European Journal of Operational Research 297 (2022) 53–65

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A two-phase kernel search variant for the multidimensional multiple-choice knapsack problem

Leonardo Lamanna, Renata Mansini, Roberto Zanotti*

Department of Information Engineering (DII), University of Brescia, Italy

ARTICLE INFO

Article history:
Received 3 August 2020
Accepted 7 May 2021
Available online 18 May 2021

Keywords:
Metaheuristics
Multidimensional multiple-choice knapsack problem
Kernel search
Matheuristic

ABSTRACT

The Multidimensional Multiple-choice Knapsack Problem (MMKP) is a complex combinatorial optimization problem for which finding high quality feasible solutions is very challenging. Recently, several heuristic approaches and a few exact algorithms have been proposed for its solution. These methods have been able to provide new best-known values for benchmark instances although many of them still remain unclosed to optimality. In this paper, we provide a new variant of the heuristic framework Kernel Search and apply it to the MMKP. The proposed variant keeps the method's main idea of solving a sequence of restricted mixed-integer subproblems but innovates by partitioning the solution process into two different phases with complementary goals. The first phase strives for feasibility and collects important information to dynamically adapt subproblems' dimension and solution time in the second phase that is focused on getting high quality solutions. This makes the global approach more scalable and efficient.

Computational results on different sets of benchmark instances demonstrate that our method is extremely effective outperforming all state-of-the-art heuristics for the MMKP. The method compares extremely well also with respect to exact approaches running for five hours. The proposed algorithm is able to improve the best-known value of 185 out of 276 open benchmark instances and gets 8 out of 17 optimal solutions for the closed ones. The average deviation from optimality is always negligible.

1. Introduction

The Multidimensional Multiple-choice Knapsack Problem (MMKP) is one of the most complex problems in the large knapsack family. It comes from the combination of the Multiple-choice Knapsack Problem (MCKP) and the Multidimensional Knapsack Problem (MDKP or MKP). In the MCKP, items are partitioned into groups, and exactly one item from each group has to be selected. In the MDKP, more than one resource is available, and each item is associated with a vector of weights representing the non-negative amounts of resources it consumes and the selected items cannot exceed the available resources capability. MDKP is a strongly NP-hard combinatorial optimization problem which arises in many application contexts such as capital budgeting, cargo loading in Shih (1979), cutting stock problems in Gilmore and Gomory (1966) and asset-backed securitization in Mansini and Speranza (2002).

Let $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ be a collection of n disjoint sets of items (groups or classes) with possibly different cardinalities $n_i = |G_i|$, $i = 1, \ldots, n$, and let $R = \{1, \ldots, m\}$ be a set of resources, each one available for a positive quantity c_r, $r = 1, \ldots, m$. Each item, denoted as a pair (i, j), $i = 1, \ldots, n$ and $j = 1, \ldots, n_i$, has a non-negative profit p_{ij} and consumes a predefined non-negative amount w_{ij}^r for each resource $r \in R$. We indicate with N the set of all items. The MMKP looks for a subset of items that maximizes the overall profit, while selecting exactly one item for each group without violating the resource constraints. MMKP can be formulated as follows:

$$\text{Maximize } z = \sum_{i=1}^{n} \sum_{j=1}^{n_i} p_{ij} x_{ij} \tag{1}$$

subject to:

$$\sum_{i=1}^{n} \sum_{j=1}^{n_i} w_{ij}^r x_{ij} \leq c_r \qquad r = 1, \ldots, m, \tag{2}$$

$$\sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1, \ldots, n, \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, n, \; j = 1, \ldots, n_i. \tag{4}$$

* Corresponding author.
E-mail addresses: l.lamanna@unibs.it (L. Lamanna), renata.mansini@unibs.it (R. Mansini), roberto.zanotti@unibs.it (R. Zanotti).

https://doi.org/10.1016/j.ejor.2021.05.007
0377-2217/

where binary variable x_{ij} takes value 1 if item (i, j) in group G_i is selected, and 0 otherwise. Constraints (2) impose that the availability of each resource is not exceeded, whereas constraints (3) ensure that exactly one item per group is selected.

Over the last decade, several papers have appeared on the MMKP, related to exact and heuristic methods. Nevertheless, researchers are still far from optimally solving many of the available benchmark instances. This is one of the main reasons, along with the applications relevant to many different problems (chip multiprocessor runtime resource management and global routing of wiring in circuits, just to name a few), for which MMKP represents an interesting playground for evaluating efficiency and effectiveness of new general purpose methods.

Kernel Search (KS) is a well-known heuristic framework initially proposed for the solution of Mixed-Integer Linear Programming (MILP) problems and successfully applied to the multi-dimensional knapsack problem by Angelelli, Mansini, and Speranza (2010), and to a portfolio selection problem by Angelelli, Mansini, and Speranza (2012). The method is based on two main components:

- the identification, in a general MILP problem, of a kernel set of variables, i.e. a set of variables that will more likely be selected (take value different from zero) in an optimal solution (*promising variables*);
- the construction and solution of a sequence of small MILP subproblems, each one taking into account the kernel set (possibly updated) plus a selected set of additional variables (the current *bucket*).

When initially proposed, the method was presented in two variants. The first one, called Basic KS, is characterized by the construction of a kernel set of variables and the solution of a sequence of MILP subproblems constructed by considering the variables not in the kernel set partitioned into buckets and sorted according to a predefined criterion. At each iteration, the kernel set is updated by including the variables belonging to the current bucket that are selected in the solution found by the current MILP suproblem (the size of the kernel set monotonically increases). The method terminates when all, or a predefined number of, buckets are considered. The second variant, called Iterative Kernel Search, modifies the Basic KS by allowing the sequence of buckets, used to construct the MILP suproblems, to be analyzed more than once (*bucket iterations*). The procedure stops when no more items are added to the kernel set.

One of the most critical aspects of the method is the way variables are selected to enter the kernel set after its initial construction, i.e. during the solution of MILP subproblems (iterative phase). A very large kernel set usually guarantees better solutions, but also implies higher computing times to solve the resulting MILP subproblems. On the contrary, a very small kernel set increases algorithm's efficiency and the probability to get a solution very quickly but worsens its performance.

Recently, Guastaroba, Savelsbergh, and Speranza (2017) propose the first innovative variant of the Kernel Search, called Adaptive Kernel Search (AKS), where the focus is put on choosing the appropriate kernel size. The method is adaptive in the sense that it tries to assess the difficulty of an instance by considering the time required to solve the first subproblem, and then adjusts the kernel size based on this instance characterization. The authors claim the resulting framework is simple allowing the user to " *... control the trade-off between solution quality and computational effort with just a few parameters*". Indeed, it is extremely hard to evaluate the difficulty of an instance by looking at the time required to solve a subproblem. There are problems where finding a feasible solution of low quality is an easy task (for instance, in the MDKP a feasible solution of bad quality can be immediately determined), although the problem is extremely difficult in practice. Finally, the initial subproblem constructed on the kernel set might be infeasible. The authors suggest to overcome the latter case by iteratively enlarging the set of variables that are part of the initial kernel until a feasible solution is reached, although such a procedure might require a large amount of time and terminate by considering the whole problem. To cope with the monotonic increase of the kernel set size, the authors exploit a parameter p corresponding to the max number of iterations allowed to a kernel variable before removing it (if no longer selected in a solution). This is a relevant issue, since one of the Kernel Search's main drawbacks is that the kernel size is non-decreasing. However, in the computational results, they set p to a value such that "*... no variable is ever removed from the kernel... "*, making the parameter almost unnecessary.

This paper provides several contributions. We introduce a new variant of Kernel Search called Two-phase Iterative Kernel Search (TIKS). The method changes both how the kernel set is constructed and how the buckets are defined and taken into account. The solution of a problem is tackled through two phases in an adaptive way: a *feasibility-oriented* phase and a *quality-oriented* one. The first phase aims at finding feasible solutions in a small amount of time, possibly scrolling the buckets as many times as possible in a coarse way. As in Guastaroba et al. (2017), we make use of an adaptivity concept to estimate the difficulty of an instance and adapt the algorithm's structure to its solution. Differently from what is done in the mentioned paper, we do not consider only the first subproblem but also the whole sequence of buckets and the solution of the resulting subproblems in the feasibility-oriented phase. The goal of the second phase is complementary to the first one. The algorithm focuses on seeking higher quality solutions. To this aim it assigns to the solution of each subproblem a larger computing time by allowing a deeper search of the solution space. To improve efficiency, the feasible (possibly of good quality) solution received as input from the first phase is used to define the kernel set and its value works as a cut-off value when solving subproblems.

The resulting method has some important features. It is extremely simple, flexible and general enough to be used to tackle other combinatorial optimization problems. Moreover, it can be easily controlled by setting only a few parameters. With respect to the original KS, TIKS has the following additional features:

1. it consists of two sequential KSs, one for each of the two phases the method is divided into;
2. it adapts dynamically the MILP subproblems' dimension by making it grow within each phase. The idea is to tackle larger size subproblems when a good quality feasible solution is at hand in order to use its value as an objective cutoff;
3. in the feasibility-oriented phase, the kernel size is redefined at each new bucket iteration in order to improve global efficiency in subproblems' solution. To this aim, the kernel variables added during a bucket iteration, but not selected in the best incumbent solution, are removed;
4. in the quality-oriented phase, the kernel set is not reduced at each bucket iteration. The idea is to keep all variables in the kernel set in order to guarantee a higher quality solution;
5. the solution time assigned to each subproblem in the quality-oriented phase depends on instance information collected from the first phase. The rationale is the following: the higher the number of subproblems of the first phase that terminate reaching the time limit without finding a feasible solution, the higher the instance difficulty and thus the larger the computing time assigned when tackling the integer subproblems in the second phase.

Computational results show the effectiveness of our approach. At present, TIKS represents the most performing solution algorithm available in the literature for the MMKP, with 185 new best-known values out of 276 instances and 8 optimal solutions. Friedman, Ne-

menyi, Wilcoxon Signed Rank, and Sign tests show that TIKS provides results that are statistically different from all the other state-of-the-art algorithms.

The paper is organized as follows. In Section 2, we analyze and classify the heuristic approaches existing in the literature for the MMKP. In Section 3, we describe the main features of the implemented method, focusing on the differential aspects with respect to known Kernel Search implementations. In Section 4, to validate our method, we compare it with the best state-of-the-art heuristic and exact algorithms on known sets of benchmark instances. We have also run some statistical tests on the results obtained by TIKS and the best performing heuristics available in the literature. Finally, in Section 5, we draw some conclusions and future developments.

2. Literature review

The recent literature provides a gamut of metaheuristic approaches for solving the MMKP. In the following, we classify them according to their main components into three different classes:

- *Relaxation-based methods.* All the methods belonging to this class exploit the problem mathematical formulation and the use of information provided by some relaxations. Moser, Jokanovic, and Shiratori (1997) make use of Lagrangian relaxation by using the multiplier method. Parra-Hernandez and Dimopoulos (2005) propose a heuristic called HMMKP which uses the Lagrangian multipliers to calculate pseudo-utility variable values and exploits them in order to find and then improve an initial feasible solution. Akbar, Rahman, Kaykobad, Manning, and Shoja (2006) study a heuristic based on convex hull; the method maps the multidimensional resource consumption to a single dimension (using vector penalty) thus reducing the search space. Hanafi, Mansi, and Wilbaut (2009) propose an iterative relaxation-based heuristic. The method is characterized by the construction of a series of small subproblems generated by exploiting information provided by different relaxations, mainly based on MILP relaxation. Cherfi and Hifi (2010) use an adaptation of a column generation approach to solve large scale MMKP instances. The method finds an initial solution and builds the restricted Integer Linear Programming (ILP) problem that contains the columns associated with such a solution. Then, both a column generation procedure and a greedy rounding solution are applied to the same ILP. Then, new nodes and subproblems are created by using some branching strategies. Crévits, Hanafi, Mansi, and Wilbaut (2012) define a semi-continuous relaxation heuristic approach where constraints are added to the LP relaxation to force variables to take values close to 0 or 1. The heuristic generates a sequence of lower and upper bounds, where upper bounds are obtained from the problem relaxation and lower bounds are obtained from subproblem resolution. Following the idea of developing an iterative relaxation-based heuristic framework, Mansi, Alves, Valério de Carvalho, and Hanafi (2013) introduce a hybrid strategy, based on the introduction of new cuts and a reformulation procedure to improve the performance of the heuristic. The authors describe some variants of the method that provide new best-known values on benchmark instances. Ren, Feng, and Zhang (2012) combine Ant Colony Optimization (ACO) with Lagrangian Relaxation (LR). The proposed method applies ACO in order to generate solutions and exploits some repair operator to convert possibly infeasible solutions into feasible ones, and then further improve the feasible ones. Lagrangian Relaxation is used to compute a domain-based heuristic factor that, along with a pheromone measure, is used to determine solutions in ACO. Finally, Hifi and Wu (2015) propose a Lagrangian heuristic-based neighborhood search. A Lagrangian relaxation of the MMKP is used to generate a series of neighborhoods, which are explored by using a local search based on a reducing strategy to intensify search, and a moving strategy to diversify. In Caserta and Voß (2019), the authors describe a novel method based on Lagrangian framework. Information provided by the Lagrangian relaxation is exploited to identify a “core” subproblem. Next, an exogenous constraint is added in order to explore a reduced portion of the search space using a MILP solver.
- *Local search and metaheuristics.* Frequently, a combination of a smart sorting of the items considering resources consumption and a simple local search can produce effective results. Khan, Li, Manning, and Akbar (2002) exploits Toyoda’s concept of aggregate resources (see Toyoda, 1975) to sort the not-yet-picked items by taking into account either the current level of resources used or the resources requirement of the items. The greedy heuristic starts with finding a feasible solution and is followed by a local search based on the exchange of items belonging to the same group, focusing on the resource saving and the increase in the solution value. Voudouris and Tsang (1999) present a local search method called Guided Local Search (GLS). A set of features for the candidate solutions is defined and, when LS is trapped into local optima, certain features are selected and penalized. LS searches the solution space by using the objective function augmented by the accumulated penalties. Hifi, Michrafy, and Sbihi (2004) propose two heuristic approaches, a Constructive Procedure (CP) and a local search based on a swapping move preserving feasibility, called Complementary Constructive Procedure (CCP). Trajectories of the solutions are oriented by decreasing the cost function with a penalty term. In Hifi, Michrafy, and Sbihi (2006), the same authors further extend the ideas developed in Hifi et al. (2004) and present two variants of a Reactive Local Search (RLS). The first method considers an initial solution and uses a fast sequential procedure of single and double swaps to improve and diversify the solution found by CCP. In the second method, called Modified RLS (MRLS), a tabu list is considered to avoid cycling. Guo, White, Wang, Li, and Wang (2011) propose a Genetic Algorithm as well as Sasikaladevi and Arockiam (2012), whereas Hiremath and Hill (2013) apply a standard tabu search with recency based memory, creating a method called First Level Tabu Search (FLTS). Shojaei, Basten, Geilen, and Davoodi (2013) propose a new parameterized Compositional Pareto-algebraic Heuristic (CPH) based on the principles of Pareto algebra. The adjective compositional indicates the presence of incremental computation of solutions. The method iteratively combines pairs of groups by using the product-sum operation that takes all possible combinations of items from the two groups, adding their values and resource usage per dimension. The resulting elements can be seen as partial solutions to the MMKP instance. This procedure is iteratively repeated. A partial solution that does not satisfy resource constraints or is dominated by some other partial solution is discarded.
- *Variable fixing procedures*. These methods may take advantage of a variable fixing procedure aiming at reducing the original problem size. In Gao, Lu, Yao, and Li (2017) a two-step iterative procedure is proposed. The authors make use of the concept of “pseudo-gap” and sum up the reduced costs of subsets of non-basic variables to derive new cuts. Such cuts are then added to the problem that is solved, in the second step, by a MILP solver. Chen and Hao (2014) introduce the well-known “reduce and solve” heuristic which combines two main ingredients, simple techniques for problem reduction (based on group fixing and variable fixing rules), and the optimal solution of the resulting ILP by means of a MILP solver. Two different variants of the approach, called PEGF and PERC, are devised according to two

different ways of combining the group and the variable fixing. Both fixing procedures take into account information provided by the linear relaxation.

In the literature a relevant place has been played also by exact methods. The most performing algorithm in this class has been recently proposed in Mansini and Zanotti (2020). The method, named YACA, exploits the concept of core as introduced by Mansini and Speranza (2012). The items in the core are the ones for which we find difficult to determine whether they will be selected in an integer optimal solution. YACA uses core items to build restricted subproblems of increasing size, exactly solved through a recursive variable-fixing procedure until an optimality condition is satisfied. The method strongly relies on a logic cut formulated from reduced costs to fix variables in the recursive process. In the computational result section, our heuristic algorithm will also be compared to the solutions obtained by this exact method on the same benchmark instances.

Finally, to place TIKS in the aforementioned classification of the existing literature, we can say that it belongs mainly to the first class since it exploits the mathematical formulation of the problem and information provided by the LP relaxation. Indeed, TIKS builds and solves a sequence of restricted subproblems and uses the reduced costs as a measure of efficiency to identify the most promising variables and sort them. Furthermore, since it iteratively tries to fix the value of some variables when a new feasible solution value is found, the method can be seen as an approach belonging to the last class. On the other side, the idea of adjusting parameters in an adaptive way through the two phases makes the method an advanced metaheuristic.

3. The solution algorithm

In this section, we briefly describe the basic Kernel Search as initially proposed, emphasizing its pros and cons. Finally, we focus on the new variant and its features.

In the basic Kernel Search, as initially proposed in Angelelli et al. (2010) and Angelelli et al. (2012), the continuous relaxation of the original problem is solved and used to identify an initial kernel set of variables (Λ) and the partition of remaining ones into a predefined number q of groups called buckets ($B_i, i = 1, \ldots, q$). Kernel set represents the set of variables that, with higher probability, will take a value larger than zero in the integer optimal solution (*promising variables*). They include all variables taking a positive value in the continuous relaxation optimal solution plus some additional variables selected among those with lower reduced costs (in absolute value). An initial solution is then obtained by tackling a restricted integer problem formulated on the initial kernel set of variables and by setting to zero all the remaining ones. Afterwards, a sequence of *MILP subproblems* is solved by means of a MILP solver, each one restricted to a subset of variables Λ_i including the ones in the kernel set Λ plus the ones belonging to the current bucket B_i. To improve efficiency, an additional constraint imposing the selection of at least one variable from B_i is added. From now on, we will refer to this constraint as to the *bucket constraint*. The best solution value (incumbent integer solution value) found in solving each subproblem is used as cut-off value for the subsequent ones. Variables belonging to a bucket and selected in the solution of the corresponding subproblem enter the kernel set Λ. The size of the kernel set increases monotonically.

The goal of solving each restricted subproblem is two-fold: possibly finding a better feasible solution, and identifying further variables to insert into the kernel set Λ. The idea is that the final kernel set should contain most of (hopefully all) the variables that will be selected in an optimal solution. Although restricted, it might be the case that solving the subproblems comes out to be computationally cumbersome. A solution time limit can be set to tackle subproblems. The best feasible solution, possibly the optimal one (if any is found), is provided as output. If one subproblem is proven to be infeasible, the next one is considered. The algorithm stops as soon as the subproblem constructed on the last bucket is tackled.

The iterative variant of the method, called Iterative Kernel Search, (IKS) allows to scroll the buckets more than once. As soon as the last bucket has been considered, provided that a new solution has been found and the kernel set has been updated, the method restarts from the first bucket until no new variables enter the kernel set.

On one side, Kernel Search has some evident advantages. It is problem-independent and thus can be applied to a wide variety of problems provided that a mathematical formulation is given. The method requires very little implementation effort because most of the work is done by a standard MILP solver called to solve the restricted subproblems. Finally, it is a robust method as a change in the model (e.g. addition of new constraints) would not require any relevant change in the heuristic. On the other side, the effectiveness of the method is strongly influenced by the potential infeasibility which can arise when the original problem is split into subproblems. Moreover, effectiveness and efficiency of Kernel Search highly depend on how variables are sorted and selected to enter the kernel set and the buckets, the size of subproblems (they should not be too large to be easily solvable, but not too small to avoid a quality downgrade) and the type of buckets used (overlapping or disjoint sets of variables). A wrong choice of the sorting rule or of the size of the buckets can affect badly the results obtained.

3.1. The two-phase iterative kernel search

We introduce a new variant of IKS, that tries to cope with the main drawbacks emphasized above. The proposed method encompasses two main phases that complement each other. The first phase focuses on feasibility trying to quickly find feasible solutions, whereas the second one concentrates on getting high quality solutions. In each phase a different implementation of IKS is run. Each phase is characterized by its own bucket size and its kernel set construction. Within the same iteration, buckets have all the same size and do not overlap. Nevertheless, in order to diversify buckets construction, we increase bucket size of a fixed amount when a new bucket iteration starts. When solving each subproblem, a time limit on the computing time is set and a *bucket constraint* as well as a cut-off constraint on the best integer solution found so far are added. Each subproblem resolution terminates with one of four different values of status. We say the status is *feasible* if the subproblem resolution terminates by providing a feasible (not proved to be optimal) solution. If no feasible solution is obtained and the time limit is reached, the status is *timeLimit*. If the subproblem results to be infeasible due to the cutoff value provided as input, its status is *cutoff*. Finally, status is *optimal* if the subproblem solution terminates successfully closing to optimality the subproblem before reaching the time limit. The algorithm stops when a maximum computing time $t_{\max}$ is reached. In particular:

1. **Feasibility-oriented phase (Phase 1)**. This phase is strongly devoted to finding feasible solutions in a coarse way, without caring too much about their quality. A time limit t_1 is assigned to each subproblem solution and the whole phase terminates as soon as a predefined percentage γ of the total resolution time $t_{\max}$ has elapsed. The initial kernel set contains all variables with positive value in the continuous relaxation. The initial size of the buckets is set equal to a predefined parameter value b_1. Once the buckets have been constructed, they are used (along the kernel set) one after the other to construct restricted sub-

problems. When solving a restricted subproblem, the variables selected in the solution and belonging to the bucket used for its construction are added to the kernel set. At the end of each bucket iteration, the kernel set is updated by removing all variables entered from the buckets but not selected in the incumbent integer solution. This allows to limit the kernel set size and the dimension of subproblems to be solved in the subsequent bucket iteration. To change buckets construction, at the end of each bucket iteration, the bucket size is increased by a fixed amount Δ_1.

This phase plays an important role in evaluating the difficulty of the instance at hand and provides information to set parameters for a more effective search in the second phase. If the percentage of subproblems, out of all the tackled ones, that terminate with a status different from *timeLimit* is greater than or equal to a predefined value θ, then the instance is classified as *fast*, otherwise it is *slow*. The latter type is featured by subproblems quite difficult to solve, where finding a feasible solution is a *slow process* either because the size of the subproblem is too large or the allotted time is too short. Thus, the subsequent quality-oriented phase is characterized by a lower initial bucket size b_2^{slow}, a higher computing time limit t_2^{slow}, and a lower increase of the bucket size Δ_2^{slow} at the end of each bucket iteration.

The reverse is true if the instance is classified as *fast*, since the percentage of subproblems with a positive exit (feasible solution, optimal solution or cut-off) is larger. In such a case, in the second phase, we set a initial bucket size $b_2^{fast} \geq b_2^{slow}$ and a higher increase of the bucket size Δ_2^{fast} at the end of each bucket iteration. Since the second phase is devoted to quality, the computing time limit is also increased for the fast instances to a value $t_2^{fast} > t_1$, but smaller than t_2^{slow}.

2. **Quality-oriented phase (Phase 2)**. This phase differs from the initial one for parameters setting and is mainly directed to an exhaustive search of the subproblems to find high quality solutions. For this reason, a larger computing time than in the first phase is assigned to each subproblem solution. Differently from Phase 1, the kernel set is not shrinked at the end of each bucket iteration, but keeps all the bucket variables taking value different from zero when solving subproblems due to the presence of the bucket constraint.

Algorithm 1 provides the pseudo-code of TIKS. It receives as input an instance $\mathcal{I}$ of the problem and the time limit $t_{\max}$ assigned to the algorithm. It produces as output the best feasible solution found $\mathbf{x}^*$ and its value z^*. In Line 1, the values of all parameters are initialized. In particular, there are three tuples of parameters, one associated with the first phase of the algorithm and two with the second one (one for the slow and one for the fast case). Each tuple indicates the initial bucket size, its size increase at each bucket iteration, and the time limit imposed to solve each subproblem.

The algorithm starts by solving the linear programming relaxation problem through function SolveLPRelaxation in Line 2. The optimal solution $\mathbf{x}^{LP}$ and its value z^{LP} as well as the value of reduced costs $\mathbf{r}^{LP}$, associated with such a solution, are used to sort items (and corresponding variables) by function SortItems (Line 3). We consider the classical sorting rule used by Kernel Search: variables in the basis are sorted in non-increasing order of their LP relaxation value, whereas out-of-the-basis variables are sorted in non-decreasing order of their absolute reduced cost values.

In Line 4, function SelectKernelSet identifies the composition of the kernel set Λ, by selecting a predefined number of variables among the first ones in the ordered set N, as the most promising. In our case, Λ consists of all variables with a strictly positive value in the LP relaxation optimal solution.

Algorithm 1 TIKS($\mathcal{I}, t_{\max}$).

1: Initialize parameters (b_1, Δ_1, t_1), $(b_2^{slow}, \Delta_2^{slow}, t_2^{slow})$, $(b_2^{fast}, \Delta_2^{fast}, t_2^{fast})$, γ, θ
2: $(\mathbf{x}^{LP}, \mathbf{r}^{LP}, z^{LP}) \leftarrow$ SolveLPRelaxation($\mathcal{I}$)
3: $N \leftarrow$ SortItems($\mathcal{I}, \mathbf{x}^{LP}, \mathbf{r}^{LP}$)
4: $\Lambda \leftarrow$ SelectKernelSet($\mathcal{I}, \mathbf{x}^{LP}, \mathbf{r}^{LP}, N$)
5: $\bar{\Lambda} \leftarrow \Lambda$
6: $n_{sub}, n_{pos} \leftarrow 0$
7: $phase_1 \leftarrow true$
8: $t_{\max}^1 \leftarrow \gamma t_{\max}$
9: $(b, \Delta, t) \leftarrow (b_1, \Delta_1, t_1)$
10: $(\mathbf{x}^*, z^*, \bar{S}, t_{elapsed}, status) \leftarrow$ Solve($\mathcal{I}, \Lambda, \emptyset, \emptyset, \emptyset, 0, t$)
11: **if** $status$! = $timeLimit$ **then**
12: $n_{pos} \leftarrow n_{pos} + 1$
13: **end if**
14: **if** $status = feasible$ **or** $status = optimal$ **then**
15: $(N_1^*, N_0^*) \leftarrow$ VariableFixing($z^*, z^{LP}, \mathbf{x}^{LP}, \mathbf{r}^{LP}$)
16: $(\bar{\Lambda}, N \setminus \bar{\Lambda}) \leftarrow$ RemoveOptimalItem($N_1^*, N_0^*, N, \bar{\Lambda}$)
17: **end if**
18: $n_{sub} \leftarrow n_{sub} + 1$
19: **while** $t_{elapsed} < t_{\max}$ **do**
20: $\mathcal{B} \leftarrow$ BuildBuckets($N \setminus \bar{\Lambda}, b$)
21: **for all** $B \in \mathcal{B}$ **do**
22: $t \leftarrow \min(t, t_{\max} - t_{elapsed})$
23: $(\mathbf{x}^{ILP}, z^{ILP}, \bar{S}, t_{used}, status) \leftarrow$ Solve($\mathcal{I}, \bar{\Lambda} \cup B, B, N_1^*, N_0^*, z^*, t$)
24: **if** $status$! = $timeLimit$ **then**
25: $n_{pos} \leftarrow n_{pos} + 1$
26: **end if**
27: $n_{sub} \leftarrow n_{sub} + 1$
28: $t_{elapsed} \leftarrow t_{elapsed} + t_{used}$
29: **if** $status = optimal$ **or** $status = feasible$ **then**
30: $(\mathbf{x}^*, z^*) \leftarrow (\mathbf{x}^{ILP}, z^{ILP})$
31: $\bar{\Lambda} \leftarrow \bar{\Lambda} \cup \bar{S}$
32: $(N_1^*, N_0^*) \leftarrow$ VariableFixing($z^*, z^{LP}, \mathbf{x}^{LP}, \mathbf{r}^{LP}$)
33: $(\bar{\Lambda}, N \setminus \bar{\Lambda}) \leftarrow$ RemoveOptimalItem($N_1^*, N_0^*, N, \bar{\Lambda}$)
34: **end if**
35: **end for**
36: $b \leftarrow b + \Delta$
37: **if** $phase_1 = true$ **then**
38: $\bar{\Lambda} \leftarrow$ UpdateKernelSet($\Lambda, \mathbf{x}^*$)
39: **end if**
40: **if** $t_{elapsed} \geq t_{\max}^1$ **and** $phase_1 = true$ **then**
41: **if** $\frac{n_{pos}}{n_{sub}} \geq \theta$ **then**
42: $(b, \Delta, t) \leftarrow (b_2^{fast}, \Delta_2^{fast}, t_2^{fast})$
43: **else**
44: $(b, \Delta, t) \leftarrow (b_2^{slow}, \Delta_2^{slow}, t_2^{slow})$
45: **end if**
46: $phase_1 \leftarrow false$
47: $t \leftarrow \min(t, t_{\max} - t_{elapsed})$
48: $(\mathbf{x}^*, z^*, \bar{S}, t_{used}) \leftarrow$ Solve($\mathcal{I}, \bar{\Lambda}, \emptyset, N_1^*, N_0^*, z^*, t$)
49: $t_{elapsed} \leftarrow t_{elapsed} + t_{used}$
50: **end if**
51: **end while**
52: **return** $(\mathbf{x}^*, z^*)$

The necessary parameters to start the feasibility-oriented phase are then set (Lines 7–9). The boolean flag $phase_1$, that identifies whether we are in the feasibility phase or not, is set to true, whereas the time limit of this phase, $t_{\max}^1$, is set as a percentage γ of the total time. Parameters (b, Δ, t) are then set to (b_1, Δ_1, t_1).

These three values are specific for the feasibility-oriented phase, and will be changed in the quality-oriented one.

TIKS iteratively solves different subproblems, each one associated with a predefined set of variables. A subproblem is formulated by restricting the original problem to the selected subset of variables while setting the remaining ones to zero. Function Solve calls the MILP solver at hand to tackle the current subproblem. It takes as inputs the instance of the problem, the subset of variables to construct the subproblem, an additional subset of variables corresponding to the current bucket (possibly empty as in the initial subproblem), a subset of variables that are permanently fixed to 1 (possibly empty), a subset of variables that are permanently fixed to 0 (possibly empty), a cutoff value (representing the incumbent feasible solution value), and the maximum amount of time allowed to solve the subproblem. The set of variables permanently fixed is determined by the procedure VariableFixing that makes use of the classical fixing procedures based on comparing the absolute reduced cost values to the gap between the LP relaxation value and the value of a feasible solution (further details about this fixing rules applied to MMKP can be found in Chen and Hao (2014) and Mansini and Zanotti (2020)). Function Solve produces as outputs the best solution (if any) identified within the time limit and its value or NULL if no solution is found, the subset of variables $\bar{S}$ that have been selected in the current bucket (if any is considered) due to the bucket constraint, the time used to solve the subproblem $t_{elapsed}$, and the status of the optimization.

In Line 10, the algorithm tackles the first subproblem considering the variables in the set Λ while setting the remaining ones to zero. Notice that, before solving the first subproblem no feasible solution is available. Thus, in this first call of function Solve, the additional set of variables corresponding to the bucket and the sets of variables permanently fixed to 1 and 0 are empty, the cutoff value is 0, and thus set $\bar{S}$ is empty. If the first subproblem terminates with a status different from *timeLimit*, the counter associated with a positive termination n_{pos} is incremented by one (Lines 11–13). If a feasible solution is found, the method calls the VariableFixing procedure to check if the gap between the current lower bound value (represented by the value of the feasible solution at hand) and the upper bound value (provided by the optimal solution of the LP relaxation) is lower than the absolute value of the reduced cost of some of the out-of-the-basis variables. If this is the case, the value taken by these variables in the optimal solution of the LP relaxation is optimal for the integer problem as well. These variables can thus be removed from the problem. The routine provides as output the sets of items N_1^* and N_0^*, possibly void, referring to variables that are set to their optimal value 1 or 0, respectively. Such variables are removed from both the kernel set $\bar{\Lambda}$ and the sorted set of variables $N \setminus \bar{\Lambda}$. This is done by function RemoveOptimalItem that provides as output the two updated sets. Note that, once some variables have been fixed to optimality, their values are correctly taken into account when solving subsequent subproblems.

If *status* comes out to be equal to *timeLimit* and no feasible solution is found, the algorithm goes straight to Line 18 and then moves to the next subproblem solution. Notice that, in Line 18, the counter n_{sub} indicating the number of tackled subproblems is updated. At each iteration of the main while-loop (Lines 19–51), the algorithm partitions the variables not belonging to the kernel set into a sequence $\mathcal{B}$ of buckets, each of size b (Line 20). Then, for each bucket $B \in \mathcal{B}$, a new subproblem that considers the variables in the kernel set plus the ones in B is solved (Line 23), with a time limit t. If a solution that has a value better than the incumbent one is found, the incumbent is updated (Line 30), the variables from B that have been selected in this solution are added to the kernel set (Line 31), and functions VariableFixing and RemoveOptimalItem are run. After all the buckets have been considered, b is increased by Δ (Line 36). If the algorithm is in the feasibility-oriented phase, the kernel set is updated (Line 38): the new kernel set $\bar{\Lambda}$ now contains only the variables of the original kernel set Λ plus the variables selected in the best integer solution $\mathbf{x}^*$. Then, if the algorithm is in the feasibility-oriented phase and the time given to such a phase has run out, the switch to the quality-oriented phase is performed: b, Δ, and t are updated (Lines 41–45).

If the problem is classified as slow, the tuple of parameters $(b_2^{slow}, \Delta_2^{slow}, t_2^{slow})$ is used (Line 42); otherwise the tuple $(b_2^{fast}, \Delta_2^{fast}, t_2^{fast})$ is chosen (Line 44). A subproblem that considers only the current kernel set is then solved (Line 48). The algorithm terminates by returning the best integer solution found and its objective function value.

4. Experimental analysis

In this section, we report the results obtained by TIKS when solving different sets of benchmark instances. TIKS is coded in Java 8 and Gurobi 9.0 is used as a MILP solver. All tests have been run on a PC with an Intel Core I7-5930K 3.5 GHz processor and with 64 GB of RAM.

We set the algorithm's stopping rule to $t_{\max} = 1200$ s. A fraction $\gamma = \frac{1}{3}$ is assigned to Phase 1 and the remaining time to Phase 2. After some preliminary tests, we have finally decided the setting of parameters as follows. The initial bucket size should not be larger than 200 and reduce when the number of resources m, the number of groups n, and the average number of items per group n_{med} increase. Thus we set $b_1 = \frac{200}{Log(n \cdot m \cdot n_{med}) - 3}$. In the first phase, the increment of the bucket size is set to $\Delta_1 = 0.1 b_1$ and the time limit t_1 for each subproblem solution is set to 40 seconds. We set parameter θ equal to $\frac{1}{3}$. An instance is classified as *fast* if $\frac{n_{pos}}{n_{sub}}$ is greater than or equal to θ and *slow* otherwise. If an instance is *slow*, $(b_2^{slow}, \Delta_2^{slow}, t_2^{slow}) = (\frac{b_1}{2}, \frac{\Delta_1}{2}, 5t_1)$. On the other side, if the instance is classified as fast, we set $(b_2^{fast}, \Delta_2^{fast}, t_2^{fast}) = (b_1, \Delta_1, 3t_1)$.

4.1. Benchmark instances

We validate our algorithm by using all the sets of instances available in the literature for the MMKP. The benchmark instances are characterized by different values of the number of resources (m), the number of total groups (n), the number of total items ($|N|$), and the number of items for each group (n_i, $i = 1, \ldots, n$). In particular, the tested collection of instances includes:

- **Class A**: a set of 7 instances (I07–I13) proposed by Khan et al. (2002) with $m = 10$, n_i set to a constant value $\bar{n} = 10$ for all $i = 1, \ldots, n$, and n raising from 100 to 400 with step 50. The instances are correlated, since the profit of each item depends on the amount of resources it consumes.
- **Class B**: a set of 20 instances (INST01-INST20) introduced by Hifi et al. (2006) according to the procedure used for the instances of Class A. In particular, $m = 10$, n ranges in the discrete interval $[50, 350]$, and n_i is set to a constant value $\bar{n} = 10, 20$, and 30 for all $i = 1, \ldots, n$.
- **Class C**: a set of 10 instances (INST21-INST30) introduced by Shojaei et al. (2013). The instances have an irregular structure since the number n_i of items included in each group i is no longer a constant. Moreover an item does not necessarily make use of all the resources. The set of resources is set equal to $m = 10, 20, 30$, and 40, whereas $n = 100, 200, 300, 400$, and 500 and the maximum number of items per group $n_{\max}$ is set equal to 10 or 20.

Table 1
Class A.

Instance	m	n	$\bar{n}$	$\lvert N\rvert$
I07	10	100	10	1000
I08	10	150	10	1500
I09	10	200	10	2000
I10	10	250	10	2500
I11	10	300	10	3000
I12	10	350	10	3500
I13	10	400	10	4000

Table 2
Class B.

Instance	m	n	$\bar{n}$	$\lvert N\rvert$
INST01	10	50	10	500
INST02	10	50	10	500
INST03	10	60	10	600
INST04	10	70	10	700
INST05	10	75	10	750
INST06	10	75	10	750
INST07	10	80	10	800
INST08	10	80	10	800
INST09	10	80	10	800
INST10	10	90	10	900
INST11	10	90	10	900
INST12	10	100	10	1000
INST13	10	100	30	3000
INST14	10	150	30	4500
INST15	10	180	30	5400
INST16	10	200	30	6000
INST17	10	250	30	7500
INST18	10	280	20	5600
INST19	10	300	20	6000
INST20	10	350	20	7000

Table 3
Class C.

Instance	m	n	$n_{\max}$	N
INST21	10	100	10	565
INST22	20	100	10	538
INST23	30	100	10	541
INST24	40	100	10	584
INST25	10	100	20	871
INST26	20	100	20	842
INST27	10	200	10	1076
INST28	10	300	10	1643
INST29	10	400	10	2223
INST30	10	500	10	2704

Table 4
Class D.

m	n	$\bar{n}$	$\lvert N\rvert$
10–25	100–700	10–25	1000–17500

- **Class D**: a set of 256 instances (MAVCH1-MAVCH256) introduced by Mansi et al. (2013) according to the method described in Khan (1998). For each combination of number of resources, groups, and items per group, four random instances have been generated. Resources are set to $m = 10, 15, 20, 25$, number of groups to $n = 100, 250, 500, 700$, and items per group n_i set to a constant value equal to $\bar{n} = 10, 15, 20,$ and 25 for all groups $i = 1, ., n$.

Features of all the classes are summarized in Tables 1–4.

4.2. Computational results

In this section, we compare TIKS with the most recent and best performing state-of-the-art heuristics and exact algorithms for the MMKP. It is worth noticing that only the most recently proposed algorithms have been tested on instances in Class D. Also Class C has not been widely used as a benchmark set.

In the following, we provide a brief overview of the methods that are considered throughout this section. We have decided to only include those algorithms that have achieved at least one best-known result on some benchmark instances. In particular:

- CHMW*: Crévits et al. (2012) design four methods, ILPH, IMIPH, IIRH, and ISCRH, respectively. For each instance, we report the best result value (indicated as CHMW* by the name of the authors) found by the four proposed variants. The methods, originally coded in C++, have been executed on a Pentium IV 3.4 GHz processor with 1 core and 4 GB of RAM. The time limit is 1 h for each variant.
- MACH*, MACH3: Mansi et al. (2013) introduce three variants MACH1, MACH2, and MACH3 of the same algorithm. For each instance in Classes A and B, we consider the best value, indicated as MACH*, found by these three approaches. The authors tested only MACH3 for Class D. The algorithms are coded in C++ and run on a Dell computer with a 2.4 GHz processor and 4 GB of RAM. Cplex 11.2 is used to solve the LP relaxation and the reduced problem at each iteration of the methods. Time limit is set to 500 seconds for each of the three variants in Classes A and B, whereas it is increased to 600 seconds in Class D.
- CPH*, CPH+OptPP*: Shojaei et al. (2013) propose two versions of their algorithm (pCPH and CPH+OptPP). In Classes A and B, we consider the best solution value (indicated as CPH*) found for each instance by these two versions. The time limit is set to 1200 seconds for each algorithm. In Class C, the authors test two variants of CPH+OptPP with a time limit equal to 1 h. We consider the best solution value (indicated as CPH+OptPP*) obtained by these two variants. The algorithms are coded in C on a PC with a 2.8 GHz Intel CPU and 12 GB of RAM.
- PE*: Chen and Hao (2014) propose two methods PERC and PEGF. For each instance we report the best result value (indicated as PE*) found by the two variants tested with a time limit of 1 h each. The algorithms are coded in C. Authors have run the tests on a PC with an Intel Xeon E5440 2.83 GHz processor with 4 cores and 2 GB of RAM. Cplex 12.4 is used to solve LP relaxations and integer subproblems. No results are available on Class D instances.
- LHBNS*, LHBNS2: Hifi and Wu (2015) test three variants of their heuristic approach for Classes A and B, with a time limit of 1 h each. For each instance we report the best result value (indicated as LHBNS*) found by these three variants. In Class D, we report the result of the most performing variant (LHBNS2), with a time limit equal to 1 h. The algorithms are implemented in C++ and tested on an Intel Pentium Core i5-2500 3.3 GHz processor with 4 cores. Auxiliary problems are solved using Cplex 12.4.
- IPGE: proposed by Gao et al. (2017). We report the results obtained by the authors with a time limit of 1 h. The algorithm is implemented in C++, compiled using Visual Studio 2010 with Cplex 12.5. Experiments are carried out on an Intel Core i3-3220 3.3 GHz CPU machine with 2 cores and 4 GB RAM, running a Windows 7 32 Bit system.
- YACA: introduced by Mansini and Zanotti (2020). This is an exact algorithm. We report the results obtained by the authors with a time limit of 5 h for Classes A, B, and C, and of 1 h for Class D. YACA is coded in Java 8 and executed on an Intel Core i7 2.93 GHz with 4 cores and 8 GB of RAM. Gurobi 6.5.1 is used to solve restricted subproblems.

Table 5
TIKS and TIKS^2 vs state-of-the-art heuristic methods: Classes A and B.

Instance	TIKS (1200s 6 cores)	TIKS^2 (3600s 2 cores)	IPGE (3600s 2 cores)	LHBNS* (3·3600s 4 cores)	PE* (2·3600s 4 cores)	MACH* (3·500s -)	CPH* (2·1200s -)	CHMW* (4·3600s 1 core)
I07	**24,595**	**24,595**	**24,595**	24,592	24,592	24,590	24,592	24,587
I08	**36,895**	36,894	36,893	**36,895**	36,894	36,888	36,886	36,889
I09	49,188	**49,189**	49,187	49,182	49,185	49,182	49,185	49,183
I10	**61,481**	61,480	61,479	61,480	61,478	61,480	61,465	61,471
I11	**73,792**	73,791	73,791	73,787	73,791	73,789	73,782	73,779
I12	86,095	86,095	86,094	**86,097**	86,095	86,094	86,084	86,083
I13	98,443	98,443	98,443	98,444	98,441	98,440	98,437	**98,445**
INST01	**10,738**	**10,738**	**10,738**	**10,738**	**10,738**	**10,738**	10,733	10,728
INST02	**13,598**	**13,598**	**13,598**	**13,598**	**13,598**	**13,598**	**13,598**	**13,598**
INST03	**10,955**	**10,955**	10,952	10,949	10,947	10,949	**10,955**	10,943
INST04	**14,457**	14,456	14,456	14,445	14,456	14,456	14,452	14,445
INST05	17,061	**17,065**	17,061	17,060	17,061	17,057	17,059	17,055
INST06	**16,845**	16,838	16,843	16,834	16,840	16,835	16,830	16,832
INST07	16,442	**16,444**	16,442	16,440	16,440	16,442	16,440	16,440
INST08	17,518	17,518	**17,521**	17,512	17,514	17,508	17,509	17,511
INST09	17,762	17,762	**17,763**	17,761	**17,763**	17,760	17,757	17,760
INST10	**19,320**	19,318	**19,320**	19,315	19,316	19,316	19,316	**19,320**
INST11	**19,449**	**19,449**	19,446	19,439	**19,449**	19,441	19,441	19,446
INST12	**21,744**	21,743	21,742	21,737	21,741	21,738	21,738	21,738
INST13	**21,580**	**21,580**	**21,580**	21,577	21,578	21,577	21,577	**21,580**
INST14	32,874	**32,875**	32,873	32,874	**32,875**	32,872	32,872	32,872
INST15	**39,165**	**39,165**	39,163	**39,165**	39,162	39,161	39,161	39,160
INST16	43,366	43,366	**43,367**	43,366	**43,367**	43,366	43,363	43,363
INST17	54,362	54,362	**54,363**	54,361	**54,363**	**54,363**	54,360	54,360
INST18	**60,469**	60,468	**60,469**	60,468	60,467	60,467	60,465	60,467
INST19	64,931	**64,933**	**64,933**	64,931	64,932	64,932	64,932	64,932
INST20	75,617	75,616	75,616	75,615	75,616	**75,618**	75,615	75,611
#Best	15	12	11	5	7	4	2	4
#Ubest	5	3	1	1	0	1	0	1
Sum OBJ	1,018,742	1,018,736	1,018,728	1,018,662	1,018,699	1,018,657	1,018,604	1,018,598
Gap	0.006%	0.007%	0.008%	0.021%	0.014%	0.020%	0.025%	0.030%

Table 6
TIKS and TIKS^2 vs state-of-the-art heuristic methods: Class C.

Instance	TIKS (1200s 6 cores)	TIKS^2 (3600s 2 cores)	IPGE (3600s 2 cores)	PE* (2·3600s 4 cores)	CPH+OptPP* (2·3600s -)
INST21	44,282	44,280	**44,284**	44,280	44,270
INST22	**41,984**	41,980	41,964	41,966	41,976
INST23	**42,600**	42,592	42,536	42,584	42,562
INST24	41,964	41,938	**41,998**	41,876	41,918
INST25	44,156	44,156	44,156	**44,159**	44,156
INST26	44,874	44,872	44,869	**44,879**	44,869
INST27	87,632	87,630	**87,634**	87,630	87,616
INST28	134,652	**134,654**	**134,654**	134,642	134,634
INST29	**179,230**	179,228	179,222	179,228	179,206
INST30	214,230	214,230	**214,242**	214,230	214,198
#Best	3	1	5	2	0
#Ubest	3	0	4	2	0
Sum OBJ	875,604	875,560	875,559	875,474	875,405
Gap	0.014%	0.024%	0.025%	0.042%	0.045%

4.3. Comparison with state-of-the-art-heuristics

The first part of the computational results analysis takes into account a comparison of TIKS with the most performing heuristic methods. Tables 5 and 6 refer to the results obtained on the first three classes and have the same structure. For each instance, the tables report the value found by each method. The column captions also indicate the time limit and the number of processing cores used originally by each method. For instance, the time associated with PE* is 7200s since PE* takes the best value between the results obtained by two methods running each one for 1 h. Similarly, the time for CHMW* is 4 times 3600s combining the results of four approaches running for 1 h each, whereas MACH* has a time limit of 1500s, since each one of the three methods MACH1, MACH2, and MACH3 runs for 500s. Finally, TIKS has a time limit of only 20 min. In order to perform a fair comparison and evaluate the impact of the computing platform on the results, we have also tested TIKS with a 3600s time limit, but only using 2 of the 6 available cores on the machine. We have not performed any parameter tuning for this version (which might have led to better performance), and used the same parameters setting as the original TIKS. This version is labeled as TIKS^2.

Each entry of line *#Best* indicates the number of times, out of all solved instances, a method has reached the best value comparatively to the other ones, whereas the line *#UBest* specifies the number of times, out of the ones in *#Best*, a method is the only one to achieve such a value. When an algorithm gets the best value the corresponding figure is emphasized in bold. Line Sum OBJ provides the sum of the solution values of all the instances in the class. This figure provides an immediate evaluation of the global performance allowing a ranking of the methods in terms of global solution value instead of number of best solutions found. Entries in line Gap report the average percentage deviation (out of all instances) of each algorithm solution value with respect to the best-known value in each instance.

Table 5 shows the results for classes A and B. In these instances, TIKS is able to find the best solution value in 15 instances out of 27, strictly being the best method in 5 cases and having the highest Sum OBJ value. The method is also the one with the lowest average gap from the best-known values. With TIKS^2, the drop in performance is minimal. The method reaches 12 best values, and 3 unique best values. It is interesting to notice that, in some cases, TIKS^2 obtains better results than TIKS. In particular, for instance INST05, it obtains a new best-known value. IPGE, as well as PE*, reaches good quality solutions. In particular, IPGE hits the

Table 7
TIKS vs LHBNS2 on Class D instances.

		TIKS (1200s 6 cores)	LHBNS2 (3600s 4 cores)
m	10	63	0
	15	59	2
	20	58	4
	25	58	6
n	100	59	2
	250	59	4
	500	59	3
	700	61	3
$\bar{n}$	10	57	4
	15	61	2
	20	59	4
	25	61	2
Total		238	12
Sum OBJ		5,21,01,161	5,20,91,810
Gap		0.003%	0.027%

best value in 11 instances while beating all the other methods in 1 instance. At the double of the time required by IPGE, the same figures for PE* reduce to 7 and 0, respectively. CPH and CHMW* are the less performing, whereas MACH* and LHBNS* perform only slightly better (they have the same performance differing only for a few units in the value of Sum OBJ). Although CHMW* reaches a number of best solutions higher than CPH, the latter has a higher value of Sum OBJ proving that, on average, it finds higher solution values than CHMW*. This is also confirmed by the Gap value, that is equal to 0.025% for CPH* and to 0.030% for CHMW*.

In Table 6, we summarize the results for the instances of Class C. Only a limited set of methods, namely PE*, IPGE, and CPH+OptPP*, have been tested on this class. In this set of instances, TIKS is fairly competitive, being the best method in 3 instances. IPGE is the most performing approach when considering #UBest values. However, TIKS obtains a Sum OBJ value and a Gap value that are considerably better than the ones obtained by IPGE. TIKS[2] is less competitive in terms of number of best values, but its Sum OBJ and Gap values are better than the ones of the other methods except for TIKS. CPH+OptPP* is not competitive at all. Although it is never able to find a best value on set C, we have kept its column being one of the few methods applied on these instances. Interesting enough, all the other methods have a complementary behavior.

In Tables 7 and 8 we report a brief overview of the results over Class D instances. In each table, we compare two heuristics at a time by counting the number of times one method provides a performance strictly better than the other. The 256 instances are grouped by the number of resources m, the number of groups n, and the number of items per group $\bar{n}$. Each group consists of 64 instances. Notice that if, in a table line, the total number of the instances is lower than 64, this means that in the excluded instances the two compared algorithms have obtained the same performance. For instance, in the first line of Table 7 with $m = 10$, TIKS and LHBNS2 have got the same result in 1 instance, whereas for $m = 20$ this has happened in 2 instances. Detailed results can be downloaded from the website https://or-dii.unibs.it/index.php?page=tiks.

TIKS has a performance over set D strongly better than LHBNS2, in terms of number of best results, Sum OBJ, and Gap value. However, it is worth observing that the number of best results obtained by LHBNS2 tends to improve when the number of resources increases, whereas this is not the case for TIKS whose performance slightly reduces. In these instances, the behavior of both methods do not seem to be affected by the size and number of the groups. Out of all the 256 instances the two methods get the same result in 6 instances.

In Table 8, we extend the comparison of TIKS with MACH3, the only variant tested by Mansi et al. (2013) on Class D. To make a fair comparison, since the time limit of MACH3 is set to 600s on Class D, we reduce the time limit of TIKS to 600s and to 200s, testing these two additional variants of our approach. Also with respect to MACH3, the performance of TIKS is extremely good. This is true also when TIKS time limit is reduced to a few minutes. When instances are small for number of items, groups and resources, MACH3 and TIKS running for 200s are comparable. When instances become larger this is no longer true and TIKS becomes the best approach. Moreover, differently from LHBNS2, MACH3 does not improve its performance when the number of resources increases. All variants of TIKS outperform MACH3 when Sum OBJ and Gap values are taken into account.

In Table 9, we report some statistics regarding the execution of TIKS for the instances in Class D. Once again, instances are grouped according to the value of m, n, and $\bar{n}$, so the values reported in each column are averages over each group, consisting of 64 instances. Column TTB indicates the time to best, Column Imp2 measures the percentage improvement of the objective function due to Phase 2 with respect to Phase 1. Best1 reports the percentage of instances where the algorithm finds the best solution in Phase 1. Columns Sub1 and Sub2 indicate the number of subproblems solved in Phase 1 and Phase 2, respectively. As already explained, Column $\frac{n_{pos}}{n_{sub}}$ reports the percentage of subproblems with a positive termination out of all the tackled subproblems in Phase 1. Finally, Column Fast indicates the percentage of instances classified as fast. Notice that the TTB value is always around half of the time limit assigned to the algorithm. The Imp2 value seems apparently to be low, but the order of magnitude of the objective function is always between 10^4 and 10^6, thus a small percentage improvement is quite significant in terms of number of units. Interesting enough, Phase 2 is more effective when the number of resources increases. In fact, the average percentage improvement is 7 times higher when the number of resources is equal to 25 with respect to those instances with 10 resources. In most cases, according to Column Best1, the best solution value is found in Phase 2 (81% on average). On average, the number of subproblems solved in Phase 2 is lower than the one of Phase 1. There are a few exceptions, mainly due to a couple of instances (with $m = 10$, $n = 700$, and $\bar{n} = 20$), for which the algorithm solves over 10,000 subproblems in Phase 1, indicating that the initial size of the buckets could have been set to a larger value. The number of fast instances (Column Fast) shows a clear decreasing trend when the number of resources rises. No evident trend can be identified when the size or the number of groups increases. Similar conclusions can be drawn for Column $\frac{n_{pos}}{n_{sub}}$.

4.4. Comparison with state-of-the-art exact algorithms

In this section, we compare TIKS with Gurobi (solving the mathematical formulation of the problem) and YACA, that is, at present, the most performing algorithm available in the literature for the MMKP.

In Tables 10 and 11, we report the solution values on all the instances of Classes A, B, and C obtained by TIKS, YACA, and Gurobi. In Table 10, we report the results for YACA and Gurobi with a time limit of 3600s, while in Table 11 their time limit is 5 hours. As in previous tables, we emphasize in bold the best solution value. Moreover, in Table 11, a "*" is added to indicate that the corresponding method has proved that the solution found is optimal within the time limit. We also report the best bound found by Gurobi after 18000s (Column UB Gurobi). In Table 10, YACA gets slightly better #Best and #UBest values, but, on average, TIKS out-

Table 8
TIKS (1200s, 600s, 200s) vs MACH3 (600s) on Class D instances.

		TIKS (1200s 6 cores)	MACH3 (600s -)	TIKS (600s 6 cores)	MACH3 (600s -)	TIKS (200s 6 cores)	MACH3 (600s -)
m	10	61	3	56	6	29	29
	15	63	0	60	3	37	26
	20	63	1	57	7	36	26
	25	63	1	61	3	43	20
n	100	62	2	55	8	25	32
	250	62	2	57	6	24	38
	500	62	1	59	4	42	21
	700	64	0	63	1	54	10
$\bar{n}$	10	63	0	59	4	30	30
	15	62	2	55	8	33	28
	20	64	0	62	2	46	16
	25	61	3	58	5	36	27
Total		250	5	234	19	145	101
Sum OBJ		52,101,161	52,082,188	52,095,273	52,082,188	52,085,728	52,082,188
Gap		0.003%	0.053%	0.018%	0.053%	0.047%	0.053%

Table 9
TIKS statistics for Class D instances.

		TTB	Imp2	Best1	Sub1	Sub2	$\frac{n_{pos}}{n_{sub}}$	Fast
m	10	633.0	0.01%	19%	565.7	259.2	66%	89%
	15	614.9	0.03%	22%	13.5	6.1	40%	50%
	20	624.4	0.05%	22%	11.8	6.7	30%	23%
	25	634.9	0.07%	13%	11.9	5.7	30%	17%
n	100	540.5	0.09%	23%	19.5	11.3	46%	50%
	250	694.7	0.04%	9%	33.4	109.0	41%	42%
	500	686.7	0.01%	17%	14.8	10.7	39%	45%
	700	585.2	0.01%	25%	535.3	146.7	39%	42%
$\bar{n}$	10	628.7	0.05%	19%	39.5	22.4	44%	45%
	15	602.5	0.03%	16%	22.4	62.4	40%	48%
	20	652.1	0.04%	19%	527.5	179.8	42%	39%
	25	623.9	0.03%	22%	13.6	13.2	40%	47%
Average		626.8	0.04%	19%	150.7	69.4	41%	45%

performs YACA, when Sum OBJ and Gap are considered. Interestingly enough, when considering Table 11, TIKS obtains the same result of YACA and Gurobi in 5 instances, whereas it reaches the same result of YACA always beating Gurobi in other 5 instances and the same result as Gurobi always beating YACA in 3 instances. In one instance (INST24), it gets the best result improving the value of YACA by 10 units and the one of Gurobi by even 30 units. When TIKS is less performing than Gurobi and/or YACA the difference is of 1 or 2 units in almost all the cases. Notice that TIKS gets 6 optimal solutions. Although reaching a lower number of #UBest, even in this case TIKS has a Sum OBJ value larger than Gurobi and very close to YACA. Its Gap value is also lower than the one of Gurobi and slightly higher than YACA's one. It is worth noticing that, in Classes A and B, when an instance is not solved to optimality, the best bound is fairly close to the value obtained by the best of the three methods. This is not the case for some of the instances in Class C, signaling that these values might be far from the optimal ones.

In Mansini and Zanotti (2020), the authors tested YACA also on Class D by reducing the time limit of the exact algorithm to 1 h. In Table 12, we compare the two approaches. TIKS is far more competitive getting 203 best values whereas YACA only 34. The two methods get the same values in other 19 instances. The Gap value confirms this difference in performance. As expected, the performance of YACA, being an exact algorithm, decreases when m, n, and $\bar{n}$ increase. On the contrary, TIKS almost keeps the same level of performance independently of the value of these parameters.

Table 10
TIKS vs YACA (3600s) and Gurobi (3600s): Classes A, B, and C.

Instance	TIKS (1200s 6 cores)	YACA (3600s 4 cores)	Gurobi (3600s 4 cores)
I07	**24,595**	24,594	**24,595**
I08	**36,895**	**36,895**	36,894
I09	49,188	**49,189**	49,182
I10	**61,481**	61,479	61,477
I11	73,792	**73,793**	73,781
I12	86,095	**86,096**	86,091
I13	**98,443**	98,441	98,429
INST01	**10,738**	**10,738**	**10,738**
INST02	**13,598**	**13,598**	**13,598**
INST03	**10,955**	10,950	**10,955**
INST04	**14,457**	14,456	**14,457**
INST05	**17,061**	**17,061**	**17,061**
INST06	**16,845**	16,843	16,840
INST07	16,442	**16,444**	**16,444**
INST08	17,518	**17,524**	17,515
INST09	17,762	**17,766**	17,762
INST10	19,320	**19,321**	19,316
INST11	**19,449**	**19,449**	19,446
INST12	21,744	**21,745**	21,741
INST13	**21,580**	21,578	21,578
INST14	**32,874**	**32,874**	32,872
INST15	**39,165**	39,163	**39,165**
INST16	43,366	**43,367**	43,365
INST17	54,362	**54,363**	54,360
INST18	**60,469**	**60,469**	60,462
INST19	64,931	**64,935**	64,932
INST20	**75,617**	**75,617**	75,615
INST21	44,282	**44,284**	**44,284**
INST22	**41,984**	41,976	41,958
INST23	**42,600**	42,560	42,530
INST24	**41,964**	41,932	41,872
INST25	44,156	**44,159**	44,156
INST26	**44,874**	44,872	44,872
INST27	87,632	**87,636**	87,620
INST28	134,652	**134,656**	134,636
INST29	179,230	**179,232**	179,188
INST30	**214,230**	214,228	214,200
#Best	21	24	9
#Ubest	9	14	0
Sum OBJ	1,894,346	1,894,283	1,893,987
Gap	0.008%	0.012%	0.026%

Table 11
TIKS vs YACA (18000s) and Gurobi (18000s): Classes A, B, and C.

Instance	TIKS (1200s 6 cores)	YACA (18000s 4 cores)	Gurobi (18000s 4 cores)	UB Gurobi (18000s 4 cores)
I07	**24,595**	**24,595**	**24,595***	24,595
I08	36,895	**36,896**	36,895	36,899
I09	49,188	**49,189***	**49,189***	49,189
I10	61,481	61,481	**61,482***	61,482
I11	73,792	**73,794***	73,783	73,795
I12	86,095	**86,097***	86,091	86,098
I13	98,443	**98,445**	**98,445***	98,445
INST01	**10,738**	**10,738***	**10,738***	10,738
INST02	**13,598**	**13,598***	**13,598***	13,598
INST03	**10,955**	10,954	**10,955***	10,955
INST04	**14,457**	**14,457***	14,456	14,461
INST05	17,061	**17,063**	17,060	17,070
INST06	**16,845**	**16,845***	**16,845***	16,845
INST07	16,442	**16,444**	**16,444**	16,451
INST08	17,518	**17,524***	17,519	17,525
INST09	17,762	**17,767**	17,762	17,773
INST10	19,320	19,322	**19,325**	19,329
INST11	**19,449**	**19,449**	19,446	19,455
INST12	21,744	**21,745**	**21,745**	21,748
INST13	**21,580**	21,578	**21,580**	21,587
INST14	32,874	32,874	**32,875**	32,882
INST15	**39,165**	39,163	**39,165**	39,171
INST16	43,366	**43,367**	43,366	43,376
INST17	54,362	**54,363**	54,361	54,370
INST18	**60,469**	**60,469**	60,468	60,476
INST19	64,931	**64,935**	64,932	64,941
INST20	**75,617**	**75,617**	75,615	75,625
INST21	44,282	**44,284**	**44,284***	44,284
INST22	41,984	**41,986**	41,964	42,040
INST23	**42,600**	**42,600**	42,584	42,712
INST24	**41,964**	41,954	41,934	42,208
INST25	44,156	**44,159***	**44,159***	44,159
INST26	44,874	**44,884**	44,872	44,967
INST27	87,632	**87,636***	87,628	87,638
INST28	134,652	**134,656**	134,648	134,662
INST29	179,230	**179,236**	179,222	179,240
INST30	**214,230**	**214,230**	**214,230**	214,250
#Best	14	30	17	
#UBest	1	14	3	
Sum OBJ	1,894,346	1,894,394	1,894,260	
Gap	0.008%	0.005%	0.012%	

4.5. Statistical analysis of the results

In the first three classes (A-C), TIKS and IPGE are by far the best performing heuristic methods. To evaluate whether the difference in performance between these two methods is statistically significant, we apply two nonparametric tests, namely the two-tailed Wilcoxon Signed Rank Test and the two-tailed Sign Test, with a significance level α equal to 0.01. The null hypothesis claims that the two heuristics perform equally well. The two tests yield a p-value equal to $5.6 \cdot 10^{-4}$ and $6.6 \cdot 10^{-5}$, respectively. This allows us to discard the null hypothesis and state that the difference between these two heuristics is statistically significant. On the contrary, we cannot discard the null hypothesis when comparing TIKS[2] and IPGE.

To evaluate the statistical significance of the results for Class D, we apply the Friedman test (Friedman, 1937) on the objective function values obtained by TIKS, MACH3, LHBNS2, and YACA. The choice of a different test is due to the fact that we are now jointly comparing more than two algorithms. Considering a significance level α equal to 0.01, the test yields a p-value equal to $2.2 \cdot 10^{-16}$, which allows us to reject the null-hypothesis (equal performance of all algorithms). We can then state that the performance of all the algorithms are significantly different.

Table 12
TIKS vs YACA: Class D.

		TIKS (1200s 6 cores)	YACA (3600s 4 cores)
m	10	30	19
	15	51	10
	20	61	3
	25	61	2
n	100	44	16
	250	49	9
	500	55	4
	700	55	5
$\bar{n}$	10	45	13
	15	53	10
	20	53	5
	25	52	6
Total		203	34
Sum OBJ		52,101,161	52,087,907
Gap		0.003%	0.029%

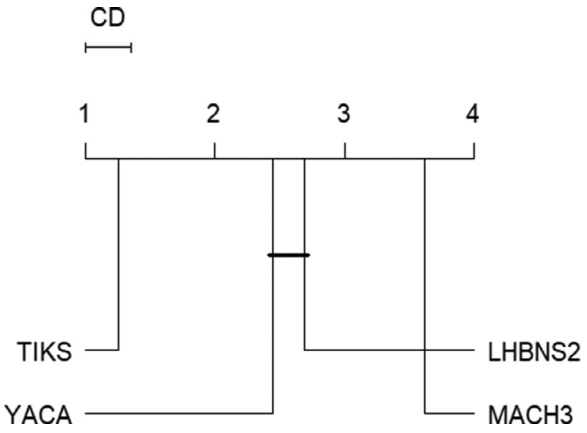

Fig. 1. Comparison between TIKS, YACA, LHBNS2, and MACH3. CD for $\alpha = 0.01$.

To further investigate the results for Class D, we check whether the algorithms are different in a pairwise manner. In order to do that, we apply the post-hoc test of Nemenyi (1963). The method ranks the algorithms for each instance. If the average difference between the ranks of two algorithms (computed out of all instances) is greater than a Critical Difference (CD), then the two algorithms have a significantly different performance. CD is a constant value computed according to a chosen significance level α, equal to 0.01 in this case. All pairs of algorithms produce results that are significantly different, except for YACA and LHBNS2. In Demšar (2006), the author introduces a way to visually represent CD and the difference between algorithms. We use this representation in Fig. 1. In the upper line, each algorithm is connected to the corresponding average rank (between 1 and 2 for TIKS, between 2 and 3 for YACA and LHBNS2, and between 3 and 4 for MACH3). CD value is also represented. If the difference between the ranks is less than CD, a bold line is drawn across the lines belonging to the two algorithms, indicating that their results are not statistically different (like for YACA and LHBNS2).

4.6. New best-known values

In this section, we summarize the performance of TIKS considering both heuristic and exact algorithms. Out of the 37 instances belonging to Classes A, B, and C, 22 are still open (no optimal solution available). In Table 13, we report, for each one of these instances, its current best-known value (Column BK) and the papers achieving it. In 5 cases, TIKS is able to reach the current best-

Table 13
Best known values for open instances from Classes A, B, and C.

Instance	BK	Reference
I08	36,896	Mansini and Zanotti (2020)
INST05	17,065	TIKS^2
INST07	16,444	Mansini and Zanotti (2020), TIKS^2
INST09	17,767	Mansini and Zanotti (2020)
INST10	19,325	Mansini and Zanotti (2020)
INST11	19,449	Chen and Hao (2014), Mansini and Zanotti (2020), TIKS, TIKS^2
INST12	21,745	Mansini and Zanotti (2020)
INST13	21,580	Crévits et al. (2012), Gao et al. (2017), Mansini and Zanotti (2020), TIKS, TIKS^2
INST14	32,875	Chen and Hao (2014), Mansini and Zanotti (2020), TIKS^2
INST15	39,165	Hifi and Wu (2015), Mansini and Zanotti (2020), TIKS, TIKS^2
INST16	43,367	Chen and Hao (2014), Gao et al. (2017), Mansini and Zanotti (2020)
INST17	54,363	Mansi et al. (2013), Chen and Hao (2014), Gao et al. (2017), Mansini and Zanotti (2020)
INST18	60,469	Gao et al. (2017), Mansini and Zanotti (2020), TIKS
INST19	64,935	Mansini and Zanotti (2020)
INST20	75,618	Mansi et al. (2013)
INST22	41,986	Mansini and Zanotti (2020)
INST23	42,600	Mansini and Zanotti (2020), TIKS
INST24	41,998	Gao et al. (2017)
INST26	44,884	Mansini and Zanotti (2020)
INST28	134,656	Mansini and Zanotti (2020)
INST29	179,236	Mansini and Zanotti (2020)
INST30	214,242	Gao et al. (2017)

Table 14
Number of best-known values for Class D instances.

		TIKS		YACA		LHBNS2		MACH3	
		#Best	#UBest	#Best	#UBest	#Best	#UBest	#Best	#UBest
m	10	44	30	33	18	1	0	2	1
	15	53	49	13	10	3	1	0	0
	20	55	53	3	3	6	4	1	1
	25	54	53	3	2	6	6	1	1
n	100	46	41	20	15	5	2	1	0
	250	50	43	15	9	4	3	2	2
	500	54	49	8	4	4	3	1	1
	700	56	52	9	5	3	3	0	0
$\bar{n}$	10	47	40	19	13	5	3	0	0
	15	50	48	11	10	3	2	2	2
	20	54	48	11	5	5	4	0	0
	25	55	49	11	5	3	2	2	1
Total		206	185	52	33	16	11	4	3

known value. TIKS^2 reaches 6 best-known values, one of which is a new result.

In Table 14, we report a global comparison among all the algorithms that have been tested on Class D instances. As in the previous tables, instances are grouped according to the values of m, n, and $\bar{n}$. For each algorithm, we indicate the number of times it holds the best-known value for an instance with one or more other algorithms (Column #Best) and the number of times it is the only one holding a best-known value for a certain instance (Column #UBest). Even when compared with both heuristic and exact algorithms, TIKS performs very well. TIKS improves the best-known value of 185 out of 256 instances, and for 21 additional instances it reaches the best-known value identified by one of the other algorithms. The optimal solution is known for only two of the instances in Class D, namely MAVCH51 and MAVCH59, thanks to YACA. In both cases, TIKS is able to reach the optimal value.

5. Conclusions

In this paper, we provide a new solution approach for the Multidimensional Multiple-choice Knapsack Problem. The method works through two complementary phases trying first to get feasibility and then to reach high quality solutions. The method enhances in a very simple way the Iterative Kernel Search (a heuristic framework thought for MILP problems) by modifying the way the kernel set is constructed at each bucket iteration, how the size of the buckets is chosen, and how computing time is allotted for each subproblem.The resulting method is a competitive approach that provides the highest number of best-known values on well-known sets of benchmark instances and the best average performance. As a future development, we would like to work on the identification of critical information obtained from the feasibility-oriented phase in order to develop more sophisticated rules to set the parameters of the quality-oriented phase. Finally, we intend to use the proposed method to solve other combinatorial problems.

Acknowledgments

The authors would like to thank three anonymous referees and the Area Editor for their helpful comments and suggestions which helped improve an initial version of the manuscript.

References

Akbar, M. M., Rahman, M. S., Kaykobad, M., Manning, E. G., & Shoja, G. C. (2006). Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers & Operations Research, 33*(5), 1259–1273.

Angelelli, E., Mansini, R., & Speranza, M. G. (2010). Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research, 37*, 2017–2026.

Angelelli, E., Mansini, R., & Speranza, M. G. (2012). Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications, 51*(1), 345–361.
Caserta, M., & Voß, S. (2019). The robust multiple-choice multidimensional knapsack problem. *Omega, 86*, 16–27.
Chen, Y., & Hao, J. K. (2014). A "reduce and solve" approach for the multiple-choice multidimensional knapsack problem. *European Journal of Operational Research, 239*(2), 313–322.
Cherfi, N., & Hifi, M. (2010). A column generation method for the multiple-choice multi-dimensional knapsack problem. *Computational Optimization and Applications, 46*, 51–73.
Crévits, I., Hanafi, S., Mansi, R., & Wilbaut, C. (2012). Iterative semi-continuous relaxation heuristics for the multiple-choice multidimensional knapsack problem. *Computers & Operations Research, 39*(1), 32–41.
Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research, 7*, 1–30.
Friedman, M. (1937). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association, 32*(200), 675–701.
Gao, C., Lu, G., Yao, X., & Li, J. (2017). An iterative pseudo-gap enumeration approach for the multidimensional multiple-choice knapsack problem. *European Journal of Operational Research, 260*(1), 1–11.
Gilmore, P. C., & Gomory, R. E. (1966). The theory and computation of knapsack functions. *Operations Research, 14*(6), 1045–1074.
Guastaroba, G., Savelsbergh, M., & Speranza, M. G. (2017). Adaptive kernel search: A heuristic for solving mixed integer linear programs. *European Journal of Operational Research, 263*(3), 789–804.
Guo, J., White, J., Wang, G., Li, J., & Wang, Y. (2011). A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software, 84*(12), 2208–2221.
Hanafi, S., Mansi, R., & Wilbaut, C. (2009). Iterative relaxation-based heuristics for the multiple-choice multidimensional knapsack problem. In M. J. Blesa, C. Blum, L. Di Gaspero, A. Roli, M. Sampels, & A. Schaerf (Eds.), *Hybrid metaheuristics* (pp. 73–83). Berlin, Heidelberg: Springer.
Hifi, M., Michrafy, M., & Sbihi, A. (2004). Heuristic algorithms for the multiple–choice multidimensional knapsack problem. *Journal of the Operational Research Society, 55*, 1323–1332.
Hifi, M., Michrafy, M., & Sbihi, A. (2006). A reactive local search-based algorithm for the multiple-choice multi-dimensionalknapsack problem. *Computational Optimization and Applications, 33*, 271–285.
Hifi, M., & Wu, L. (2015). Lagrangian heuristic-based neighbourhood search for the multiple-choice multi-dimensional knapsack problem. *Engineering Optimization, 47*(12), 1619–1636.
Hiremath, C. S., & Hill, R. R. (2013). First-level tabu search approach for solving the multiple-choice multidimensional knapsack problem. *International Journal of Metaheuristics, 2*(2), 174–199.
Khan, S. (1998). *Quality adaptation in a multi-session adaptive multimedia system: Model, algorithms and architecture*. Department of Electrical and Computer Engineering, University of Victoria Ph.D. thesis.
Khan, S., Li, K. F., Manning, E. G., & Akbar, M. M. (2002). Solving the knapsack problem for adaptive multimedia systems. *Studia Informatica Universalis, 2*, 157–178.
Mansi, R., Alves, C., Valério de Carvalho, J. M., & Hanafi, S. (2013). A hybrid heuristic for the multiple choice multidimensional knapsack problem. *Engineering Optimization, 45*(8), 983–1004.
Mansini, R., & Speranza, M. G. (2002). A multidimensional knapsack model for asset-backed securitization. *Journal of the Operational Research Society, 53*(8), 822–832.
Mansini, R., & Speranza, M. G. (2012). Coral: An exact algorithm for the multidimensional knapsack problem. *INFORMS Journal on Computing, 24*(3), 399–415.
Mansini, R., & Zanotti, R. (2020). A core-based exact algorithm for the multidimensional multiple choice knapsack problem. *INFORMS Journal on Computing,, 32*(4), 1061–1079.
Moser, M., Jokanovic, D. P., & Shiratori, N. (1997). An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 80*, 582–589.
Nemenyi, P. (1963). *Distribution-free multiple comparisons*. Princeton University Ph.D. thesis.
Parra-Hernandez, R., & Dimopoulos, N. J. (2005). A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, 35*(5), 708–717.
Ren, Z.-G., Feng, Z.-R., & Zhang, A.-M. (2012). Fusing ant colony optimization with Lagrangian relaxation for the multiple-choice multidimensional knapsack problem. *Information Sciences, 182*(1), 15–29.
Sasikaladevi, N., & Arockiam, L. (2012). Genetic approach for service selection problem in composite web service. *International Journal of Computer Applications, 44*(4), 22–29.
Shih, W. (1979). A branch and bound method for the multiconstraint zero–one knapsack problem. *Journal of the Operational Research Society, 30*, 369–378.
Shojaei, H., Basten, T., Geilen, M., & Davoodi, A. (2013). A fast and scalable multidimensional multiple-choice knapsack heuristic. *ACM Transactions on Design Automation of Electronic Systems, 18*(4), 51:1–51:32.
Toyoda, Y. (1975). A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science, 21*(12), 1417–1427.
Voudouris, C., & Tsang, E. (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research, 113*(2), 469–499.