A new exact algorithm to solve the Multiple-Choice Multidimensional Knapsack Problem:

The Utility Sorted Branch and Bound Algorithm

Antonio Kreß[a,*], Florian Kiel[a], Joachim Metternich[a]

[a]*Technical University Darmstadt, Otto-Berndt-Str. 2, 64287 Darmstadt, Germany*

Abstract

This paper offers an exact algorithm for one of the most complex members of the Knapsack Problem family: The Multiple-choice Multidimensional Knapsack Problem (MMKP). Based on a literature survey, only a few exact algorithms exist for the MMKP. The proposed algorithm excludes ranges of the possible solution space through systematic variations of ranked elements by utility sorting. Based on logical rules, successor nodes of a feasible node cannot include the optimal solution. Besides a literature survey as well as a detailed description of the algorithm and the test design, the results of a computational study on 11,160 instances are presented. The efficiency of the algorithm depends on the number of classes, elements per class, number of restrictions, and the degree of restrictiveness. Using the distribution of a search quotient shows the relative number of examined solutions being investigated. Testing against a brute-force algorithm reveals that all problem instances could be solved exactly while reducing the number of node enumerations. Furthermore, the structure of the algorithm allows its usage on other members of the knapsack problem family.

Keywords: Combinatorial optimization, MMKP, utility sorting, branch and bound

* Corresponding author. Tel.: +49 6151 16-26053; fax: +49 6151 16-20087.
E-mail address: a.kress@ptw.tu-darmstadt.de

1. Introduction

The knapsack problem is an optimization problem of combinatorics (Kellerer et al. 2011). It describes the selection of a subset of elements, each having a weight and a utility value. The total weight of all selected elements should not exceed a given limit. Under this condition, the utility value of the selected elements is to be maximized. Knapsack problems belong to the 21 classic NP-complete problems. Many real-world situations, in which a limited capacity cannot satisfy the entire demand, can be explained mathematically by this problem, e. g project budgeting.

The Multiple-choice Multidimensional Knapsack Problem (MMKP) is a variant of the knapsack problem (KP). As in the KP, the utility u_{ij} of the selected elements should be maximized; formalized by the objective function U (1). The MMKP generalizes the KP in two ways. On the one hand, the number of possible resource constraints is no longer limited to one, but to any natural number k, formalized by formula (2). Each element has k non-negative weights w_{ijk}. The term "multidimensional" originates from this consideration. On the other hand, it is no longer possible to decide only whether an element is to be selected or not, but various options are possible out of which one is selected, formalized by formula (3). In the KP, these options are simply "include" or "do not include". This is represented in the MMKP by a set J of classes that contain different objects as options. Each class j in J contains a set I of selectable elements. KPs that are extended in this way are called Multiple-Choice Knapsack Problems (MCKP). Binary variables x_{ij} are still used for modelling in formula (4). Unlike the KP, it is not trivial to find a feasible solution. In the KP, a feasible solution would be that no element was selected; this is not possible in the MMKP. Formula (5) implies that there is at least one feasible solution. Briefly: In the MMKP, exactly one element from each defined class must be selected subject to several resource constraints. So, the MMKP can be stated as:

Maximize	$U = \sum_{i=1}^{I}\sum_{j=1}^{J} u_{ij}x_{ij}$	$\forall\, i \in \{1,\dots,I\}, j \in \{1,\dots,J\}$	(1)
Subject to	$\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ijk}x_i \leq C_k$	$\forall\, k \in \{1,\dots,K\},$ $i \in \{1,\dots,I\},\ j \in \{1,\dots,J\}$	(2)
	$\sum_{j=1}^{J} x_{ij} = 1$	$\forall\, i \in \{1,\dots,I\},\ j \in \{1,\dots,J\}$	(3)
	$x_{ij} \in \{0,1\}$	$\forall\, i \in \{1,\dots,I\},\ j \in \{1,\dots,J\}$	(4)
	$\sum_{i=1}^{I} \min_{1\leq j\leq J}\{w_{ijk}\} \leq C_k \leq \sum_{i=1}^{I} \max_{1\leq j\leq J}\{w_{ijk}\}$	$\forall\, k \in \{1,\dots,K\},$ $i \in \{1,\dots,I\},\ j \in \{1,\dots,J\}$	(5)

Like the knapsack problem, the MMKP is known to be NP-hard. Several practical applications can be modelled with the MMKP: e.g., Resource allocation (Gavish & Pirkul, 1982), telecommunications (Watson, 2001), capital budgeting (Pisinger, 2001), and logistics (Basnet & Wilson, 2005).

The paper is organized as follows: The second chapter gives an overview of previously published heuristics and exact algorithms for the MMKP. The third chapter describes the developed algorithm based on utility sorting. The theoretical running time and the test design for the computational study are

described in the fourth chapter, the results of which are presented in the fifth chapter. The sixth chapter closes with a short conclusion and an outlook.

2. Literature survey

Algorithms for solving integer optimization problems can be divided into two areas: On the one hand, there are so-called heuristics that find the best possible solution with short computing time and on the other hand, there are exact methods that find the optimal solution. Heuristics are mainly used when the computing time is too high for exact methods or when no exact methods have been developed so far. Heuristics can be further subdivided into those, which have the goal of a shortest possible computing time and those, which have the goal to approximate the exact solution in the best possible way (Shojaei et al., 2013).

For the solution of MMKP, there are mainly heuristics that apply the special characteristics of Multiple-Choice Knapsack Problems to Multidimensional Knapsack Problems (Kellerer et al., 2011). At first, the basic principles of existing heuristics shall be mentioned following this scheme. The first heuristics is by Moser et al. (1997). It uses the concept of Lagrange multipliers, which reduces the number of resource constraints, thus decreasing complexity (Kellerer et al. 2004). For the objects with the highest utility - starting in the respective class - the algorithm exchanges the objects of a class with the smallest Lagrange multiplier until the first valid solution is found or no further changes are possible. In a further step, it is examined whether objects with a higher utility are exchangeable. Akbar et al. (2001) use a dynamic greedy approach, in which after the successful search for a valid start solution the objects with the highest relevance score within a class are exchanged, provided that the solution remains valid (Akbar et al. 2001). The relevance indices change after each exchange. The application of this algorithm shows a shorter running time with results closer to the optimal solution compared to previously developed algorithms. Parra-Hernandez and Dimopoulos (2005) use a Lagrange relaxation for the multiple-choice condition. This allows better solutions to be found, while the running time is higher than with the concepts mentioned above. The idea of using convex hulls to restrict the solution space is developed by Akbar et al. (2006). The convex hulls describe the convex multidimensional polygon, which can be formed in a utility-resource diagram and encloses all objects. In the same year, Ykman-Couvreur (2006) developed the fastest method so far - according to the author - based on a static greedy approach. Hifi et al. (2006) use reactive local search to develop two algorithms, both based on the exchange of two objects from two different classes. A solution is generated using a two-step procedure that allows leaving a local search area. In 2009, the "column generation approach" is developed by Cherfi (2009), which uses a branch-and-bound procedure with the help of fractional variables after a permissible starting solution by a greedy algorithm that is not described in detail. This approach is especially suitable for larger numbers of classes and objects. In the same year, Hanafi (2009) presents a heuristic based on iterative relaxation. Shojaei et al. (2009) develop a composite heuristic based on the Pareto algebra. In the following year, Cherfi (2010) develops a hybrid algorithm that

combines the above-mentioned approach with local search. The principle of ant colony optimization is transferred by Iqbal et al. (2010) to the MMKP. A heuristic by Mansi (2011) is based on LP relaxations and the fixation of some binary variables during runtime. Crévits (2012) publishes an approach based on an iterative semi-continuous relaxation. Shojaei et al. further develop their mentioned approach and integrate the possibility of using multi-core processors more efficiently for the MMKP by parallelization (Shojaei et al. 2013). Another hybrid algorithm using Pareto-algebra in a branch-and-bound approach is developed by Zennaki (2013). The taboo search is applied to the MMKP for the first time in the same year (Hiremath and Hill 2013). Htiouech (2013) uses a new approach that works with strategic oscillation and replacement restrictions. Chen (2014) implements the "reduce and solve" approach in 2014, in which individual elements that are likely to be part of the optimal solution are fixed and other elements that are unlikely to be part of the optimal solution are no longer considered. Hifi (2014) combines in the same year the local neighborhood search with a Lagrange relaxation. In the same year, Duenas (2014) applies the MMKP to factory planning, using an evolutionary algorithm with a three-dimensional chromosome. Xia et al. (2015) develop a fast stochastic heuristic that uses an iterative perturbative search paradigm with penalty weights for dimensions. An improved approximate binary search algorithm is presented by He et al. (2016). Vasko et al. (2016) compare five metaheuristics for the MMKP with the result, that no significant differences can be observed. Gao et al. (2017) derive pseudo-cuts from pseudo-gaps to outperform one state-of-the-art lower bound. Hwang et al. (2018) use an integer programming local search approach. Admi et al. (2020) determine the best strategy for a genetic-algorithm approach by evaluating the performance of several heuristics. As mentioned, there are many available heuristics for the MMKP.

The first exact procedure is developed and published by Khan (1998). For this purpose, a branch-and-bound method is used, which utilizes lower and upper bounds, calculated by the simplex algorithm. Each new level of nodes considers a different class. Sbihi and Hifi (2007) use a similar approach, integrating the "best first" strategy. The fastest exact algorithm so far is from Ghasemi (2011), which uses the "core" concept for a branch-and-bound procedure. Hifi (2012) publishes an exact procedure using the long-range multipliers applied to the dualized problem. Ghassemi-Tari et al. (2018) develop a discriminatory branch-and-bound method with three methods to select branches. The algorithm starts with an empty partial solution and selects alternatives iteratively. Mansini & Zanotti (2020) develop a method that exactly solves sub-problems of the MMKP by a recursive variable-fixing technique until an optimality criterion is satisfied. In conclusion, it can be stated that the number of exact algorithms is smaller than the number of heuristics. The main reason for this is that the MMKP is NP-complete; running times grow exponentially for larger problem instances. Nevertheless, for certain practical problems, it is important to find the optimal solution. A weakness of the presented exact methods is that upper and lower bounds are often calculated in an effortful way and incomplete solutions are frequently determined during the procedures.

3. USBB algorithm

The algorithm developed in this paper to solve the MMKP is based on the branch-and-bound principle. The problem is iteratively divided into several branches and conclusions about the optimal solution are drawn using suitable bounds. In contrast to other methods, each node contains one element from each class. Each node can therefore contain the optimal solution. In other methods, only the last examined node contains a complete solution.

First, the branching procedure is described. In the first step of the algorithm, all elements of a class are sorted in descending order according to their utility value u_{ij}. The element with the highest utility in a class is assigned rank 1, and so on. If two elements have the same utility value, the element with a smaller relative resource utilization η_{ij} is preferred:

$$\eta_{ij} = \sum_{k=1}^{K} \frac{w_{ijk}}{c_k} \tag{6}$$

The node with the first elements of each class has the highest possible total utility. If this node is feasible, it forms the optimal solution. However, if one of the K restrictions is violated, further nodes must be formed. The branching is done by exchanging one element in a class with the next higher rank. This is possible until the last element in each class has been exchanged, whereby the node with the smallest possible total utility is represented. The creation of further nodes is done across all J classes.

Figure 1 shows the structure of the corresponding decision tree of a problem instance with J classes and i elements per class. The nodes are specified with the rank of their elements over all classes. The next generated node (2,1,…,1) contains the element with rank 2 from the first class and the two highest-ranking elements of all other classes. Here it can be seen that nodes that have already been formed do not have to be formed again, as would be the case with node (2,2,…,1) in the third row: This could be formed from the node (2,1,…,1) or by node (1,2,…,1).

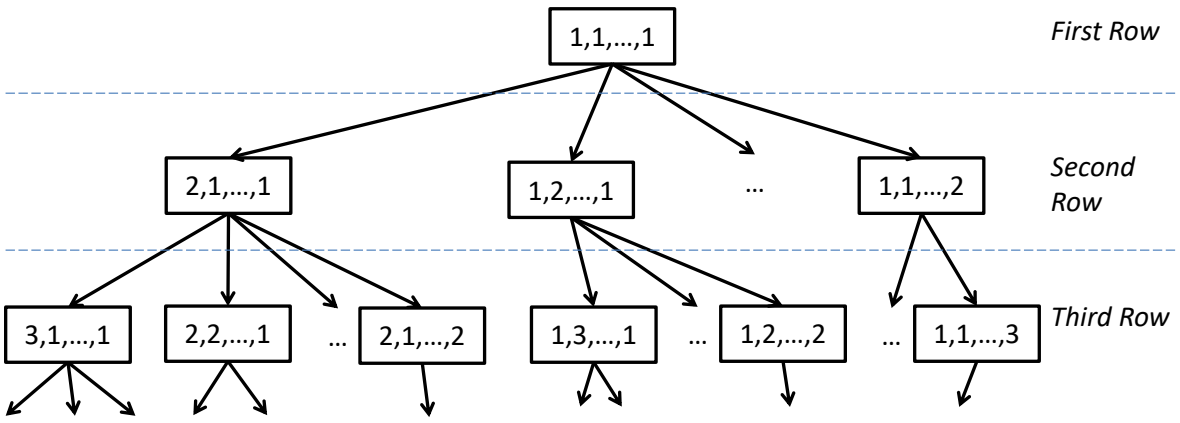

Figure 1: Structure of the decision tree

From the structure of the decision tree, it follows that subsequent nodes cannot have a higher total utility value U than the node from which they were formed. Therefore, a feasible node does not have to

be branched further. None of the following nodes can contain the optimal solution vector since the objective function value cannot improve. If the objective function value of a formed node is smaller than a feasible node found so far, no further nodes must be formed either. Also, in this case the optimal solution cannot be found in the following nodes, because their objective function value is smaller. This reduces the search space so that not all possible nodes have to be examined. The earlier a valid solution is found, the fewer nodes must be considered in total. Table 1 illustrates the formation of further nodes based on the two rules.

	Increased utility value compared to the best solution found so far?		
Node is feasible?		*Yes*	*No*
	Yes	• Update result • Do not create a successor node	• Do not create a successor node
	No	• Create successor node	• Create successor node

Table 1: Rules for building new nodes

In the following, the procedure of the algorithm, which is shown in Figure 2, is described in more detail. As already explained previously, the elements are sorted according to their utility value (USBB1). In the following step USBB2, this sorting is saved by assigning ranks to each element. This step allows a faster determination of the utility value u_{ij} and the respective weights w_{ij} of the selected elements. In the USBB3 step, the initial solution is formed by selecting the first elements of each class and then adding them to the list of open nodes.

The iterative search includes the steps 4 to 11. In step 4, the system checks with each iteration whether there are already formed nodes that have not been checked yet. If this is the case, the first element from the list of open nodes is selected as the new current node. Since newly formed nodes are later placed at the end of the list, the nodes are examined according to the first-in-first-out principle. In step 6, the utility of the current node is calculated by summing the individual utility values u_{ij}. If the utility U of this node is lower than the best utility already found, this node is not examined further but deleted from the list of open nodes (step 11). The comparison of the utility values takes place in step 7. However, if the node has a higher utility, it will be checked in step 8 for its feasibility. An infeasible node is branched as described (step 10). A feasible node with the highest utility found so far represents a new best solution and is saved as such (step 9). After the scan is complete, the node is deleted from the list of formed nodes in step 11. If this is the last node to be examined, the best node found is displayed as the solution. This is therefore the optimal solution for the examined problem instance of MMKP.

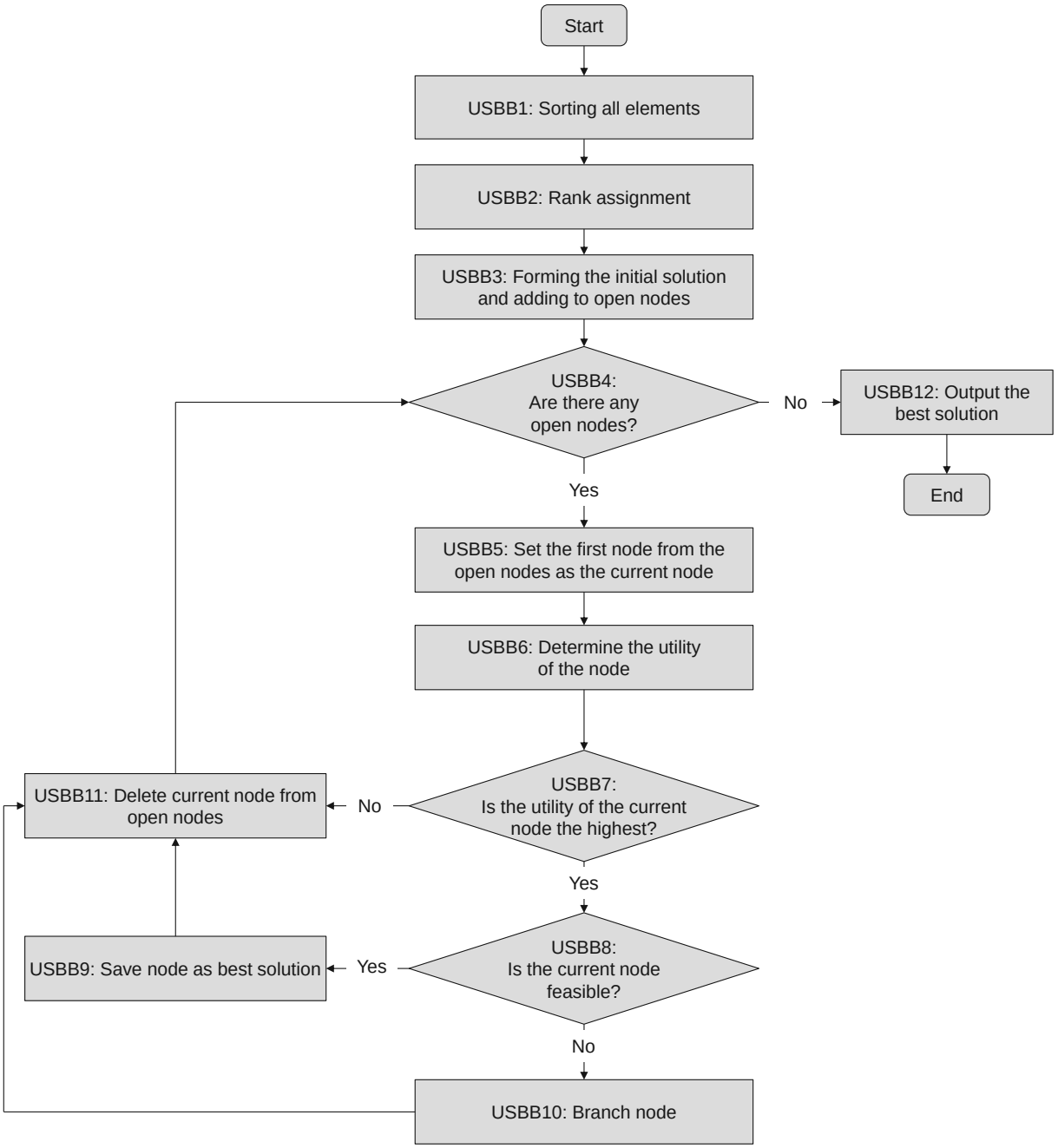

Figure 2: Flow chart of the USBB algorithm

4. Test design

4.1 Theoretical runtime

In the following, the theoretical runtime is determined using the worst, average and best-case scenarios (Goldreich & Vadhan, 2007). The **worst case** regarding the runtime occurs when the last node to be examined represents the only feasible solution. In this case the preparatory steps (USBB1-3) will be performed. The search loop (USBB4-11) must now be run through a total of $I^J - 1$ times. Since none of the nodes examined so far has fulfilled the feasibility condition, a node to be newly examined cannot be excluded by determining its usefulness alone, so that the respective weights w_{ijk} for the K different

dimensions must be formed and compared with the maximum values of the resource constraints C_k. For the last node to be examined, a feasible solution is found for the first time, which is stored. Since there is no further node, this leads to the termination of the search process. If the case is assumed that the considered instances have more than two classes and resource constraints and therefore $J > 2$ and $K \geq 2$ is valid, the asymptotic runtime O^W results in:

$$O^W(I, J, K) = I^J JK \tag{7}$$

The runtime of the algorithm thus grows exponentially in the worst case and depends mainly on the number of elements and classes. This applies to all exact algorithms in the worst case for the MMKP.
In the **best case**, the starting solution is already feasible. Due to the way in which these are formed, this represents the optimum, so that branching, and examination of the other nodes can be omitted. In the Ω notation this can be illustrated as follows:

$$\Omega(I, J, K) = I^2 J \tag{8}$$

The best case behaviour is therefore polynomial. Based on the differences between the two extreme scenarios (best case and worst case) the runtime is significantly dependent on the number of nodes examined. The more nodes that have to be examined, the more often the search loop must be run.
If the variable D now describes the number of required loops passed and p^A the probability that the examined node represents the optimum, the runtime in the **average case** L^A can be specified as follows:

$$L^A = \sum_{D=1}^{I^J} (D - 1) p^A JK \tag{9}$$

Since D can lie between 1 and I^J and one assumes that p^A is evenly distributed, which means that $p^A = \frac{1}{I^J}$, the asymptotic running time in the average case O^A can also be specified in the following way:

$$O^A(I, J, K) = \frac{1}{2} \cdot I^J JK = \frac{1}{2} O^W \tag{10}$$

The determined duration corresponds to half of the asymptotic duration in the worst case, and thus also grows exponentially. This is primarily because large parts of the solution space can be systematically excluded. Compared to the total running time, the running time required for sorting is relatively short.

4.2 Computational study

The described analysis of the runtime does not consider the fathoming of nodes and the branches following them. In order to include this factor in the evaluation of the algorithm as well, a computational study was carried out and then evaluated.

To investigate the runtime of the USBB algorithm more closely, a computational study was realized. Altogether 11,160 problem instances were created for this purpose. These instances differed by their combinations of the parameters I, J and K. Furthermore, the severity of the resource constraints was introduced as an additional parameter. This is understood to be a multiple of the expected value of the weights of the J elements.

The respective values for the utility values u_{ij} and weights w_{ijk} of the elements were generated randomly. Here, the following restrictions were defined:

- The utility value u_{ij} of an element lies between 0 and 200.
- The weight w_{ijk} of an element lies between 0 and 100.

The expected value of the weights μ^W from J is therefore $50J$. The values 0.75, 1 and 1.25 were selected as factors by which this value is multiplied (variable M). Ten instances were created for each of the parameter combinations listed above. This procedure was also used by Khan et al. (2002). For the later evaluation, the average values of the results of the ten instances were formed.

5. Results

The key value which should be determined by means of the computational study is the number of nodes which had to be examined until the optimal solution was found. This number was set in relation to all possible nodes, in the following referred to as search quotient Q. This value corresponds to the product of probability p^A and the number of iterations D which were used to determine the runtime in the average case:

$$Q = p^A \cdot D \tag{11}$$

For the 11,160 problem instances generated, an arithmetic mean of 0.34 with a standard deviation of 0.35 and a median of 0.17 for the search quotient could be determined. Figure 3 shows the histogram of the distribution. Here, the classes are divided in steps with a distance of 0.05. The comparison to a brute-force algorithm showed that the USBB algorithm could find the optimal solution in every instance.

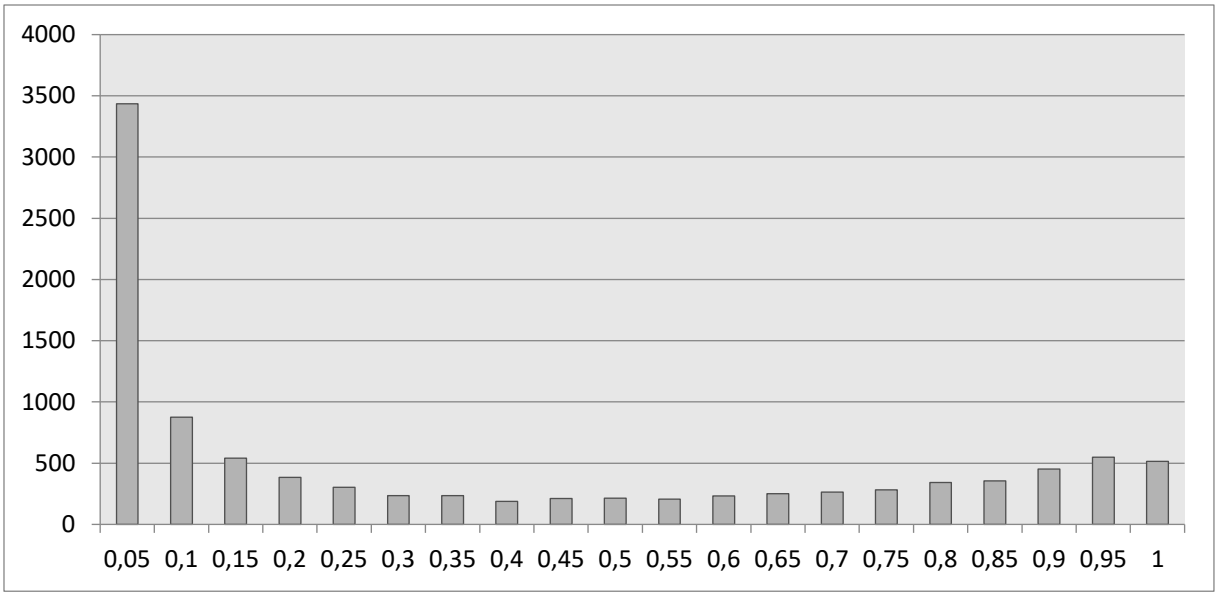

Figure 3: Frequency distribution of the search quotient Q

With the help of this data, the hypothesis that the probability for a node that corresponds to the searched-for optimum is equally distributed over all nodes can now be tested. This is done by means of a chi-square test. For this purpose, the following null hypothesis H_0 is formulated:

H_0: The frequencies of the expressions of the search quotient are equally distributed.

A test value of $\chi^2 = 1.85$ was obtained. Thus, with a probability of $p = 0.999$, the null hypothesis can be rejected. This result shows the limits of the determined runtime in the average case. In order to better illustrate the factors influencing the search quotient Q, the heat maps in Figure 4 - 6 were created. The respective values of Q are shown as a dependency of the number of classes and elements. The number of resource constraints K was set to two.

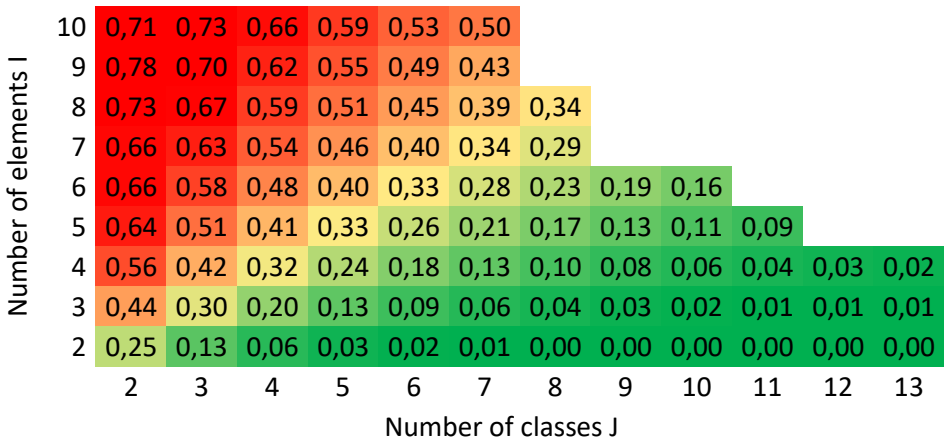

Figure 4: Heat map with $K = 2$ (number of resource constraints) and $M = 0.75$ (factor)

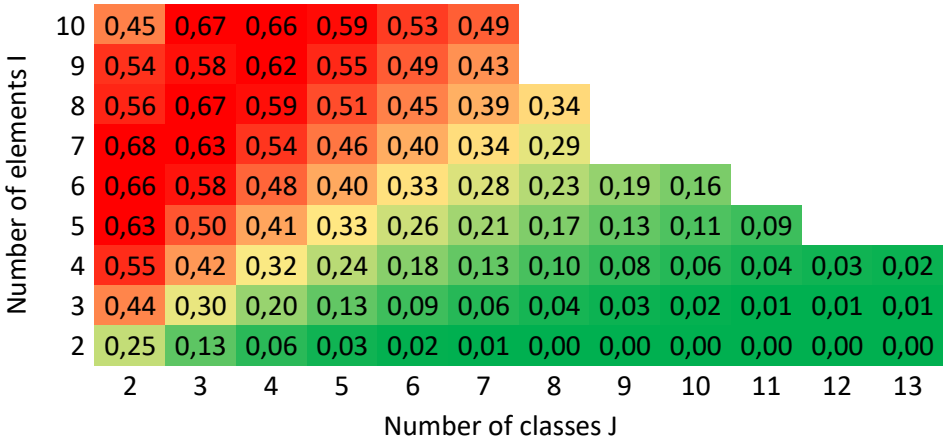

Figure 5: Heat map with $K = 2$ (number of resource constraints) and $M = 1$ (factor)

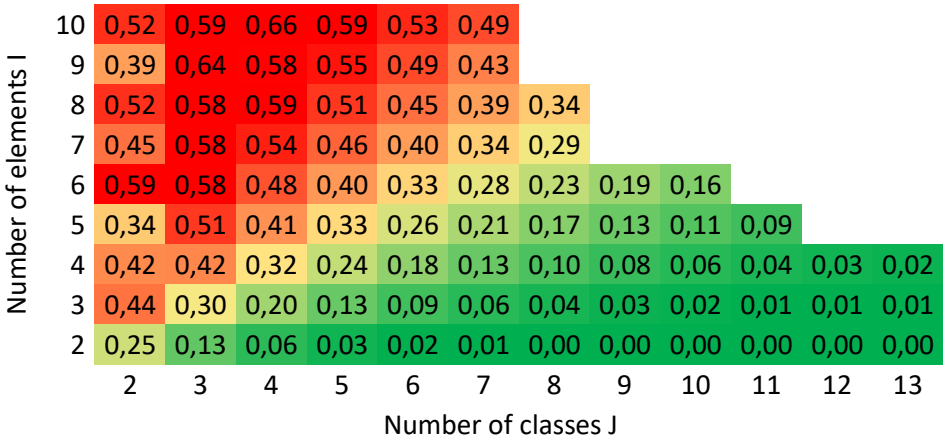

Figure 6: Heat map with $K = 2$ (number of resource constraints) and $M = 1.25$ (factor)

For visual support, the fields were coloured. Fields with values in the lower quantile of the distribution were coloured green and fields with values in the upper quantile were coloured red. Fields with values between the two quantiles were coloured using a colour gradient. For problem instances with more than two classes, with an increasing number of elements, the proportion of examined nodes also increases. This property leads to the fact that the runtime increases correspondingly strongly with the number of elements. The more classes are used, the larger parts of the solution space can be excluded.

6. Conclusions and outlook

In this article, a new exact algorithm was presented to solve the MMKP: the USBB algorithm. The algorithm sorts its elements by their utility in a decision tree, in which every node represents a complete solution. Two logical rules can be used to exclude large solution spaces. The running time of the algorithm was examined both theoretically and empirically. It could be shown that the theoretical

running time is on average $O^{A}(I, J, K) = \frac{1}{2} I^{J} JK$. To test the empirical running time of the developed algorithm, a total of 11,160 problem instances were generated for a computational study. For this purpose, a search quotient was introduced, which represents the ratio of the examined nodes to all possible nodes. On average, 66 % of all possible solutions can be excluded by the structure based on utility sorting. A higher number of classes increases the search quotient. Small and medium-sized instances could be solved exactly by comparison with a brute-force algorithm.

There are also further possibilities for using the algorithm. The proposed sorting in the decision tree can also be applied for the MCKP, because the number of constraints does not influence the the structure of the algorithm. Furthermore, non-linear resource restrictions can be considered, while the target function must be linear. If the target function is non-linear (as for example in the quadratic KP), no solution spaces can be excluded by the utility sorting. For a better performance, the procedure of the algorithm can be parallelised, as the individual branches can be divided between several processor cores. The presented structure of the decision tree also opens further possibilities for the development of heuristics for the MMKP. The algorithm can be applied, for example, to the configuration of factories and learning factories (Kreß & Metternich, 2020).

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Admi Syarif, A. S., Anggraini, D., Kurnia Muludi, K. M., Wamiliana, W., & Gen, M. (2020). Comparing Various Genetic Algorithm Approaches for Multiple-choice Multi-dimensional Knapsack Problem. *International Journal of Intelligent Engineering & System, 13*(5), 455-462.

Akbar, M. M., Manning, E. G., Shoja, G. C., & Khan, S. (2001, May). Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *International Conference on Computational Science* (pp. 659-668). Springer, Berlin, Heidelberg.

Akbar, M. M., Rahman, M. S., Kaykobad, M., Manning, E. G., & Shoja, G. C. (2006). Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers & operations research*, *33*(5), 1259-1273. Basnet, C., & Wilson, J. (2005). Heuristics for determining the number of warehouses for storing non-compatible products. International transactions in operational research, 12(5), 527-538.

Chen, Y., & Hao, J. K. (2014). A “reduce and solve” approach for the multiple-choice multidimensional knapsack problem. *European Journal of Operational Research*, *239*(2), 313-322.

Cherfi, N., & Hifi, M. (2009). Hybrid algorithms for the multiple-choice multi-dimensional knapsack problem. *International Journal of Operational Research*, *5*(1), 89-109.

Cherfi, N., & Hifi, M. (2010). A column generation method for the multiple-choice multi-dimensional knapsack problem. *Computational Optimization and Applications*, *46*(1), 51-73

Crévits, I., Hanafi, S., Mansi, R., & Wilbaut, C. (2012). Iterative semi-continuous relaxation heuristics for the multiple-choice multidimensional knapsack problem. *Computers & Operations Research*, *39*(1), 32-41.

Duenas, A., Di Martinelly, C., & Tütüncü, G. Y. (2014, September). A Multidimensional Multiple-Choice Knapsack Model for Resource Allocation in a Construction Equipment Manufacturer Setting Using an Evolutionary Algorithm. In: IFIP *International Conference on Advances in Production Management Systems*. Springer, Berlin, Heidelberg. 539-546

Gao, C., Lu, G., Yao, X., & Li, J. (2017). An iterative pseudo-gap enumeration approach for the multidimensional multiple-choice knapsack problem. *European Journal of Operational Research*, *260*(1), 1-11.

Gavish, B., & Pirkul, H. (1985). Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. *Mathematical programming*, *31*(1), 78-105.

Ghasemi, T., & Razzazi, M. (2011). Development of core to solve the multidimensional multiple-choice knapsack problem. *Computers & Industrial Engineering*, *60*(2), 349-360.

Ghassemi-Tari, F., Hendizadeh, H., & Hogg, G. L. (2018). Exact Solution Algorithms for Multi-dimensional Multiple-choice Knapsack Problems. Current Journal of Applied Science and Technology

Goldreich, O., & Vadhan, S. (2007). Special issue on worst-case versus average-case complexity editors' foreword. *computational complexity*, *16*(4), 325-330.

Hanafi, S., Mansi, R., & Wilbaut, C. (2009, October). Iterative relaxation-based heuristics for the multiple-choice multidimensional knapsack problem. In *International Workshop on Hybrid Metaheuristics* (pp. 73-83). Springer, Berlin, Heidelberg.

He, C., Leung, J. Y., Lee, K., & Pinedo, M. L. (2016). An improved binary search algorithm for the Multiple-Choice Knapsack Problem. *RAIRO-Operations Research*, *50*(4-5), 995-1001.

Hifi M, Michrafy M, Sbihi A (2004). An exact algorithm for the multi-choice multidimensional knapsack problem. *Université Panthéon-Sorbonne (Paris 1), Cahiers de la Maison des Science Economiques*. S. 1–16.

Hifi, M., Michrafy, M., & Sbihi, A. (2006). A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem. *Computational Optimization and Applications*, *33*(2-3), 271-285.

Hifi, M., & Wu, L. (2012). An equivalent model for exactly solving the multiple-choice multidimensional knapsack problem. *International Journal of Combinatorial Optimization Problems and Informatics*, *3*(3), 43-58.

Hifi, M., & Wu, L. (2015). Lagrangian heuristic-based neighbourhood search for the multiple-choice multi-dimensional knapsack problem. *Engineering Optimization*, *47*(12), 1619-1636.

Hiremath, C. S., & Hill, R. R. (2013). First-level tabu search approach for solving the multiple-choice multidimensional knapsack problem. *International Journal of Metaheuristics*, *2*(2), 174-199.

Htiouech, S., Bouamama, S., & Attia, R. (2013). OSC: solving the multidimensional multi-choice knapsack problem with tight strategic Oscillation using Surrogate Constraints. *International Journal of Computer Applications*, *73*(13), 1-7.

Hwang, J. (2018). An Integer Programming-based Local Search for the Multiple-choice Multidimensional Knapsack Problem. *Journal of the Korea Society of Computer and Information*, *23*(12), 1-9.

Iqbal, S., Bari, M. F., & Rahman, M. S. (2010, September). Solving the multi-dimensional multi-choice knapsack problem with the help of ants. In International Conference on Swarm Intelligence (pp. 312-323). Springer, Berlin, Heidelberg.

Khan, M. (1998). Quality adaptation in a multisession multimedia system: Model, algorithms and architecture (Doctoral dissertation).

Khan, S., Li, K. F., Manning, E. G., & Akbar, M. M. (2002). Solving the knapsack problem for adaptive multimedia systems. *Stud. Inform. Univ.*, *2*(1), 157-178.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack Problems. Springer, Berlin, Heidelberg.

Kreß, A., & Metternich, J. (2020). System development for the configuration of learning factories. *Procedia Manufacturing*, *45*, 146-151.

Mansi, R., Alves, C., Valerio de Carvalho, J. M., & Hanafi, S. (2013). A hybrid heuristic for the multiple choice multidimensional knapsack problem. *Engineering Optimization*, *45*(8), 983-1004.

Mansini, R., & Zanotti, R. (2020). A Core-Based Exact Algorithm for the Multidimensional Multiple Choice Knapsack Problem. INFORMS Journal on Computing.

Moser, M., Jokanovic, D. P., & Shiratori, N. (1997). An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, *80*(3), 582-589.

Parra-Hernandez, R., & Dimopoulos, N. J. (2005). A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *35*(5), 708-717.

Pisinger, D. (2001). Budgeting with bounded multiple-choice constraints. *European Journal of Operational Research*, *129*(3), 471-480.

Shojaei, H., Basten, T., Geilen, M., & Davoodi, A. (2013). A fast and scalable multidimensional multiple-choice knapsack heuristic. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *18*(4), 1-32.

Shojaei, H., Ghamarian, A., Basten, T., Geilen, M., Stuijk, S., & Hoes, R. (2009, July). A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In *Proceedings of the 46th Annual Design Automation Conference* (pp. 917-922).

Vasko, F. J., Lu, Y., & Zyma, K. (2016). An empirical study of population-based metaheuristics for the multiple-choice multidimensional knapsack problem. *International Journal of Metaheuristics*, *5*(3-4), 193-225.

Watson, R. K. (2001). Packet Networks and optimal admission and upgrade of service level agreements: applying the utility model. *MA Sc. thesis, ECE, University of Victoria*.

Xia, Y., Gao, C., & Li, J. (2015, September). A stochastic local search heuristic for the multidimensional multiple-choice knapsack problem. In *Bio-Inspired Computing-Theories and Applications* (pp. 513-522). Springer, Berlin, Heidelberg.

Ykman-Couvreur, C., Nollet, V., Catthoor, F., & Corporaal, H. (2011). Fast multidimension multichoice knapsack heuristic for MP-SoC runtime management. *ACM Transactions on Embedded Computing Systems (TECS)*, *10*(3), 1-16.

Zennaki, M. (2013). A New Hybrid Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem. Wseas Transactions on Information Science and Application, 7(10), 219-229.

Appendix

Pseudo code of the USBB algorithm

```
//USBB1: Sort all elements according to their utility in each class
//USBB2: Assign ranks to all elements
    //Set the maximum utility to zero
    UtilityMax = 0
//USBB3: Start with the elements of rank 1 of each class
    For j = 1 To ClassCount
        i(j) = 1
    Next k
    //Create a NodeList with this node
    NodeList.Add Array( i(1), ... , i(ClassCount))
//USBB4: As long as the NodeList is not empty, proceed
Do
   //USBB5: Assign the values of the first element of the node list to the variables i(j)
   For j = 1 To ClassCount
       i(j) = NodeList(1)(j)
   Next j
```

```
    //USBB6: Calculate the utility and resource consumption of the node
    UtilityCurrent = 0
    For k = 1 To ResourceCount
        ResourceConsumption(k) = 0
    Next k
    For j = 1 To ClassCount
        UtilityCurrent = UtilityCurrent + Utility(j,i(j))
        For k = 1 To ResourceCount
            ResourceConsumption(k) = ResourceConsumption(k) + Resource(k,j,o)
        Next k
    Next j
//USBB7: Check if the utility of the current node is the highest found
    If UtilityCurrent > UtilityMax
        Then
//USBB8: Check the feasibility of the current node
        If ResourceCurrent(k) <= ResourceMax(k) For Each k
            Then
            //USBB9: If permissible and utility is increased, update the result
            UtilityMax = UtilityCurrent
            Else
            //USBB10: If not allowed and utility is increased, create successor nodes
            For j = 1 To ClassCount
                //Do not create more successor nodes than there are objects in the class.
                If i(j)<ObjectCount
                    Then
                    //Do not create successor nodes if this child has already been created.
                    If AlreadyAdded( i(1), ... , i(j) + 1, ... , i(ClassCount) = False
                        Then
```

```
                    NodeList.Add Array( i(1), ... , i(j) + 1 , ... i(ClassNumber)

                    AlreadyAdded( i(1), ... , i(j) + 1, ... , i(ClassCount) = True

                End If

        Next j

        //USBB11: Remove the node just checked

        NodeList.Remove(1)

Loop While NodeList.Count <> 0
```