European Journal of Operational Research 298 (2022) 89–98

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A branch-and-bound algorithm for the quadratic multiple knapsack problem

Krzysztof Fleszar

Suliman S. Olayan School of Business (OSB), American University of Beirut (AUB), P.O. Box 11-0236, Riad El-Solh, Beirut 1107 2020, Lebanon

ARTICLE INFO

Article history:
Received 27 April 2020
Accepted 9 June 2021
Available online 16 June 2021

Keywords:
Packing
Branch-and-bound
Transportation problem
Fractional knapsack problem
Local search

ABSTRACT

We present a new branch-and-bound algorithm for the Quadratic Multiple Knapsack Problem. A key component of our algorithm is a new upper bound that divides the pairwise item values among individual items, estimates the maximum potential value contributed by each individual item, and calculates the upper bound via a transportation model. A local search is used to adjust the division of pairwise item values in order to improve the upper bound. Reduced costs from the solution of the transportation problem are used to forbid some item-to-knapsack assignments. Information from the upper bound is also used in selecting the item to branch on next. Computational experiments are carried out on three sets of benchmark instances from the literature, two sets of smaller instances with 20–35 items and 3–10 knapsacks per instance, and one set of larger instances with 40–60 items and 3–10 knapsacks per instance. Our algorithm finds and verifies optimal solutions for all small benchmark instances in less than one hour of CPU time apiece and outperforms the best previously proposed algorithms for the problem in terms of both the average computation time and the number of instances for which optimality is verified. For the larger benchmark instances, our algorithm obtains smaller average gaps than the best previously proposed methods.

1. Introduction

The Quadratic Multiple Knapsack Problem (QMKP) is defined by a set of items $N = \{1, \dots, n\}$ and a set of knapsacks $M = \{1, \dots, m\}$. Each knapsack $k \in M$ has a capacity c_k. Each item $i \in N$ has a positive weight, w_i, and an individual value, v_i. Additionally, each pair of items $i, j \in N$, $i < j$, has a pairwise value, v_{ij} (for convenience, $v_{ji} = v_{ij}$). If an item, $i \in N$, is packed in a knapsack, $k \in M$, w_i units of capacity c_k are used, and v_i units are added to the total solution value. Additionally, if a pair of items, $i, j \in N, i < j$, is packed in the same knapsack, v_{ij} units are added to the total solution value. The QMKP is to decide the subset of items to pack in each knapsack such that each item is packed in at most one knapsack, no knapsack capacity is exceeded, and the total solution value is maximized.

The QMKP is a generalization of the classic 0–1 knapsack problem, one of the most widely studied problems in discrete optimization (Martello & Toth, 1990). The generalization is two-fold: multiple knapsacks are available instead of one, and pairwise item values are used in addition to individual item values. The QMKP was first posed by Hiley & Julstrom (2006), who presented a project management application, in which a manager must form teams of people (items) to assign to projects (knapsacks) such that overall productivity (total value) is maximized without exceeding project budgets (capacities). Productivity in this application depends on the individual productivity of each person, as well as on the pairwise productivities of team members. Other applications can be found in capacity planning in computer networks (Gerla, Kleinrock, & Gerla, 1977), capital budgeting (Mathur, Salkin, & Morito, 1983), product-distribution system design (Elhedhli & Coffin, 2005), and in the steel industry (Dawande, Kalagnanam, Keskinocak, Ravi, & Salman, 2000).

The QMKP can be formulated as the following binary quadratic programming model, in which a binary variable x_{ik} indicates whether item i is packed in knapsack k:

$$\max \sum_{k \in M} \left(\sum_{i \in N} v_i x_{ik} + \sum_{i \in N} \sum_{j \in N : i < j} v_{ij} x_{ik} x_{jk} \right) \tag{1}$$

$$\sum_{i \in N} w_i x_{ik} \le c_k \qquad \forall k \in M \tag{2}$$

$$\sum_{k \in M} x_{ik} \le 1 \qquad \forall i \in N \tag{3}$$

E-mail addresses: kfleszar@gmail.com, kf09@aub.edu.lb
URL: https://sites.google.com/view/kfleszar

https://doi.org/10.1016/j.ejor.2021.06.018
0377-2217/© 2021 Elsevier B.V. All rights reserved.

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N, k \in M \tag{4}$$

Objective (1) calculates the total solution value as the sum of individual and pairwise values of items packed in each knapsack. Constraints (2) ensure knapsack capacities are respected, constraints (3) ensure that each item is packed in at most one knapsack, and constraints (4) define the domain of the decision variables. Due to the quadratic objective, the problem is much harder to solve to optimality than the linear single- and multiple-knapsack problems.

Since the work of Hiley & Julstrom (2006), who introduced a greedy heuristic, a hill-climbing algorithm, and a genetic algorithm, several heuristic approaches have been proposed. The most recent meta-heuristics were proposed by Qin, Xu, Wu, & Cheng (2016); Tlili, Yahyaoui, & Krichen (2016), and Peng, Liu, Lü, Kochengber, & Wang (2016). In contrast, only two papers attempted to solve the problem exactly. Bergman (2019) proposed a branch-and-price algorithm, in which columns represent solutions for single knapsacks, the pricing problem is a quadratic single knapsack problem with side constraints, and branching is performed by fixing a column variable to zero or one. More recently, Galli, Martello, Rey, & Toth (2021) proposed polynomial-size linear formulations and relaxations for solving the QMKP.

In this paper, we propose a branch-and-bound algorithm for the QMKP with identical knapsacks with branching carried out by selecting an item and attempting to pack it to each knapsack in which it fits, or excluding it from being packed in any knapsack. A key component is a new upper bound that divides the pairwise values among individual items, estimates the maximum potential value contributed by each individual item, and calculates the upper bound value via a transportation model. A local search is used to improve the upper bound value by adjusting the divisions of pairwise values. Reduced costs from the solution of the transportation problem are used to forbid some item-to-knapsack assignments. Information from the upper bound is also used in selecting the item to branch on next.

The rest of the paper is organized as follows. Section 2 describes our new branch-and-bound algorithm, Section 3 presents the results of our computational experiments, and Section 4 summarizes the contributions of this work.

2. The branch-and-bound algorithm

The notation used throughout is shown in Table 1. Without loss of generality, we assume that all problem parameters are integers and items are sorted by non-decreasing weights ($w_{i-1} \le w_i$ $\forall i \in \{2, \dots, n\}$). We also assume that all knapsacks have identical capacities, $c_k = c$ $\forall k \in M$. While all weights are assumed to be positive, individual and pairwise item values can be positive, zero, or negative.

Our branch-and-bound algorithm searches the solution space in the best-bound-first fashion. The solution space is subdivided by taking decisions on item-to-knapsack assignments. Each solution subspace is represented by a 'node' data structure that stores information about the decisions taken so far and potential future decisions. A 'heap' data structure is used to store all nodes representing solutions subspaces needing exploration.

2.1. Node information

A node in our branch-and-bound algorithm contains the following information:

- $N_k \subseteq N$: set of items packed in knapsack $k \in M'$
- $C_k \subseteq N$: set of candidate items for packing in knapsack $k \in M'$
- UB(node): upper bound value calculated for the node
- Branch(node): the item to branch on next
- Reeval(node): if the best solution value is increased to or above this value, the upper bound for the node should be reevaluated in order to forbid some additional item-to-knapsack assignments.

Sets N_k and C_k are defined for each $k \in M'$, where $M' = M \cup \{m+1\}$ is a set that includes all knapsacks and an additional dummy knapsack, $m+1$. Set N_{m+1} contains items that are excluded from packing in any of the original knapsacks, and set C_{m+1} contains items that may in the future search be excluded from packing in any knapsack. Each item $i \in N$ must be in exactly one set N_k (when it is packed in knapsack k or excluded from packing if $k = m+1$) or in at least one set C_k (when it is not packed yet).

The total value of the partial solution represented by sets N_k for $k \in M$ is:

$$V(N_1, \dots, N_m) = \sum_{k \in M} \left(\sum_{i \in N_k} v_i + \sum_{i \in N_k} \sum_{j \in N_k : i < j} v_{ij} \right) \tag{5}$$

The slack (unused space) in knapsack $k \in M$ is calculated as:

$$s_k = c - \sum_{i \in N_k} w_i \tag{6}$$

Throughout, we will assume that s_k is automatically updated whenever set N_k changes. Note that all candidate items $i \in C_k$ for each knapsack $k \in M$ (excluding the dummy knapsack) must satisfy $w_i \le s_k$.

2.2. Overall algorithm

The pseudo-code of our branch-and-bound algorithm is presented in Fig. 1. First, an initial solution is constructed in line 1 using the heuristic described in Section 2.4 and its value is stored in variable 'Best'. Throughout the algorithm, whenever a solution better than the best is found, the best solution and the variable 'Best' are updated.

In line 2, a root node representing the whole solution space of the problem is created: for all knapsacks $k \in M'$ (including the dummy knapsack), N_k is empty and C_k contains all items N. In line 3, the upper bound procedure described in Section 2.3 is called for the root node. This procedure calculates an upper bound value for the node and stores it in UB(node). Additionally, it determines the item to branch on, Branch(node), and value Reeval(node), which indicates whether reevaluating the upper bound can beneficial in the future. In line 4, the root node is stored in the heap. Note that the heap is a standard data structure that allows logarithmic-time addition of a new node and logarithmic-time removal of a node with the largest upped bound value.

In line 5, the main branch-and-bound loop begins, which is continued until the heap is empty or the time limit is reached. In line 6, a node with the largest upper bound is removed from the heap. In lines 7–9, the upper bound is reevaluated if necessary, and if the new upper bound is sufficiently improved, the node is pruned by bound. In line 10, the item to branch on, i, is retrieved from the node. Then, the branching loop begins, in which nodes are created by assigning item i to each knapsack $k \in M'$ for which i is among its candidate items C_k. Note that in the last iteration of the branching loop, i is assigned to the dummy knapsack $k = m+1$ (provided that $i \in C_{m+1}$), which is equivalent to excluding item i from packing in any knapsack.

In the branching loop, a copy of the node is created in line 13. Then, item i is packed in knapsack k by adding it to N_k (line 14) and removing it from all sets of candidate items (line 15). Then, the upper bound procedure is called for the new node (line 16), and a heuristic described in Section 2.4 is called for the new

Table 1
Notation.

i, j	indices of items
k	index of a knapsack
$N = \{1, \dots, n\}$	set of all items
$U \subseteq N$	set of unpacked items
$M = \{1, \dots, m\}$	set of all knapsacks
$M' = M \cup \{m+1\}$	set of all knapsacks and an additional dummy knapsack $m+1$
$c_k = c$	capacity of knapsack $k \in M$
w_i, v_i	weight and individual value of item $i \in N$
$v_{ij} = v_{ji}$	pairwise value of items $i, j \in N$, $i < j$, if both are packed in the same knapsack
N_k	set of items packed in knapsack $k \in M'$; N_{m+1} is a set of items that are excluded from packing in any knapsack
$V(N_1, \dots, N_m)$	value of a (partial) solution defined by sets N_k for $k \in M$; $V(N_1, \dots, N_m) = \sum_{k \in M} \left(\sum_{i \in N_k} v_i + \sum_{i \in N_k} \sum_{j \in N_k : i<j} v_{ij} \right)$
$s_k = c - \sum_{i \in N_k} w_i$	slack in knapsack $k \in M$; automatically updated whenever N_k changes
C_k	set of candidate items for packing in knapsack $k \in M'$; C_{m+1} is a set of items that can be excluded from packing in any knapsack
$C_{ik} \subseteq C_k \setminus \{i\}$	set of 'useful' candidate items for packing in knapsack $k \in M$ together with item $i \in C_k$
$C = \bigcup_{k \in M'} C_k$	set of all candidate items
$C' = C \cup \{n+1\}$	set of all candidate items and an additional dummy item $n+1$
$v(i, N_k)$	value that would be contributed to the total solution value if a candidate item $i \in C_k$ were packed in knapsack $k \in M$; $v(i, N_k) = v_i + \sum_{j \in N_k} v_{ij}$
α_{ijk}	fraction of v_{ij} allocated to item i when calculating the maximum potential contribution p_{ik} of candidate item $i \in C_k$ if put in knapsack $k \in M$
p_{ik}	maximum potential contribution of item $i \in C_k$ if put in knapsack $k \in M$
p'_{ik}	modified maximum potential contribution of item $i \in C_k$ if put in knapsack $k \in M$; used for selecting an item to branch on next
y_{ijk}, y^*_{ijk}	decision variable and its optimal value for the FKP(i, k, α) (See Section 2.3)
f_{ik}, f^*_{ik}	decision variable and its optimal value for the TP(α) (See Section 2.3)
Best	objective value of current best solution
node, node2	nodes in the branch-and-bound search
heap	set of nodes; a structure that allows adding a new node and extracting a node with the largest upper bound in logarithmic time
UB(node)	upper bound value calculated for the node
Branch(node)	the item to branch on next
Reeval(node)	if the best solution value is increased to or above this value, the upper bound for the node should be reevaluated in order to forbid some additional item-to-knapsack assignments

```
1:  Run heuristic to obtain the initial best solution; save its value in Best;
2:  node ← create a root node with $N_k = \emptyset, C_k = N \ \forall k \in M'$;
3:  CalcUB(node);                                   // calculate the upper bound for the root node
4:  heap ← {node};                                  // initialize the heap with the root node
5:  while heap $\neq \emptyset$ and time limit not reached do
6:      node ← remove from heap node with largest UB(node);
7:      if Reeval(node) $\leq$ Best then                       // if reevaluation useful
8:          CalcUB(node);                                   // recalculate upper bound for node
9:          if UB(node) $\leq$ Best then continue;                 // prune node by bound
10:     $i$ ← Branch(node);                                   // extract item for branching
11:     foreach $k \in M' : i \in C_k$ do          // branch by trying to assign $i$ to knapsack $k$
12:         if $1 < k \leq m$ and $N_k$(node) $= N_{k-1}$(node) $= \emptyset$ then continue;   // sym. breaking
13:         node2 ← node;                                       // create a copy of node
14:         $N_k$(node2) ← $N_k$(node2) $\cup \{i\}$;                  // pack item $i$ in knapsack $k$
15:         foreach $l \in M'$ do $C_l$(node2) ← $C_l$(node2) $\setminus \{i\}$;  // remove $i$ from all candidate sets
16:         CalcUB(node2);                              // calculate upper bound for new node
17:         Run heuristic for node2 and update best solution and Best if appropriate;
18:         if Best improved then Remove from heap all nodes with UB $\leq$ Best;   // prune heap
19:         if UB(node2) $\leq$ Best then continue;                // prune node by bound
20:         heap ← heap $\cup$ {node2};                                  // add node to heap
21:     end
22: Resulting solution: the best solution with value Best;
23: Resulting UB: if heap $\neq \emptyset$, $\max_{\text{node} \in \text{heap}}$ UB(node); otherwise Best (optimal solution found);
```

Fig. 1. The branch-and-bound algorithm.

node (line 17). If the value 'Best' is improved by this heuristic or by another heuristic used inside the upper bound procedure, all nodes in the heap that can be pruned by bound are removed from the heap (line 18). Also, if the upper bound value for the new node is sufficiently low, the new node is pruned by bound (line 19). Otherwise, the new node is added to the heap (line 20).

Line 12 skips an empty knapsack, k, if the previously considered knapsack, $k-1$, is also empty, for, in such a case, adding item i to knapsack k would create a node with a partial solution symmetrical to one already created in an earlier iteration.

When the main branch-and-bound loop is finished, the resulting solution is the best solution obtained so far and its value is stored in the variable 'Best' (line 22). If any unexplored nodes remain in the heap (in case the algorithm is stopped due to the time limit), the resulting upper bound equals the largest upper of the remaining nodes. Otherwise, the best solution is optimal and the upper bound equals 'Best' (line 23).

2.3. Upper bound

Let $v(i, N_k)$ be the value that will be contributed to the total solution value if a candidate item $i \in C_k$ is packed in knapsack $k \in M$, calculated as:

$$v(i, N_k) = v_i + \sum_{j \in N_k} v_{ij} \tag{7}$$

This value includes the individual value v_i as well as all pairwise values v_{ij} for all $j \in N_k$, i.e., those related to all items currently packed in knapsack k. However, packing i in k may lead to additional contributions v_{ij} for $j \in C_k \setminus \{i\}$, i.e., those related to other candidate items that may be packed in the same knapsack later. In the following, we will try to estimate the maximum potential contribution of item i in knapsack k, which will help determine a good upper bound value.

Let C_{ik} be the set of 'useful' candidate items for packing in knapsack $k \in M$ together with item $i \in C_k$. Set C_{ik} includes only those candidate items $j \in C_k \setminus \{i\}$ that satisfy the capacity constraint, $w_j \le s_k - w_i$, and have positive pairwise value together with item i, $v_{ij} = v_{ji} > 0$. Items with zero pairwise contribution are excluded because they would not have any impact on the calculation of the upper bound but would increase the computation time. Items with negative contributions must be excluded in order for the upper bound to work correctly, keeping in mind that some candidate items $j \in C_k \setminus \{i\}$ may be later excluded from packing or packed in a knapsack different from k and negative contributions may be completely avoided.

To find the maximum potential contribution of adding item $i \in C_k$ to knapsack $k \in M$, we solve the following fractional knapsack problem FKP(i, k, α) in which items $j \in C_{ik}$ with values $\alpha_{ijk} v_{ij}$ and weights w_j can be packed in a knapsack with capacity $s_k - w_i$:

$$\max \sum_{j \in C_{ik}} \alpha_{ijk} v_{ij} y_{ijk} \tag{8}$$

$$\sum_{j \in C_{ik}} w_j y_{ijk} \le s_k - w_i \tag{9}$$

$$0 \le y_{ijk} \le 1 \qquad \forall j \in C_{ik} \tag{10}$$

Parameter α_{ijk} indicates the fraction of the pairwise value v_{ij} that is allocated to the potential contribution of item i in knapsack k. The α_{ijk} parameters are introduced in order to divide the pairwise value $v_{ij} = v_{ji}$ among potential contributions of items i and j, so that we can avoid double-counting these values at a later stage. The α_{ijk} parameters are non-negative numbers that satisfy:

$$\alpha_{ijk} + \alpha_{jik} = 1 \quad \forall k \in M, i, j \in C_k, i < j \tag{11}$$

As is well known in the knapsack problem literature, an optimal solution of the FKP(i, k, α) can be obtained by sorting items $j \in C_{ik}$ in the order of non-increasing value-to-weight ratios $\alpha_{ijk} v_{ij}/w_j$, and packing them in this order in the knapsack until the capacity is exhausted (assuming positive values and weights, which is the case here). Note that this will result in packing some items $j \in C_{ik}$ with the larger ratios completely ($y^*_{ijk} = 1$), then partially packing at most one item with the largest ratio among the remaining items ($0 < y^*_{ijk} < 1$), and finally excluding all the rest ($y^*_{ijk} = 0$).

The maximum potential contribution of item $i \in C_k$ in knapsack $k \in M$ is:

$$p_{ik}(\alpha) = v(i, N_k) + \text{the optimal value of FKP}(i, k, \alpha) \tag{12}$$

$v(i, N_k)$, which is defined by Eq. (7), includes the individual value v_i of item i as well as all pairwise values v_{ij} for $j \in N_k$, while the optimal value of FKP(i, k, α) includes parts of pairwise values v_{ij} for $j \in C_{ik}$. Note that while the optimal value of FKP(i, k, α) is never negative, $v(i, N_k)$ could be be negative, and hence the maximum potential contribution $p_{ik}(\alpha)$ may also be negative.

With the maximum potential contributions, $p_{ik}(\alpha)$, for all knapsacks $k \in M$ and items $i \in C_k$ computed, an upper bound on the total objective function value in a node can be calculated as:

$$\text{UB(node)} = V(N_1, \dots, N_m) + \text{the optimal value of TP}(\alpha) \tag{13}$$

$V(N_1, \dots, N_m)$ is the total value of the current partial solution, as defined by equation (5). TP(α) is the following maximization transportation problem with decision variables f_{ik} denoting the number of units (of weight) transported from source (item) $i \in C'$ to sink (knapsack) $k \in M'$:

$$\max \sum_{i \in C'} \sum_{k \in M'} p_{ik}(\alpha) f_{ik}/w_i \tag{14}$$

$$\sum_{k \in M'} f_{ik} = w_i \qquad \forall i \in C' \tag{15}$$

$$\sum_{i \in C'} f_{ik} = s_k \qquad \forall k \in M' \tag{16}$$

$$f_{ik} \ge 0 \qquad \forall i \in C', k \in M' \tag{17}$$

In this model, the sinks are defined by set M' and represent all knapsacks, including the dummy knapsack $m+1$. The sources are defined by set $C' = C \cup \{n+1\}$, where $C = \bigcup_{k \in M'} C_k$ is the set of all candidate items, and $n+1$ denotes a dummy item. The dummy item is introduced in order to allow the transportation model to leave some space empty in some knapsacks, to cater for the case when filling this space would decrease the objective value due to the contribution $p_{ik}(\alpha)$ being negative for some $i \in C$ and $k \in M$. The supply of the dummy item source is set to $w_{n+1} = \sum_{k \in M} s_k$ and the demand of the dummy knapsack sink is $s_{m+1} = \sum_{i \in C} w_i$. This balances the total supply and the total demand, as required in a transportation model with equality constraints. It also allows for all fractional item-to-knapsacks assignments, including the extreme case of leaving all remaining space in all knapsacks unoccupied, which could happen if all maximal potential contributions were negative.

Note that equation (12) defines maximal potential contributions $p_{ik}(\alpha)$ only for $i \in C_k$ and $k \in M$, while model TP(α) requires $p_{ik}(\alpha)$ values for all $i \in C'$ and $k \in M'$. The missing values are set as follows. We set $p_{i,m+1}(\alpha) = 0$ for all $i \in C_{m+1}$, so that excluding an item from packing has no impact on the objective value of the transportation problem, provided that this item is a valid candidate for exclusion. Similarly, we set $p_{n+1,k} = 0$ for each $k \in M'$, so that the dummy item can be used to fill space in any knapsack with no impact on the objective value. For $k \in M'$ and $i \in C \setminus C_k$, $p_{ik}(\alpha)$ is set to a large negative penalty in order to prevent packing an item in a knapsack (when $k \in M$) or excluding the item from packing (when $k = m+1$) when such a decision is forbidden in the branch-and-bound node.

Theorem 1. *For any non-negative parameters α_{ijk} satisfying condition (11),* UB(node) *defined by Eq. (13) is a valid upper bound on the optimal value of the QMKP in the branch-and-bound node defined by sets of items packed in knapsacks, N_k, and sets of candidate items, C_k, for $k \in M'$.*

Proof. For UB(node) to be a valid upper bound, Eq. (13) has to calculate exactly or overestimate the sum of all individual and pairwise values of the optimal allocation of items to knapsacks, assuming that items in N_k are already packed in knapsack $k \in M$ and items in C_k may still be packed. The individual values, v_i, and the pairwise values, v_{ij}, for all $i, j \in N_k$ (both i and j are packed), are exactly accounted for in the first element, $V(N_1, \dots, N_m)$, of

Eq. (13). The remaining individual and pairwise values are accounted for or overestimated by the optimal objective function value of the transportation problem, TP(α). The decision variables in the transportation problem correspond to packing items in knapsacks (possibly fractionally). The objective in the transportation model is defined such that packing a candidate item $i \in C_k$ in knapsack $k \in M$ contributes a maximum potential value of $p_{ik}(\alpha)$ defined by Eq. (12) to the upper bound. The first element in this equation is $v(i, N_k)$, which includes the individual value of candidate item i and all pairwise values, v_{ij}, between candidate item i and any packed item $j \in N_k$ (i is a candidate and j is packed). So far, we have accounted exactly for all individual item values as well as all pairwise values between two items i and j, where i is either packed or a candidate and j is packed. What remains is the pairwise values, v_{ij}, where both i and j are candidate items. In the rest of the proof, we will show how these values are correctly accounted for.

Observe that the second element in Eq. (12) defining $p_{ik}(\alpha)$ is the optimal value of the fractional knapsack problem, FKP(i, k, α). This problem calculates the maximum potential contribution that a candidate item $i \in C_k$ can have in knapsack k due to pairwise values $\alpha_{ijk}v_{ij}$ of candidate item i with another candidate item $j \in C_k \setminus \{i\}$. If the optimal solution of the transportation model, TP(α), packs two different items $i, j \in C_k$ in the same knapsack $k \in M$ (optimal values $y^*_{ik} = w_i$ and $y^*_{jk} = w_j$), a value $\alpha_{ijk}v_{ij}$ or larger will be accounted for in the potential contribution $p_{ik}(\alpha)$ for item i and, similarly, a value $\alpha_{jik}v_{ji}$ or larger will be accounted for in the potential contribution $p_{jk}(\alpha)$ for item j. Hence, the total value accounted for in UB(node) will be at least $\alpha_{ijk}v_{ij} + \alpha_{jik}v_{ji} = (\alpha_{ijk} + \alpha_{jik})v_{ij} = v_{ij}$. If only one of the two items i, j is packed in knapsack k in the solution of the transportation model, only part of the pairwise value v_{ij} will be accounted for in UB(node), which will overestimate the actual zero contribution in the QMKP. If neither of the two items is packed in knapsack k, the value is zero correctly.

Note that the above analysis would be correct if knapsack and transportation problems were solved as binary assignment problems, that is, assuming that items are either fully packed or excluded. The fractional assignments allowed in models TP(α) and FKP(i, k, α) are further relaxations. Hence, the resulting value, UB(node), is a valid upper bound. □

The above upper bound calculation works for any non-negative values α_{ijk} satisfying condition (11). This leads to an opportunity to further improve the upper bound value by finding the α_{ijk} values that will minimise it. However, this leads to a highly non-linear problem that would need to be solved at every branch-and-bound node, at some, possibly considerable, computational expense. Hence, we use a simple local search heuristic to find good, rather than optimal, values. We start with all α_{ijk} values set to 0.5. Whenever we detect a pair of items $i, j \in C_k$ for a knapsack $k \in M$ such that $\alpha_{ijk}v_{ij}$ is at least partially included in the upper bound value, while $\alpha_{jik}v_{ji}$ is not, we decrease α_{ijk} and increase α_{jik} by the same amount in an attempt to reduce the upper bound value. Note that a contribution $\alpha_{ijk}v_{ij}$ is included in the upper bound value if $y^*_{ijk} > 0$, i.e., item j is used when calculating the maximum potential contribution for item i to knapsack k, and $f^*_{ik} > 0$, i.e., when the transportation model assigns at least part of item i to knapsack k.

Fig. 2 shows the pseudo-code of our upper bound procedure CalcUB(node), the logic of which is described above. This procedure is called for each branch-and-bound node. First, information about the packed items and the candidate items is extracted from the node in line 2. Then, α_{ijk} and local search parameters are initialized in lines 3–4. Parameter 'nrep' denotes the number of iterations of the local search. Parameter 'step' is the value by which α_{ijk} will be modified, and parameter 'mult' is the value by which 'step' will be multiplied after each local search iteration. The remainder of the procedure is the local search loop that tries to determine the best possible value of the upper bound for the given node.

Lines 6–13 ensure that sets C_k for $k \in M'$ contain only valid candidate items and fixes items in knapsacks whenever possible. Candidate items with weights exceeding the slack of the corresponding knapsack are dropped (line 8). This check is necessary, because any time an item is packed in a knapsack (through branching or because of item fixing in line 11), the slack in the knapsack is reduced, which could make some remaining candidate items for that knapsack invalid. Then, if an item is a candidate for one knapsack only, it is fixed in that knapsack without branching (line 11). If an item is a candidate for no knapsacks (including the dummy knapsack), the decision node is pruned by infeasibility (line 12).

Lines 14–18 determine the maximum potential contribution of each candidate item $i \in C_k$ for knapsack $k \in M$. First, the set 'useful' candidate items, C_{ik}, that can be packed together with item i in knapsack k is determined. Then, the fractional knapsack problem FKP(i, k, α) described earlier is solved by sorting the 'useful' candidate items and selecting highest value-to-weight items until knapsack capacity is exhausted. Based on this solution, the maximum potential contribution p_{ik} is determined. Additionally, a modified contribution p'_{ik} not using the α_{ijk} parameters is determined. The modified contributions are used at the end of the upper bound procedure to choose the next item for branching.

Lines 20–22 set trivial potential contributions of all items in the dummy knapsack (zero), of the dummy item in all knapsacks (zero), and of all items in the knapsacks in which they are forbidden (penalty). The penalty used for forbidden assignments is set to the sum of all v_i and all v_{ij} values for all $i, j \in N$.

In line 23, the transportation problem TP(α) is solved. Based on its solution, the fractional upper bound value is calculated in line 24. Given that all problem parameters are integer, the optimal objective value must also be integer. Hence, the rounded down value is used in line 25 to update the upper bound value for the node. If the updated upper bound is not greater than the value of the best solution found so far, the node is pruned by bound in line 26.

Lines 27–32 exploit the reduced costs $\bar{c}_{ik}$ obtained from the solution of TP(α) to forbid some item-to-knapsack assignments. A negative reduced cost indicates the rate with which the objective function value would decrease if the corresponding variable is forced into the solution. Given that packing an item in a knapsack corresponds to increasing variable f_{ik} from zero to w_i, the reduced upper bound is calculated in line 29 as UBfixed = $\lfloor$UBfrac + $\bar{c}_{ik}w_i\rfloor$. If this reduced upper bound is not greater than the value of the best solution, assignment of item i to knapsack k is forbidden by excluding i from C_k in line 30. The lowest value of UBfixed is saved in Reeval(node) in line 31. This value will be used only if no reduced cost works and the node is saved for later consideration in the search. If this node is later considered for branching and the best solution value is increased to or above Reeval(node), the upper bound procedure is called again in order to forbid some additional item-to-knapsack assignments (see lines 7–9 in Fig. 1). If any assignments were forbidden with the help of reduced costs and the local search is in the last iteration, line 32 repeats the local search loop again. Accordingly, the number of repetitions of the local search loop may exceed 'nrep'. However, this is useful, since the recalculated upper bound could turn out to be stronger after assignment forbidding.

Lines 33–38 update the values of α_{ijk} parameters in an attempt to improve the upper bound. For each knapsack $k \in M$ and each pair of items $i, j \in C_k$, the optimal solutions of FKP(i, k, α) and TP(α) are checked to detect when a contribution $\alpha_{ijk}v_{ij}$ is included in the upper bound calculation ($y^*_{ijk} > 0$ and $f^*_{ik} > 0$) and contribution $\alpha_{jik}v_{ji}$ is not ($y^*_{jik} = 0$ or $f^*_{jk} = 0$). If such a case is detected,

```
1:  procedure CalcUB(node)
2:      Extract $N_0, \dots, N_{m+1}, C_1, \dots, C_{m+1}$ from node;
3:      $\alpha_{ijk} \leftarrow 0.5 \; \forall k \in M, i, j \in C_k, i \neq j$;        // initialize $\alpha_{ijk}$ values
4:      rep $\leftarrow$ 1, nrep $\leftarrow$ 20, step $\leftarrow$ 0.3, mult $\leftarrow$ 0.8;        // initialize local search parameters
5:      while rep $\leq$ nrep do
            // Forbid assignments if insufficient space, and fix items in knapsacks if possible
6:          repeat
7:              for $k \in M$ and $i \in C_k$ do
8:                  if $w_i > s_k$ then $C_k \leftarrow C_k \setminus \{i\}$;        // forbid assignment if insufficient space
9:              for $i \in N \setminus \bigcup_{k \in M'} N_k$ do        // for each undecided item
10:                 $K \leftarrow \{k \in M' : i \in C_k\}$;        // find candidate knapsacks
11:                 if $|K| = 1$ then $\{k\} \leftarrow K; C_k \leftarrow C_k \setminus \{i\}; N_k \leftarrow N_k \cup \{i\}$;        // fix if one
12:                 if $|K| = 0$ then UB(node) $\leftarrow$ Best; return;        // prune if no knapsacks
13:         until no item is fixed;
            // Determine maximum potential contributions and calculate upper bound
14:         for $k \in M$ and $i \in C_k$ do
15:             $C_{ik} \leftarrow \{j \in C_k \setminus \{i\} : w_j \leq s_k - w_i \wedge v_{ij} > 0\}$;        // find 'useful' candidate items
16:             Solve FKP$(i, k, \alpha)$ and save solution in $y^*_{ijk}$;
17:             $p_{ik} \leftarrow v(i, N_k) + \sum_{j \in C_{ik}} \alpha_{ijk} v_{ij} y^*_{ijk}$;        // find max. potential contribution
18:             $p'_{ik} \leftarrow v(i, N_k) + \sum_{j \in C_{ik}} v_{ij} y^*_{ijk}$;        // find contribution for branching
19:         $C \leftarrow \bigcup_{k \in M'} C_k$; $C' \leftarrow C \cup \{n+1\}$; $M' \leftarrow M \cup \{m+1\}$;
20:         for $i \in C_{m+1}$ do $p_{i,m+1} \leftarrow 0$;        // set contributions in dummy knapsack
21:         for $k \in M'$ do $p_{n+1,k} \leftarrow 0$;        // set contributions for dummy item
22:         for $k \in M'$ and $i \in C \setminus C_k$ do $p_{ik} \leftarrow -$penalty;        // penalize forbidden assignments
23:         Solve TP$(\alpha)$, save solution in $f^*_{ik}$ and reduced costs in $\bar{c}_{ik}$;
24:         UBfrac $\leftarrow V(N_1, \dots, N_m) + \sum_{i \in C'} \sum_{k \in M'} p_{ik} f^*_{ik} / w_i$;        // fractional upper bound
25:         if $\lfloor$UBfrac$\rfloor$ < UB(node) then UB(node) $\leftarrow \lfloor$UBfrac$\rfloor$;        // update node upper bound
26:         if UB(node) $\leq$ Best then return;        // prune node by bound
            // Forbid assignments using reduced costs
27:         RCworked $\leftarrow$ false; Reeval(node) $\leftarrow$ UB(node);
28:         for $k \in M'$ and $i \in C_k$ do
29:             UBfixed $\leftarrow \lfloor$UBfrac + $\bar{c}_{ik} w_i \rfloor$;        // calculate UB if item fixed in knapsack
30:             if UBfixed $\leq$ Best then $C_k \leftarrow C_k \setminus \{i\}$; RCworked $\leftarrow$ true;        // forbid assignment
31:             Reeval(node) $\leftarrow$ min{Reeval(node), UBfixed};
32:         if RCworked and rep = nrep then continue;        // run again even if last repetition
            // Modify $\alpha_{ijk}$ values to try and improve the upper bound value
33:         alphaChanged $\leftarrow$ false ;
34:         if rep < nrep then
35:             for $k \in M$ and $i, j \in C_k$ do
36:                 if $y^*_{ijk} > 0$ and $f^*_{ik} > 0$ and ($y^*_{jik} = 0$ or $f^*_{jk} = 0$) then
37:                     $\alpha_{ijk} \leftarrow \max\{0, \alpha_{ijk} - \text{step}\}$; $\alpha_{jik} \leftarrow 1 - \alpha_{ijk}$;        // modify $\alpha_{ijk}$ and $\alpha_{jik}$
38:                     alphaChanged $\leftarrow$ true ;
39:         step $\leftarrow$ step $\times$ mult; rep $\leftarrow$ rep + 1;
40:         if not RCworked and not alphaChanged then break;
        // Run a heuristic and determine the next item for branching
41:     Run heuristic based on the last TP$(\alpha)$ solution; update the best solution and Best;
42:     for $i \in C$ do $K \leftarrow \{k \in M : i \in C_k\}$; $S_i \leftarrow (\sum_{k \in K} p'_{ik})/|K|$;        // find average contributions
43:     Branch(node) $\leftarrow \arg\max_{i \in C} S_i$;        // save item with max. avg. contribution for branching
44: end
```

Fig. 2. The upper bound procedure.

α_{ijk} is decreased by 'step', but not below zero, and α_{jik} is increased by 'step', but not above one (line 37), changing the division of the pairwise value v_{ij} among maximum potential contributions for items i and j. This change does not guarantee improving the upper bound every time, but does so considerably on average.

Line 39 reduces the 'step' by which values of α_{ijk} are modified and increases the iteration number 'rep' by one. In line 40, the local search is stopped early before reaching the maximum number of iterations 'nrep' in case no assignments were forbidden using reduced costs and no changes of α_{ijk} parameters were made; otherwise, if 'rep' ≤ 'nrep', the next iteration of the loop begins.

After the local search loop is completed, a heuristic solution is generated in line 41, based on the solution of the last TP(α). This heuristic starts from the partial solution defined by sets N_k for $k \in M$ and adds each item $i \in C_k$ to knapsack $k \in M$ if $f^*_{ik} = w_i$. Additionally, an attempt is made to assign each remaining unassigned

```
1:  $U \leftarrow N$; $N_k \leftarrow \emptyset\ \forall k \in M$;        // initialize sets of unpacked and packed items
2:  foreach $k \in M$ do                                                       // pack one knapsack $k$ at a time
3:      $i \leftarrow \arg\max_{i \in U: w_i \le s_k} v(i, U \setminus \{i\})/w_i$;   // find $i$ with max. value density w.r.t. $U$
4:      Move $i$ from $U$ to $N_k$;                                            // pack item $i$ in knapsack $k$
5:      while $\exists i \in U : w_i \le s_k$ do                               // while items that fit in knapsack $k$ exist
6:          $i \leftarrow \arg\max_{i \in U: w_i \le s_k} v(i, N_k)/w_i$;      // find $i$ with max. value density w.r.t. $N_k$
7:          Move $i$ from $U$ to $N_k$;                                        // pack item $i$ in knapsack $k$
8:      Move $i$ from $N_k$ back to $U$;                                       // unpack from knapsack $k$ the last packed item $i$
9:      $i \leftarrow \arg\max_{i \in U: w_i \le s_k} v(i, N_k)$;              // find $i$ with max. value w.r.t. $N_k$
10:     Move $i$ from $U$ to $N_k$;                                            // pack item $i$ in knapsack $k$
11: end
```

Fig. 3. The modified greedy heuristic.

item (in an arbitrary order) to any available knapsack in which it fits if that increases total solution value. If the resulting solution is better than the best solution found so far, the best solution and the value 'Best' are updated accordingly.

Finally, in lines 42–43, for each candidate item $i \in C$, an average modified contribution is calculated, and the index of the item with the largest average value is saved for branching. Notably, some early experiments showed that using the modified contributions p'_{ik} instead of p_{ik} for the branching decision leads to better results.

The upper bound procedure described above is negatively affected by a symmetry that occurs when two or more knapsacks are empty. In order to avoid this, we combine such empty knapsacks into one when solving TP(α). For example, if knapsacks 1–3 are empty ($N_1 = N_2 = N_3 = \emptyset$ and $s_1 = s_2 = s_3 = c$), we consider knapsack 1 with capacity $3c$ and exclude knapsacks 2 and 3 from consideration when solving TP(α). This modification requires a number of additional small modifications across the upper bound procedure, including: potential contributions for the excluded knapsacks do not have to be calculated in lines 14–18; when forbidding an assignment of an item to the combined knapsack in lines 27–32, the item should be forbidden in all original knapsacks; α_{ijk} parameters can be updated only for the combined knapsack in lines 33–38; and the heuristic in line 41 must distribute items among the combined knapsacks without exceeding their capacities. Given that these modifications would further complicate the already quite complex algorithm, they were not reflected in the pseudo code shown in Fig. 2. In order to allow for future reproduction of our results, the C++ code of our branch-and-bound algorithm is posted online at https://sites.google.com/view/kfleszar/research.

The upper bound procedure is the most time consuming component of our overall algorithm, with solutions of TP(α) taking most of the computation time, followed by solutions of FKP(i, k, α). TP(α) is solved using a fast procedure developed in Fortran by Sustarsic (1999) and converted by us to C++. Still, these solutions take more than half of the computation time of the overall algorithm. Solutions of FKP(i, k, α), each of which is dominated by the sorting step, also take considerable computation time since the problem is solved a large number of times within the procedure.

2.4. Heuristic solutions

An initial solution of our branch-and-bound algorithm is obtained using a modified version of the greedy heuristic of Hiley & Julstrom (2006), further improved by a simple tabu search.

The modified greedy heuristic, shown in Fig. 3, starts with all items unpacked, stored in set U, and fills one knapsack at a time. The first item packed in each knapsack is the item that fits and has the largest density (lines 3–4), where density is the value defined by equation (7) calculated with respect to all other unpacked items divided by the weight. Then, while items can still be added to the knapsack, items with the largest densities with respect to those already packed in the knapsack are packed (lines 5–7). The last step, which is not present in the original heuristic of Hiley & Julstrom (2006), is to replace the last packed item with the one that fits and has the largest value (rather than density) with respect to those items already packed in the knapsack (lines 8–10). While this additional step often selects the same last item as in Hiley & Julstrom (2006), it sometimes assigns a larger and more valuable item.

The solution obtained by the modified greedy heuristic is further improved by a simple tabu search that searches the space of all feasible moves, including additions (packing an unpacked item in one of the knapsacks), removals (removing a packed item), transfers (moving an item from one knapsack to another), and swaps (exchanging two items between their respective knapsacks or between a knapsack and the set of unpacked items). In each iteration, all moves are evaluated and the move with the best change of objective function value is adopted. If the adopted move worsens the current objective function value, a reverse move is marked as tabu for a specified number of iterations (tabu tenure) in order to avoid reversing the move in some following iterations. The best solution obtained in the process is saved. We set the tabu tenure to 10 iterations and run the tabu search until there is no improvement of the best solution in the last 10,000 iterations.

Heuristic solutions are also computed during the branch-and-bound search. A variant of the modified greedy heuristic is called for some of the new nodes created through branching (see line 17 in the algorithm in Fig. 1). Since this heuristic starts with a partial solution defined by sets N_k for $k \in M$, it proceeds slightly differently than the heuristic shown in Fig. 3. It first fills all partially filled knapsacks, which obviates the need for lines 3–4, and then fills all remaining knapsacks. In order to limit the computational time taken by this heuristic, it is executed for each of the first 10,000 branch-and-bound nodes, then for every hundredth node up to the first one million, and then for each thousandth.

A different heuristic is called at the end of the upper bound procedure (Fig. 2, line 41). This heuristic completes the partial solution represented by sets N_k for $k \in M$ based on the solution of the transportation problem TP(α), as described in Section 2.3.

3. Computational experiments

Our algorithm is coded in C++ and compiled using Microsoft Visual Studio 2017. Tests are run on a PC with Intel Core i7-3770 3.4 GHz and 32 GB of RAM. All data sets, detailed results, and the source code are published online at https://sites.google.com/view/kfleszar/research.

We compare our algorithm with the branch-and-price algorithm BP1, the best of the algorithms proposed by Bergman (2019). The results of BP1 were obtained on a PC with Intel Core i7-4770 3.4 GHz and 32 GB of RAM, the processor of which has the same

Table 2
Results for small instances ($n = 20, \dots, 35$).

set	subset	#	BNB				BP1				Gurobi-QM				Cplex-RLT1			Cplex-MDRLT1		
			opt	%gap	avgT	maxT	opt	%gap	avgT	maxT	opt	%gap	avgT	maxT	opt	%gap	avgT	opt	%gap	avgT
HJ	$n = 20$	45	**45**	**0.000**	**0.1**	**0.5**	**45**	**0.000**	3.6	53.8	**45**	**0.000**	16.3	212.7	**45**	**0.000**	4.2	**45**	**0.000**	2.1
	$n = 25$	45	**45**	**0.000**	**1.6**	**12.3**	44	0.003	95.8	3600.0	41	0.716	515.9	3600.1	**45**	**0.000**	62.1	**45**	**0.000**	120.0
	$n = 30$	45	**45**	**0.000**	**15.6**	**93.6**	**45**	**0.000**	151.2	1882.1	30	4.431	1711.0	3600.1	40	0.469	727.3	44	0.070	462.4
	$n = 35$	45	**45**	**0.000**	**372.2**	**3532.8**	44	0.007	455.2	3600.0	15	17.142	2647.3	3600.1	29	2.154	1974.2	23	2.458	2335.0
	$m = 3$	60	**60**	**0.000**	**66.1**	**1052.9**	59	0.006	382.2	3600.0	46	3.172	1006.1	3600.1	56	0.204	619.9	54	0.403	547.3
	$m = 5$	60	**60**	**0.000**	191.7	3532.8	**60**	**0.000**	**67.2**	**566.1**	36	7.971	1687.5	3600.1	45	1.644	1198.6	50	1.091	843.6
	$m = 10$	60	**60**	**0.000**	**34.3**	**937.0**	59	0.002	80.0	3600.0	49	5.574	974.3	3600.1	58	0.119	257.5	53	0.403	798.8
	$d = 0.25$	60	**60**	**0.000**	**13.5**	**216.4**	**60**	**0.000**	37.8	456.5	59	0.116	238.3	3600.1	59	0.049	365.2	57	0.188	428.9
	$d = 0.50$	60	**60**	**0.000**	166.1	3532.8	**60**	**0.000**	**133.2**	**2411.6**	42	4.529	1354.8	3600.1	48	1.391	906.2	51	0.953	834.3
	$d = 0.75$	60	**60**	**0.000**	**112.4**	**2003.0**	58	0.008	358.4	3600.0	30	12.072	2074.8	3600.1	52	0.528	804.5	49	0.755	926.5
	all	180	**180**	**0.000**	**97.4**	**3532.8**	178	0.003	176.4	3600.0	131	5.572	1222.6	3600.1	159	0.656	692.0	157	0.632	729.9
SS	$m = 3$	75	**75**	**0.000**	**32.7**	**317.5**	73	0.006	312.5	3600.0	62	0.265	969.5	3600.7						
	$m = 5$	75	**75**	**0.000**	197.0	1279.7	**75**	**0.000**	**134.5**	N/A	29	0.875	2804.7	3600.4						
	$m = 10$	75	**75**	**0.000**	68.6	**2152.8**	74	0.003	**57.0**	3600.0	73	0.009	583.1	3600.0						
	all	225	**225**	**0.000**	**99.4**	**2152.8**	222	0.003	168.0	3600.0	164	0.383	1452.5	3600.7						

clock speed as that of the processor of our PC, but is one generation newer and marginally faster.

Additionally, we compare our algorithm with the QMKP model (1)–(4) solved using Gurobi version 9.00, run on the same PC as our algorithm with the number of threads set to one and parameter PreQLinearize left at its default value. This parameter controls the linearization of the quadratic model and preliminary testing showed that the default value (automatic linearization) provides the best results, almost the same as value 1 (strong LP relaxation) and much better than the other values, 2 (compact relaxation) and 0 (leave Q matrices unmodified).

Finally, we compare our algorithm with the two best performing linear models proposed by Galli et al. (2021), called RLT1 and MDRLT1. The published results of these models were obtained using Cplex version 12.10 run with the number of threads set to one on a PC with AMD Ryzen 7 2700X 3.7 GHz and 64 GB of RAM. The processor in this PC is known to be slightly faster than the processor of our PC in single-threaded computations.

3.1. Data sets

We use the same two sets of small benchmark instances as Bergman (2019). The first small set, HJ, is based on the generating scheme of Hiley & Julstrom (2006). For each combination of $n \in \{20, 25, 30, 35\}$, $m \in \{3, 5, 10\}$, and density $d \in \{0.25, 0.5, 0.75\}$, five instances were generated, for a total of 180 instances. Density is the fraction of pairwise item values that are not zero. Individual values and the non-zero pairwise values were drawn from $U(0, 100)$, weights from $U(1, 50)$, and capacities were set to $\lfloor 0.8/\mathrm{m} \times \sum_{i \in N} w_i \rfloor$.

The second small set, SS, which is based on the generating scheme of Saraç & Sipahioglu (2014), contains a total of 225 instances with $n = 30$, $m \in \{3, 5, 10\}$, $d = 0.5$. Weights were drawn from uniform distributions with varying upper limits, and both individual and pairwise values of items were generated randomly in such a way that they have various degrees of correlation with item weights. Capacities were set in the same way as for set HJ.

Additionally, we use the set of larger benchmark instances proposed by Galli et al. (2021). This set was generated in the same way as set HJ, except that the number of items was set to $n \in \{40, 45, 50, 55, 60\}$. This set contains 225 instances.

3.2. Tests on small instances

We first present the computational performance of algorithms on the two sets of small benchmark instances, HJ and SS, with 20–35 items in each problem instance.

Table 2 shows a comparison of our algorithm, denoted BNB, with BP1 of Bergman (2019), the quadratic model QM (1)–(4) solved using Gurobi 9.00, and the linear models RLT1 and MDRLT1 of Galli et al. (2021) solved using Cplex 12.10. All methods are run with a one-hour time limit per instance. The first two columns of the table show the set and the subset for which results are reported. The third column ('#') shows the cardinality of each subset. Then, for each of the compared methods, the columns show, respectively: the number of instances for which optimal solution is verified ('opt'), the average percentage optimality gap, calculated as $100\% \times (\mathrm{UB} - \mathrm{Best})/\mathrm{Best}$ ('%gap'), and finally the average ('avgT') and maximum ('maxT') computation times in seconds. The best values among the three compared methods are boldfaced. Note that one maxT value, that for BP1 and set SS, is reported as not available ('N/A'). This is because Bergman (2019) reports average computation times for groups of five instances in set SS, and when all instances in a subset are solved to optimality, the maximum computation time is not available. Also note that the maximum times for all subsets and the results for set SS are not shown in the table for models RLT1 and MDRLT1, as they are not reported by Galli et al. (2021).

On average, our BNB performs better than all the other methods. BNB solves all instances in both sets to optimality, never reaching the time limit of one hour per instance, although getting close to it for one instance.[1] With the same time limit per instance, BP1 does not verify optimality of 2 instances in set HJ and 3 in set SS, although optimality gaps for these instances are very small. The linear models, RLT1 and MDRLT1, do not verify optimality of, respectively, 21 and 23 instances in set HJ. The quadratic model, QM, does no verify optimality for about 27% of the instances in each of the two sets. Moreover, optimality gaps for these instances are rather large.

Analysis of the performance of the algorithms across different subsets of instances reveals additional insights. Clearly, for all

[1] For instance 157 in set HJ, 'randomQMKP_35_50_2', BNB took almost 59 minutes to find and verify an optimal solution. For all other instances in both sets HJ and SS, it took less than 36 minutes apiece.

Table 3
Results for large instances ($n = 40, \ldots, 60$).

subset	#	BNB			Cplex-RLT1		Cplex-MDRLT1	
		opt	%gap	avgT	%gap	avgT	%gap	avgT
$n = 40$	45	21	1.615	2442.2	6.014	9217.0	6.142	8183.2
$n = 45$	45	12	4.283	3012.4	11.562	10346.4	11.680	9772.1
$n = 50$	45	2	9.520	3529.6	17.043	10792.4	18.377	10486.4
$n = 55$	45	0	14.164	3600.4	21.287	10795.2	25.214	10800.0
$n = 60$	45	0	18.748	3600.3	26.094	10794.7	30.620	10800.0
$m = 3$	75	16	9.942	3034.4	10.465	9760.7	15.952	9271.5
$m = 5$	75	9	11.993	3405.5	19.241	10712.7	22.331	10637.5
$m = 10$	75	10	7.063	3271.0	19.495	10694.0	16.937	10116.0
$d = 0.25$	75	21	8.448	2900.5	16.055	9932.2	14.431	8539.8
$d = 0.50$	75	4	12.862	3435.3	19.344	10639.3	23.410	10686.3
$d = 0.75$	75	10	7.689	3375.2	13.801	10595.8	17.379	10798.9
all	225	35	9.666	3237.0	16.400	10389.1	18.407	10008.3

methods, problem instance difficulty grows rapidly with the number of items, n. This is expected as increasing n increases proportionately problem instance size, regardless of the values of all other parameters. For the subset of HJ with $n = 20$, all methods solve all instances to optimality, although our BNB is much faster than all the other methods. Indeed, for all subsets of HJ with $n \in \{20, 25, 30\}$, BNB is on average at least an order of magnitude faster than all the other algorithms. For the largest instances in set HJ with $n = 35$ items, as well as for all instances in set SS, which all have $n = 30$ items, BNB is still the fastest, but its computation time advantage over BP1 is not as marked as for the smaller instances of set HJ. It is noteworthy that our BNB as well as models RLT1 and MDRLT1 solve to optimality all instances in the subset of HJ with $n = 25$ items, with BNB taking at most 12.3 seconds per instance, while BP1 does not solve one of these instances to optimality in one hour.

As for the number of knapsacks, instances with $m = 5$ knapsacks are harder to solve for BNB, QM, RLT1, and MDRLT1 than instances with $m \in \{3, 10\}$. This is clear from the average computation times shown in Table 2 and, in the cases of QM, RLT1, and MDRLT1, also from the number of instances with verified optimality. On the other hand, the most difficult instances for BP1 are those with $m = 3$ knapsacks, as is evident from the average computation times. We believe BNB, QM, RLT1, and MDRLT1 find instances with $m = 3$ easy because of relatively smaller solution space. Also, while the solution space is much larger when $m = 10$, these instances are also easy for BNB, QM, RLT1, and MDRLT1 because very few items on average fit in each knapsack, which leads to opportunities for significant reductions early in the branching process. On the other hand, BP1 finds instances with $m = 3$ knapsacks most difficult because of the relatively larger average number of items per knapsack and, thus, a relatively larger solution space in each pricing problem.

As for the density, instances of set HJ with a low density of $d = 0.25$ are the easiest for all methods. While for BP1 and QM, the difficulty keeps increasing as density increases to 0.5 and then to 0.75, BNB, RLT1, and MDRLT1 find instances with $d = 0.5$ slightly harder than those with $d = 0.75$. Lower density should make instances easier for all methods. Hence, the fact that instances with density 0.5 are slightly harder for BNB, RLT1, and MDRLT1 than instances with density 0.75 is somewhat surprising.

Overall, BNB is on average better than BP1: given the same time limit, BNB solves all instances to optimality while BP1 does not verify optimality for 5 out of 405 instances, and the average computation time of BNB is about 1.7-1.8 times smaller than that of BP1 for both sets, HJ and SS. However, average computation time of BP1 is better than that of BNB for some subsets, most notably when the number of knapsacks $m = 5$ and when density $d = 0.5$. This difference in performance indicates that there might be an opportunity for designing an even better exact algorithm that combines the strengths of both approaches.

Additionally, our BNB has better performance than mathematical models QM, RLT1, and MDRLT1. BNB is on average much faster for all subsets of small benchmark problem instances, and solves all instances to optimality in one hour per instance, while the mathematical models do not verify optimality for some larger instances. However, since QM, RLT1, and MDRLT1 are all mathematical models, they have the advantage of being simple to formulate and use. It is also easy to modify them to solve variants of the QMPK. Finally, their performance might improve in the future as performance of mixed-integer linear and quadratic solvers such as Gurobi and Cplex improves.

3.3. Tests on larger instances

We now present the computational performance of algorithms on the set of larger benchmark instances with 40–60 items in each instance.

Table 3 shows a comparison of our BNB with the linear models RLT1 and MDRLT1 of Galli et al. (2021) solved using Cplex 12.10. Quadratic model QM is not tested, as it was already shown in the previous section to be inferior to RLT1 and MDRLT1. BNB was run with a one-hour time limit per instance, while RLT1 and MDRLT1 were run with a three-hour time limit per instance. Columns in Table 3 have the same meaning as in Table 2. Maximum times are not presented as each method reaches its respective time limit for each subset reported in Table 3. The numbers of instances solved to optimality are not shown in the table for models RLT1 and MDRLT1, as they are not reported by Galli et al. (2021).

The results on the larger benchmark problem instances reveal similar insights to those observed for the small instances in the previous section. Even though our BNB has a shorter time limit, it achieves smaller average optimality gaps than RLT1 and MDRLT1 in all subsets of problem instances presented in Table 3. As before, problem instance difficulty increases for all methods as the number of items, n, increases. Also as before, instances in subsets with the medium number of knapsacks, $m = 5$, and medium density, $d = 0.50$, are on average harder to solve for all presented methods than instances in all the other subsets.

4. Conclusions

We presented a new branch-and-bound algorithm for the QMKP. Its strength derives from a new upper bound that divides the pairwise item values among individual items, estimates maximal potential contribution by each individual item, and casts the problem of upper bound calculation into a transportation model. Other significant contributions include: using local search to improve the upper bound by adjusting the division of pairwise item values; forbidding some item-to-knapsack assignments using reduced costs from the transportation model; and judicious choice of the item to branch on.

The results of computational experiments show that our branch-and-bound algorithm outperforms the best previously proposed algorithms, branch-and-price algorithm BP1 of Bergman (2019) and linear models RLT1 and MDRLT1 of Galli et al. (2021), in terms of both average computation time and number of instances with optimality verified within a time limit of one hour per instance. However, BP1 exhibits better performance for some types of problem instances. This difference in performance indicates that there may be an opportunity for designing an even better exact algorithm that combines the strengths of both approaches.

A limitation of our algorithm is that it is designed for the QMKP with identical knapsacks, while all the other approaches can solve the problem with varying knapsack capacities. However, with some additional coding, it should be possible to extend our algorithm to the case of varying knapsack capacities. Moreover, our algorithm is highly optimized for the QMKP and is not easy to extend to similar types of problems, which is not the case for mathematical models such as RLT1 and MDRLT1. Additionally, mathematical models have the advantage of potentially benefiting from future performance improvements in mathematical solvers such as Gurobi and Cplex.

Acknowledgement

I would like to express my deepest gratitude to Dr. Khalil S. Hindi for his invaluable help and unwavering support throughout my research career.

References

Bergman, D. (2019). An exact algorithm for the quadratic multiknapsack problem with an application to event seating. *INFORMS Journal on Computing, 31*(3), 477–492. https://doi.org/10.1287/ijoc.2018.0840.

Dawande, M., Kalagnanam, J., Keskinocak, P., Ravi, R., & Salman, F. (2000). Approximation algorithms for the multiple knapsack problem with assignment restrictions. *Journal of Combinatorial Optimization, 4*(2), 171–186. https://doi.org/10.1023/A:1009894503716.

Elhedhli, S., & Coffin, J.-L. (2005). Efficient production-distribution system design. *Management Science, 51*(7), 1151–1164. https://doi.org/10.1287/mnsc.1050.0392.

Galli, L., Martello, S., Rey, C., & Toth, P. (2021). Polynomial-size formulations and relaxations for the quadratic multiple knapsack problem. *European Journal of Operational Research, 291*(3), 871–882. https://doi.org/10.1016/j.ejor.2020.10.047. https://linkinghub.elsevier.com/retrieve/pii/S0377221720309383

Gerla, M., Kleinrock, L., & Gerla, M. (1977). On the topological design of distributed computer networks. *IEEE Transactions on Communications, 25*(1), 48–60. https://doi.org/10.1109/TCOM.1977.1093709.

Hiley, A., & Julstrom, B. (2006). The quadratic multiple knapsack problem and three heuristic approaches to it. In *GECCO 2006 - Genetic and evolutionary computation conference: 1* (pp. 547–552).

Martello, S., & Toth, P. (1990). *Knapsack problems. algorithms and computer implementations*. John Wiley & Sons.

Mathur, K., Salkin, H., & Morito, S. (1983). A branch and search algorithm for a class of nonlinear knapsack problems. *Operations Research Letters, 2*(4), 155–160. https://doi.org/10.1016/0167-6377(83)90047-0.

Peng, B., Liu, M., Lü, Z., Kochengber, G., & Wang, H. (2016). An ejection chain approach for the quadratic multiple knapsack problem. *European Journal of Operational Research, 253*(2), 328–336. https://doi.org/10.1016/j.ejor.2016.02.043.

Qin, J., Xu, X., Wu, Q., & Cheng, T. (2016). Hybridization of tabu search with feasible and infeasible local searches for the quadratic multiple knapsack problem. *Computers and Operations Research, 66*, 199–214. https://doi.org/10.1016/j.cor.2015.08.002.

Saraç, T., & Sipahioglu, A. (2014). Generalized quadratic multiple knapsack problem and two solution approaches. *Computers and Operations Research, 43*(1), 78–89. https://doi.org/10.1016/j.cor.2013.08.018.

Sustarsic, A. M. (1999). *An efficient implementation of the transportation problem*. UNF Digital Commons.

Tlili, T., Yahyaoui, H., & Krichen, S. (2016). An iterated variable neighborhood descent hyperheuristic for the quadratic multiple knapsack problem. *Studies in Computational Intelligence, 612*, 245–251.