INTERNATIONAL
TRANSACTIONS
IN OPERATIONAL
RESEARCH

Intl. Trans. in Op. Res. XX (20XX) 1–13
DOI: xx.xxxx/itor.xxxxx

Multi-dimensional indexing on dynamic programming for multi-objective knapsack problem

Marcos Daniel Valadão Baroni[a,*] and Flávio Miguel Varejão[a]

[a]*Departamento de Informática/Universidade Federal do Espírito Santo (UFES), Av. Fernando Ferrari, 514, Vitória - ES, Brazil*
E-mail: marcos.baroni@aluno.ufes.br [Baroni]; fvarejao@inf.ufes.br [Varejão]

Received DD MMMM YYYY; received in revised form DD MMMM YYYY; accepted DD MMMM YYYY

Abstract

This work proposes a performance improvement of a state of art dynamic programming algorithm for solving the multi-objective knapsack problem by using a multi-dimensional indexing strategy for handling large amount of intermediate solutions, especially for high dimensional cases. Through several computational experiments the applicability and efficiency of the strategy is shown, considerably reducing the number of solution comparisons executed which resulted in an algorithm speedup up to 2.3 for bi-dimensional cases and up to 15.5 for 3-dimensional cases.

Keywords: multi-objective knapsack problem; dynamic programming; multi-dimensional indexing

1. Introduction

Many real applications like project selection, capital budgeting and cutting stock involves optimizing multiple objectives that are usually conflicting. Some of those problems can be modelled as a multi-objective knapsack problem. Several exact approaches have been proposed in the literature to solve the multi-objective knapsack problem. Examples of such approaches are a ε-constraint method presented in [**?**], a branch and bound algorithm [**?**], a labeling algorithm [**?**] and a dynamic programming algorithm proposed in [**?**] which is the approach currently achieving the best results. Some later contributions for the dynamic programming approach are an algorithmic improvement for the bi-objective case [**?**] and some techniques for reducing its memory usage [**?**].

One of the main difficulties on multi-objective optimization problems is the large cardinality of the set of non-dominated (or *efficient*) solutions, which has motivated research to provide an approximation of the solution set [**? ?**]. Indeed, it is well-known, in particular, that most multi-objective combinatorial optimization problems are *intractable*, in the sense that the number of non-dominated points is exponential

*Research supported by Fundação de Amparo à Pesquisa do Espírito Santo.

in the size of the instance [**?**].

It is also well known that the use of a proper data structure usually has considerable impact on the efficiency of an algorithm, especially when dealing with large amount of data. In this work we propose a performance improvement on the state of art dynamic programming algorithm for the multi-objective knapsack problem by applying a multi-dimensional indexing strategy for handling the large amount of intermediate solutions.

Section 2 of this paper reviews the basic concepts of multi-objective optimization and defines the multi-objective knapsack problem. Section 3 introduces the dynamic programming algorithm. Section 4 presents the proposal of multi-dimensional indexing by using a k-d tree. Section 5 analyzes the approach through computational experiments and Section 6 provides conclusions and lines for future research.

2. The Multiobjective Knapsack Problem

A general multi-objective optimization problem can be described as a vector function f that maps a decision variable (solution) to a tuple of m objectives. Formally:

$$\begin{aligned} &\max \boldsymbol{y} = f(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\big) \\ &\text{subject to } \boldsymbol{x} \in X \end{aligned}$$

where $\boldsymbol{x}$ is the *decision variable*, X denotes the set of feasible solutions, and $\boldsymbol{y}$ is the *objective vector* (or *criteria vector*) where each objective has to be maximized.

Considering two decision vectors $\boldsymbol{a}, \boldsymbol{b} \in X$, $\boldsymbol{a}$ is said to *dominate* $\boldsymbol{b}$ if, and only if $\boldsymbol{a}$ is at least as good as $\boldsymbol{b}$ in all objectives and better than $\boldsymbol{b}$ in at least one objective. For shortening we will say that $\boldsymbol{a}$ dominates $\boldsymbol{b}$ by saying $dom(\boldsymbol{a}, \boldsymbol{b})$. Formally:

$$dom(\boldsymbol{a}, \boldsymbol{b}) = \begin{cases} \forall i \in \{1, 2, \ldots, m\} : f_i(\boldsymbol{a}) \geqslant f_i(\boldsymbol{b}) \text{ and} \\ \exists j \in \{1, 2, \ldots, m\} : f_j(\boldsymbol{a}) > f_j(\boldsymbol{b}) \end{cases}$$

A feasible solution $\boldsymbol{a} \in X$ is called *efficient* if it is not dominated by any other feasible solution. The set of all efficient solutions of a multi-objective optimization problem is known as *Pareto optimal*. Solving a multi-objective problem consists in providing its Pareto optimal set. It is worth remarking that the size of Pareto optimal set for this problem tends to rapidly grow mainly with the number of objectives.

An instance of a multi-objective knapsack problem (MOKP) with m objectives consists of an integer capacity $W > 0$ and n items. Each item i has a positive weight w_i and nonnegative integer profits $p_i^1, p_i^2, \ldots, p_i^m$. Each profit p_i^k represents the contribution of the i-th item for k-th objective. A solution is represented by a set $\boldsymbol{x} \subseteq \{1, \ldots, n\}$ containing the indexes of the items included in the solution. A solution is feasible if the total weight included in the knapsack does not exceed its capacity. Formally the

definition of the problem is:

$$\begin{aligned} \max\ & f(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\big) \\ \text{subject to}\ & w(\boldsymbol{x}) \leqslant W \\ & \boldsymbol{x} \subseteq \{1, \ldots, n\} \end{aligned}$$

where

$$f_j(\boldsymbol{x}) = \sum_{i \in \boldsymbol{x}} p_i^j \quad j = 1, \ldots, m$$

$$w(\boldsymbol{x}) = \sum_{i \in \boldsymbol{x}} w_i$$

The MOKP is considered a $\mathcal{NP}$-Hard problem since it is a generalization of the well-known $0-1$ knapsack problem, in which $m = 1$. It is quite difficult to determine the Pareto optimal set for the MOKP, especially for high dimensional instances, in which the Pareto optimal set tends to grow exponentially. Even for the bi-objective case, small problems may prove intractable. For this reason we are interested in developing efficient methods for handling large solution sets, which may bring tractability to previously intractable instances.

Considering two solutions $\boldsymbol{x}, \boldsymbol{y} \subseteq \{1, \ldots, n\}$, $\boldsymbol{y}$ is called *extension* of $\boldsymbol{x}$, denoted as $ext(\boldsymbol{y}, \boldsymbol{x})$, iff $\boldsymbol{x} \subseteq \boldsymbol{y}$. Any set $\boldsymbol{e} \subseteq \{1, \ldots, n\}$ such that $\boldsymbol{x} \cap \boldsymbol{e} = \emptyset$ is called an *extender* of $\boldsymbol{x}$. If $w(\boldsymbol{x}) + w(\boldsymbol{e}) \leqslant W$ than $\boldsymbol{e}$ is called a *feasible extender* of $\boldsymbol{x}$. A solution $\boldsymbol{x}$ is called *deficient* if it has available space to fit one or more item, i.e., $w(\boldsymbol{x}) + min\{w_i : i \notin \boldsymbol{x}\} \leqslant W$. We say $\boldsymbol{x}$ *knapsack-dominates* $\boldsymbol{y}$, denoted as $dom_k(\boldsymbol{x}, \boldsymbol{y})$, if $\boldsymbol{x}$ dominates $\boldsymbol{y}$ and does not weight more than $\boldsymbol{y}$. Formally:

$$dom_k(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} dom(\boldsymbol{x}, \boldsymbol{y}) & \text{and} \\ w(\boldsymbol{x}) \leqslant w(\boldsymbol{y}) \end{cases} \tag{1}$$

The concept of knapsack dominance was proposed by Weingartner and Ness [**?**]. Figure 1 illustrates the concept for a problem with $m = 1$. Any solution in the cross-hatched area knapsack-dominates the marked solution. The knapsack dominance concept is the basis of the algorithm considered in this work which will be presented in the next section.

3. The Dynamic Programming algorithm

The algorithm addressed in this work can be seen as a MOKP specialization of the classical Nemhauser and Ullmann's algorithm proposed in [**?**] for generically solving knapsack problems. The basic algorithm will be presented in Section 3.1. Three optimizations for reducing the number of partial solutions will then be presented: reordering of items (3.2), the avoidance of deficient solutions (3.3) and the elimination of unpromising partial solutions (3.4).

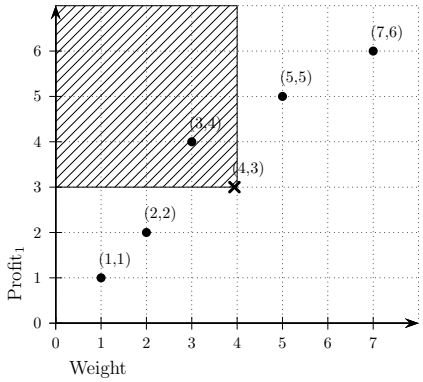


Fig. 1: A knapsack-dominated solution.

3.1. The Nemhauser-Ullmann algorithm

The Nemhauser and Ullmann's algorithm generically solves knapsack problems by applying the concept of knapsack dominance to remove partial solutions that will not generate efficient ones. A basic multi-objective version of the algorithm is presented by Algorithm 1.

Algorithm 1 Nemhauser and Ullmann's algorithm for MOKP

1: **function** DP($\boldsymbol{p}, \boldsymbol{w}, W$)
2: $S^0 = \{\emptyset\}$
3: **for** $k \leftarrow 1, n$ **do**
4: $S^k_* = S^{k-1} \cup \{\boldsymbol{x} \cup k \mid \boldsymbol{x} \in S^{k-1}\}$ ▷ solutions extension
5: $S^k = \{\boldsymbol{x} \mid \nexists \boldsymbol{a} \in S^k_* : dom_k(\boldsymbol{a}, \boldsymbol{x})\}$ ▷ partial dominance filter
6: **end for**
7: $P = \{\boldsymbol{x} \mid \nexists \boldsymbol{a} \in S^n : dom(\boldsymbol{a}, \boldsymbol{x}) \mid w(\boldsymbol{x}) \leqslant W\}$ ▷ dominance/feasibility
8: **return** P
9: **end function**

The algorithm begins by defining an initial solution set S^0 containing only the empty solution (line 2). At a k-th stage the algorithm receives a set S^{k-1} exclusively containing solutions composed by the first $k-1$ items, i.e., $\forall \boldsymbol{x} \in S^{k-1}, \boldsymbol{x} \subseteq \{1, \ldots, k-1\}$. The set S^{k-1} is then expanded by adding a copy of its solutions but now including the k-th item (line 4). This new set is defined by S^k_* having twice the cardinality of S^{k-1}. Set S^k is then defined by selecting from S^k_* all solutions that are not knapsack dominated by any other existing solution (line 5). The last step of the algorithm (line 7) consists of selecting the efficient solutions, i.e., that are not dominated by any other partial solution according to their objective values. Any partial solution in the context of stages prior than n-th stage will be considered *efficient* if it is not knapsack dominated by any other partial solution.

Regarding its simplicity Algorithm 1 is considerably powerful. However the exponential growth potencial of MOKP solutions sets may severely compromise its performance. One way of tackling

this is reducing the number of partial solutions handled through the algorithm stages. Following, three optimizations for achieving this reduction, proposed in the Bazgan's algorithm [**?**], are described.

3.2. Input order of items

The first important issue to be considered is the order in which the items are introduced in the algorithm. It is well-known that good knapsack problem solutions are generally composed of items with the best cost-benefit rate. Therefore, the prioritization of those items tends to lead to better solutions.

An appropriate cost-benefit rate for the single objective case may be directly derived from the profit/weight ratio of the items. For the multi-objective case however there is no such natural measure. Thus, one may mainly consider item orders defined over the aggregation of their ranks derived from cost-benefit from all objectives.

We denote $\mathcal{O}^j$ as the set of items ordered by descending order of profit/weight ratio regarding objective j. Formally:

$$\mathcal{O}^j = (o_1^j, \ldots, o_n^j), \qquad cb^j(o_1^j) \geqslant \ldots \geqslant cb^j(o_n^j)$$

where function $cb^j(a) = p_a^j / w_a$ is the cost-benefit function of item a regarding objective j. Let r_i^j be the rank of item i in order $\mathcal{O}^j$ and $\mathcal{O}^{sum}$, $\mathcal{O}^{min}$ and $\mathcal{O}^{max}$ denotes orders according to increasing values of the sum, minimum and maximum ranks respectively. Formally:

$$\begin{aligned} r_i^j &= \max\{k \mid o_k^j = i\} \\ r_i^{sum} &= \textstyle\sum_{j=1}^{m} r_i^j \\ \mathcal{O}^{sum} &= (o_1, \ldots, o_n), \qquad r_{o_1}^{sum} \leqslant \ldots \leqslant r_{o_n}^{sum} \end{aligned}$$

The orders $\mathcal{O}^{min}$ and $\mathcal{O}^{max}$ are conceived in the same manner as $\mathcal{O}^{sum}$, except for the fact that $\frac{r_i^{sum}}{m}$ is added up in their ranks as tie breaking criteria. The notation $\mathcal{O}(\boldsymbol{s})$ will be used to denote order $\mathcal{O}$ over a restrict set $\boldsymbol{s}$ of items and $\mathcal{O}_{rev}$ to denote the reverse order of $\mathcal{O}$.

The order of interest is the one that generates the smallest partial solutions sets. According to literature, experimental tests have reported $\mathcal{O}^{max}$ superior to others for inputting items on the algorithm and was adopted in this work as the cost-benefit order of choice.

3.3. Deficient solutions avoidance

At k-th stage of the algorithm a copy of all previous solutions is strictly added to the new solution set without adding the k-th item (line 4). However preserving solutions with too much available capacity may generate unnecessary deficient solutions. If a partial solution $\boldsymbol{x} \in S^{k-1}$ has enough space to fit all remaining items, i.e. , $w(\boldsymbol{x}) + \sum_{i=k}^{n} w_i \leqslant W$, $\boldsymbol{x}$ may be discarded and only $\boldsymbol{x} \cup \{k\}$ kept, once keeping $\boldsymbol{x}$ will certainly lead to deficient solutions. It is worth noting that this optimization only regards the available capacities of solutions and the weights of remaining items.

3.4. Elimination of unpromising partial solutions

Another way of reducing the amount of partial solutions generated is by analyzing their potential to generate an efficient solution regarding their available capacity, their current quality and the set of remaining items. This may be done by computing an upper bound for their objective functions and a set of lower bounds values over the current set of partial solutions. If the upper bound of a solution is dominated by any known lower bound this partial solution can be safely discarded.

Lower bound values of partial solutions can be computed by greedily filling their available capacity with remaining items. As in this case the order in which the items are inserted is relevant, items must be prioritized according to order $\mathcal{O}^{max}$. Given a partial solution $\boldsymbol{x}$ and the set $\boldsymbol{s}$ of remaining items, the lower bound function $lb(\boldsymbol{x}, \boldsymbol{s})$ can be defined as:

$$lb(\boldsymbol{x}, \boldsymbol{s}) = f(\boldsymbol{x}) + f(\boldsymbol{y})$$

where

$$\boldsymbol{y} = \left\{ o_i \;\middle|\; \sum_{j=1}^{i} w_{o_j} \leqslant W - w(\boldsymbol{x}) \right\}$$
$$(o_1, \ldots, o_k) = \mathcal{O}^{max}(\boldsymbol{s})$$

The upper bound of a partial solution most be an upper limit for each objective function considering any feasible extender defined on the remaining items. It must be an optimistic measure, ensuring that any higher value is unfeasible. For this reason those values will be computed separately for each j-th objective, according to the order $\mathcal{O}^j$. Given a partial solution $\boldsymbol{x}$ and the set $\boldsymbol{s}$ of remaining items, the upper bound function $ub(\boldsymbol{x}, \boldsymbol{s})$ can be formally defined as:

$$ub(\boldsymbol{x}, \boldsymbol{s}) = (u_1, \ldots, u_m)$$

where

$$u_j = f_j(\boldsymbol{x}) + f_j(\boldsymbol{y}) + r.p_i^j$$
$$\boldsymbol{y} = \left\{ o_i^j \;\middle|\; \sum_{l=1}^{i} w_{o_l^j} \leqslant W - w(\boldsymbol{x}) \right\}$$
$$(o_1^j, \ldots, o_k^j) = \mathcal{O}^j(\boldsymbol{s})$$
$$r = \frac{W - w(\boldsymbol{x}) - w(\boldsymbol{y})}{w_{o_{i+1}^j}}$$

This upper bound definition was presented in [**?**] and is the same one adopted in [**?**].

These three optimization proposals were applied on Algorithm 1 for defining algorithm Algorithm 2 proposed by [**?**] which is considerably faster once it handles much less partial solutions. Further theoretical support for the proposals can be found in the original paper.

Algorithm 2 begins by defining an initial solution set S^0 containing only the empty solution (line 1). Then the items order is defined according to the chosen cost-benefit order (line 2). For each k-th stage of the algorithm, a set S^k of partial solutions are generated after the set S^{k-1} previously defined (lines 5-7). At line 5 all solutions in the previous set are extended by item o_k except for those which are not feasible. At line 6 all solutions in the previous set are copied, except for those with too much left capacity

Algorithm 2 Bazgan's DP algorithm for the MOKP

1: **function** BAZDP($\boldsymbol{p}, \boldsymbol{w}, W$)
2: $S^0 = \{\emptyset\}$
3: $o_1, \ldots, o_n = \mathcal{O}^{max}$
4: **for** $k \leftarrow 1, n$ **do**
5: $S^k_* = \{\boldsymbol{x} \cup \{o_k\} \mid \boldsymbol{x} \in S^{k-1} \wedge w(\boldsymbol{x}) + w_{o_k} \leqslant W\}$
6: $\cup \{\boldsymbol{x} \mid \boldsymbol{x} \in S^{k-1} \wedge w(\boldsymbol{x}) + w_{o_k} + \ldots + w_{o_n} > W\}$
7: $S^k = \{\boldsymbol{x} \in S^k_* \mid (\nexists \boldsymbol{a} \in S^k_*)[dom_k(\boldsymbol{a}, \boldsymbol{x}) \vee dom(lb(\boldsymbol{a}), ub(\boldsymbol{x}))]\}$
8: **end for**
9: **return** S^n
10: **end function**

(deficient solution avoidance). At line 7 the knapsack dominance and lower bound filter are applied. Any solution which is knapsack dominated by any existing solution or has an upper bound dominated by the lower bound of any existing solution is discarded.

For efficiency reasons the original authors suggest to apply the knapsack dominance check operation in parallel with the construction of the new solution set S^k_*. The elimination of unpromising solutions, which is more computationally expensive, is implemented as a final step in which a set of lower bounds is primarily generated out of the partial solutions in S^k_*. Then the upper bound of each partial solution is generated and compared with the lower bound set. The partial solutions for set S^k_* are also generated and stored in ascending order of weight, which allows us to simplify the knapsack dominance check operation, once their weights do not need to be explicitly evaluated.

Despite the effort to reduce the number of partial solutions Algorithm 2 still suffers from the size of efficient solutions set, especially with higher number of objectives, since the enumeration of all solutions is necessary. For this reason it becomes crucial the development of a method that operates efficiently on large sets of solution.

Operations on lines 2-6 demand linear time and do not present difficulties. However the dominance check operation of line 7 may represent the computational bottleneck of the algorithm. This operation must cover the entire set of solutions and if the right strategy is not considered, the algorithm will have to compare each solution individually, which represents a great computational effort. For the bi-dimensional case [**?**] proposes to use an AVL tree for indexing solutions by its first objective value, which improves significantly its performance over the use of a list.

This work proposes to increase the performance of dominance check operations by multi-dimensional indexing partial solutions. The strategy may reduces the computational complexity of the algorithm, minimizing the impact of solution growth, which may even bring viability to previously unfeasible instances. This proposal is discussed in Section 4.

4. Indexed dominance checking

The main operation in Algorithm 2 is filtering all partially efficient solutions from the early generated set (line 7). Ensuring that a partial solution has no dominant may demand quadratic effort on the total

number of solutions if implemented as pairwise comparison. This operation tends to be the computational bottleneck of the algorithm since it is executed on every stage and all others operations have linear computational complexity. For this reason the optimization of this operation is crucial for an efficient algorithm.

If solutions are mapped into points in a multi-dimensional space, it can be deduced from equation (1) that this operation corresponds on checking whether a point exists in a certain region. Formally:

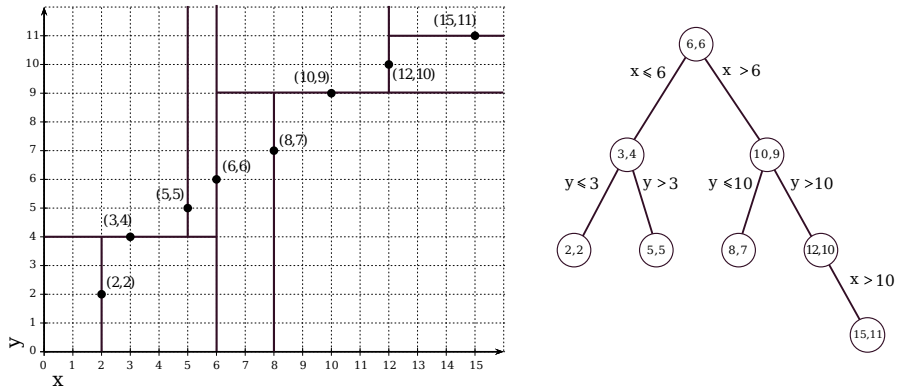
$$\text{if } \; dom_k(\boldsymbol{y}, \boldsymbol{x}) \; \text{ then } \; pnt(\boldsymbol{y}) \in R(\boldsymbol{x})$$

where

$$pnt(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x}), w(\boldsymbol{x})\big)$$
$$R(\boldsymbol{x}) = \big\{a \in \mathbb{R}^{m+1} \;\big|\; a_{m+1} \leqslant w(\boldsymbol{x}) \text{ and } a_i \geqslant f_i(\boldsymbol{x}),\ i \in \{1, \ldots, m\}\big\}$$

The problem of determining whether a point exists in a certain region of space is known in geometric computation as *range search*[**?**] and is usually solved with the use of a k-d tree[**?**]. The k-d tree is a type of binary search tree for indexing multi-dimensional data with simple construction and low space usage. Despite its simplicity, it efficiently supports nearest neighbour search and range search operations [**?**] and for those reasons k-d tree is widely used on spacial geometry algorithms [**? ?**], clustering [**? ?**] and graphic rendering algorithms [**?**].

Like a standard binary search tree, the k-d tree subdivides data at each recursive level of the tree. Unlike a standard binary tree, that uses only one key for all levels of the tree, the k-d tree uses k keys and cycles through these keys for successive levels of the tree. Figure 2 presents (a) points on a plane indexed by a (b) 2-d tree. The first and third level of the 2-d tree indexes x component while second level indexes y component. Each point branches a subregion in two – showed on the figure by a thicker line – according to the component being indexed.



(a) points displayed on a plane (b) points arranged on a kd-tree

Fig. 2: Example of points indexed in a k-d tree.

Figure 3 presents an example of dominance check operation with indexed solutions using a 2-d tree.

The gray area has no intersection with the dominant (cross-hatched) area, therefore solutions inside it are not evaluated. The efficiency of this pruning action grows with the amount of points.

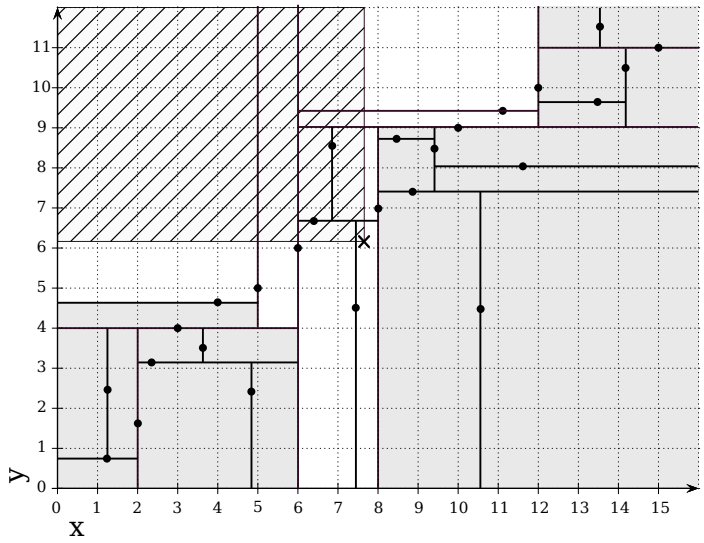


Fig. 3: Example of dominance check operation using k-d tree for solution indexing.

Concerning the efficiency of the k-d tree, it is important to consider the number of dimensions it is indexing. As a general rule, a k-d tree is suitable for the efficiently indexing of n elements if n is much greater than 2^k. Otherwise, when k-d tree is used with high-dimensional data, most of the elements in the tree will be evaluated and the efficiency is no better than exhaustive search [**?**].

The use of k-d tree is applied on Algorithm 2 to index sets for which there is a demand of dominance check operation. These sets are (a) the set S_*^k of intermediate partial solutions and (b) the set of upper bound solutions. In both cases there is no need to evaluate the weight of the solution, therefore a k-d tree indexing up to all m objective values may be used.

It is expected that the use of a k-d tree assists the dominance check operation by pruning a larger amount of points than a single dimensional indexing, which may demand a smaller number of solution evaluation, thus increasing the algorithm performance.

5. Computational experiments

Several computational experiments were performed with the objective of verifying the efficiency of multi-dimensional indexing on the algorithm, especially for instances with higher dimensions. All instances were generated in the same manner as described in [**?**]. Four types of bi-objective instances were considered:

Type A) Random instances: $p_i^j \in [1, 1000], w_i \in [1, 1000]$.
Type B) Unconflicting instances: $p_i^1 \in [111, 1000]$,

$p_i^2 \in [p_i^1 - 100, p_i^1 + 100]$,
$w_i \in [1, 1000]$.

Type C) Conflicting instances: $p_i^1 \in [1, 1000]$,
$p_i^2 \in [max\{900 - p_i^1; 1\}, min\{1100 - p_i^1, 1000\}]$,
$w_i \in [1, 1000]$.

Type D) Conflicting instances with correlated weight: $p_i^1 \in [1, 1000]$,
$p_i^2 \in [max\{900 - p_i^1; 1\}, min\{1100 - p_i^1, 1000\}]$,
$w_i \in [p_i^1 + p_i^2 - 200, p_i^1 + p_i^2 + 200]$.

where $\in [a, b]$ denotes uniformly randomly generated in range $[a, b]$. Instances of type B are considered the easiest ones while type D are considered the hardest. For all instances, we set $W = \frac{1}{2}\left\lfloor \sum_{k=1}^{n} w^k \right\rfloor$. For each type and each value of n 10 different instances were generated. The experiments were run on a Intel® Core™ i5-3570 3.40HGz computer with 4GB of RAM and the algorithms were implemented in C programming language.

Instance			AVL tree	2-d tree	
Type	n	$\|ND\|$	time (s)	time (s)	speedup
A	40	38.1	0.06	0.06	1.0
	60	73.1	1.12	0.88	1.3
	80	125.6	19.81	11.89	1.7
	100	180.4	165.24	76.50	2.2
	120	233.9	708.53	361.87	2.0
B	100	3.1	0.02	0.08	0.3
	200	10.0	0.80	5.09	0.2
	300	24.9	9.45	88.30	0.1
	400	36.2	95.39	730.04	0.1
	500	53.7	255.57	2824.65	0.1
C	20	36.6	0.00	0.00	1.0
	40	102.8	0.65	0.42	1.5
	60	231.9	28.98	14.09	2.1
	80	358.0	564.10	241.54	2.3
	100	513.8	3756.57	1605.19	2.3
D	20	174.9	0.15	0.12	1.3
	30	269.3	16.82	7.60	2.2
	40	478.0	395.76	186.67	2.1
	50	553.4	2459.48	1417.94	1.7

Table 1: Average CPU-time for bi-objective instances.

Table 1 presents results on bi-objective instances where $|ND|$ is the size of the solution set. The last column of the table shows the speedup of using 2-d tree. Fig. 4 presents the number of solutions evaluations for bi-objective cases. The horizontal axis presents the number of items. Each presented value is the average for 10 instances.

It can be noted that the use of 2-d tree increased the performance of the algorithm with a speedup up to

2.3 on instances of type A, C and D, which have large solution sets. In most cases occurred the reduction of almost an order of magnitude in the number of solution evaluations. For instances of type B the use of 2-d tree had a poor performance, even with the reduction in the number of evaluations. This is probably to the small size of the solution set for which the use of the structure is not efficient.

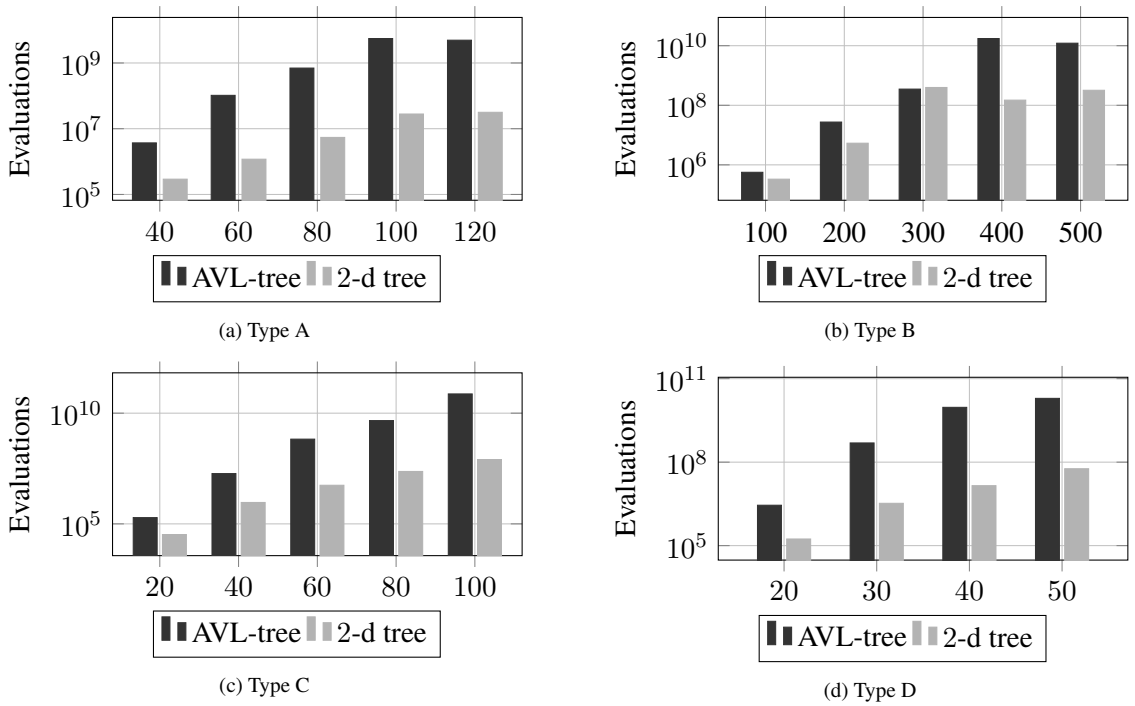


Fig. 4: Average number of solution evaluations for bi-objective instances.

For the experiments with 3-objective cases we considered the generalization introduced in [**?**] for the bi-dimensional types A and C, and proposed the generalization of types B and D as follows:

Type A) Random instances: $p_i^j \in [1, 1000]$
$w_i \in [1, 1000]$

Type B) Unconflicting instances: $p_i^1 \in [111, 1000]$,
$p_i^2 \in [p_i^1 - 100, p_i^1 + 100]$,
$p_i^3 \in [p_i^1 - 100, p_i^1 + 100]$,
$w_i \in [1, 1000]$.

Type C) Conflicting instances: $p_i^1 \in [1, 1000],\ p_i^2 \in [1, 1001 - p_i^1]$
$p_i^3 \in [max\{900 - p_i^1 - p_i^2; 1\}, min\{1100 - p_i^1 - p_i^2, 1001 - p_i^1\}]$
$w_i \in [1, 1000]$.

Type D) Conflicting instances with correlated weight: $p_i^1 \in [1, 1000]$
$p_i^2 \in [1, 1001 - p_i^1]$
$p_i^3 \in [max\{900 - p_i^1 - p_i^2; 1\}, min\{1100 - p_i^1 - p_i^2, 1001 - p_i^1\}]$

$w_i \in [p_i^1 + p_i^2 + p_i^3 - 200, p_i^1 + p_i^2 + p_i^3 + 200]$.

Instance			AVL tree	2-d tree		3-d tree	
Type	n	$\|ND\|$	time (s)	time (s)	speedup	time (s)	speedup
A	50	557.5	41.2	21.3	1.9	18.5	2.2
	60	1240.0	485.9	247.8	1.9	79.9	6.0
	70	1879.3	3179.5	1038.	3.0	614.5	5.1
	80	2540.5	6667.9	3796.0	1.7	2943.9	2.2
	90	3528.5	24476.5	12916.7	1.8	3683.7	6.6
B	100	18.0	0.1	0.3	0.3	0.3	0.3
	200	65.4	11.4	34.4	0.3	29.1	0.4
	300	214.2	307.7	631.5	0.5	583.2	0.5
	400	317.0	4492.9	8464.9	0.5	5402.2	0.8
C	20	254.4	0.06	0.05	1.2	0.03	2.17
	30	1066.6	9.69	4.18	2.3	1.30	7.46
	40	2965.5	471.68	153.21	3.1	30.50	15.5
D	20	4087.7	23.6	10.9	2.17	1.9	12.5
	30	8834.5	8914.2	3625.3	2.5	1019.5	8.7

Table 2: Average CPU-time for 3-objective instances.

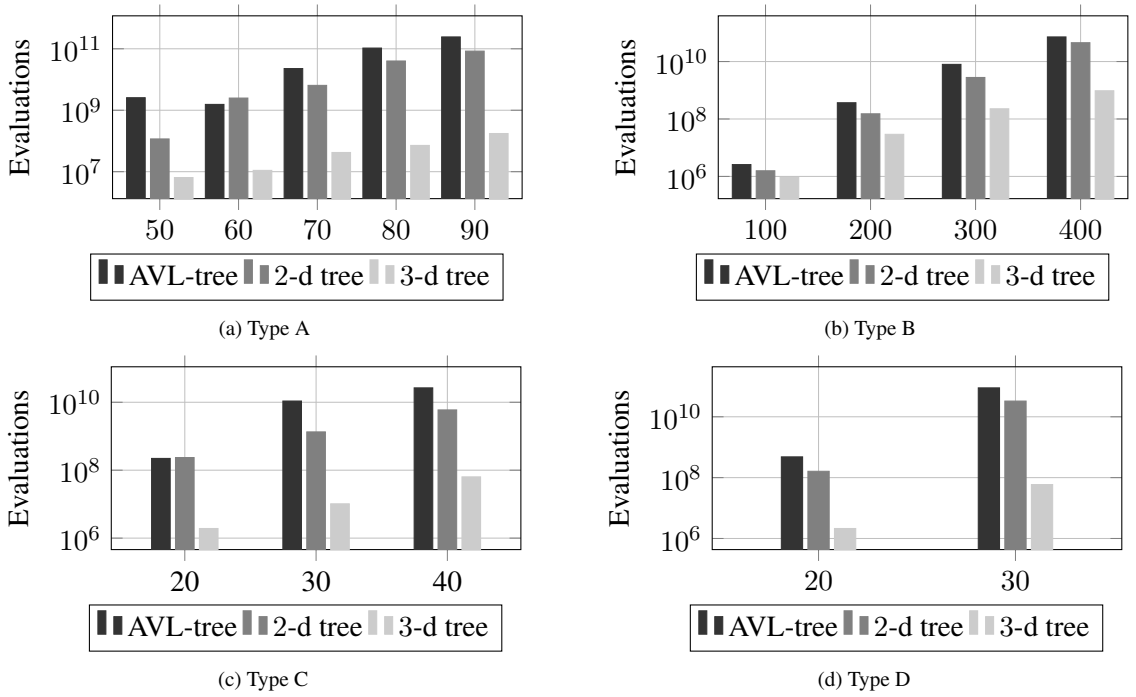


Fig. 5: Average number of solution evaluations for 3-objective instances.

Table 2 shows results on 3-objective instances for which were used AVL tree, 2-d tree and 3-d tree. Fig. 5 presents the number of solutions evaluations for 3-objective cases. Each presented value is the average for 10 instances.

The use of multi-dimensional indexing for all 3-objective cases increased the performance of the algorithm with 3-d tree outperforming 2-d tree on types A, C and D. The use of k-d tree still had lower performance on type B.

6. Conclusions and future remarks

This paper shows the application of a multi-dimensional indexing structure for exactly solving the MOKP with a dynamic programming algorithm and investigating its efficiency. Through computational experiments we showed that multi-dimensional indexing is applicable to the problem requiring considerably less solution evaluations, especially on hard instances, which resulted in a algorithm speedup 2.3 for bi-dimensional cases and up to 15.5 on 3-dimensional cases. The multi-dimensional indexing was not efficient on instances for which the set of solutions is relatively small.

A promising line of future research is to investigate the performance of multi-dimensional indexing on heuristic and approximate approaches for the MOKP as well as others multi-objective problems with the same requirement of handling a high number of multi-dimensional intermediary states.