Development of Core to Solve the Multidimensional Multiple-Choice Knapsack Problem

Taha Ghasemi 1**, Mohammadreza Razzazi 1,2

1 Software Systems R&D Lab, Computer Engineering and IT Department, Amirkabir University of Technology, #424 Hafez Avenue, P. O. Box 15875-4413, Tehran, Iran {tghasemi, razzazi}@aut.ac.ir

2 Institute for Studies in Theoretical Physics and Mathematics (IPM), Tehran, Iran

Abstract

The multidimensional multiple-choice knapsack problem (MMKP) is an extension of the 0-1 knapsack problem. The core concept has been used to design efficient algorithms for the knapsack problem but the core has not been developed for the MMKP so far. In this paper, we develop an approximate core for the MMKP and utilize it to solve the problem exactly.

Computational results show that the algorithm can solve large uncorrelated instances (up to 100 classes and 100 items in each class) and correlated instances with small number of constraints (up to 5) efficiently. In particular, it solves recently published hard instances for the MMKP in less than a second. The algorithm consumes negligible memory, and compared with the best previous exact algorithm for the MMKP performs significantly faster.

Keywords
Multidimensional Multiple-Choice Knapsack, Core, Branch and Bound, Integer Linear Programming, Combinatorial Optimization.

1 Introduction

In the 0-1 knapsack problem (KP), there is a set of items, each with a weight and a profit, and we want to select a subset of them such that we gain the maximum profit from them while their total weight is less than a specific amount (known as the knapsack capacity). The multidimensional multiple-choice knapsack problem (MMKP) is an extension of the KP. In the MMKP, we have several classes (or groups) of items and several limited resources. Each item gives us a profit and consumes a specific amount from each resource. The goal is to choose one item from each class to maximize the profit sum while satisfying the constraints on the resources. Formally, the MMKP can be stated as:

\[
\begin{align*}
\text{max} & \quad \sum_{i \in C} \sum_{j \in N_i} p_{ij} x_{ij} \\
\text{s.t.} & \quad \sum_{i \in C} \sum_{j \in N_i} w_{ij} x_{ij} \leq b_k \quad k \in \{1, \ldots, m\} \\
& \quad \sum_{j \in N_i} x_{ij} = 1 \quad i \in C \\
& \quad x_{ij} \in \{0, 1\} \quad i \in C, j \in N_i
\end{align*}
\] (1)

where \(C\) is the set of classes, \(N_i\) is the set of items of class \(i\), \(m\) is the number of available resources, \(b_k\) is the available amount from resource \(k\), \(p_{ij}\) is the profit of the \(j^{th}\) item of class \(i\), \(w_{ij}^k\) is its consumption from resource \(k\), and \(x_{ij}\) is a decision 0-1 variable determining whether the corresponding item is chosen or not. Constraints (2) restrict the usage of resources, and Constraints (3) are multiple-choice or generalized upper bound (GUB) constraints and state that exactly one item from each class must be chosen. If we relax the integrality constraints (4) and allow \(x_{ij}\)'s to take fractional values from 0 to 1, we obtain the Linear MMKP (LMMKP). Let \(g = |C|\), \(n_i = |N_i|\), and \(n = \max_{i \in C} \{n_i\}\). If we only have one resource available, that is \(m=1\), the problem is called multipe-
choice knapsack problem (MCKP). The problem reduces to the KP if we only have two items in each class, one of them corresponds to one of the knapsack items and the other item has zero profit and weight. In this setting, when \( m = 1 \) we obtain the KP, and for \( m > 1 \) we obtain the multidimensional knapsack problem (MKP). There is another extension of the KP known as the multiple knapsack problem (which is also abbreviated by MKP). In this problem, we want to select a subset of items and assign them to \( m \) knapsacks such that the capacity of each knapsack is respected. Note that MMKP is not a generalization of this problem.

The MMKP appears in configuration problems. In these problems, we have a system composed of several components and for each component there are several candidates. We want to choose one candidate for each component so that the overall system gives us the maximum profit while the design constraints are satisfied. Applications of this kind include: the optimal composition of workflow tasks from several options for each task (Jäger, 2007), determining the optimal quality of service for each session in a multimedia system (Shahadatullah-Khan, 1998), menu planning, and budgeting. It can also be used to solve integer separable nonlinear knapsack problems by sampling each nonlinear function (possibly nonconvex) at integer points in the domain of the corresponding variables and group them as the items of a class.

Since the MMKP contains the KP as a special case, it is NP-hard. Thus, many researchers have tried to develop heuristic approaches to solve the problem (Akbar & et al., 2001, 2006; Cherfi & Hifi, 2008; Hifi et al., 2006; Hiremath & Hill, 2007; Khan, 1998; Parra-Hernandez & Dimopoulos, 2005; Ren & Feng, 2010). There are only two algorithms, specific to the MMKP, for solving the problem exactly: BBLP (Shahadatullah-Khan, 1998) and EMKP (Sbihi, 2007), both are best-first branch-and-bound (B&B) algorithms. In BBLP, each class is mapped to a level in the tree with its items as the branches of the nodes of that level. The upper bound at each node is calculated by solving the corresponding LMMKP and the arrangement of the levels is determined using the highest fractional heuristic. At each step, the best node, which is the node with the highest upper bound among the available nodes, is selected for branching. Reaching the first feasible node will terminate the algorithm. More recently, Sbihi (Sbihi, 2007) has proposed a new algorithm named EMKP. The idea is to enumerate all possible solutions from the highest objective value solution down to the lowest one until reaching the first feasible solution. The enumeration is performed using a best-first B&B algorithm. The B&B tree is organized similar to BBLP, but items in each class are sorted in non-increasing order of their profits. The algorithm forms an auxiliary problem by summing the constraints into one constraint and uses the solution of this auxiliary problem for the upper bound testing. The best node is the one that has the largest value after adding the most profitable items of the remaining (unassigned) classes of that node. At each step, the best node is selected and two new nodes, a son node and a brother node, are developed. The son node corresponds to assigning the next remaining class its first (most profitable) item while the brother node corresponds to assigning the next profitable item to the current class. Again, the algorithm terminates when reaching the first feasible solution.

The core was first proposed and used for solving large (uncorrelated) knapsack problems. Balas and Zemel (Balas & Zemel, 1980) empirically observed that the solution of the linear relaxation of the KP (LKP) is only different in a few items from the optimal solution of the KP and called the interval containing these items the core. Since the core cannot be identified before solving the problem, they used an approximation of the core by estimating its size around the break item. They used this core in a B&B algorithm to obtain a good lower bound. The approximate core was used in the later B&B approaches with some enhancements but still with a fixed sized core (S Martello & Toth, 1990). Pisinger introduced the idea of the dynamic expansion of the core and used it in a B&B algorithm (expknap) (D. Pisinger, 1994) and later in a dynamic programming algorithm (minknap) (D. Pisinger, 1997). Minknap has advantages over expknap due to its minimal expansion of the core and using domination among dynamic programming states. Minknap and its later enhancement (Silvano Martello et al., 1999) are now the best algorithms for solving the KP instances. Pisinger also applied these ideas to the MCKP and obtained an efficient dynamic programming algorithm for solving the MCKP (D. Pisinger, 95). In multidimensional cases, Puchinger et al. (Puchinger et al., 2006) have recently proposed a way to identify the core in the MKP and applied it in a memetic framework and a relaxation guided variable neighborhood search to obtain an approximate approach to solve the MKP.

Following the success of using core in solving knapsack problems, in this paper we derive an approximate core for the MMKP and use it to solve the problem exactly. To the best of our knowledge, no one has developed the core for the MMKP so far and, more generally, no one has applied it to solve multidimensional problems exactly. By viewing the core concept in a more abstract way, we propose a three-step procedure to define the core. The approach is to take a base point in the solution space and then define an ordering relation among the variables. This ordering is then used for defining an appropriate subspace around the base point, which constitutes the core. We then follow these steps for the MMKP and empirically choose the best candidate for each step.

To solve the problem exactly, an enumeration method and proper incorporation of the core into it must be decided. Dynamic programming encounters curse of dimensionality in the multidimensional cases and requires considerable memory. The basic idea of this paper is to incorporate expansion of the core into a B&B algorithm while retaining the minimal expansion of the core like in dynamic programming approaches. The B&B tree is
arranged based on the core so that the core is expanded during the depth first navigation of the tree. This way the current core is completely enumerated before the expansion and will be expanded if required. This is in contrast with the previous usages of the core in the B&B approaches that took fixed sized cores or expknap algorithm, which may expand the core before complete enumeration. For efficient navigation of the solution space, we use two fathoming tests, which are effective and fast. We use propagation of the mixed surrogate constraint, which is a tight constraint and can be used to prune sub trees with little computation. Moreover, we use the linear programming solution of the surrogate-relaxed problem for the upper bound testing and fixations of variables whenever the algorithm reaches a new incumbent.

The paper is organized as follows. Section 2 gives some background about the core and MCKP. In section 3, the core for the MMKP is developed. In section 4, the exact algorithm and its complexity analysis are presented and finally in section 5, the computational experiences and comparison of the algorithms are discussed.

2 Background

This section, in the first part, gives more details about the core concept and its usage in the exact approaches. MCKP is a MMKP with only one constraint and has well-established properties. Later, we relax MMKP to MCKP and use it to define core and do fathoming tests. So in the next part of this section, properties of MCKP and its linear programming solution are discussed.

2-1 Core and Knapsack Problems

Before 1980, one usual method to solve the KP was as follows. First, solve the LKP and use its solution for computing some lower bounds and upper bounds. Then use these bounds with a variable fixation method in an enumeration procedure like B&B to solve the problem. The LKP is solved exactly using the following well-known heuristic algorithm. Let \( p_i, w_i, x_i \) be the profit, weight, and decision variable of item \( i \), \( n \) be the total number of items, and \( c \) be the knapsack capacity.

1. For each item compute the efficiency: \( e_i = \frac{p_i}{w_i} \).
2. Sort the items in non-increasing order of efficiency.
3. Put the items in the knapsack until it is filled: \( \sum_{i=1}^{b-1} w_i < c \).

The \( b^{th} \) item, which caused the violation of the knapsack capacity, is called the break item. The optimal solution of the LKP is obtained by setting:

\[
x_i = \begin{cases} 
1 & 1 \leq i < b \\
\left(c - \sum_{i=1}^{b-1} w_i \right)/w_b & i = b \\
0 & b < i \leq n 
\end{cases}
\]  

(5)

Since the LKP is a relaxation of the KP, its optimal value is an upper bound for the KP. In addition, setting \( x_b = 0 \) will generate a feasible solution, which can be used as a lower bound. From now on, we assume that the items are sorted according to the above ordering.

If the number of items in the KP is large, sorting of all items will take a significant portion of the solution time (for uncorrelated instances). The core was first proposed to avoid complete sorting of the items in large knapsack problems (Balas & Zemel, 1980). Balas and Zemel observed that comparing the optimal solution of the KP with the optimal solution of the LKP, the difference is just in a small number of items around the break item. They called the interval containing these items the core. Therefore, the core contains the variables whose efficiencies are near the efficiency of the break item. In Figure 1, the optimal solutions of a sample LP and the associated LKP are depicted and items in the core are specified.

![Figure 1: Core in a sample knapsack](image-url)
More precisely the core \( C \) is defined as: (excluding cases with no break item)

\[
s = \min \{ i \in N \mid x_i^* = 0 \}, \quad t = \max \{ i \in N \mid x_i^* = 1 \}
\]

\[
C = [s, t]
\]

(6)

where \( x^* \) is the optimal solution of the KP and \( N \) is the set of sorted items according to their efficiencies.

Balas and Zemel claimed that the core size, i.e. \( t-s \), is small compared to the total number of items (Balas & Zemel, 1980) (although later studies corrected this claim (David Pisinger, 1999)). Hence, having the core defined we can solve the KP more easily by just concentrating on the items in the core and fixing the variables before \( s \) to 1 and after \( t \) to 0 and avoid sorting of all items. By fixing the variables to 1, some part of the knapsack capacity is consumed and the items in the core with the remaining capacity constitute a small knapsack problem, namely the core problem, which must be solved:

\[
\begin{align*}
& \max \sum_{i=s}^{t} p_i x_i, \\
& \sum_{i=s}^{t} w_i x_i \leq c - \sum_{i=r}^{t} w_i, \quad x_i \in \{0,1\} \quad i \in C
\end{align*}
\]

(7)

Unfortunately, the core, with the above definition, cannot be found before solving the problem exactly. Thus, one must develop an approach to obtain an approximate core. In the KP, the approximate core can be any interval containing the break item. For example, Balas and Zemel obtained the break item with a partitioning algorithm in \( O(n) \) and took 25 items on each side as the approximate core (Balas & Zemel, 1980). Later Martello and Toth suggested \( 2\sqrt{n} \) as the size of the approximate core (S Martello & Toth, 1990).

The approximate core was used in B&B methods in the following general way. Define the core problem and solve it exactly or approximately. The result will be a (near) optimal solution. Try to prove the solution is optimum by upper bound testing and variable fixation. If this process fails, use the solution as a good starting lower bound in an ordinary B&B method.

One problem with the above approach is that if the process of proving the optimality is failed, the core will be no longer used in the continuation of the process. Pisinger introduced the novel idea of the dynamic expansion of the core while solving the problem (D. Pisinger, 1994, 1997). This way the size of the core does not remain fixed and is used throughout the solution process. This also leads to finding the exact core during the process. He used this idea in a B&B method (expknnap) (D. Pisinger, 1994) and later in a dynamic programming approach (minknap) (D. Pisinger, 1997). Expknnap has the disadvantage of expanding the core before complete enumeration so it does not ensure the minimal expansion property. That is, despite possibility of finding the optimal solution within the current core, it may expand the core. In contrast, minknap at each step completely enumerates the current core before expansion. The algorithm solves the problem using a forward dynamic programming. It starts with a 1-item core like expknnap. At each step, it keeps the partial solutions resulting from the enumeration of the core in the dynamic programming states. If the optimal solution is not found (or its optimality cannot be proved), it symmetrically expands the core from one side and introduces a new item into the core, then completely enumerates the new core using the dynamic programming states of the previous steps and generates new states. This process is repeated until all other items outside the core can be fixed. Minknap besides minimal expansion takes advantage of the domination among dynamic programming states and is now considered as one of the best algorithms for solving the KP.

To solve multidimensional problems exactly, dynamic programming is not a good candidate compared to B&B method because of its intensive memory consumption. However, B&B approaches have not used core as efficient as dynamic programming approaches. In this paper, we no longer consider the core as a way to avoid sorting, but as a guide for the enumeration process to search in the promising region of the solution space. Based on this view, we arrange the B&B tree completely based on the core and obtain the minimal expansion property. In this setting, the core acts more like a variable selection heuristic in a B&B algorithm. This approach is explained in section 4.

2-2 Multiple Choice Knapsack

MCKP is a MMKP with only one constraint. In this section some basic properties of MCKP is briefly described. It has been shown that MCKP and its linear relaxation (LMCKP) have the following three properties (D. Pisinger, 95). We use the notation of MMKP for MCKP except the constraint number indexes are dropped.
1. If \( \frac{w_i}{w_j} \geq \frac{p_i}{p_j} \) and \( p_j \leq p_a \) then \( x_j = 0 \) in every optimal solution of MCKP. In other words, item k dominated item j in class i because it has less weight while give us more profit. Figure 2 shows items in a sample class. Each point represents an item in that class. Items that are outside of the shaded region are dominated by at least one item.

2. If \( \frac{p_i}{w_i} - \frac{p_j}{w_j} > 0 \) then \( x_{ik} = 0 \) in every optimal solution of LMCKP. This property implies that in order to solve LMCKP, in each class we only need to consider items that make lines of upper convex hull. Other items are "lp-dominated" by these items. So e.g. only bolded points in Figure 2 are of interest for LMCKP. Let \( L_i \) be the set of non-lp-dominated items in class i.

3. In every optimal solution of MCKP variables of all classes, except at most one class, are 0 or 1. Exceptional class has two fractional variables which are the two end points of a line in the upper convex hull.

Using properties 2 and 3, we can optimally solve LMCKP with the following greedy algorithm:

1. Remove items in \( N_i - L_i \), and sort remained items in increasing order of their weights in each class.
2. Select the first (lightest) item from each class and set \( p = \sum_{i \in C_i} p_i \), \( w = \sum_{i \in C_i} w_i \), \( x_{i,j} = 1 \) \( \forall i \in C \).
3. Calculate slope of lines: \( \lambda_{ij} = (p_j - p_{i,j-1})/(w_j - w_{i,j-1}) \).
   This value indicates change of efficiency if we choose item j instead of item j-1 in class i.
4. Sort lines in decreasing order of their slopes. Start from the first line.
5. Select next line with the slope \( \lambda_{ij} \). If \( w + w_j \geq b \) go to step 6. Else set: \( p = p + p_{ij} - p_{i,j-1} \), \( w = w + w_j - w_{i,j-1} \), \( x_{i,j} = 1 \), \( x_{i,j-1} = 0 \). Repeat this step unless there are no more lines.
6. If \( w = b \) algorithm is finished with an integer solution and p as the optimal value. Else assume \( \lambda_{ij} \) was the slope of the last line selected. Calculate two fractional variables: \( x_j = (b - w)/(w_j - w_{i,j-1}) \), \( x_{i,j-1} = 1 - x_j \). And the optimal value would be \( p + (b - w)\lambda_{ij} \).

We will refer to the last line selected in step 5 as the break line in LMCKP. In the worst case we have \( n_g \) lines so the algorithm is of \( O(n_g \log(n_g)) \).

3 Development of Core

3-1 Approach

According to the definition of Balas and Zemel, the core is the minimal interval of items which contains items with different values in the optimal solution of the KP and the optimal solution of the associated LKP (Balas & Zemel, 1980). Thus, the (exact) core can be seen as a subspace of the solution space containing the optimal solution. In the KP, this subspace is obtained by fixing the items before the core interval to 1 and after the core interval to 0, assuming the items are sorted in non-increasing order of efficiency. Note that there is another definition of the core which defines the core as a subset of items that must be changed to convert the associated LKP optimal solution to the KP optimal solution (Goldberg & Marchetti-Spaccamela, 1984). However, we will not consider it here.

In an approximate core, the core may no longer contain the exact solution. In general, we can define three metrics for the quality of an approximate core: size, accuracy, and exactness. The size of the core is the size of the subspace that the core induces. The core accuracy is defined as the ratio of the value of a best solution in the core subspace to the value of an optimal solution of the problem. The core exactness indicates whether the core subspace has one of the optimal solutions or not. An approach for deriving an approximate core for a problem may result in a core with different qualities for different instances of the problem. Hence, the above measures can be interpreted as random variables. For example, the core exactness is an indicator random variable, whose
expectation shows the probability that the core is exact over all instances of the problem for a particular core-deriving approach.

Overall quality of the core depends on its usage. For example, we can develop an approximate algorithm by a complete enumeration of the approximate core. In this approach, an approximate core, which is smaller and more accurate, is more desirable. Nevertheless, as long as the remaining subspace outside the core contains an optimal solution this process cannot be finished. Hence, it is also desirable for the core to be exact as much as possible and the probability of its exactness to be increased rapidly as the size of the core is increased. This also indicates why the core-based approaches are very efficient for randomly generated (uncorrelated) instances but are not sufficient for solving harder problems like problems with large duality gaps. This is in part because, according to the approaches developed so far, an exact core at most can lead the algorithm to reach the optimal solution earlier. However, for proving the optimality of the solution the algorithm needs to search the remaining subspace. In uncorrelated instances, this subspace is pruned easily and the algorithm finishes very quickly but in hard instances, the remaining subspace still needs an extensive enumeration.

Development of an approach to derive a high quality approximate core for a problem is not a trivial task. We propose the following three steps for deriving an approximate core empirically. This procedure is a direct generalization of the approach used to obtain an approximate core for the KP.

1) Take an arbitrary (even infeasible) solution $\bar{x}$ in the solution space. We call this solution the base solution. The base solution acts as a center of the core and the core will be defined around it. If this point is close to an optimal solution, we can have a small exact core. Since an optimal solution can be close to the boundary of the feasibility region, we do not force the base solution to be feasible. For simplicity, here we also assume the base solution is integral. The base solution can be a (rounded) solution of a relaxation of the problem, or a heuristic solution.

2) Define a total ordering relation among the variables. Let $x^*$ be an optimal solution and denote the probability of change for item $i$ as $P_{ch}(i) = \Pr\{x^*_i \neq \bar{x}_i\}$, where this probability is defined over all instances of the problem. We are interested in an ordering with the following property:

$$x_i \leq_o x_j \Rightarrow P_{ch}(i) > P_{ch}(j) \quad \forall i, j \in N$$

where $N$ is the set of variables and $\leq_o$ is the ordering relation on the set $N$. Variables at the beginning of this ordering are more probable to take different values in the optimal solution than the base solution. One empirical way to obtain this relation is to first statistically compare the base solution with the optimal solution over several instances and estimate the probability of change for each variable. Then examine various precedence (or utility) functions to find a function that suitably gives higher precedence to variables with higher probability of change. The ordering relation is then defined based on these precedence values.

3) Define a subspace around the base solution using the ordering relation. One simple way to do this is to take the first $k$ variables in the relation as the items of the core and fix other variables to their values in the base solution. Thus, assuming items are sorted according to the ordering relation, the core is defined as

$$C = \{x_i : 1 \leq i \leq k\}$$

and subspace is defined by setting $x_i = \bar{x}_i, \ i \notin C$. In this way, the ordering can easily be exploited in the core expansion by increasing $k$. A subspace can be defined by addition of constraints to a space. In the previous method, the fixation of variables corresponds to the addition of equality constraints. Thus, we may define the subspace by addition of some constraints related to the base solution assuming the items are ordered according to the ordering relation. We will use this idea in the next section.

In the MMKP (and more generally in multiple-choice programs), we can define two orderings instead of one, one for ordering of the classes and one for ordering of the items in each class. Therefore, we can define the subspace based on these orderings. For example, we can sort the classes according to the ordering of the classes and items of each class according to the ordering of the items. Then we take first $k_1$ classes and in these classes take first $k_2$ items. Variables of the other $n-g$-classes will be set according to the base solution. Variables of the other $n-g$-classes will be set to zero. This can be viewed from another point of view. Let $t = \sum_{i=1}^{n} n_i$ and $x \in \{0,1\}^t$ be a t-dimensional vector representing an integral solution of the MMKP. The solution can be
represented in another way as a g-dimensional vector $s$ where $s_i \in N_i$. $s_i$ indicates which item of class $i$ is selected. These two representations are equivalent according to (10).

$$j = s_i \iff x_{i,j} = 1 \land \forall k \in N_i, k \neq j \ x_{i,k} = 0$$

We will refer to this representation as the integer representation of the solution. In this representation, the core may be viewed as a subset of variables and a subset of values that a variable can take on.

3-2 The MMKP Core

In this section, we apply the proposed approach to the MMKP to derive an approximate core for it. At first, we define some metrics to measure the quality of different choices. Then, we follow these steps, for each step examine various candidates, and choose the best one according to the defined metrics.

3-2-1 Metrics

To compare different candidates for each step, we define the following metrics. Let $\tilde{s}$ and $s^*$ be the integer representations of base and optimal solutions, respectively.

1. The Hamming distance of $\tilde{s}$ with $s^*$: $h = |\{i \in C | \tilde{s}_i \neq s^*_i\}|$.
2. The Manhattan distance of $\tilde{s}$ with $s^*$: $d_m = \sum_{i \in C} |\tilde{s}_i - s^*_i|$.
3. The Chebyshev distance of $\tilde{s}$ with $s^*$: $d_{\infty} = \max_{i \in C} |\tilde{s}_i - s^*_i|$.
4. The position of the last class that is different in the solutions: $l_c = \max_{i \in C} \{i | s^*_i \neq \tilde{s}_i\}$.
5. The time required for a complete enumeration of all solutions in the core space.
6. The accuracy of the core, which is the ratio between the values of a best solution in the core space and an optimal solution of the problem.

$n_d$ indicates the quality of the base solution. $d_m$ and $d_{\infty}$ are metrics for the quality of ordering relations on items. $l_c$ measures the size of the exact core in the dimension of classes and indicates the quality of class ordering relations. The last two metrics can be used for estimating the quality of the obtained core (and thus the quality of the subspace-defining approaches). Each metric was computed by averaging over 100 randomly generated instances. These instances were generated using the method described in section 5 for generating uncorrelated instances with $r=1000$. To find an optimal solution, which is required in the computations of these metrics, we used a simple B&B algorithm. If an instance of the problem had multiple optimal solutions, we used one of them as a fixed optimal solution in all calculations. We also threw away infeasible instances. Thus, in this section, we always assume there is only one optimal solution for the problem.

3-2-2 Base Solution

We can relax the MMKP to the MCKP using the surrogate technique. In the surrogate technique, multiple constraints are replaced with a linear combination of them (Glover 1975), that is we multiply each constraint with a multiplier, and then add these constraints together generating one combined constraint. In this way, we can reduce the number of constraints of a problem. The new problem obtained in this way is a relaxation of the original problem since every feasible solution for the original problem must satisfy all the constraints, and thus it satisfies any linear combination of them, and thus it is also feasible for the new problem. Although we can use any desired numbers as multipliers, we would like to choose the multipliers so that the resulting problem has the smallest gap with the original problem (the so-called surrogate duality gap). One good and simple heuristic for computing such multipliers is to use the dual values of the linear relaxation of the problem, which also gives the “strongest” surrogate constraint (Fred Glover, 1968). Thus, to obtain an MCKP instance from an MMKP instance, we surrogate relax all the resource usage constraints of MMKP (that is constraint 2 in the formulation of the problem) into one constraint using the dual values of the LMMKP solution as the surrogate multipliers. We refer to the resulting problem as the associated MCKP and its linear relaxation as the associated LMCKP. Note that since the associated MCKP already has GUB constraints, their incorporation in the surrogate constraint does not make the problem tighter.

For computing the base solution, we considered three candidates: the rounded solution of the associated LMCKP, the optimal solution of the associated MCKP, and a heuristic solution of the problem. The linear programming solution of the MMKP is not a good candidate because it may contain many fractional values. Since we want to have an integer solution, this creates many options for rounding the LMMKP solution. On the contrary, the LMCKP solution contains at most one class with two-fractional values. The LMCKP solution was computed
using the algorithm presented in (Sinha & Zoltners, 1979) and rounded by setting the fractional item with larger weight to zero and the other variable to one. We also used the exact solution of the associated MCKP to test the relation between optimal solutions of these problems. A heuristic algorithm presented in (Parra-Hernandez & Dimopoulos, 2005) was applied for computing a heuristic solution of the MMKP. These 3 candidates were compared by $n_d$. Table 1 reports the results.

<table>
<thead>
<tr>
<th>Category</th>
<th>LMCKP</th>
<th>MCKP</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,10,5)</td>
<td>1.96 (5.7)</td>
<td>2.2 (6.1)</td>
<td>2.25 (4.5)</td>
</tr>
<tr>
<td>(10,10,10)</td>
<td>3.37 (6.3)</td>
<td>3.62 (6.8)</td>
<td>4.55 (6.5)</td>
</tr>
<tr>
<td>(30,10,5)</td>
<td>2.2 (5.7)</td>
<td>2.43 (5.3)</td>
<td>2.8 (4.8)</td>
</tr>
<tr>
<td>(30,10,10)</td>
<td>4.52 (5.6)</td>
<td>4.76 (6.2)</td>
<td>6.48 (5.9)</td>
</tr>
<tr>
<td>(10,30,5)</td>
<td>1.84 (5.5)</td>
<td>1.91 (5.3)</td>
<td>1.59 (4.5)</td>
</tr>
<tr>
<td>(10,30,10)</td>
<td>3.52 (6.4)</td>
<td>3.8 (6.9)</td>
<td>4.25 (6.07)</td>
</tr>
<tr>
<td>AVG</td>
<td>2.90</td>
<td>3.12</td>
<td>3.65</td>
</tr>
</tbody>
</table>

Table 1: Average (standard deviation) of $n_d$ for different base solutions (100 instances per category)

In column 1, categories of generated instances are given. A category is a triple (g,n,m). For each category, 100 instances were generated and the average and standard deviation of $n_d$ over these instances for each candidate was computed. In the last row, the average of $n_d$ over different categories is reported. As the results of Table 1 shows, the optimal solution of the LMCKP gives the smallest number of different classes and, compared to the solution of the MCKP, it can be computed efficiently. Thus, it was considered as the base of the core. According to the results obtained for these random instances, usually at most three classes of the LMCKP solution have values different from the corresponding classes in the optimal solution of the MMKP. For an intuitive example, suppose that the optimal solution of an MMKP instance with 10 classes and 10 items in each class is (2,6,3,3,7,5,0,1,9,3), represented in the integer form. We expect the rounded solution of the LMMKP to have 2 or 3 different classes, for example it might be (2,1,3,5,7,5,0,8,9,3), where different classes are bolded. The different classes may be distributed in the solution vector, hence, in the ordering of classes we try to order the classes in such a way that different classes are placed in the beginning of the solution vector. In addition, the value of a class in the LMMKP may be far from the MMKP optimal solution. For example, this difference in the 8th class is 7. We also would like to order items in each class so that this difference becomes as small as possible which gives us a smaller core. These two orderings are described in the next subsection.

3-2-3 Ordering Relations

We should find two ordering relations: one for ordering of items in each class and one for ordering of classes. For ordering of items, we consider two candidates. One option is to use the reduced costs of items. Items with small absolute reduced costs have more potential to be changed without affecting the objective value too much. Thus, in this option, we order the items in each class according to the non-decreasing order of their absolute reduced costs. The other option is to generalize the KP approach for ordering of items. The KP approach defines the profit to weight ratio of each item as its efficiency, and sorts the items in decreasing order of their efficiencies. It then takes items around the break item as the candidates for the core. Intuitively, an item with efficiency near the efficiency of the break item, gives nearly the same profit while uses nearly the same resource, and thus is usually competing to be the choice of an optimal solution. Thus, we can view this approach as sorting the items according to the difference of their efficiency from the efficiency of the break item. In multidimensional case, we have several resources and thus we should somehow aggregate the resource usage of each item to be able to define an efficiency measure for each item. A natural choice is to use a weighted average of usages, and hence we use the following formula to compute the efficiency of each item:

$$e_i = \frac{p_i}{\sum_{j=1}^{m} n_j \cdot w_j} \quad i \in C, j \in N_i$$

(11)

where $r_k$ is the weight of constraint $k$ and indicates its relative importance. Motivated from computing the surrogate constraint, which gives us one relatively tight constraint out of several constraints, we used the dual values of the LMMKP solution as the constraints weights. In other words, we define the efficiency of each item as the ratio of its profit to its weight in the associated MCKP. The ordering relation is then defined by sorting the items in class $i$ according to increasing order of $|e_i - e_j|$. This ordering relation states that we expect items with efficiency near the efficiency of the base solution item, have more potential to be in the optimal solution.
To compare these options, we ordered the items in each class using these orderings. These options were then compared by computing $d_u$ and $d_v$ metrics. Again, for each category, which is a triple $(g,n,m)$, we randomly generated 100 instances and computed the average and standard deviation of these metrics over these instances. For each instance, we computed these metrics in the following way. We first ordered its items in each class using the target ordering relation. Then, we computed the base solution using the method of last subsection and for each class, we manually put the selected item of that class in the base solution, in the first position of the list of items of that class. Note that after this reordering, the integer representation of the base solution simply become a $g$-dimensional vector with all entries are one. Next, we computed the optimal solution of the instance. Since the items are sorted and then the optimal solution is computed, the integer representation of the optimal solution indicates the ranks (or positions) of the optimal items in the sorted lists. Closeness of these ranks to one indicates the ordering relation quality. Finally, we used the optimal solution to calculate the metrics. The results are shown in Table 2.

<table>
<thead>
<tr>
<th>Category</th>
<th>$d_u$</th>
<th>$d_v$</th>
<th>$d_u$</th>
<th>$d_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,10,5)</td>
<td>1.16  (5.9)</td>
<td>2.19 (5.1)</td>
<td>2.31 (5.3)</td>
<td>3.95 (3.9)</td>
</tr>
<tr>
<td>(10,10,10)</td>
<td>2.11  (6.1)</td>
<td>2.71 (6.9)</td>
<td>5.03 (4.1)</td>
<td>6.05 (4.6)</td>
</tr>
<tr>
<td>(30,10,5)</td>
<td>1.07  (7.0)</td>
<td>2.81 (5.6)</td>
<td>2.43 (5.3)</td>
<td>5.26 (5.2)</td>
</tr>
<tr>
<td>(30,10,10)</td>
<td>1.39  (7.4)</td>
<td>3.02 (7.1)</td>
<td>4.95 (5.4)</td>
<td>8.10 (7.0)</td>
</tr>
<tr>
<td>(10,30,5)</td>
<td>1.13  (6.0)</td>
<td>6.42 (5.1)</td>
<td>2.24 (4.8)</td>
<td>10.54 (20.2)</td>
</tr>
<tr>
<td>(10,30,10)</td>
<td>2.42  (6.7)</td>
<td>6.85 (5.5)</td>
<td>5.26 (5.0)</td>
<td>14.67 (22.7)</td>
</tr>
<tr>
<td>AVG</td>
<td>1.54</td>
<td>4.00</td>
<td>3.70</td>
<td>8.09</td>
</tr>
</tbody>
</table>

The results clearly show that ordering of items based on their reduced costs, is a better candidate. In the reduced cost ordering, $d_u$ and $d_v$ are less sensitive to the increase of number of items. In both orderings, the computed metrics are increased considerably when the number of constrains are increased. Combining $d_u$ and $d_v$ we can conclude that using the reduced cost ordering, the values of the variables in the optimal solution are not too different from their values in the base solution. Figure 3 displays the empirical probability distribution of $d_u$, if we sort the items according to the reduced cost ordering. This figure has been obtained by computing $d_u$ over 1000 randomly generated instances from (10,30,5) category. These instances were generated according to the method described in section 5 to generate uncorrelated instances with $r=1000$.

For ordering of classes, we used the upper convex hull lines in the associated LMCKP (Sinha & Zoltners, 1979). Intuitively, in the MCKP, the slopes of upper convex lines play the same role as the efficiencies in the KP. Hence, like the core in the KP, assuming the lines are sorted according to non-increasing slopes of lines, classes with lines near the break line must have more potential for change. We define the break difference for each class as the minimum difference between the slope of an upper convex line of that class and the slope of the break line in the associated LMCKP:

\[
\Delta_i = \min_{j \in L_i} \| \lambda^* - \lambda_j \|_1, \quad i \in C
\]  

Where $\lambda^*$ is the slope of the break line, $L_i$ is the set of non-lp-dominated items in class $i$, and $\lambda_j$ is the slope of the $j^{th}$ line in class $i$. The classes were ordered according to increasing $\Delta_i$. The MCKP core used by Pisinger motivated this ordering (D. Pisinger, 1995). Figure 4 displays the empirical probability distribution of $l_c$, if we sort classes according to the above ordering. Computations are over 1000 randomly generated instances from (30,10,5) category. Again, these instances were generated according to the method described in section 5 to generate uncorrelated instances with $r=1000$. As this figure demonstrates, after ordering, it is highly expected that only classes at the beginning take on values different from their values in the base solution.
3-2-4 Subspace Definition

We examined three techniques to define the core subspace. The first technique is based on the observation that $E[l_c]$ and $E[d_{\infty}]$ are small. Thus after ordering, we can take first $E[l_c]$ classes and in those classes just take first $E[d_{\infty}]$ items for the core. One should estimate these two parameters such that the resulting core has an acceptable size and accuracy. Based on our computational results, we observed that $l_c$ is almost independent of $n$ and $d_{\infty}$ is almost independent of $g$ and they are bounded from above by $g$ and $n$ respectively. Thus, we used the following empirical formulas to estimate their values:

$$
\tilde{l}_c = \min\{g, \alpha g + \beta m\} \\
\tilde{d}_{\infty} = \min\{n, \alpha' n + \beta' m\}
$$

(13)

The second technique is based on the observation that the Manhattan distance of the base solution is not too far from the optimal solution. Therefore, we can define the core as a subspace having the Manhattan distance of at most $E[d_{\infty}]$. This restriction can be enforced by adding the following soft constraint to the problem:

$$
\sum_{i \in C, j \in N_i} (j - 1)x_{ij} \leq E[d_{\infty}] \\
$$

(14)

This constraint is valid, because if $j$th item in class $i$ is chosen, it contributes to the Manhattan distance from the base solution by $j-1$. Also, to have a control over the dimension of classes, observe that we expectedly have $E[n_j]$ classes taking different values than the base solution. Hence, by adding the following soft constraint to the problem we can enforce this in the subspace:

$$
\sum_{i \in C, j \in N_i, j \neq i} x_{ij} \leq E[n_j] \\
$$

(15)

$E[d_{\infty}]$ and $E[n_j]$ were predicted using the following empirical formula:

$$
\tilde{d}_{\infty} = \tilde{n}_j = m
$$

(16)

In the third technique, we took a completely different approach by defining a total ordering among all items and taking the first $k$ items as the core. We used the reduced cost ordering for ordering of all items. In this way, different classes may have different number of items in the core. Since the reduced cost of at least one item in each class is zero, each class has at least one item in the core. We used the following empirical relation to compute $k$:

$$
k = \alpha^* g + \beta^* m
$$

(17)

Table 3 displays the computed results for the overall core obtained using each of the three mentioned techniques.
Table 3: Average (standard deviation) of the accuracy (in percent) and enumeration time (in milliseconds) for different subspace definition approaches numbered from 1 to 3 (100 instances per category)

<table>
<thead>
<tr>
<th>Category</th>
<th>Accuracy</th>
<th>Enumeration time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(10,10,5)</td>
<td>99.75 (0.00)</td>
<td>99.84 (0.05)</td>
</tr>
<tr>
<td>(10,10,10)</td>
<td>99.77 (0.00)</td>
<td>99.96 (0.00)</td>
</tr>
<tr>
<td>(30,10,5)</td>
<td>99.92 (0.00)</td>
<td>99.75 (0.05)</td>
</tr>
<tr>
<td>(30,10,10)</td>
<td>99.91 (0.00)</td>
<td>99.99 (0.00)</td>
</tr>
<tr>
<td>(10,30,5)</td>
<td>99.75 (0.01)</td>
<td>99.99 (0.06)</td>
</tr>
<tr>
<td>(10,30,10)</td>
<td>99.94 (0.00)</td>
<td>99.99 (0.00)</td>
</tr>
<tr>
<td>AVG</td>
<td>99.84</td>
<td>99.90</td>
</tr>
</tbody>
</table>

The accuracy is reported in percent and the time of complete enumeration in milliseconds. We experimentally set $\alpha = \alpha' = 0.15$, $\beta = \beta' = 0.7$, $\alpha'' = 1.5$, and $\beta'' = 2$ which showed good tradeoff between size and accuracy of the resulting core. As the table shows, technique 1 enables us to define a core which is relatively both accurate and small. It is also easy to be incorporated in a dynamic expansion approach. Hence, it is chosen as the technique of deriving the core in this paper. In summary, to obtain a core for the MMKP, we first surrogate relax the problem to the MCKP using the dual values of the LMMKP solution as the surrogate multipliers and use the LMCKP solution as the base solution. We then order the classes and items according to increasing break differences and absolute reduced costs, respectively. Then define the subspace or the approximate core by taking the first $\tilde{l}_c$ classes and the first $\tilde{a}_d$ items in those classes.

### 4 The Exact Algorithm

In this section, the main parts of a B&B algorithm that solves the MMKP exactly are presented. The outline of this algorithm is presented below. The details of the steps are described in the following subsections. At the end, the memory complexity of the algorithm is analyzed.

#### 4-1 The Tree Structure and Navigation Method

Instead of taking a fixed sized core, we arrange the B&B tree in a way that the core will be expanded during its navigation. Suppose we have sorted the classes and items according to the MMKP core defined in the previous section. The tree is naturally arranged based on the orderings defined in the core as illustrated in Figure 5. The tree has $g$ levels. Level $i$ of the tree corresponds to $(g-i+1)$th class. The branches in level $i$ correspond to the items of class $i$. An expanded node at the $i$th level and $j$th branch is represented by $n_{ij}$.

![Figure 5: The B&B tree structure (the actual tree is the product of all levels). The classes and items are sorted based on the orderings defined in the core. The first items constitute the base solution.](image-url)
**Algorithm** Core-Based B&B  
**Input.** An instance of MMKP problem.  
**Output.** An exact feasible solution to the problem or declaring infeasibility.  

//Step 1: Preprocessing  
1. Solve the corresponding LMMKP problem. Let lmmkp be the computed solution.  
2. If lmmkp is integral or infeasible, terminate the algorithm.  
3. Use dual values of lmmkp to compute the associated MCKP problem.  
4. Solve the associated LMCKP problem. Let lmckp be the computed solution. Also, store the upper convex lines in a doubly linked list.

//Step 2: Constraint Processing  
1. Order the constraints based on their tightness.  
2. Remove negative coefficients.  
3. Use dual values of lmmkp to compute the mixed surrogate constraint and add it to the constraints.

//Step 3: Core Computation (according to the approach defined in section 3)  
1. Round lmckp to obtain the base solution.  
2. Use the break differences in lmckp to order the classes.  
3. Use the reduced costs in lmmkp to order the items in each class.

//Step 4: Navigation  
1. Structure the tree using the core.  
2. Set the incumbent to null and set $X$ to null.  
3. Navigate the tree in a depth-first order:  
   a. For each node $n_{i,j}$:  
      i. Set $X_{g^{-1}+1} = j$.  
      ii. Compute the upper bound by using lmckp and navigation of upper convex lines.  
      iii. If $X$ violates one of the constraints or cannot lead to a feasible solution or the computed upper bound is no better than the incumbent, cut the node.  
   b. Whenever reach the bottom of the tree and $X$ is a better solution than the incumbent:  
      i. Update the incumbent with $X$.  
      ii. Update the mixed surrogate constraint.  
      iii. Fix the variables using lmckp.  
4. Report the incumbent or if it is null declare infeasibility.

The tree is navigated in a depth-first order. During the navigation when taking $n_{i,j}$ we set $X_{g^{-1}+1} = j$, where $X$ is the integer representation of the current partial solution and $X_k$ is its $k^{\text{th}}$ component. Because the classes in the core are sorted according to decreasing probability of change, those at the top of the tree are less likely to take on different values than their base solution values. Therefore, by this type of navigation, we fix classes with more confidence about their values (making good decisions at first) and try to find the values of other classes. This type of navigating also consumes memory proportional to the depth of the tree, which is negligible. When the algorithm reaches the bottom of the tree, it updates the best solution (incumbent) if a better feasible solution is found. Note that since our base solution is feasible and we manually put the items of the base solution in the first positions, the first incumbent, which the algorithm reaches it, is the base solution.

After adding the fathoming tests of the next subsection, when the algorithm navigates this tree it simulates the following core-expansion behavior. The algorithm begins by the base solution (when reaches the bottom of the tree). If the algorithm cannot prove the optimality of this solution, it defines a core with one class (class 1) in it and fixes the variables of other classes (this actually happens when the algorithm backtracks one level higher). The algorithm then completely enumerates the subspace induced by this core or in other words, it exactly solves the core problem defined by this core. If the optimality cannot be proved using the fathoming tests, the algorithm then expands the core with the next most probable class (again when backtracks one level higher) and completely enumerates the subspace of this core while the variables of out-of-core classes are fixed, and so on. It is easy to see that the algorithm maintains the minimal expansion property along the dimension of classes that is it expands the core after it completely searches the core for an optimal solution. Since the items in the core are also sorted, the algorithm first examines the most probable candidates in each class and thus we can expect to reach an optimal solution quickly. As mentioned before, this usage of core resembles variable and node selection heuristics in the B&B approach.
4-2 Fathoming Tests

In order to prune the sub-trees effectively, we test a node for fathoming at three steps. If a node fails in one of these tests, it will be fathomed. For the first test, we use the propagation of the mixed surrogate constraint. Let the general 0-1 program be:

\[
\max \{ c^T x \mid Ax \leq b, x \in \{0,1\}^n \}
\]  

where \( c \) is an \( n \)-dimensional (column) vector of objective coefficients, \( A \) is an \( m \times n \) matrix of non-negative constraints coefficients, \( b \) is an \( m \)-dimensional vector of right hand sides of constraints. In (Hanafi & Glover, 2006; Osorio & Hernandez, 2004), the surrogate constraint with the objective function constraint is paired resulting in the following constraint, namely the mixed surrogate constraint, for (18):

\[
(u^T A - c^T)x \leq u^T b - LB
\]  

where \( u \) is an \( m \)-dimensional vector of dual multipliers and \( LB \) is a lower bound for the problem instance. This constraint can be used for variable fixation and generation of nested logic cuts for multidimensional knapsack problems.

We observed that because this constraint is tight, in addition of its usage for fixation, it can be propagated like other constraints and effectively be used for fathoming test. Note that in the mixed surrogate constraint, simple upper bounding constraints are excluded to get a stronger constraint. For the MMKP the GUB constraints are also excluded to get the same effect. Because the result is a valid constraint if it is violated at any node, that node can be fathomed. We used the value of the incumbent as the \( LB \), hence the right hand side of this constraint is changing dynamically during the navigation of the tree.

In the second test, we simply check whether a node leads to a feasible solution or not. This is done by (20). \( n_{i,j} \) is fathomed if

\[
\exists k \in \{1,\ldots,m\}: \sum_{i-g}^j w_{i,j}^k + \sum_{i-g}^j \min_{l \in N_i}^j v_{i,j}^k > b^k
\]  

Equation (20) states that even if one of the constraints is violated by adding the consumed amount of that constraint so far and the minimum consumption of the remaining classes, then there is no feasible solution from this node. In each node, we keep the sum of the consumed amount of each constraint (including the mixed surrogate constraint), so by using information of the parent node we can compute the above checks incrementally.

Finally, we compute an upper bound for a node using the associated LMCKP and compare it with the incumbent value. If it is less than or equal to the incumbent, the node will be fathomed. This step is put last because it is the most time-consuming step. We used the algorithm presented in (Sinha & Zoltners, 1979) to solve the LMCKP. To do it faster, however, at the root node, the algorithm solves the associated LMCKP once, obtains the break line, and stores the lines of the upper convex hull in a sorted doubly linked list. The lines will not be computed in other nodes. For other nodes, instead of starting from the first line, we keep track of changes in the capacity consumption with respect to the root node, and start from the break line. If we have extra capacity, we move forward in the lines (from the break line) until the knapsack is filled. If the capacity is exceeded then we move backward in the lines (from the break line) until the capacity is no longer violated.

4-3 Variable Fixation and Preprocessing

In order to reduce the search space even further, we fix the variables whenever the algorithm reaches a new incumbent using (21).

\[
UB(\text{MMKP} \mid x_g = 1) < LB \Rightarrow x_g = 0
\]  

where \( UB(\text{MMKP} \mid x_g = 1) \) is an upper bound function for the MMKP with the added constraint \( x_g = 1 \), and \( LB \) is the value of the current incumbent. UB can be computed as in the third fathoming test, but for faster calculation we used a weaker upper bound for the associated LMCKP proposed in (D. Pisinger, 95).

The algorithm preprocesses the problem at two steps. First, it rearranges the constraints. During the second fathoming test, we check the constraints in order until finding a possibly violated one. If we consider tighter constraints at first, the expected number of constraints checking is reduced. We compute the following measure for the tightness of constraints:
\[ t^k = \left( \sum_{i \in C} \sum_{j \in \mathcal{N}_i} w_{ij}^k \right) / b^k \]  

(22)

and put constraints with more \( t^k \) at first. At the second preprocessing step, each constraint is replaced by constraint (23), which defines the same solution space.

\[ \sum_{i \in C} \sum_{j \in \mathcal{N}_i} (w_{ij}^k - \min_{l} \{w_{ij}^l\}) \leq b^k - \sum_{j \in \mathcal{N}_i} \min_{l} \{w_{ij}^l\} \]  

(23)

This will cause negative coefficients to be removed and calculation of (20) become easier because we are sure now that those minimums are equal to zero. It is common in B&B algorithms to compute a lower bound (by means of some heuristic algorithms) in the preprocessing phase. We did not use a heuristic here because as mentioned in section 4.1 we expect the algorithm to reach a near optimal solution early during its navigation.

### 4-4 Memory Complexity Analysis

Due to the depth first navigation, we have at most \( g \) nodes alive, where \( g \) is the number of classes and thus is the depth of the tree. In each node, to be able to calculate step 4.3.a.iii incrementally, for each constraint we maintained its amount of consumption. Hence, the navigation, overall, consumes \( O(mg) \) memory. In addition, for the third fathoming test we kept the convex lines computed at the root of the LMCKP for further uses. Having \( O(ng) \) lines, in total the space complexity of the algorithm is \( O((m+n)g) \).

### 5 Computational Results

To examine the algorithms in practice, they were coded in C++\(^1\) and tested on a P4 3.4 GHz computer with 1GB of memory. We used Coin CLP library (Coin-OR project, 2008) for solving the LMMKP and computing the dual values. We compared our algorithm with our implementation of EMKP algorithm\(^2\) (Sbihi, 2007). According to (Sbihi, 2007), EMKP completely outperforms the previous algorithm developed in (Shahadatullah-Khan, 1998) thus it can be considered as the best specific algorithm for the MMKP so far. We also compared the algorithms against CBC (Coin-OR project, 2008), a general branch-and-cut framework, with its default settings. The maximum available memory for each algorithm was 512MB and the maximum time was set to one hour.

To test our algorithm, we used both of available instances for MMKP and our own generated instances. Han et al. (Han et al., 2010) investigated how to generate hard instances for MMKP. As a result, they generated a large number of instances with different types of correlation between profits and weights and different degrees of tightness of constraints. These instances are available at http://enstb.org/~gsimon/Resources/MMKP/. In all instances, \( g = 10 \), \( n = 5 \), and \( m = 5 \). Table 4 compares the running time of the algorithms against these instances. The instances are categorized based on their generation method (refer to (Han et al., 2010) for the details about these methods and their names). Each category contains 100 to 2000 instances. In each category, we reported the average and standard deviation of running times of the algorithms against the instances of that category. As the results show, the core-based algorithm significantly performs faster than EMKP and CBC. The core-based algorithm has a small standard deviation, which makes it a stable algorithm for different instances. In fact, the maximum time taken by the core-based algorithm to solve an instance was 0.24 seconds while this was 26.22 and 1889.41 seconds in EMKP and CBC, respectively. EMKP managed to solve harder instances more quickly than CBC, which was mainly due to its best-first nature, but in simpler instances, CBC outperformed EMKP. As there are more than 46000 instances with different levels of difficulties in this test case, we can conclude that the core-base algorithm can solve small instances of MMKP very efficiently.

We also examined the algorithms against the large MMKP instances of Khan (Khan, 2002) and Hifi (Hifi et al., 2006), but except for the first six instances of Khan, none of the algorithms could solve them in the given time and memory. Optimum values of these instances are still unknown. Although again the core-based algorithm solved those six instances faster than the other methods. This shows that we still need to develop more powerful exact algorithms to solve large instances of the MMKP.

---

\(^1\) The source code is obtainable by request from the first author

\(^2\) As pointed out in (Han et al., 2010), there are some mistakes or ambiguities in the presented outline of EMKP but these errors do not exist in the body of the paper. In particular, the paper precisely specifies both of how to choose the best node and how to correctly prune infeasible nodes.
We also generated our own instances. Three classes of test problems were generated. In the random or uncorrelated class (UC), the profits and weights were drawn from a uniform random generator in the range \([1,r]\), where \(r\) is a parameter. The right hand sides were computed according to (24).

\[
b^k = \alpha \sum_{i \in C} \max_{j \in N_i} \{w^k_{ij}\} + (1-\alpha) \sum_{i \in C} \min_{j \in N_i} \{w^k_{ij}\}
\]

where \(\alpha \ (0 \leq \alpha \leq 1)\) determines the tightness of the constraints and was set to 0.5 here. In the second class, we generated correlated instances. Experiences in the knapsack problems have shown that instances having correlation between their profits and weights are more difficult to solve. In (Freville & Plateau, 1994), the following equation for the multidimensional knapsacks is proposed to generate correlated instances by computing profits from weights.

\[
p_i = \frac{\sum_{k=1}^{n} w^k_{ij}}{m+r/2} \quad q_i = \alpha \quad i \in C, j \in N_i
\]

where \(U\) is a uniform random generator. In the weakly correlated class (WC), the weights and right hand sides were computed like class UC and the profits were computed using (25). Strongly correlated (SC) instances were computed similarly but with the value of \(q_i\) fixed to one. Instances were generated in different configurations. A configuration is a triple \((g,n,m)\). It is assumed that all classes have the same number of items, that is \(n = n_i \quad i \in C\). In each configuration, 100 instances were generated and solved.

<table>
<thead>
<tr>
<th>Category</th>
<th>Core-based</th>
<th>EMKP</th>
<th>CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-U-U</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.05)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-U-D(U)</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-C(U)-U</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-C(L)-U</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-C(U)-D(U)</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-C(L)-D(U)</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-R-U</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-L-U</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-L-D(U)</td>
<td>0.00 (0.00)</td>
<td>0.02 (0.04)</td>
<td>0.02 (0.04)</td>
</tr>
<tr>
<td>G-U-W</td>
<td>0.00 (0.00)</td>
<td>0.78 (0.98)</td>
<td>0.13 (0.09)</td>
</tr>
<tr>
<td>G-U-D(W)</td>
<td>0.00 (0.00)</td>
<td>0.69 (0.87)</td>
<td>0.15 (0.09)</td>
</tr>
<tr>
<td>G-C(U)-W</td>
<td>0.00 (0.00)</td>
<td>5.43 (5.76)</td>
<td>0.05 (0.06)</td>
</tr>
<tr>
<td>G-C(U)-D(W)</td>
<td>0.00 (0.00)</td>
<td>5.22 (5.90)</td>
<td>0.05 (0.06)</td>
</tr>
<tr>
<td>G-C(L)-W</td>
<td>0.00 (0.00)</td>
<td>4.60 (4.91)</td>
<td>0.06 (0.06)</td>
</tr>
<tr>
<td>G-C(L)-D(W)</td>
<td>0.00 (0.00)</td>
<td>3.76 (3.94)</td>
<td>0.06 (0.07)</td>
</tr>
<tr>
<td>G-R-W</td>
<td>0.00 (0.00)</td>
<td>0.72 (0.97)</td>
<td>0.19 (0.12)</td>
</tr>
<tr>
<td>G-L-W</td>
<td>0.01 (0.01)</td>
<td>0.41 (0.53)</td>
<td>3.93 (9.08)</td>
</tr>
<tr>
<td>G-L-D(W)</td>
<td>0.00 (0.01)</td>
<td>0.35 (0.45)</td>
<td>1.52 (3.31)</td>
</tr>
<tr>
<td>G-U-S</td>
<td>0.00 (0.02)</td>
<td>1.35 (1.66)</td>
<td>0.02 (0.02)</td>
</tr>
<tr>
<td>G-U-D(S)</td>
<td>0.00 (0.01)</td>
<td>1.23 (1.44)</td>
<td>0.02 (0.02)</td>
</tr>
<tr>
<td>G-C(U)-S</td>
<td>0.00 (0.01)</td>
<td>6.72 (5.86)</td>
<td>0.01 (0.01)</td>
</tr>
<tr>
<td>G-C(U)-D(S)</td>
<td>0.00 (0.01)</td>
<td>6.93 (5.94)</td>
<td>0.00 (0.01)</td>
</tr>
<tr>
<td>G-R-S</td>
<td>0.00 (0.00)</td>
<td>1.25 (1.63)</td>
<td>4.13 (16.83)</td>
</tr>
<tr>
<td>G-R-D(SU)</td>
<td>0.00 (0.00)</td>
<td>1.20 (1.68)</td>
<td>0.10 (0.43)</td>
</tr>
<tr>
<td>G-R-D(SI)</td>
<td>0.00 (0.00)</td>
<td>1.51 (1.74)</td>
<td>0.34 (2.67)</td>
</tr>
<tr>
<td>G-L-S</td>
<td>0.04 (0.05)</td>
<td>0.72 (1.01)</td>
<td>332.19 (507.77)</td>
</tr>
<tr>
<td>G-L-D(S)</td>
<td>0.04 (0.05)</td>
<td>0.68 (0.88)</td>
<td>406.60 (558.58)</td>
</tr>
<tr>
<td>G-L-D(SU)</td>
<td>0.01 (0.02)</td>
<td>0.30 (0.53)</td>
<td>71.78 (146.39)</td>
</tr>
<tr>
<td>G-L-D(SI)</td>
<td>0.01 (0.02)</td>
<td>1.48 (2.39)</td>
<td>78.29 (210.04)</td>
</tr>
<tr>
<td>G-L-D(SUI)</td>
<td>0.02 (0.04)</td>
<td>4.56 (3.81)</td>
<td>264.93 (513.27)</td>
</tr>
<tr>
<td>G-C(L)-S</td>
<td>0.02 (0.04)</td>
<td>4.93 (3.88)</td>
<td>209.41 (427.02)</td>
</tr>
<tr>
<td>G-C(L)-D(S)</td>
<td>0.02 (0.04)</td>
<td>6.93 (5.94)</td>
<td>0.05 (0.06)</td>
</tr>
</tbody>
</table>

Table 4: Average running time (standard deviation) of the algorithms in seconds for different categories of instances reported in (Han et al., 2010). Each category contains at least 100 instances.
Table 5 shows the results of running the algorithms against instances of class UC with different configurations. For each configuration, 100 instances are generated and the average and standard deviation of the running time of the algorithms over these instances in seconds are reported. \(r\) was set to 1000.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Core-based</th>
<th>EMKP</th>
<th>CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,10,2)</td>
<td>0.00 (0.01)</td>
<td>0.00 (0.01)</td>
<td>0.03 (0.05)</td>
</tr>
<tr>
<td>(10,10,5)</td>
<td>0.02 (0.03)</td>
<td>0.09 (0.15)</td>
<td>0.96 (0.66)</td>
</tr>
<tr>
<td>(10,10,10)</td>
<td>0.02 (0.00)</td>
<td>3.93 (8.64)</td>
<td>3.46 (2.81)</td>
</tr>
<tr>
<td>(20,50,2)</td>
<td>0.01 (0.01)</td>
<td>0.05 (0.26)</td>
<td>0.12 (0.19)</td>
</tr>
<tr>
<td>(20,50,5)</td>
<td>0.01 (0.01)</td>
<td>0.90 (3.99)</td>
<td>0.84 (1.06)</td>
</tr>
<tr>
<td>(20,50,10)</td>
<td>0.02 (0.03)</td>
<td>(space)</td>
<td>4.15 (4.86)</td>
</tr>
<tr>
<td>(20,50,15)</td>
<td>0.30 (1.15)</td>
<td>(space)</td>
<td>20.35 (28.96)</td>
</tr>
<tr>
<td>(50,20,2)</td>
<td>0.01 (0.01)</td>
<td>(space)</td>
<td>0.26 (0.56)</td>
</tr>
<tr>
<td>(50,20,5)</td>
<td>0.02 (0.00)</td>
<td>(space)</td>
<td>1.59 (1.90)</td>
</tr>
<tr>
<td>(50,20,10)</td>
<td>0.09 (0.41)</td>
<td>(space)</td>
<td>8.00 (11.14)</td>
</tr>
<tr>
<td>(50,20,15)</td>
<td>2.96 (9.88)</td>
<td>(space)</td>
<td>43.21 (77.00)</td>
</tr>
<tr>
<td>(100,100,2)</td>
<td>0.09 (0.01)</td>
<td>(space)</td>
<td>5.91 (14.81)</td>
</tr>
<tr>
<td>(100,100,5)</td>
<td>0.14 (0.02)</td>
<td>(space)</td>
<td>22.81 (33.19)</td>
</tr>
<tr>
<td>(100,100,10)</td>
<td>0.48 (1.23)</td>
<td>(space)</td>
<td>96.29 (136.17)</td>
</tr>
<tr>
<td>(100,100,15)</td>
<td>116.42 (962.64)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
</tbody>
</table>

A ‘space’ (‘time’) entry indicates that the algorithm has ran out of its memory (time) for at least one instance in that configuration. In the instances of class UC, the core-based algorithm performs very fast. Due to the best first nature of EMKP, it cannot solve medium to large instances because of memory shortage. It is also clear that CBC managed to solve more instances than EMKP.

Table 6 displays the performance of the algorithms against instances of class WC.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Core-based</th>
<th>EMKP</th>
<th>CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,10,2)</td>
<td>0.00 (0.01)</td>
<td>21.77 (19.43)</td>
<td>0.73 (0.18)</td>
</tr>
<tr>
<td>(10,10,5)</td>
<td>0.01 (0.01)</td>
<td>20.86 (18.62)</td>
<td>2.18 (1.24)</td>
</tr>
<tr>
<td>(10,10,10)</td>
<td>0.05 (0.05)</td>
<td>37.74 (35.93)</td>
<td>6.63 (5.57)</td>
</tr>
<tr>
<td>(20,50,2)</td>
<td>0.01 (0.01)</td>
<td>(space)</td>
<td>45.75 (35.09)</td>
</tr>
<tr>
<td>(20,50,5)</td>
<td>26.89 (31.82)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(50,20,2)</td>
<td>0.02 (0.01)</td>
<td>(space)</td>
<td>59.51 (48.65)</td>
</tr>
<tr>
<td>(50,20,5)</td>
<td>94.80 (97.01)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(100,100,2)</td>
<td>2.90 (0.28)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(100,100,5)</td>
<td>(time)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
</tbody>
</table>

In WC instances, the duality gap is high. Therefore, EMKP considers too many infeasible states before reaching the optimal state. Hence, even for small instances, EMKP encounters the lack of memory. The highness of duality gap makes the fathoming tests and variable fixation less effective. Thus, the core-based algorithm also must spend more time exploring the tree to prove the optimality of the solution.

None of the algorithms could solve SC instances when the number of constraints was greater than 5. Table 7 summarizes the results of solving SC instances with 2 and 3 constraints.
Table 7: Average running time (standard deviation) of the algorithms in seconds for different configurations of class SC (100 instances per configuration), r=1000

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Core-based</th>
<th>EMKP</th>
<th>CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,10,2)</td>
<td>0.06 (0.06)</td>
<td>(space)</td>
<td>137.10 (98.04)</td>
</tr>
<tr>
<td>(20,10,2)</td>
<td>0.35 (2.74)</td>
<td>(space)</td>
<td>314.65 (260.86)</td>
</tr>
<tr>
<td>(10,20,2)</td>
<td>0.03 (0.04)</td>
<td>(space)</td>
<td>256.50 (192.07)</td>
</tr>
<tr>
<td>(20,20,2)</td>
<td>0.06 (0.05)</td>
<td>(space)</td>
<td>375.84 (202.04)</td>
</tr>
<tr>
<td>(50,20,2)</td>
<td>0.43 (1.70)</td>
<td>(space)</td>
<td>444.87 (249.13)</td>
</tr>
<tr>
<td>(20,50,2)</td>
<td>0.05 (0.04)</td>
<td>(space)</td>
<td>269.27 (156.83)</td>
</tr>
<tr>
<td>(50,50,2)</td>
<td>0.30 (0.83)</td>
<td>(space)</td>
<td>436.11 (295.18)</td>
</tr>
<tr>
<td>(10,10,3)</td>
<td>9.29 (6.63)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(20,10,3)</td>
<td>157.07 (237.43)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(10,20,3)</td>
<td>33.67 (35.28)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(20,20,3)</td>
<td>(time)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
<tr>
<td>(50,20,3)</td>
<td>(time)</td>
<td>(space)</td>
<td>(time)</td>
</tr>
</tbody>
</table>

It is clear that the core-based algorithm significantly performs better than the other two algorithms when the number of constraints is small even in hard instances. To analyze why the algorithm performs fast on random instances, we measured some metrics about these instances and behavior of the algorithm with respect to them. The results for r=10000 are shown in Table 8.

Table 8: Average run time metrics of the core-based algorithm for different configurations in class UC (100 instances per configuration), r=10000

<table>
<thead>
<tr>
<th>Configuration</th>
<th>%T\text{init}</th>
<th>%Fixed</th>
<th>%Dgap</th>
<th>#Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(20,20,2)</td>
<td>31.00</td>
<td>95.05</td>
<td>0.03</td>
<td>0.00</td>
</tr>
<tr>
<td>(20,20,5)</td>
<td>43.00</td>
<td>92.37</td>
<td>0.13</td>
<td>0.92</td>
</tr>
<tr>
<td>(20,20,10)</td>
<td>65.40</td>
<td>87.43</td>
<td>0.37</td>
<td>94.86</td>
</tr>
<tr>
<td>(20,50,2)</td>
<td>56.00</td>
<td>98.26</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>(20,50,5)</td>
<td>81.00</td>
<td>96.94</td>
<td>0.05</td>
<td>1.58</td>
</tr>
<tr>
<td>(20,50,10)</td>
<td>89.79</td>
<td>94.93</td>
<td>0.15</td>
<td>80.42</td>
</tr>
<tr>
<td>(50,20,2)</td>
<td>74.00</td>
<td>95.57</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>(50,20,5)</td>
<td>98.34</td>
<td>93.88</td>
<td>0.03</td>
<td>11.18</td>
</tr>
<tr>
<td>(50,50,2)</td>
<td>85.15</td>
<td>91.37</td>
<td>0.07</td>
<td>942.32</td>
</tr>
<tr>
<td>(50,50,5)</td>
<td>99.50</td>
<td>98.47</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>(50,50,10)</td>
<td>99.32</td>
<td>97.60</td>
<td>0.01</td>
<td>3.69</td>
</tr>
<tr>
<td>(50,50,10)</td>
<td>87.08</td>
<td>96.70</td>
<td>0.02</td>
<td>765.76</td>
</tr>
<tr>
<td>(100,100,2)</td>
<td>100.00</td>
<td>99.21</td>
<td>0.00</td>
<td>0.10</td>
</tr>
<tr>
<td>(100,100,5)</td>
<td>99.71</td>
<td>98.86</td>
<td>0.00</td>
<td>10.34</td>
</tr>
<tr>
<td>(100,100,10)</td>
<td>84.58</td>
<td>98.56</td>
<td>0.00</td>
<td>7764.91</td>
</tr>
<tr>
<td>AVG</td>
<td>79.59</td>
<td>95.68</td>
<td>0.05</td>
<td>645.07</td>
</tr>
</tbody>
</table>

The descriptions of the columns in Table 8 are:

- %T\text{init} is the time required to solve the LMMKP and the associated LMCKP, reported in percent with respect to the total running time.
- %Fixed is the percent of variables fixed at the end of the algorithm.
- %Dgap is the duality gap with respect to the LMMKP solution, reported in percent.
- #Nodes is the total number of tree nodes in thousands examined during the navigation of the tree.

The descriptions of the columns in Table 8 are:

- %T\text{init} is the time required to solve the LMMKP and the associated LMCKP, reported in percent with respect to the total running time.
- %Fixed is the percent of variables fixed at the end of the algorithm.
- %Dgap is the duality gap with respect to the LMMKP solution, reported in percent.
- #Nodes is the total number of tree nodes in thousands examined during the navigation of the tree.

%Fixed shows that a large portion of variables in random instances are fixed. T\text{init} column shows that more than half of the time is spent in the preprocessing step and in contrast, a low number of nodes reveals that after preprocessing, the algorithm prunes the remaining subspace very quickly. Hence, using the core, the algorithm can find the (near) optimal solution very quickly, and using the variable fixation it can fix a significant portion of the variables, and since Dgap is low it can prune the remaining part of the tree very quickly.

We also analyzed the behavior of our algorithm with respect to the changes in values of g, n, and m. Figure 6 displays the change of expected running time of the algorithm with respect to the changes in g and n with r=10000 and m=5, for randomly generated instances from class UC. In Figure 7, the change of expected running time by varying number of constraints (i.e. m) is displayed. Figure 6 was obtained by fixing g=50 and n=50 and varying r for randomly generated instances. The expectations were computed over 100 instances.
As Figure 6 reveals, the algorithm has almost linear running time with respect to $g$ and $n$ in randomly generated instances of class UC. The steeper slope at the $g$ dimension indicates that the algorithm has a larger running time constant with respect to the number of classes. As Figure 7 shows, by increasing the number of constraints, the performance of the algorithm degrades exponentially.

6 Conclusion

In this paper a B&B algorithm to solve the multidimensional multiple-choice knapsack problem (MMKP) exactly was developed. At first, a core for the MMKP was identified. This core was defined with the solution of the associated LMCKP as the base point and by ordering the classes and items according to the difference with the break line and the reduced costs respectively. The B&B tree was built according to the defined orderings in the core and navigated in a depth first manner. The pruning of the sub-trees was done at three steps: checking the mixed surrogate constraint, checking the feasibility, and checking the upper bound computed with the aid of the associated LMCKP. We also used the variable fixation and constraint ordering for improving the performance of the algorithm. The computational results showed that the algorithm works significantly better than the previous best algorithm especially for uncorrelated instances and instances with a few number of constraints.

The computational analysis showed some drawbacks of the core-based approach. When the number of constraints was increased, the algorithm became more instable and time-consuming. This fact may be explained by insufficient power of the core to define the promising subspace, especially because our base solution in these situations is usually outside of the feasible region. It also takes more time for the algorithm to solve correlated instances. In these cases, significant portion of the time is spent to prove the optimality. The core can only help the algorithm to find the optimal solution faster but it does not help to prove the optimality of the incumbent faster (at least according to the approaches have proposed so far). This problem is exactly what previously the KP community encountered for solving the strongly correlated instances of the KP so it seems, one must look for other techniques for solving the problem. To solve hard instances of the MMKP, our preliminary attempts in using cutting planes (F. Glover & al., 1997) or combination with an enumeration method (James & Nakagawa, 2005) was unsuccessful. It is also possible to generate broader classes of test problems with different characteristics. In addition, more analytical approaches for analyzing the size of the core or the average case behavior of the algorithm like (Beier & Vöcking, 2004) are desirable.

Acknowledgment

We would like to thank referees for their helpful comments. The authors also would like to thank ‘Institute for Studies in Theoretical Physics and Mathematics’ (IPM) for their support from this research.

References


