Variable Size Bin Packing Problem*

(VSBPP). Given a set of \bar{n} items of weight $w_1, \dots, w_{\bar{n}}$ and a set of d bins of size c_1, \dots, c_d , the VSBPP asks for an assignment of every item to a bin so that the total weight assigned to any bin does not exceed its size. In our case the packing must also satisfy the additional condition that all the items assigned to a bin belong to the same class. We solved the resulting problem through an adaptation of the *Next-Fit Decreasing* heuristic for the classical *Bin Packing Problem* (a VSBPP in which all bins have the same size). Procedure **NFpack** below considers a class at a time and packs all items of the current class (by decreasing weight) into consecutive bins (by decreasing capacity). A new knapsack is opened when either the next item does not fit into the current knapsack or a new class is considered. The procedure prematurely terminates when no feasible packing is found:

Procedure NFpack:

```

assign the first item,  $OJ(1)$ , to knapsack  $OM(d)$  and set  $s := d$ ,  $\bar{c} := c_{OM(d)} - w_{OJ(1)}$ ;
for  $j := 2$  to  $|J|$  do
    if  $w_{OJ(j)} \leq \bar{c}$  and the class of  $OJ(j)$  is the same as that of  $OJ(j-1)$  then
        assign item  $OJ(j)$  to knapsack  $OM(s)$  and set  $\bar{c} := \bar{c} - w_{OJ(j)}$ ;
    else
        begin
             $s := s - 1$ ;
            if  $s = 0$  or  $w_{OJ(j)} > c_{OM(s)}$  then return false
            else assign item  $OJ(j)$  to knapsack  $OM(s)$  and set  $\bar{c} := c_{OM(s)} - w_{OJ(j)}$ ;
        end
    end for
return the feasible packing found.

```

The performance of this heuristic reduction is affected by the sorting of items which, in turn, is affected by the numbering of classes. Preliminary computational experiments showed that the use of different numberings produces similar results.

As $|J|$ is bounded by n , procedure **NFpack** requires $O(n)$ time. Procedure **Reduction** needs $O(n \log n)$ time for the preliminary sorting of knapsacks and items. At each iteration, before invoking **NFpack**, the ordering OJ can be built in $O(n)$ time by scanning the ordering ON . As there are m iterations, the overall time complexity of our preprocessing is $O(n \log n + nm)$. It is interesting to observe that the time complexity of **NFpack** ($O(n)$, plus the time for the preliminary sorting) is the same as that of the classical next-fit decreasing algorithm for the original (simpler) bin packing problem.

2.2 Pseudo-polynomial time reduction

An additional preprocessing, that however requires a pseudo-polynomial time, is possible. For any knapsack i , we can compute, for every class k , the maximum feasible knapsack filling \bar{c}_{ik} that can be obtained with item set S_k . Let $\bar{c}_i = \max_{k \in K} \{\bar{c}_{ik}\}$: If $\bar{c}_i < c_i$ then we know that $c_i - \bar{c}_i$ capacity units will never be used. Hence we can set $c_i = \bar{c}_i$, thus making the formulation more tight.

Determining each \bar{c}_{ik} value amounts to solving a *Subset Sum Problem* (SSP), i.e., a special

knapsack problem with \bar{n} items, in which $p_j = w_j$ for $j = 1, \dots, \bar{n}$. The SSP is known to be (weakly) \mathcal{NP} -hard and solvable, through dynamic programming, in pseudo-polynomial time $O(\bar{n}c_i)$. A naive implementation would then require mr such solutions (one per knapsack and class), i.e., $O(\sum_{k \in K} \sum_{i \in M} |S_k| c_i) = O(\sum_{k \in K} |S_k| \sum_{i \in M} c_i) = O(n \sum_{i \in M} c_i)$ time in total. Recall however that, due to the knapsack sorting, $c_{\max} = c_{OM(m)}$ is the largest capacity. It follows that, for every class k , the dynamic programming list of the SSP induced by the largest knapsack contains all states induced by the other (smaller) knapsacks. We can thus perform a single computation per class and hence preprocess the whole instance in $O(nc_{\max})$ time.

3. Relaxations

Kataoka and Yamada [11] have studied the *Lagrangian relaxation* of capacity constraints (4) of model (1)-(6) with multipliers λ_{ik} . They also investigated the *Lagrangian dual* problem, i.e., the problem of determining the multipliers producing the best (minimum) upper bound value. Indeed, it is shown in [11] that: (i) an optimal solution to the Lagrangian dual exists, in which all multipliers have the same value; (ii) the resulting upper bound has the same value as that of the continuous relaxation of the model. Such properties generalize analogous results obtained by Martello and Toth [18] for the Lagrangian relaxation of the capacity constraints of the MKP.

An alternative method for computing upper bounds can be obtained by replacing, for each knapsack i , the associated r constraints (4) with their *surrogate relaxation*. We obtain:

$$\sum_{k \in K} \pi_{ik} \sum_{j \in S_k} w_j x_{ij} \leq \sum_{k \in K} \pi_{ik} c_i y_{ik} \quad i \in M \quad (7)$$

where $\pi \in R_+^{m \times r}$ is an array of non-negative multipliers. As to the *surrogate dual* problem, Martello and Toth [16] proved that, for the MKP, an optimal dual solution assigns an identical positive value to all multipliers. We next show that the same property extends to the MKAP.

Property 1 *The optimal dual solution to the surrogate relaxation of the MKAP, defined by (1)-(3) and (5)-(7), is $\pi_{ik} = \alpha$ (α any positive constant value) for $i = 1, \dots, m$, $k = 1, \dots, r$.*

Proof. First observe that the optimal dual solution must have, for any $i \in M$, at least one $\pi_{ik} > 0$ as otherwise its value would provide the trivial upper bound $\sum_{j \in N} p_j$ (corresponding to packing all items into a knapsack having all null multipliers). Consider any knapsack $i \in M$ and let $k^*(i)$ be a class k associated with the highest π_{ik} value. An optimal solution to the surrogate relaxation will associate to each knapsack i the largest possible capacity, i.e., it will have $y_{ik^*(i)} = 1$ and $y_{ik} = 0$ for $k \neq k^*(i)$ ($i \in M$). Hence, the right-hand side of (7) can be written as $\pi_{ik^*(i)} c_i$ ($i \in M$). By dividing the resulting constraints by $\pi_{ik^*(i)}$, we get

$$\sum_{k \in K} \frac{\pi_{ik}}{\pi_{ik^*(i)}} \sum_{j \in S_k} w_j x_{ij} \leq c_i \quad i \in M. \quad (8)$$

For any knapsack i we have $\pi_{ik}/\pi_{ik^*(i)} \leq 1$ for all k . It follows that an optimal set of multipliers (i.e., such that the lowest surrogate solution value is attained) consists in having

identical positive π_{ik} values for all i and k . \square

An optimal set of multipliers is thus $\pi_{ik} = 1 \ \forall \ i, k$. With such choice, and given that in (3) we can impose equality, the surrogate constraint (7) becomes

$$\sum_{j \in N} w_j x_{ij} \leq c_i \quad i \in M \quad (9)$$

By observing that variables y_{ik} no longer play any role, the resulting MKP-*based relaxation* is given by (1), (2), (9), and (5). Intuitively, this corresponds to allowing each knapsack to contain items of different classes, i.e., to solve a classical MKP without the additional class restriction. The resulting problem remains however strongly \mathcal{NP} -hard, i.e., it can be expected to only be practically computable for small instances.

A computationally more tractable relaxation can be obtained from the previous one by further relaxing, in a surrogate fashion, constraints (9) with m unit multipliers, i.e., by replacing the m capacity constraints (9) with a unique constraint

$$\sum_{i \in M} \sum_{j \in N} w_j x_{ij} \leq \sum_{i \in M} c_i. \quad (10)$$

By introducing new variables $\xi_j = \sum_{i \in M} x_{ij}$, we obtain our final KP-*based relaxation*:

$$\max \sum_{j \in N} p_j \xi_j \quad (11)$$

$$\sum_{j \in N} w_j \xi_j \leq \sum_{i \in M} c_i \quad (12)$$

$$\xi_j \in \{0, 1\} \quad j \in N, \quad (13)$$

which disregards the item classes and defines a single knapsack problem with capacity equal to the sum of the given knapsack capacities. This problem is known to be weakly \mathcal{NP} -hard and solvable in pseudo-polynomial time $O(n \sum_{i \in M} c_i)$ through dynamic programming.

4. Heuristic algorithms

We present in this section a heuristic approach for the MKAP, consisting of two phases: a constructive heuristic and a metaheuristic post-optimization procedure.

4.1 Constructive heuristic

The constructive algorithm decomposes the MKAP into two separate problems:

- assign a different set of knapsacks to each class;
- find a feasible packing of a subset of items into the knapsacks assigned to the corresponding classes.

The following procedure heuristically determines an assignment of a disjoint subset of knapsacks, M_k , to each class $k \in K$, trying to mimic a given optimal solution ξ of the *KP-based relaxation* (11)-(13):

Procedure Assign:

for each class $k \in K$ **do**

$$P_k := \sum_{j \in S_k} p_j \xi_j;$$

$$W_k := \sum_{j \in S_k} w_j \xi_j$$

end for;

sort the classes according to decreasing P_k/W_k ratios and let $OK(k)$ be the k -th element in the ordering;

$$\overline{M} := M;$$

for $\bar{k} := 1$ **to** r **do**

$$k := OK(\bar{k});$$

determine, if any, a subset $\hat{M} \subseteq \overline{M}$ such that $\sum_{i \in \hat{M}} c_i$ is closest to, and not below, W_k ;

if such a set exists **then**

$$M_k := \hat{M}, \overline{M} := \overline{M} \setminus M_k$$

else

$$M_k := \overline{M}, \overline{M} := \emptyset;$$

for $k' := \bar{k} + 1$ **to** r **do** $M_{OK(k')} := \emptyset$;

break

end if

end for.

The procedure starts by computing the total profit P_k and weight W_k assigned to each class k in the solution of the relaxed problem. A class k at a time is then considered: if possible, a subset of unassigned knapsacks of total capacity not lower than W_k is determined and assigned to the class. When the total residual knapsack capacity is below the threshold, all free knapsacks are assigned to the current class, and no knapsack is assigned to the remaining classes.

Given an optimal solution to the *KP-based relaxation*, the procedure needs $O(n + r \log r)$ time for pricing and sorting the r classes. The second loop performs r iterations, each requiring the exact solution of an SSP in minimization form, in which the capacity is W_k , items correspond to knapsacks and weights to knapsack capacities. The solution to such problem can be obtained, through dynamic programming, in $O(mW_k)$ time. By observing that $\sum_{k \in K} W_k \leq \sum_{i \in M} c_i$, the overall time complexity for the second loop is $O(m \sum_{i \in M} c_i)$. It turns out that the overall time complexity for the assignment of a different set of knapsacks to each class is given by that of the preceding phases (preprocessing followed by the solution of the relaxed problem), i.e., it is bounded by $O(n \log n + nm + n \sum_{i \in M} c_i) = O(n \log n + n \sum_{i \in M} c_i)$.

The second problem (find a feasible item packing into the knapsacks) can be solved by separately determining, for each class, a feasible packing of a subset of its items to the

assigned knapsacks. For each class, the induced problem can then be solved as an MKP, namely:

Procedure MKpack:

for each class k **do**

 solve the MKP instance induced by item set S_k and knapsack set M_k ;

 let N_i be the set of items assigned to knapsack i ($i \in M_k$)

end for;

define the MKAP solution that allocates item set N_i to each knapsack $i \in M$;

return the feasible packing found.

The MKP is strongly \mathcal{NP} -hard, and hence the procedure would require, in the worst case, an exponential time if implemented with an exact algorithm. We adopted instead the polynomial-time heuristic algorithm MTHM by Martello and Toth [17] (Fortran implementation available at <http://www.or.deis.unibo.it/knapsack.html>), whose time complexity is $O(n^2)$. This gives a time complexity $O(n^2 + n \sum_{i \in M} c_i)$ for the sequence [Preprocessing (Section 2) \rightarrow KP-based relaxation (Section 3) \rightarrow Constructive heuristic].

4.2 Metaheuristic refinement

Once a feasible solution has been obtained, a randomized metaheuristic approach is performed to improve it. Define a dummy class S_0 , and assign it a knapsack set M_0 containing all knapsacks left empty by **MKpack**. The method iteratively explores three neighborhoods:

Neighborhood 1: assign an empty knapsack of M_0 to a class;

Neighborhood 2: re-assign a non-empty knapsack;

Neighborhood 3: interchange two non-empty knapsacks.

The algorithm initially defines the *incumbent solution* as the one returned by **MKpack**. It operates on a *current solution* (initially, the incumbent), making use of three user defined parameters τ_1, τ_2 , and τ_3 . At each iteration, τ_1, τ_2 , and τ_3 solutions in the corresponding neighborhoods of the current solution are randomly selected and evaluated by running an MKP heuristic for each class involved in the move. As soon as a solution improving on the incumbent is obtained, the incumbent is updated and the process is re-started with the new incumbent. When, at the end of an iteration, none of the three neighbors has improved on the incumbent, the best solution obtained in the iteration becomes the current solution. The following function receives as input a pair (item set S' , knapsack set M') and possibly a second (non empty) pair (S'' , M''), and solves the MKP instance(s) associated with the pair(s): it returns the value **true** if the incumbent solution (and the corresponding knapsack partition) has been improved, and the value **false** otherwise.

Function Improve(S', M', S'', M''):

solve the MKP instance induced by item set S' and knapsack set M' ;

if $S'' \neq \emptyset$ **then**

 let $\overline{M}_0 \subseteq M'$ be the resulting set of empty knapsacks;

 solve the MKP instance induced by item set S'' and knapsack set $M'' \cup \overline{M}_0$;

endif

if the overall resulting solution improves on the incumbent **then**

 update the incumbent solution and partition M_0, M_1, \dots, M_r and **return true**

else return false

The following metaheuristic procedure iteratively invokes **Improve** to explore the neighborhoods:

Procedure Refine:

define the current knapsack partition M_0, M_1, \dots, M_r as in the incumbent solution;

while a stopping condition does not hold **do**

for $\tau := 1$ **to** τ_1 **do** (comment: explore Neighborhood 1)

if $M_0 \neq \emptyset$ **then** randomly select a class $k \neq 0$ **else break**;

$impr := \text{Improve}(S_k, M_k \cup M_0, \emptyset, \emptyset)$;

if $impr = \text{true}$ **then break**

end for;

if $impr = \text{true}$ **then continue**;

for $\tau := 1$ **to** τ_2 **do** (comment: explore Neighborhood 2)

 randomly select a non-empty knapsack $i' \notin M_0$;

 let k' be the class associated with i' and randomly select a class $k \neq k'$;

$impr := \text{Improve}(S_{k'}, (M_{k'} \setminus \{i'\}) \cup M_0, S_k, M_k \cup \{i'\})$;

if $impr = \text{true}$ **then break**

end for;

if $impr = \text{true}$ **then continue**;

for $\tau := 1$ **to** τ_3 **do** (comment: explore Neighborhood 3)

 randomly select two non-empty knapsacks i' and i'' associated with different classes;

 let k' (resp. k'') be the class associated with i' (resp. i'');

$impr := \text{Improve}(S_{k'}, (M_{k'} \setminus \{i'\}) \cup \{i''\} \cup M_0, S_{k''}, (M_{k''} \setminus \{i''\}) \cup \{i'\})$;

if $impr = \text{true}$ **then break**

end for;

end while

return the incumbent solution.

A metaheuristic process is typically non-polynomial, and hence we control the execution

through a *stopping condition* that halts the algorithm when either the value of the incumbent solution equals that of the *KP-based relaxation* (Section 3) or a limit on the number of iterations is reached. The MKP instances are heuristically solved through the branch-and-bound algorithm MULKNAP by Pisinger [19], halted after the root node. Although this takes pseudo-polynomial time, it frequently produces better solutions than those obtained by the polynomial-time algorithm MTHM [17] that was adopted for **MKpack**.

5. Computational experiments

The procedures of Sections 3 and 4 were implemented in C language and executed on an Intel Xeon E5649 running at 2.53 GHz on literature instances and on new instances. On the basis of preliminary computational experiments, (i) procedure **Refine** was executed with $\tau_1 = 3$, $\tau_2 = \tau_3 = 10$, with a maximum number of iterations set to $n * m * r / 10$; (ii) in order to keep the CPU time low, the time-consuming reduction procedure of Section 2.2 was only executed for small instances (see below).

To the best of our knowledge, the only available benchmark for the MKAP was proposed by Kataoka and Yamada [11], who obtained them by uniformly randomly generating the weight w_j of each item j in $[1, R]$ (R being a prefixed parameter). All instances were generated with $R = 1000$, except those in Table 6, which evaluates the effect of different values of R . Different *families* of instances were then produced by generating each profit p_j ($j \in N$) as follows:

- p_j uniformly random integer in $[1, R]$ (*uncorrelated* instances, UNC in the tables);
- $p_j = \lfloor 0.6w_j \rfloor + \vartheta_j$, with ϑ_j a uniformly distributed random integer in $[1, \lfloor 0.4R \rfloor]$ (*weakly correlated* instances, WEA in the tables);
- $p_j = w_j + \lfloor 0.2R \rfloor$ (*strongly correlated* instances, STR in the tables).

The capacity of each knapsack i was always set to

$$c_i = \lfloor \rho \varphi_i \sum_{j \in N} w_j \rfloor, \quad (14)$$

with $\rho \in \{0.25, 0.50, 0.75\}$ and the φ_i being random reals in $[0, 1]$ such that $\sum_{i \in M} \varphi_i = 1$.

Instances of different size were generated:

- *small* instances: $n \in \{20, 40, 60\}$, $r \in \{2, 5\}$, and $m \in \{10, 20\}$;
- *large* instances: $n \in \{4000, 8000\}$, $r \in \{50, 100\}$, and $m \in \{200, 400, 800\}$.

For each instance, every class k contained $|S_k| = n/r$ consecutively generated items. Kataoka and Yamada kindly provided us the computer code they used for generating the instances studied in [11], allowing us to test our algorithms on the same benchmark (10 instances for each of parameters combination).

5.1 Small instances

The first set of experiments was performed on small instances. As in [11], the purpose was to evaluate the performance of the various heuristic approaches and that of an ILP solver. We considered both the instances from [11] and new instances.

Literature instances

Table 1 refers to the test bed introduced in [11], consisting of 360 instances obtained by generating 10 instances for each quadruplet (Family, r , m , n). Lalla-Ruiz and Voß [15] compared their genetic algorithm (LV in the tables) with the heuristic of Kataoka and Yamada (KY in the tables) only on the 120 small instances of family STRONG. The table compares our algorithm (denoted as MM) with both previous approaches on the whole benchmark.

Each row refers to the 10 instances generated for the quadruplet (Family, r , m , n). Columns ‘GRB (1200s)’ report the average solution values z^* and the number of provably optimal solutions obtained, from model (1)-(6), by ILP solver Gurobi (Version 7.5.2) with a time limit of 1200 seconds.

Columns ‘KY’ give the average solution values z and the average percentage error with respect to the best solution produced by Gurobi in 1200 seconds, computed as

$$\%Err = 100 \frac{z^* - z}{z^*}. \quad (15)$$

Having obtained the original computer code from the authors, we tested it on our machine. The average CPU times of KY are negligible: “far less than a second” according [11], and even below 0.01 second on our machine, and hence they are not reported.

Columns ‘LV’ give, for the strongly correlated instances only, the average solution values z and CPU times (taken from [15], where they were obtained on an Intel 3.16 GHz), and the average percentage error, computed according to (15). The same information is reported, for all instances, in columns ‘MM’ for what concerns the proposed algorithm and in columns ‘GRB(1s)’ for what concerns the execution of Gurobi with a time limit of 1 second. For each family of instances, an additional row reports the average values over the 120 tested instances, while the final row gives the overall average values over the whole benchmark.

The results show that these instances are relatively easy to solve, both for the three heuristics and for GRB(1s). GRB(1200s) exactly solved 307 out of 360 instances, the strongly correlated ones being somehow more challenging (as frequently observed in the knapsack literature). One can observe that the solution values z^* for the instances with $r = 2$ are very close to their counterparts for $r = 5$, and that the latter appear to be much easier (Gurobi found a proven optimal solution in over 92% of the cases).

The constructive heuristic KY is the fastest approach. As could be expected, LV and MM, which include metaheuristic improvements, take higher CPU times (although still far below a second), but also produce much better approximations. For the strongly correlated instances, the average percentage error of KY is 7.15. It is improved to 0.22 by LV (in 0.25 seconds on average), and further reduced to 0.12 (in 0.15 seconds on average, on a slower machine) by MM. For the other two families of instances, MM produced small average errors, about 25 times smaller than those produced by KY. When its time limit is decreased to 1

Instances				GRB (1200s)		KY (< 0.1s)		LV			MM			GRB (1s)	
Fam	r	m	n	z*	#opt	z	%Err	z	CPU	%Err	z	CPU	%Err	z	%Err
UNC	2	10	20	7438.8	10	6991.6	6.01				7438.8	0.01	0.00	7438.80	0.00
			40	16,291.4	10	15,893.7	2.44				16,257.4	0.08	0.21	16,146.10	0.89
			60	24,710.9	6	24,568.8	0.58				24,712.8	0.09	−0.01	24,463.00	1.00
		20	20	4685.8	10	4436.5	5.32				4685.8	0.00	0.00	4685.80	0.00
	40		15,455.9	10	14,798.2	4.26				15,351.4	0.13	0.68	15,436.00	0.13	
	60		24,588.1	7	23,911.9	2.75				24,518.0	0.31	0.29	24,279.90	1.25	
	5	10	20	7314.7	10	5847.6	20.06				7252.4	0.01	0.85	7314.70	0.00
			40	16,041.0	10	15,089.0	5.93				16,021.3	0.03	0.12	16,000.30	0.25
			60	24,489.4	10	23,846.1	2.63				24,456.9	0.10	0.13	24,004.10	1.98
		20	20	4685.8	10	3830.2	18.26				4685.8	0.00	0.00	4685.80	0.00
	40		15,379.7	10	13,356.5	13.16				15,273.5	0.08	0.69	15,379.70	0.00	
	60		24,445.7	10	22,944.2	6.14				24,311.2	0.20	0.55	24,208.80	0.97	
	Avg				15,460.6	9.4	14,626.2	7.29			15,413.8	0.09	0.29	15,336.92	0.54
WEA	2	10	20	5559.0	10	5061.2	8.95				5523.2	0.01	0.64	5559.00	0.00
			40	12,319.9	10	11,933.6	3.14				12,273.3	0.08	0.38	12,174.90	1.18
			60	18,707.2	0	18,659.0	0.26				18,735.7	0.12	−0.15	18,457.20	1.34
		20	20	2957.0	10	2835.5	4.11				2957.0	0.00	0.00	2957.00	0.00
	40		11,612.7	10	10,911.7	6.04				11,531.7	0.12	0.70	11,611.30	0.01	
	60		18,597.6	6	18,032.3	3.04				18,501.7	0.36	0.52	18,459.60	0.74	
	5	10	20	5430.4	10	4385.9	19.23				5292.6	0.01	2.54	5430.40	0.00
			40	12,061.4	10	11,135.5	7.68				12,032.8	0.04	0.24	12,058.40	0.02
			60	18,540.4	10	18,090.3	2.43				18,519.5	0.10	0.11	18,147.00	2.12
		20	20	2957.0	10	2483.6	16.01				2957.0	0.00	0.00	2957.00	0.00
	40		11,498.7	10	9659.3	16.00				11,473.6	0.09	0.22	11,498.70	0.00	
	60		18,447.8	10	16,859.6	8.61				18,357.5	0.24	0.49	18,265.50	0.99	
	Avg				11,557.4	8.8	10,837.3	7.96			11,513.0	0.10	0.47	11,464.67	0.53
STR	2	10	20	7134.4	10	6596.9	7.53	7134.4	0.05	0.00	7132.8	0.01	0.02	7134.40	0.00
			40	15,378.7	7	15,062.8	2.05	15,375.3	0.02	0.02	15,372.4	0.10	0.04	15,197.80	1.18
			60	23,156.0	0	23,018.8	0.59	23,194.0	0.14	−0.16	23,193.8	0.26	−0.16	22,904.50	1.09
		20	20	4150.3	10	3799.6	8.45	4150.3	0.06	0.00	4150.3	0.00	0.00	4150.30	0.00
	40		14,891.9	10	14,252.2	4.30	14,849.6	0.29	0.28	14,825.4	0.15	0.45	14,885.20	0.04	
	60		23,100.7	4	22,478.1	2.70	23,054.9	0.29	0.20	23,033.6	0.70	0.29	22,656.90	1.92	
	5	10	20	6978.5	10	5766.4	17.37	6977.6	0.14	0.01	6977.6	0.01	0.01	6978.50	0.00
			40	15,146.6	10	14,273.6	5.76	15,083.0	0.42	0.42	15,118.7	0.04	0.18	15,066.60	0.53
			60	22,996.4	1	22,483.5	2.23	22,949.9	0.23	0.20	22,993.6	0.15	0.01	22,574.80	1.83
		20	20	4150.3	10	3579.8	13.75	4150.3	0.10	0.00	4150.3	0.00	0.00	4150.30	0.00
	40		14,781.3	10	12,677.4	14.23	14,643.1	0.93	0.93	14,733.1	0.10	0.33	14,781.30	0.00	
	60		22,932.3	5	21,369.3	6.82	22,754.2	0.30	0.78	22,861.2	0.27	0.31	22,708.20	0.98	
	Avg				14,566.5	7.3	13,779.9	7.15	14,526.4	0.25	0.22	14,545.2	0.15	0.12	14,432.40
Overall Avg				13,861.5	8.5	13,081.1	7.47				13,824.0	0.11	0.30	13,744.66	0.57

Table 1: Small instances from the literature: Average solution values and percentage errors with respect to ILP solver Gurobi (1200 seconds).

second, Gurobi (column ‘GRB(1s)’) produces reasonably good solutions, but it is dominated, on average, by MM, which obtains half of its error in one tenth of its time.

The efficiency of MM mostly comes from the metaheuristic refinement, while the constructive heuristic is extremely fast but produces much worse approximation. For the three

families of Table 1 the average percentage errors of the initial solution are 6.98, 8.02, and 7.90, respectively, i.e., over one order of magnitude larger than the final errors. (A similar behavior was observed for the remaining tables.)

New instances

Considering instances in which all classes contain the same number of items (as those in Table 1) appears to be quite restrictive. In addition, the classical benchmarks for the MKP (see, e.g., [18] and [12]) usually adopt two kinds of knapsack capacities: *similar* and *dissimilar*. Those in [11], see (14), belong to the latter kind, so we refer to them as SAME/DISS. We obtained three new benchmarks by generating, from each small instance of [11], three new instances having the same values of n , r , and m , the same item set, but, respectively:

1. classes of different cardinality (benchmark DIFF/DISS);
2. knapsacks with similar capacities (benchmark SAME/SIMI);
3. classes of different cardinality and knapsacks with similar capacities (benchmark DIFF/SIMI).

Classes of different cardinality (benchmarks DIFF/*) were obtained by: (i) setting

$$|S_k| = \lfloor \varphi_k n \rfloor, (k = 1, \dots, r-1); \quad (16)$$

$$|S_r| = n - \sum_{k=1}^{r-1} |S_k|, \quad (17)$$

the φ_k values being random reals in $[0, 1]$ such that $\sum_{k \in K} \varphi_k = 1$; and (ii) randomly assigning the original items to the generated classes.

Knapsacks with similar capacities (benchmarks */SIMI) were obtained by uniformly randomly generating, for $i = 1, \dots, m-1$, capacity c_i in the interval

$$\left[\left\lfloor 0.4 \frac{\sum_{j \in N} w_j}{m} \right\rfloor, \left\lfloor 0.6 \frac{\sum_{j \in N} w_j}{m} \right\rfloor \right], \quad (18)$$

and setting $c_m = \lfloor 0.5 \sum_{j \in N} w_j / m \rfloor - \sum_{i=1}^{m-1} c_i$.

Table 2 reports the outcome of computational experiments on these instances and provides, in a synthetic way, the same information as Table 1. In particular, we don't report the results for the instances with five classes ($r = 5$): As already observed for Table 1, these instances are very similar but easier than their counterparts for $r = 2$ (and the same holds for the three new benchmarks). Also, we don't provide the solution value z of the heuristic algorithm, as it can be derived from the percentage error. The results confirm the outcome of Table 1: The constructive heuristic KY is very fast, but its average percentage errors are one order of magnitude larger than those produced by MM. Concerning Gurobi, these instances are slightly more challenging than those of Table 1, especially for what concerns their optimal solution: the solver could not obtain, in 1200 seconds, a provably optimal solution in one quarter of the cases. It produces however, in one second, good approximate solutions, although worse (average percentage error 0.48 versus 0.36) than those obtained by MM in 0.15 seconds on average.

Instances			DIFF/DISS				SAME/SIMI				DIFF/SIMI											
			GRB (1200s) <i>z</i> [*] #opt	KY %Err	MM CPU	GRB(1s) %Err	GRB (1200s) <i>z</i> [*] #opt	KY %Err	MM CPU	GRB(1s) %Err	GRB (1200s) <i>z</i> [*] #opt	KY %Err	MM CPU	GRB(1s) %Err								
Fam	UNC	2	10	20	7446.0	10	7.70	0.10	0.01	0.00	6063.1	10	7.07	0.00	0.00	0.00	6087.8	10	2.55	0.00	0.00	
		40	16,270.2	9	3.50	1.03	0.06	0.36	16,329.6	9	3.53	0.30	0.07	0.64	16,220.9	10	4.42	1.60	0.06	0.73		
		60	24,687.8	5	0.74	0.01	0.12	0.56	24,671.0	2	0.94	-0.20	0.09	1.08	24,660.7	4	0.51	-0.09	0.11	0.95		
		20	20	4685.8	10	6.00	0.00	0.00	0.00	3374.1	10	5.56	0.00	0.00	0.00	3215.5	10	1.69	0.00	0.00	0.00	
		40	15,450.6	10	4.49	1.14	0.13	0.00	12,497.9	10	4.91	0.00	0.02	0.00	12,450.3	10	3.36	0.00	0.01	0.00		
		60	24,601.5	7	3.33	0.66	0.30	1.31	24,037.0	6	4.68	0.67	0.32	1.00	24,107.0	5	4.35	1.07	0.32	0.99		
		Avg	15,523.7	8.5	4.29	0.49	0.10	0.37	14,495.5	7.8	4.45	0.13	0.08	0.45	14,457.0	8.2	2.82	0.43	0.08	0.44		
		WEA	2	10	20	5547.5	10	9.81	0.47	0.01	0.00	4502.8	10	4.91	0.00	0.00	0.00	4448.6	10	8.24	0.43	0.00
		40	12,313.5	7	1.59	0.98	0.06	0.61	12,359.2	10	3.83	0.55	0.10	0.87	12,280.8	8	3.50	2.00	0.11	1.10		
		60	18,680.0	1	1.61	-0.06	0.14	1.09	18,695.1	0	0.92	-0.18	0.16	1.34	18,692.8	1	0.71	-0.08	0.17	1.07		
20	20	2957.0	10	12.10	0.00	0.00	0.00	1811.5	10	17.00	0.00	0.00	0.00	1834.9	10	11.23	0.00	0.00	0.00			
40	11,628.7	10	5.76	1.40	0.15	0.07	9245.5	10	5.12	0.00	0.01	0.00	9072.7	10	4.02	0.00	0.01	0.00				
60	18,593.7	6	6.26	0.72	0.35	0.70	18,456.3	3	5.44	0.64	0.44	0.81	18,434.1	5	6.84	1.47	0.40	1.03				
Avg	11,620.1	7.3	6.19	0.59	0.12	0.41	10,845.1	7.2	6.20	0.17	0.12	0.50	10,794.0	7.3	5.76	0.64	0.11	0.53				
STR	2	10	20	7159.6	10	5.45	0.02	0.01	0.00	5924.2	10	8.88	0.00	0.00	0.00	6255.3	10	12.25	0.00	0.00		
		40	15,374.1	6	2.00	0.08	0.18	0.56	15,432.9	9	3.44	0.41	0.10	1.12	15,417.6	2	1.89	0.94	0.17	0.95		
		60	23,144.5	1	1.39	-0.07	0.52	0.79	23,156.3	0	0.31	-0.29	0.25	0.78	23,164.4	0	0.87	-0.14	0.44	0.97		
		20	20	4150.3	10	7.98	0.00	0.00	0.00	2647.1	10	7.65	0.00	0.00	0.00	2487.2	10	4.13	0.00	0.00	0.00	
		40	14,989.1	10	5.03	1.16	0.30	0.06	11,464.6	10	4.03	0.33	0.03	0.00	11,715.1	10	1.13	0.00	0.00	0.00		
		60	23,106.7	4	4.18	0.46	0.96	1.51	23,111.2	3	5.04	0.87	0.58	1.53	23,117.7	1	4.86	1.52	0.81	1.67		
		Avg	14,654.1	6.8	4.34	0.27	0.33	0.49	13,622.7	7.0	4.89	0.22	0.16	0.57	13,692.9	5.5	4.19	0.39	0.24	0.60		
		Overall Avg	13,932.6	7.6	4.94	0.45	0.18	0.42	12,987.7	7.3	5.18	0.17	0.12	0.51	12,981.3	7.0	4.25	0.48	0.15	0.53		

5.2 Large instances

The results of the previous section show that small-size instances are relatively easy to solve. A general purpose ILP solver can obtain, in about 75% of the cases, a provably optimal solution within a reasonable time limit (20 minutes). It also produces good approximate solutions within one second, although the proposed heuristic, MM, finds better solutions within a fraction of a second. In this section we thus evaluate the various algorithms on larger instances from the literature.

Preliminary computational experiments showed that, for large-size instances, Gurobi is generally unable to find optimal (and even near optimal) solutions within 1200 seconds. It was thus decided to avoid such time-consuming runs, and to evaluate the quality of the solutions obtained by the heuristic algorithms as the percentage gap with respect to the upper bound UB provided by the *KP-based relaxation* of Section 3, computed as

$$\%Gap = 100 \frac{UB - z}{UB}. \quad (19)$$

We examine in Table 3 the behavior of the algorithms when the number of items increases. For different combinations of r and m , and for $\rho = 0.5$, the table reports, for n increasing

			UNC				WEA				STR			
Instances			KY	MM		GRB(1s)	KY	MM		GRB(1s)	KY	MM		GRB(1s)
r	m	n	%Gap	%Gap	CPU	%Gap	%Gap	%Gap	CPU	%Gap	%Gap	%Gap	CPU	%Gap
2	10	20	8.14	2.27	0.01	2.27	13.46	5.56	0.01	4.95	10.89	3.66	0.01	3.63
		40	3.04	0.82	0.07	1.50	3.69	0.95	0.06	1.74	2.32	0.31	0.08	1.44
		60	0.65	0.06	0.09	1.07	0.45	0.04	0.12	1.52	0.86	0.10	0.26	1.35
		80	0.21	0.04	0.08	0.75	0.25	0.03	0.12	0.89	0.48	0.07	0.41	0.81
		100	0.20	0.02	0.12	0.64	0.15	0.06	0.25	0.79	0.33	0.00	0.22	0.65
		200	0.08	0.01	0.15	0.32	0.06	0.01	0.54	0.33	0.16	0.00	0.17	0.45
		400	0.03	0.01	0.81	0.29	0.02	0.01	1.01	0.27	0.04	0.00	0.29	0.28
		600	0.01	0.01	1.01	0.17	0.01	0.01	1.01	0.28	0.04	0.01	0.36	0.43
		800	0.01	0.00	0.92	1.05	0.01	0.00	1.02	1.23	0.03	0.02	0.54	0.39
		1000	0.01	0.00	1.01	2.83	0.00	0.00	1.01	1.18	0.03	0.02	0.89	0.81
Avg		1.24	0.32	0.43	1.09	1.81	0.67	0.52	1.32	1.52	0.42	0.32	1.02	
5	20	20	18.26	0.00	0.00	0.00	16.01	0.00	0.00	0.00	13.75	0.00	0.00	0.00
		40	15.64	3.53	0.07	2.86	19.11	3.92	0.07	3.71	16.90	3.42	0.10	3.10
		60	7.06	1.52	0.21	1.86	9.88	1.87	0.24	2.36	7.71	1.27	0.26	1.93
		80	3.48	0.74	0.46	3.34	4.94	0.90	0.48	3.44	4.43	0.58	0.61	2.98
		100	2.48	0.38	0.69	2.87	2.41	0.56	0.77	2.97	1.72	0.24	1.01	2.51
		200	0.40	0.11	1.01	1.49	0.36	0.11	1.01	1.51	0.61	0.16	1.02	1.37
		400	0.18	0.05	1.01	1.57	0.14	0.05	1.01	1.03	0.29	0.08	0.95	1.14
		600	0.10	0.04	1.01	2.70	0.06	0.03	1.01	1.30	0.18	0.04	1.02	0.92
		800	0.06	0.03	1.01	1.77	0.04	0.02	1.01	1.81	0.14	0.04	1.09	1.23
		1000	0.04	0.02	1.01	3.61	0.03	0.02	1.01	1.70	0.12	0.05	1.18	1.06
Avg		4.77	0.64	0.65	2.21	5.30	0.75	0.66	1.98	4.58	0.59	0.72	1.62	
Overall Avg			3.00	0.48	0.54	1.65	3.55	0.71	0.59	1.65	3.05	0.50	0.52	1.32

Table 3: Results for increasing n values (with $\rho = 0.5$): Average percentage gaps with respect to the *KP-based relaxation*.

from 20 to 1000, the average percentage gaps obtained by KY, MM, and Gurobi (all with a time limit of one second). The results for the three families of instances (UNC, WEA, and STR) are reported in different groups of columns. Each entry refers to 10 instances. The average CPU time of KY was negligible, the one of MM was around half a second. The average gaps produced by MM are less than one half of those produced by Gurobi, which in turn are one half of those produced by KY. As could be expected, the average gaps of both KY and MM clearly decrease when n grows. The same phenomenon is less evident for Gurobi. For the instances with $r = 5$ and $m = 20$, the existing algorithms perform poorly while MM performs quite well: its average percentage error (0.66) is one third of that of Gurobi (1.94) and one seventh of that of KY (4.89).

Tables 4 and 5 examine the behavior of KY and MM on very large instances with $r \in \{50, 100\}$, $m \in \{200, 400, 800\}$, and $n \in \{4000, 8000\}$. For instances of this size, Gurobi (with a time limit of one second) could never solve the LP relaxation at the root node and it always failed to even produce a feasible solution. Increasing the time limit to 30 seconds produced a feasible solution for less than 5% of the instances. For this reason, in all the remaining tables we don't report the results for Gurobi. KY and MM were always executed with a time limit of one second.

The three groups of columns in Table 4 provide the average percentage gaps (over 10 instances) for uncorrelated instances with different values of $\rho \in \{0.25, 0.50, 0.75\}$, while those in the companion Table 5 refer to the three families of instances (uncorrelated, weakly correlated, and strongly correlated) for fixed $\rho = 0.50$. It is worth observing that, in both sets of instances, the average CPU time of KY is no longer irrelevant: it is about one quarter of a second, while that of MM (not reported) was almost always equal to the one second time limit. The average percentage gap of KY is about one and a half times that of MM. Among the instances of Table 4, those with $\rho = 0.25$ appear to be more challenging for both heuristics: this could be partially due to the fact that relaxing all capacities to a single

			$\rho = 0.25$			$\rho = 0.50$			$\rho = 0.75$		
Instances			KY		MM	KY		MM	KY		MM
r	m	n	%Gap	CPU	%Gap	%Gap	CPU	%Gap	%Gap	CPU	%Gap
50	200	4000	0.20	0.02	0.18	0.13	0.02	0.13	0.08	0.02	0.11
		8000	0.07	0.02	0.07	0.04	0.02	0.06	0.03	0.03	0.06
	400	4000	0.26	0.08	0.17	0.11	0.09	0.11	0.07	0.10	0.08
		8000	0.05	0.09	0.06	0.03	0.11	0.04	0.02	0.09	0.04
	800	4000	1.69	0.22	0.97	0.42	0.27	0.21	0.09	0.27	0.08
		8000	0.05	0.26	0.06	0.03	0.27	0.04	0.02	0.30	0.03
100	200	4000	0.69	0.08	0.60	0.44	0.06	0.45	0.57	0.08	0.39
		8000	0.27	0.07	0.27	0.30	0.07	0.21	0.47	0.05	0.22
	400	4000	1.07	0.28	0.54	0.38	0.31	0.33	0.23	0.25	0.25
		8000	0.18	0.30	0.19	0.11	0.24	0.15	0.08	0.27	0.13
	800	4000	3.39	0.64	1.37	1.23	0.86	0.49	0.40	0.82	0.22
		8000	0.24	0.84	0.19	0.10	0.83	0.12	0.07	0.87	0.15
Avg			0.68	0.24	0.39	0.28	0.26	0.20	0.18	0.26	0.15

Table 4: Results on large, uncorrelated instances with different ρ values: Average percentage gaps with respect to the KP-based relaxation.

			UNC			WEA			STR		
Instances			KY		MM	KY		MM	KY		MM
r	m	n	%Gap	CPU	%Gap	%Gap	CPU	%Gap	%Gap	CPU	%Gap
50	200	4000	0.13	0.02	0.13	0.10	0.02	0.10	0.31	0.05	0.19
		8000	0.04	0.03	0.06	0.04	0.03	0.04	0.16	0.10	0.09
	400	4000	0.11	0.09	0.11	0.08	0.09	0.09	0.29	0.11	0.20
		8000	0.03	0.11	0.04	0.03	0.09	0.03	0.14	0.17	0.14
	800	4000	0.42	0.27	0.21	0.36	0.27	0.20	0.47	0.28	0.31
		8000	0.03	0.28	0.04	0.02	0.30	0.03	0.14	0.35	0.12
100	200	4000	0.44	0.06	0.45	0.35	0.06	0.35	0.62	0.07	0.40
		8000	0.30	0.07	0.21	0.19	0.06	0.15	0.35	0.12	0.30
	400	4000	0.38	0.31	0.33	0.30	0.29	0.27	0.64	0.26	0.34
		8000	0.11	0.24	0.15	0.09	0.25	0.10	0.32	0.30	0.21
	800	4000	1.23	0.87	0.49	1.15	0.68	0.49	1.21	0.79	0.49
		8000	0.10	0.83	0.12	0.08	1.09	0.09	0.31	1.10	0.22
Avg			0.28	0.27	0.20	0.23	0.27	0.16	0.41	0.31	0.25

Table 5: Results on large instances of different families with $\rho = 0.5$: Average percentage gaps with respect to the KP-based *relaxation*.

knapsack weakens the upper bound when the instance includes small capacities, as happens for small values of ρ . Table 5 shows that strongly correlated instances are most difficult to solve, as usually happens for classical knapsack problems.

All instances addressed so far were generated with $R = 1000$, R being the maximum value of profits and weights. The complexity of knapsack problems is known to be dependent on the magnitude of the input. In particular, the time complexity of the constructive heuristic of Section 4.1 is a function of the knapsack capacities. Table 6 examines the performance of the heuristics for large instances of the three families generated with different values of $R \in \{100, 1000, 10000\}$. We don't report the CPU times of the two heuristics as they were much the same as for Tables 4 and 5. The results show no remarkable difference between the two approaches: both produce high quality solutions, MM being slightly better for $R \leq 1000$, KY for $R = 10,000$. These are the three values of R tested by Kataoka and Yamada [11]. Limited additional experiments with larger values of R showed that KY generally performs better than MM also for larger R values (although for $R \geq 25,000$ both algorithms fail due to integer overflow).

Finally, following what has been done in Kataoka and Yamada [11], we tested whether a smaller number of possible objective function values has an effect on the performance of the algorithms. To this end, we considered an additional family of instances obtained by generating weights and capacities as in the other families (with $R = 1000$) but profits

- $p_j = 1$ or 100 with equal probability (*binary* instances).

In Table 7 we evaluate large binary instances against the corresponding uncorrelated instances, for different values of $\rho \in \{0.25, 0.50, 0.75\}$. As already observed in [11], binary instances with smaller ρ values, i.e., with smaller capacities, are much harder than their uncorrelated counterparts as only a fraction of high profit items can be accommodated into small knapsacks. For such instances ($\rho \leq 0.5$), the average percentage gap of both heuristics

Instances		UNC						WEA						STR					
		$R = 10^2$		$R = 10^3$		$R = 10^4$		$R = 10^2$		$R = 10^3$		$R = 10^4$		$R = 10^2$		$R = 10^3$		$R = 10^4$	
r	m	n	KY	MM	%Gap	KY	MM	KY	MM	%Gap	KY	MM	%Gap	KY	MM	%Gap	KY	MM	%Gap
50	200	4000	0.12	0.12	0.13	0.13	0.22	0.10	0.08	0.10	0.10	0.10	0.16	0.29	0.13	0.31	0.19	0.33	0.38
		8000	0.05	0.05	0.04	0.06	0.08	0.04	0.03	0.04	0.04	0.04	0.05	0.15	0.07	0.16	0.09	0.17	0.33
400	4000	400	0.09	0.09	0.11	0.11	0.15	0.08	0.06	0.08	0.08	0.11	0.12	0.26	0.12	0.29	0.20	0.35	0.36
		8000	0.04	0.04	0.03	0.04	0.31	0.03	0.02	0.03	0.03	0.03	0.26	0.14	0.07	0.14	0.15	0.31	0.31
800	4000	800	0.09	0.09	0.42	0.21	0.67	0.44	0.07	0.36	0.20	0.61	0.46	0.21	0.15	0.47	0.31	0.69	0.63
		8000	0.03	0.04	0.03	0.04	0.04	0.07	0.03	0.02	0.02	0.03	0.05	0.13	0.07	0.14	0.12	0.19	0.28
100	200	4000	0.40	0.43	0.44	0.45	0.46	0.52	0.34	0.31	0.35	0.36	0.40	0.60	0.35	0.62	0.40	0.64	0.50
		8000	0.29	0.20	0.30	0.21	0.30	0.23	0.17	0.13	0.19	0.15	0.16	0.34	0.21	0.35	0.30	0.34	0.85
400	4000	400	0.32	0.31	0.38	0.33	0.42	1.44	0.25	0.23	0.30	0.27	0.33	0.56	0.27	0.64	0.35	0.75	1.12
		8000	0.12	0.13	0.11	0.15	0.12	0.90	0.10	0.09	0.10	0.10	0.77	0.28	0.13	0.32	0.21	0.35	0.75
800	4000	800	0.47	0.28	1.23	0.49	1.33	0.58	0.41	0.24	1.15	0.49	1.31	0.65	0.59	1.21	0.49	1.44	0.75
		8000	0.10	0.10	0.12	0.12	0.12	0.19	0.07	0.07	0.08	0.09	0.09	0.26	0.14	0.31	0.22	0.36	0.43
Avg			0.18	0.16	0.28	0.20	0.32	0.43	0.14	0.11	0.23	0.16	0.27	0.32	0.17	0.41	0.25	0.48	0.56

Table 6: Results on large instances of different families and different values of R : Average percentage gaps with respect to the KP-based relaxation.

Instances		Binary						Uncorrelated					
		$\rho = 0.25$		$\rho = 0.50$		$\rho = 0.75$		$\rho = 0.25$		$\rho = 0.50$		$\rho = 0.75$	
r	m	n	KY	MM	%Gap	KY	MM	%Gap	KY	MM	%Gap	KY	MM
50	200	4000	1.57	0.63	0.35	0.58	0.61	0.01	0.01	0.20	0.18	0.13	0.08
		8000	0.65	0.35	0.29	0.45	0.45	0.01	0.00	0.07	0.07	0.06	0.06
400	4000	400	2.58	0.78	0.66	0.50	0.50	0.01	0.01	0.26	0.17	0.11	0.07
		8000	0.56	0.31	0.18	0.33	0.33	0.01	0.00	0.05	0.06	0.03	0.08
800	4000	800	6.10	3.47	2.47	1.44	1.44	0.01	0.01	1.69	0.97	0.42	0.02
		8000	0.94	0.48	0.20	0.28	0.28	0.01	0.00	0.05	0.06	0.04	0.08
100	200	4000	3.26	1.58	1.59	1.58	1.58	0.04	0.02	0.69	0.60	0.44	0.03
		8000	1.45	0.85	1.14	1.16	1.16	0.02	0.01	0.27	0.27	0.30	0.39
400	4000	400	5.78	1.62	1.88	1.24	1.24	0.02	0.01	1.07	0.53	0.38	0.22
		8000	1.40	0.70	0.55	0.74	0.74	0.01	0.01	0.18	0.19	0.11	0.47
800	4000	800	8.63	4.02	4.52	2.05	2.05	0.03	0.01	3.39	1.37	1.23	0.25
		8000	2.60	0.89	0.69	0.66	0.66	0.01	0.01	0.24	0.19	0.10	0.13
Avg			2.96	1.31	1.23	0.92	0.92	0.02	0.01	0.68	0.39	0.28	0.07
													0.15

Table 7: Results on large binary and uncorrelated instances with different ρ values: Average percentage gaps with respect to the KP-based relaxation.

is about five times larger than for uncorrelated instances. Among the two algorithms, the average percentage gap of MM is about one half that of KY.

The overall outcome of our computational experiments can be summarized as follows:

- *Small instances.* MM is the best approach for strongly correlated instances and for all benchmarks with classes of the same cardinality. Gurobi prevails for uncorrelated and weakly correlated instances of classes with different cardinalities. KY is always extremely fast, but it never gives the best solutions.
- *Large instances.* MM is the best method for instances generated with $R \leq 10,000$, while KY wins for larger R values. Gurobi is never competitive.

6. Conclusions

We have considered a variant of the multiple knapsack problem in which some assignment-type side constraints have to be satisfied. The input consists of m knapsacks and n items, which are partitioned into r classes: A knapsack can only contain items of the same class. The problem is to assign a class and a set of items to each knapsack, in such a way that no knapsack is assigned a set of items exceeding its capacity and the total profit of the assigned items is maximized. The problem finds applications in a number of logistics sectors. We have presented algorithms for the computation of upper and lower bounds. The upper bounds have been derived from the solution to Lagrangian and surrogate relaxations of a mathematical model of the problem. Lower bounds have been obtained through preprocessing techniques and a constructive heuristic coupled with a metaheuristic refinement. We have examined the computational complexity of the proposed methods as well as their practical performance through extensive computational experiments, showing that they are more effective than state-of-the-art methods, outclassing them on some benchmarks from the literature.

Acknowledgements

This research was supported by Air Force Office of Scientific Research (under award number FA9550-17-1-0067). We are indebted to Seiji Kataoka and Takeo Yamada for making available their upper and lower bounding computer codes and for providing assistance for their practical use. We thank two referees for useful comments.

References

- [1] S. Balbal, Y. Laalaoui, and M. Benyettou. Local search heuristic for multiple knapsack problem. *International Journal of Intelligent Information Systems*, 4:35–39, 2015.
- [2] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35:713–728, 2006.

- [3] G. Dahl and N. Foldnes. LP based heuristics for the multiple knapsack problem with assignment restrictions. *Annals of Operations Research*, 146:91–104, 2006.
- [4] M. Dawande, J. Kalagnanam, P. Keskinocak, F.S. Salman, and R. Ravi. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *Journal of Combinatorial Optimization*, 4:171–186, 2000.
- [5] J.E. Diaz, J. Handl, and D.-L. Xu. Integrating meta-heuristics, simulation and exact techniques for production planning of a failure-prone manufacturing system. *European Journal of Operational Research*, 266:976–989, 2018.
- [6] N.B. Dimitrov, D. Solow, J. Szmerekovsky, and J. Guo. Emergency relocation of items using single trips: Special cases of the multiple knapsack assignment problem. *European Journal of Operational Research*, 258:938–942, 2017.
- [7] A. Fukunaga. Integrating symmetry, dominance, and bound-and-bound in a multiple knapsack solver. In L. Perron and M.A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 5th International Conference, CPAIOR 2008 Paris, France, May 20-23, 2008 Proceedings*, pages 82–96, Berlin, Heidelberg, 2008. Springer.
- [8] A. Fukunaga. A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, 184:97–119, 2011.
- [9] A. Fukunaga and R. Korf. Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of Artificial Intelligence Research*, 28:393–427, 2007.
- [10] K. Jansen. Parameterized approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 39:1392–1412, 2009.
- [11] S. Kataoka and T. Yamada. Upper and lower bounding procedures for the multiple knapsack assignment problem. *European Journal of Operational Research*, 237:440–447, 2014.
- [12] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.
- [13] Y. Laalaoui and R. M’Hallah. A binary multiple knapsack model for single machine scheduling with machine unavailability. *Computers & Operations Research*, 72:71–82, 2016.
- [14] M.E. Lalami, M. Elkihel, D.E. Baz, and V. Boyer. A procedure-based heuristic for 0-1 multiple knapsack problems. *International Journal of Mathematics in Operational Research*, 4:214–224, 2012.
- [15] E. Lalla-Ruiz and S. Voß. *A Biased Random-Key Genetic Algorithm for the Multiple Knapsack Assignment Problem*, pages 218–222. Springer International Publishing, Cham, 2015.

- [16] S. Martello and P. Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. *Discrete Applied Mathematics*, 3:275–288, 1981.
- [17] S. Martello and P. Toth. A heuristic algorithm for the multiple knapsack problem. *Computing*, 27:93–112, 1981.
- [18] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, 1990.
- [19] D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114:528–541, 1999.
- [20] T. Yamada and T. Takeoka. An exact algorithm for the fixed-charge multiple knapsack problem. *European Journal of Operational Research*, 192:700–705, 2009.
- [21] L. Zhen, K. Wang, S. Wang, and X. Qu. Tug scheduling for hinterland barge transport: A branch-and-price approach. *European Journal of Operational Research*, 265:119–132, 2018.