Marcos Daniel V. Baroni

A Hybrid Heuristic for the Multi-objective Knapsack Problem

Vitória - Espírito Santo - Brasil
November 27, 2017

Marcos Daniel V. Baroni

A Hybrid Heuristic for the Multi-objective Knapsack Problem

Doctoral Thesis Proposal presented according to the regiment of the Programa de Pós-graduação em Informática of the Universidade Federal do Espírito Santo.

Universidade Feredal do Espírito Santo – UFES
Departamento de Informática
Programa de Pós-Graduação em Informática

Advisor: Dr. Flávio Miguel Varejão

Vitória - Espírito Santo - Brasil
November 27, 2017

Marcos Daniel V. Baroni

A Hybrid Heuristic for the Multi-objective Knapsack Problem

Doctoral Thesis Proposal presented according to the regiment of the Programa de Pós-graduação em Informática of the Universidade Federal do Espírito Santo.

Work approved. Vitória - Espírito Santo - Brasil, November 27, 2017:

Dr. Flávio Miguel Varejão
Advisor

Dr.ª Simone de Lima Martins
Guest

Dr. Arlindo Gomes de Alvarenga
Guest

Vitória - Espírito Santo - Brasil
November 27, 2017

Abstract

Many real applications like project selection, capital budgeting and cutting stock involves optimizing multiple objectives that are usually conflicting and can be modelled as a multi-objective knapsack problem (MOKP). Unlike the single-objective case, the MOKP is considered a NP-Hard problem with considerable intractability. This work propose a hybrid heuristic for the MOKP based on the shuffled complex evolution algorithm. A multi-dimensional indexing strategy for handling large amount of intermediate solutions are proposed as an optimization, which yields considerable efficiency, especially on cases with more than two objectives. A series of computational experiments show the applicability of the proposal to several types of instances.

Keywords: Multi-objective Knapsack Problem, Metaheuristic, Shuffled Complex Evolution, Multi-dimensional indexing

Contents

1 Introduction

In opposition to single objective optimization, multi-objective problems involve optimizing multiple criteria that are usually conflicting. These problems has typically no optimal solution, i.e., one that is best for all the objectives, but the solutions of interest – called efficient solutions – are those such that any solution which is better on one objective is necessarily worse on at least one other objective. These set of efficient solutions is called *Pareto optimal set*.

One of the most important multi-objective problem is the multi-dimensional knapsack problem (MOKP). Many real applications like project selection (**??**), capital budgeting (**??**), cargo loading (**??**) and low shop scheduling (**??**) can be modeled as MOKP. The MOKP is considered a $\mathcal{NP}$-Hard problem since it is a generalization of the well-known $0-1$ knapsack problem. This work proposes the development of a hybrid heuristic algorithm for solving the MOKP.

1.1 Motivation

Several exact approaches have been proposed in the literature for solving the MOKP. However, no exact method has proved to be effective for large multi-objective problems with more than two objectives. Even for the bi-objective case, some medium sized instances has shown to be hard to solve exactly. This reason has motivated the development of heuristic methods, which allow the approximations of the Pareto optimal solutions in a reasonable computational time.

The heuristic proposed in this work will be based on an evolutionary algorithm called shuffled complex evolution (SCE) which combines the ideas of a controlled random search with the concepts of competitive evolution and shuffling. Several applications of the SCE have been proposed for successfully solving optimization problems, among then, the multi-dimensional knapsack problem (MKP), which is also a contribution of this work. As a performance improvement for the approach, a multi-dimensional indexing strategy will be used for handling the large amount of solutions. This indexing strategy has already presented considerable performance improvement when applied to an exact algorithm for the MOKP (**??**) and is a contribution of this work as well.

1.2 Contributions and Publications

The thesis consists of the following contributions:

1. **Efficient indexing strategy for MOKP solutions:** a multi-dimensional indexing strategy for handling large amount of solutions for the MOKP. Several computational experiments show the applicability and efficiency of the strategy on a dynamic programming algorithm for exactly solving the problem. The approach considerably reduces the number of solution comparisons which resulted in an algorithm speedup of 2.3 for bi-dimensional cases and up to 15.5 for 3-dimensional cases. Publication:

    - BARONI, M. D. V; VAREJÃO, F. M. Multi-dimensional indexing on dynamic programming for multi-objective knapsack problem. *International Transactions in Operational Research*, Wiley Online Library, 2017, Submitted.

2. **A SCE algorithm for the MKP:** a shuffled complex evolution algorithm using a problem reduction technique for heuristically solving the multi-dimensional knapsack problem. The approach demanded small amount of time (less than 2 seconds) for solving the hardest instances from literature, although achieving an average solution quality of 99.0% of the best known solution. Publications:

    - BARONI, M. D. V; VAREJÃO, F. M. A shuffled complex evolution algorithm for the multidimensional knapsack problem. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications.* [S.l.]: Springer, 2015. p. 768-775.
    - BARONI, M. D. V; VAREJÃO, F. M. A shuffled complex evolution algorithm for the multidimensional knapsack problem using core concept. In: IEEE. *Evolutionary Computation (CEC), 2016 IEEE Congress on.* [S.l.], 2016 p. 2718-2723.

3. **Hybrid heuristic for MOKP:** a shuffled complex evolution algorithm for heuristically solving the MOKP, which uses the multi-dimensional indexing strategy for efficiently handling solutions. Publication:

    - BARONI, M. D. V; VAREJÃO, F. M. An efficient shuffled complex evolution algorithm for the multi-objective knapsack problem. In: IEEE *Evolutionary Computation (CEC), 2018 IEEE Congress on.* [S.l.], 2018, In Preparation

1.3 Structure

This work is structured as following. Chapter **??** presents the multi-objective knapsack problem and briefly describes its existing literature. Chapter **??** presents the multi-dimensional indexing for MOKP solutions and its use case application on a state of art exact algorithm. Chapter **??** introduces the SCE meta-heuristic and its use case for solving the multi-dimensional knapsack problem. Chapter **??** presents the proposal for the development of a SCE hybrid algorithm for heuristically the MOKP.

2 The Multi-objective Knapsack Problem

2.1 Problem Definition

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

$$\begin{aligned} f_j(\boldsymbol{x}) &= \sum_{i \in \boldsymbol{x}} p_i^j \quad j = 1, \ldots, m \\ w(\boldsymbol{x}) &= \sum_{i \in \boldsymbol{x}} w_i \end{aligned}$$

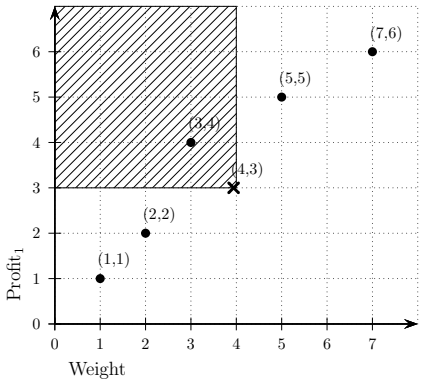


Figure 1: A knapsack-dominated solution.

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:

$$dom_k(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} dom(\boldsymbol{x}, \boldsymbol{y}) & \text{and} \\ w(\boldsymbol{x}) \leqslant w(\boldsymbol{y}) \end{cases} \tag{2.1}$$

The concept of knapsack dominance was proposed by Weingartner and Ness (**??**). Figure **??** 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 widely used and is the basis of the exact algorithm approached in Chapter **??**.

2.2 Literature Review

Several exact approaches have been proposed in the literature for solving the MOKP. Examples of such approaches are the theoretical work on (**??**) where several dynamic programming formulations are presented, a ε-constraint method presented in (**??**), the two-phase method including a branch and bound algorithm proposed in (**??**), a labeling method presented in (**??**) and the dynamic programming algorithm proposed in (**??**) which is the most efficient exact algorithm according to literature. Some later contributions

for the dynamic programming approach are an algorithmic improvement for bi-objective cases (**??**), techniques for reducing its memory usage (**??**) and a multi-dimensional solution indexing strategy for performance improvement (**??**).

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 in the size of the instance (**??**), which motivates the use of heuristics.

Most of proposed multi-objective metaheuristic are adaptations of metaheuristics originally proposed for single-objective cases. Among these we may mention a simulated annealing algorithm (**??**), a scatter search based method (**????**), a tabu search-based method (**??**), a quantum inspired artificial immune system (**??**), a hybrid genetic algorithm (**??**) and an estimation of distribution method (**??**).

One of the most popular heuristic algorithm for the MOKP is the nondominated sorting genetic algorithm (NSGA-II) proposed in (**??**). This algorithm uses sorts based on the dominance concept to obtain the convergence towards the Pareto optimal set. It also employs the crowding distance to maintain the diversity of the solution set.

Another common heuristic algorithm is the Strength Pareto Evolutionary Algorithm (SPEA-II), proposed in (**??**). SPEA-II is a genetic algorithm that, in contrast to NSGA-II, is based on the use of a external population called archive. This external population contains a limited number of non-dominated solutions during the optimization phase. At any iteration, the new non-dominated solutions of the population are compared to members of the archive with respect to dominance.

Recently a cooperative swarm intelligence algorithm called MOFPA has been proposed in (**??**), combing a firefly algorithm and a particle swarm optimization for cooperation on the exploration of the search space. A transfer function is used for discretization of the solutions. The results are compared with NSGA-II, SPEA-II and other 3 algorithms. The experiments showed that this novel approach out-performed the other algorithms in terms of convergence to optimal while preserving good diversity of solutions.

3 The Multi-dimensional Indexing

In this chapter we propose the use of a multi-dimensional indexing structure for handling MOKP solutions. The strategy is applied as a performance improvement on a state of art exact algorithm for the MOKP. Computational experiments verifies the efficiency of the approach.

3.1 Dominance Check as Range Search Operation

During the process of solving a MOKP or other multi-objective problems one of the main general operation is to check if a solution is dominated by another. An algorithm may even require selecting all non-dominated solutions from a large set of partial solutions. This may demand quadratic effort on total number of solution if implemented as pairwise comparison. However, if solutions are mapped into points in a multi-dimensional space, it can be deduced from equation (**??**) that this operation corresponds on checking whether a point exists in a certain region. Formally:

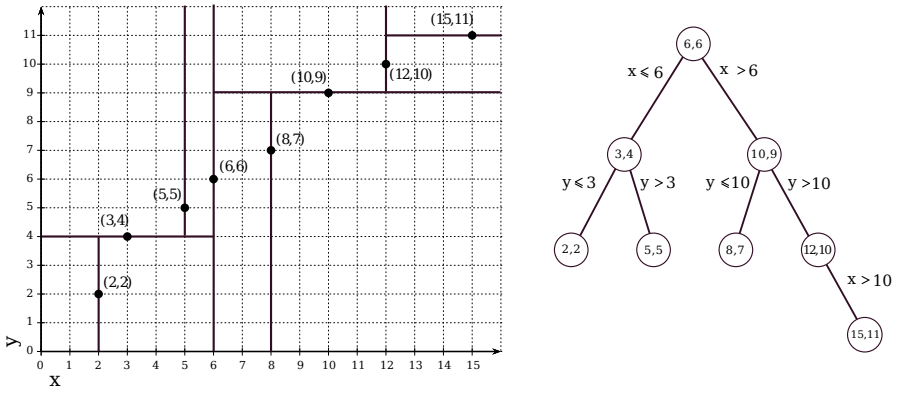
$$\text{if } \; dom_k(\boldsymbol{y}, \boldsymbol{x}) \; \text{ then } \; pnt(\boldsymbol{y}) \in R(\boldsymbol{x})$$

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 **??** 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

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

Figure **??** 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.

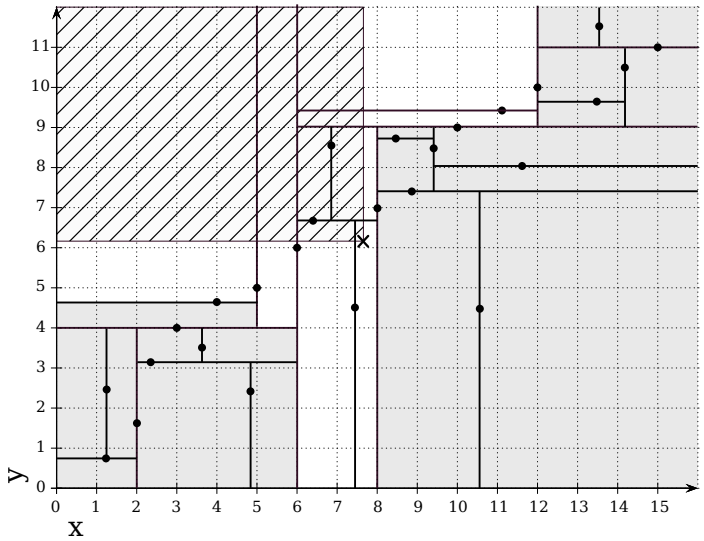


Figure 3: Example of dominance check operation using k-d tree for solution indexing.

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 (**??**).

It is expected that the use of a k-d tree assists 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 algorithm performance. The k-d tree can

also be useful in the case of querying solutions that are closest to a given one for which a nearest neighbour search may be executed.

3.2 The DP Algorithm – A Use Case

As an application of the indexing proposal we will use the k-d tree in an exact algorithm for the MOKP proposed by (**??**). The algorithm 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 first. Three optimizations will be presented next as specialization of the algorithm: reordering of items, the avoidance of deficient solutions and the elimination of unpromising partial solutions.

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 **??**.

Algorithm 1 Nemhauser and Ullmann's algorithm for MOKP

1: **function** DP($\boldsymbol{p}, \boldsymbol{w}, W$)
2: $S^0 = \left\{\emptyset\right\}$
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 **??** 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.

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.

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.

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 greedly 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 **??** for defining algorithm Algorithm **??** 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 **??** 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 (deficient solution avoidance). At line 7 the knapsack dominance and lower bound filter are applied. Any

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: $S^k_* = \left\{\boldsymbol{x} \cup \{o_k\} \;\middle|\; \boldsymbol{x} \in S^{k-1} \wedge w(\boldsymbol{x}) + w_{o_k} \leqslant W\right\}$
6: $\cup \left\{\boldsymbol{x} \;\middle|\; \boldsymbol{x} \in S^{k-1} \wedge w(\boldsymbol{x}) + w_{o_k} + \ldots + w_{o_n} > W\right\}$
7: $S^k = \left\{\boldsymbol{x} \in S^k_* \;\middle|\; (\nexists \boldsymbol{a} \in S^k_*)\Big[dom_k(\boldsymbol{a}, \boldsymbol{x}) \vee dom\Big(lb(\boldsymbol{a}), ub(\boldsymbol{x})\Big)\Big]\right\}$
8: **end for**
9: **return** S^n
10: **end function**

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 **??** still suffers from the size of efficient solutions set, especially with higher number of objectives, since the enumeration of all solutions is necessary. 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.

However, the most indicated structure seems to be a multi-dimensional one as proposed in Section **??**, which may reduces the computational complexity of the algorithm, minimizing the impact of solution growth. This performance improvement may even bring viability to previously unfeasible instances. The use of k-d tree is applied on Algorithm **??** 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.

3.2.1 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:

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

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

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

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.

Table **??** 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. **??** 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.

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:

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

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.

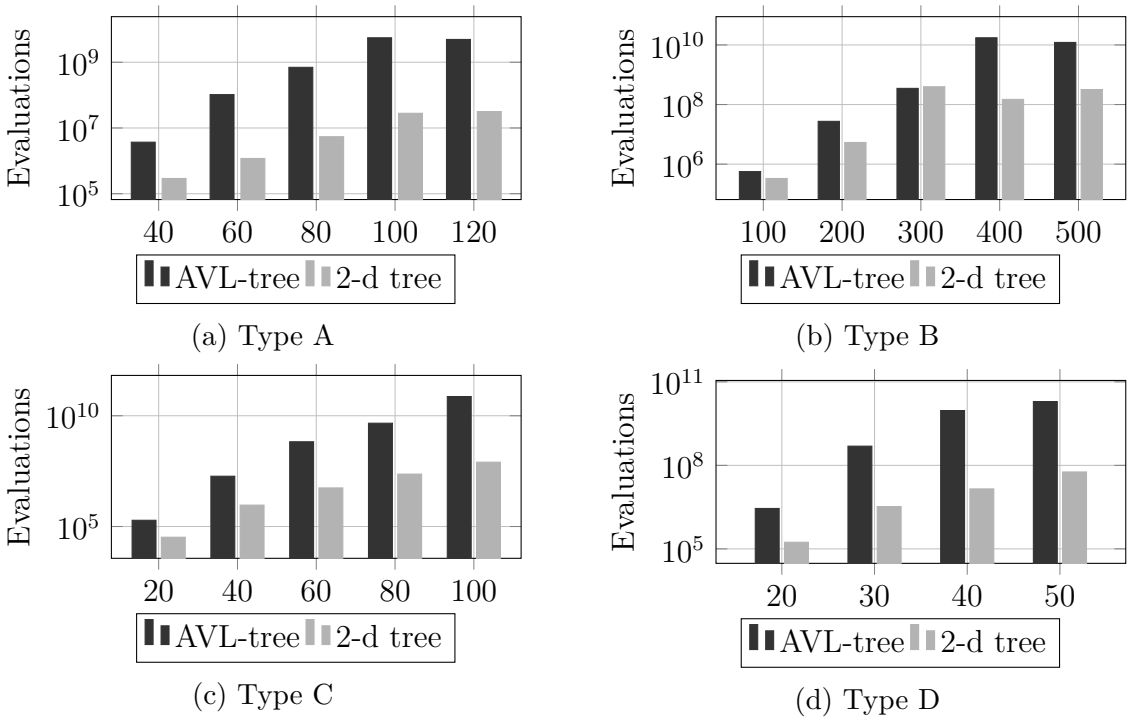


Figure 4: Average number of solution evaluations for bi-objective instances.

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	1038.0	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	2.2	**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.

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

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

D) Conflicting instances with correlated weight: $p_i^1 \in [1, 1000]$
$p_i^2 \in [1, 1001 - p_i^1]$
$p_i^3 \in [max\{900 - p_i^1 - p_i^2; 1\}, min\{1100 - p_i^1 - p_i^2, 1001 - p_i^1\}]$
$w_i \in [p_i^1 + p_i^2 + p_i^3 - 200, p_i^1 + p_i^2 + p_i^3 + 200]$.

Table **??** shows results on 3-objective instances for which were used AVL tree, 2-d tree and 3-d tree. Fig. **??** 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.

3.2.2 Conclusions

This chapter 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

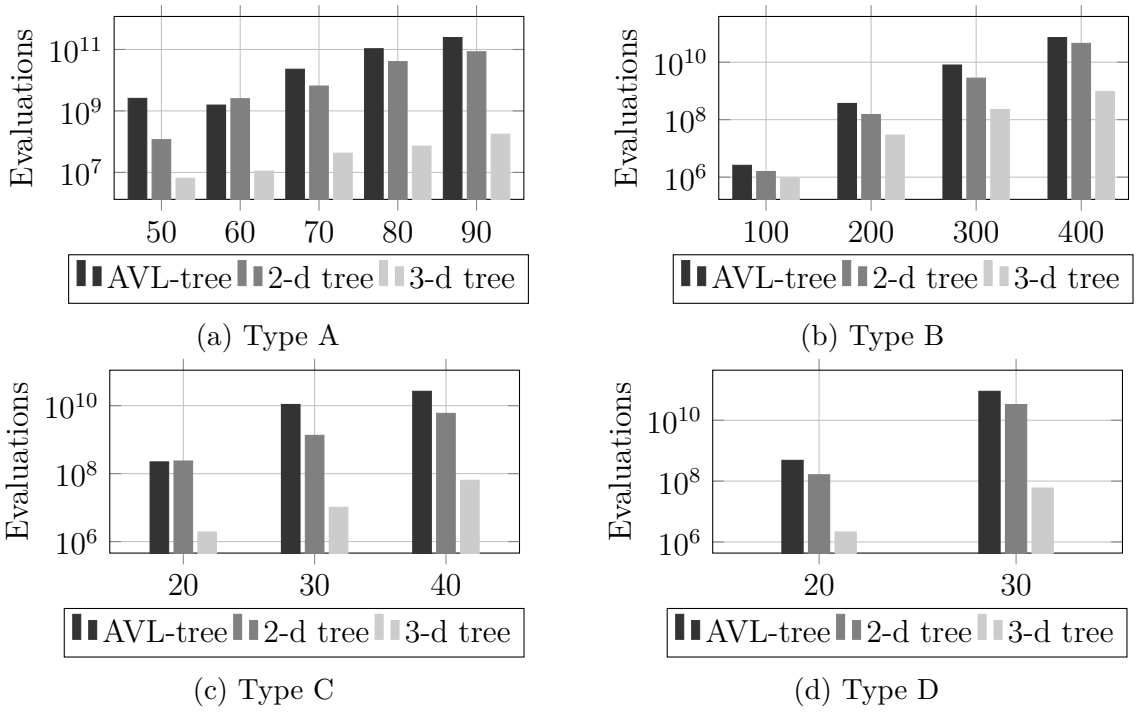


Figure 5: Average number of solution evaluations for 3-objective instances.

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 *easy* instances for which the set of solutions is relatively small.

Despite the considerable performance improvement provided by the multi-dimensional indexing strategy, several instances are still intractable due the large number of intermediate solutions, especially for 3-objective cases. For this reason this work proposes to use the indexing strategy in conjunction with an evolutionary metaheuristic which is expected to provide promising results for hard instances, particularly for many-objective instances.

The following chapter will introduce the shuffled complex algorithm and presents is application for efficiently solves the multi-dimensional knapsack problem.

4 The Shuffled Complex Evolution

The SCE is a population based evolutionary optimization algorithm proposed by Duan in (**??**) and has been successfully used on scheduling problems (**??**), project management (**??**), $0-1$ knapsack problems (**??**) and multi-dimensional knapsack problem (**????**). This chapter introduces the SCE algorithm presenting its application for solving the multi-dimensional knapsack problem with the assistance of a variable fixing procedure specific for the problem.

4.1 Definition

The SCE regards a natural evolution happening simultaneously in independent communities. The algorithm works with a population partitioned in N complexes, each one having M individuals. Initially the population of $N*M$ individuals is randomly taken from the feasible solution space. After this initialization the population is sorted by descending order according to their fitness and the best global solution is identified. The entire population is then partitioned (shuffled) into N complexes, each containing M individuals. In this shuffling process the first individual goes to the first complex, the second individual goes to the second complex, individual N goes to N-th complex, individual $(M+1)$-th goes back to the first complex, etc.

The next step after shuffling the complexes is to evolve each complex through a given fixed amount of K' steps. The individuals in each complex are sorted by descending order of fitness. In each step a subcomplex of P individuals is selected from the complex using a triangular probability distribution, where the i-th individual has a probability $p_i = \frac{2(n+1-i)}{n(n+1)}$ of being selected. The use of triangular distribution is intended to prioritize individuals with better fitness, supporting the algorithm convergence rate.

After the selection of the subcomplex, its worst individual is identified to be replaced by a new generated solution. This new solution is generated by the crossing of the worst individual and an other individual with better fitness. At first the best individual of the subcomplex is considered for the crossing. If the new solution is not better than the worst one, the best individual of the complex is considered for a crossing. If the latter crossing did not result in any improvement, the best individual of whole population is considered. Finally, if all the crossing steps couldn't generate a better individual, the worst individual of the subcomplex is replaced by a new random solution taken from the feasible solution space. This last step is important to prevent the algorithm becoming trapped in local minima. Figure **??** presents the evolving procedure described above in a flowchart diagram.

```
1: procedure New random solution
2:     v ← shuffle(1, 2, ..., n)
3:     s ← ∅                                          ▷ empty solution
4:     for i ← 1 : n  do
5:         s ← s ∪ {v_i}                              ▷ adding item
6:         if  s is not feasible then                 ▷ checking feasibility
7:             s ← s − {v_i}
8:         end if
9:     end for
10:    return s
11: end procedure
```

Figure 6: Generation of a new random solution for the MKP.

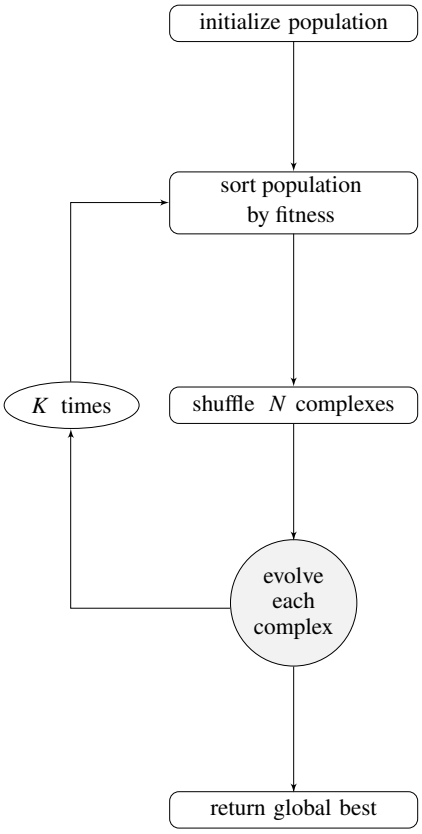


Figure 7: The shuffled complex evolution algorithm.

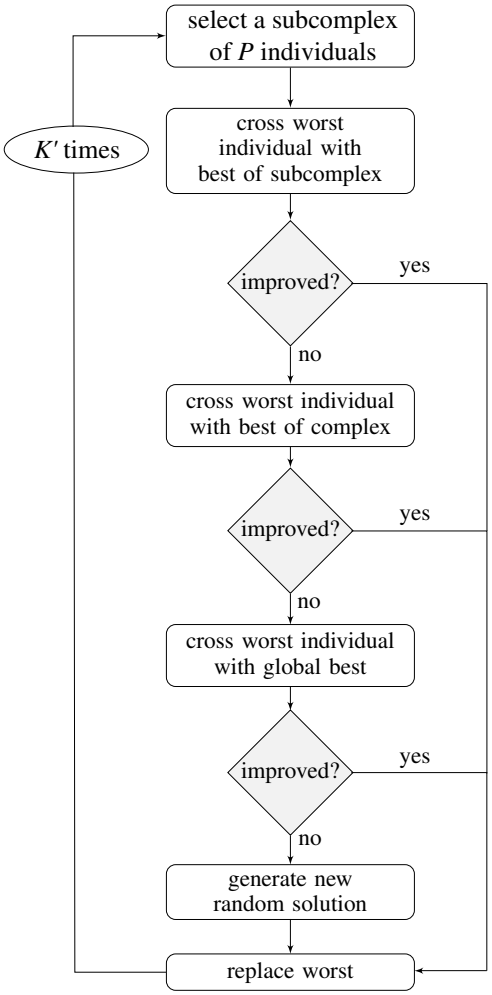


Figure 8: The evolving stage of SCE for a single complex.

After evolving all the N complexes the whole population is again sorted by descending order of fitness and the process continues until a stop condition is satisfied. Figure **??** shows the SCE algorithm in a flowchart diagram in which the stop condition is a fixed amount of K evolving steps.

4.2 A SCE for the MKP

The multi-dimensional knapsack problem (MKP) is a strongly NP-hard combinatorial optimization problem which can be viewed as a resource allocation problem and defined as follows:

$$\text{maximize} \sum_{j=1}^{n} p_j x_j \tag{4.1}$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij} x_j \leqslant c_i \quad i \in \{1, \ldots, m\} \tag{4.2}$$

$$x_j \in \{0, 1\}, \quad j \in \{1, \ldots, n\}. \tag{4.3}$$

The problem can be interpreted as a set of n items with profits p_j and a set of m resources with capacities c_i. Each item j consumes an amount w_{ij} from each resource i, if selected. The objective is to select a subset of items with maximum total profit, not exceeding the defined resource capacities. The decision variable x_j indicates if j-th item is selected. It is considered an integer programming problem (IP) since its variables x_i are restricted to be integers.

The multi-dimensional knapsack problem can be applied on budget planning scenarios and project selections (**??**), cutting stock problems (**??**), loading problems (**??**), allocation of processors and databases in distributed computer programs (**??**).

The problem is a generalization of the well-known knapsack problem (KP) in which $m = 1$. However it is a NP-hard problem significantly harder to solve in practice than the KP. Due its simple definition but challenging difficulty of solving, the MKP is often used to to verify the efficiency of novel metaheuristics.

Its well known that the hardness of a NP-hard problem grows exponentially over its size. Thereupon, a suitable approach for tackling NP-hard problems is to reduce their size through some variable fixing procedure. Despite not guaranteeing optimality of the solution, an efficient variable fixing procedure may provide near optimal solutions through a small computational effort.

As it can be noted in its description the SCE is easily applied to any optimization problem. The only steps needed to be specified is (a) the creation of a new random solution and (b) the crossing procedure of two solutions. The procedures presented in this section are based on the work in (**??**). These two procedures for the MKP are respectively presented by algorithms on Figure **??** and Figure **??**.

To construct a new random solution (Figure **??**) the items are at first shuffled in random order and stored in a list (line 2). A new empty solution is then defined (line 3). The algorithm iteratively tries to fill the solution's knapsack with the an item taken from the list (lines 4-9). The feasibility of the solution is then checked: if the item insertion let the solution unfeasible (line 6) its removed from knapsack (line 7). After trying to place all available items the new solution is returned.

```
1: procedure CROSSING(x^w : worst individual, x^b : better individual, c)
2:     v ← shuffle(1, 2, ..., n)
3:     for i ← 1 : c do
4:         j ← v_i
5:         x^w_j ← x^b_j                                   ▷ gene carriage
6:     end for
7:     if s^w is not feasible then
8:         repair s^w
9:     end if
10:    update s^w fitness
11:    return s^w
12: end procedure
```

Figure 9: Crossing procedure used on SCE algorithm.

The crossing procedure (Figure **??**) takes as input the worst solution taken from the subcomplex $x^w = (x_1^w, x_2^w, \ldots, x_n^w)$, the selected better solution $x_b = (x_1^b, x_2^b, \ldots, x_n^b)$ and the number c of genes that will be carried from the better solution. The c parameter will control how similar the worst individual will be from the given better individual. At first the items are shuffled in random order and stored in a list (line 2). Then c randomly chosen genes are carried from the better individual to the worst individual (line 5). At the end of steps the feasibility of the solution is checked (line 7) and the solution is repaired if needed. The repair stage is a greedy procedure that iteratively removes the item that less decreases the objective function. Finally the fitness of the generated solution is updated (line 10) and returned (line 11).

The following section will present the core concept for the MKP. The concept will be used as a problem reduction technique that will assist the SCE method as a pre-processing step on the algorithm. Its application on the SCE algorithm for the MKP is also a contribution of this work.

4.2.1 The Core Concept for the MKP

The core concept was first presented for the one-dimensional 0-1 knapsack problem (KP), leading to very successful KP algorithms. The main idea is to reduce the original problem by only considering a set of items for which it is hard to decide if they will occur or not in an optimal solution. This set of items is named core. The variables for all items outside the core are fixed to certain values.

The knapsack problem considers items $j = 1, \ldots, n$, associated profits p_j and associated weights w_j. A subset of these items has to be selected and packed into a knapsack having capacity c. The total profit of the items in the knapsack has to be maximized, while the total weight is not allowed to exceed c.

Before defining the core of the KP it is worth to note the solution structure of its

LP-relaxation. The LP-relaxation of an integer programming problem arises by replacing the constraint that each variable must be integer by a constraint that allows continuity of the variable. The LP-relaxation of a KP is found by replacing the constraint $x_i \in \{0, 1\}$ by $0 \leqslant x_i \leqslant 1$ for all $i \in \{1, \ldots, n\}$. If the items are sorted according to decreasing efficiency values

$$e_j = \frac{p_j}{w_j}, \tag{4.4}$$

it is known that the solution of the LP-relaxation consists of three consecutive parts: the first part contains variables set to 1, the second pare consists of at most one split item s, whose corresponding LP-values is fractional, and finally the remaining variables, which are always set to zero, form the third part.

For most instance of KP (except those with a very special structure) the integer optimal solution closely corresponds to this partitioning in the sense that it contains most of the highly efficient items of the first part, some items with medium efficiency near the split item, and almost no items with low efficiency from the third part. Items of medium efficiency constitute the core.

Balas and Zemel (**??**) gave the following precise definition of the core of a KP, based on the knowledge of an optimal integer solution x^*. Assume that the items are sorted according to decreasing efficiency and let

$$a := \min\{j | x_j^* = 0|\}, \quad b := \max\{j | x_j^* = 1\}. \tag{4.5}$$

The core is given by the items in the interval $C = \{a, \ldots, b\}$. It is obvious that the split item is always part of the core, i.e., $a < s < b$.

Figure **??** illustrates an exact core for an hypothetical KP instance with 13 items. First row represents the efficiency value e of each item. The items are sorted by descending order of efficiency. Second row represent solution array of the LP-relaxation. The third row represents the exact solution for the original problem. The 8-th item is the split item. Notice that the split item is within the exact core.

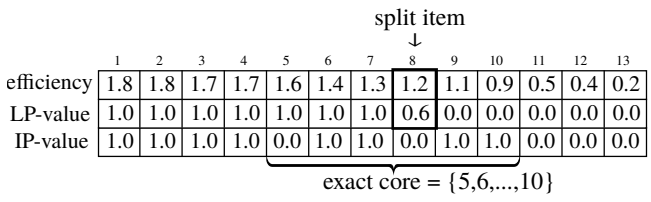


Figure 10: Example of exact core for a hypothetical KP instance.

The KP Core problem (KPC) is defined as

$$\text{maximize} \sum_{j \in C} p_j x_j + \tilde{p} \tag{4.6}$$

$$\text{subject to} \sum_{j \in C} w_j x_j \leqslant c - \tilde{w} \tag{4.7}$$

$$x_j \in \{0, 1\}, \quad j \in C. \tag{4.8}$$

with $\tilde{p} = \sum_{j=1}^{a-1} p_j$ and $\tilde{w} = \sum_{j=1}^{a-1} w_j$ respectively quantifying the total profit and the total weights of items fixed as selected. The solution of KPC would suffice to compute the optimal solution of KP, which however, has to be already partially known to determine C. Nevertheless an approximate core $C = \{s - \delta, \ldots, s + \delta\}$, of fixed size $|C| = 2\delta + 1$ is considered for a heuristic reduction of the problem.

Figure **??** exemplifies an approximate core of a hypothetical KP instance with 13 items. The first row represents the efficiency value of each item and the second row represents the value of each variable on the LP-relaxation optimal solution. The items are sorted in descending order of efficiency value. The last row illustrates the variable fixing after the defined core. Asterisks indicate free variables associated to the items in the approximate core.

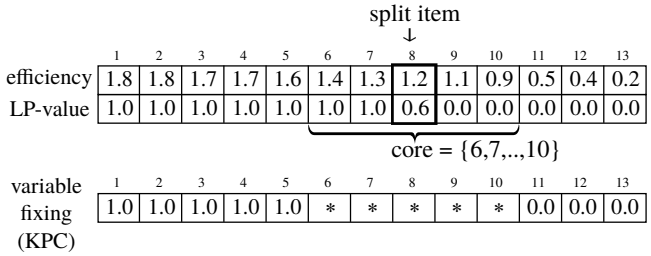


Figure 11: Example of core for a hypothetical KP instance with $n = 13$ and approximate core size of 5 ($\delta = 2$).

The previous definition of the core for KP can be extended to MKP without major difficulties, once an efficiency measure is defined for the MKP, as addressed in (**??**), even though, a proper efficiency measure for MKP is not obvious due the multidimensional weight of the items. A well accepted efficiency measure is discussed in Section **??**.

Now let x^* be an optimal solution for a MKP and assume that the items are sorted in descending order after some given efficiency measure. Then let

$$a = \min\{j | x_j^* = 0\}, \quad b = \max\{j | x_j^* = 1\}. \tag{4.9}$$

The core is given by the items in the interval $C = \{a, \ldots, b\}$, and the multidimensional

knapsack core problem (MKPC) defined as

$$\text{maximize} \sum_{j \in C} p_j x_j + \tilde{p} \tag{4.10}$$

$$\text{subject to} \sum_{j \in C} w_{ij} x_j \leqslant c_i - \tilde{w}_i, \quad i = 1, \ldots, m \tag{4.11}$$

$$x_j \in \{0, 1\}, \quad j \in C. \tag{4.12}$$

with $\tilde{p} = \sum_{j=1}^{a-1} p_j$ and $\tilde{w}_i = \sum_{j=1}^{a-1} w_{ij}, i = 1, \ldots, m$.

In contrast to KP, the solution of the LP-relaxation of MKP in general does not consists of a single fractional split item. But up to m fractional values give rise to a whole *split interval* $S = \{s_1, \ldots, s_m\}$, where s_1 and s_m are respectively the first and the last index of variables with fractional values after sorting by the given efficiency measure. Once the split interval is defined, a central index value $s = \lfloor \frac{s_1 + s_m}{2} \rfloor$ can be used as the center of an approximate core.

For defining a reasonable efficiency measure for MKP consider the most obvious form of efficiency which is a direct generalization of the one-dimensional case:

$$e_j(simple) = \frac{p_j}{\sum_{i=1}^{m} w_{ij}} \tag{4.13}$$

In this definition different orders of magnitude of the constraints are not considered and a single constraint may dominate the others. This drawback can be avoided by introducing relevance values r_i for every constraint:

$$e_j(relevance) = \frac{p_j}{\sum_{i=1}^{m} r_i w_{ij}} \tag{4.14}$$

Several proposals for setting the relevance values r_i were discussed and tested by Puchinger, Raidl and Pferschy in (**??**). According to their work setting the relevance values r_i to the values of an optimal solution to the dual problem of the MKP's LP-relaxation, as suggested in (**??**), achieved the best results and will be the one considered in this work for the development of the hybrid heuristic.

While the original MKP can be seen as a resource allocation problem, the dual problem of the MKP can be seen as a resource valuation problem. For this reason the values of the dual solution is related to the *importance* of each resource.

The split interval resulting from the efficient measure considered can be precisely characterized. Let x^{LP} be the optimal solution of the LP-relaxation of MKP. Then the following relation holds, as proved in (**??**):

$$x_l^{LP} = \begin{cases} 1 & \text{if } e_j > 1, \\ \in [0, 1] & \text{if } e_j = 1, \\ 0 & \text{if } e_j < 1. \end{cases}$$

From this relation we can note that all variables in the split interval will have efficiency measure $e_j = 1$, while less and more efficient ones will have $e_j < 1$ and $e_j > 1$ respectively.

Figure **??** exemplifies an approximate core of a hypothetical MKP with 13 variables and 3 dimensions. Notice that the LP-relaxation solution has now 3 fractional variables that defines the center of the split interval.

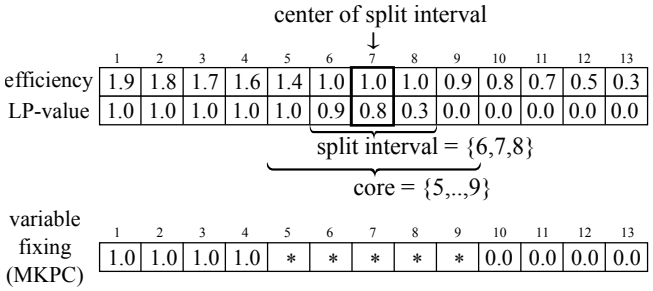


Figure 12: Example of core for a hypothetical MKP instance with $n = 13$, $m = 3$ and approximate core size of 5 ($\delta = 2$).

The core will be applied as a problem reduction procedure for fixing a given number of variables of the problem, before being inputted to the SCE algorithm. The following section presents computational experiments for evaluating the approach.

4.2.2 Computational Experiments

To verify the efficiency of the proposed metaheuristic, several computational experiments was executed considering the original SCE for the MKP without using the problem reduction procedure – as proposed in (**??**) – and the SCE with the reduction procedure. For brevity the SCE with the reduction procedure will be referred to as SCEcr .

For the experiments, two sets of MKP instances was used: a first set composed by 270 instances provided by Chu and Beasley ((**??**)) and a second set composed by 11 instances provided by Glover and Kochenberger in (**??**). These instances are all available at (**??**).

The best known solution for all instances were taken from the literature and were found by different algorithms which we had no access to the implementation. In mosts cases those are exact algorithms which took minutes or hours of execution time.

The set of MKP instances provided by Chu and Beasley was generated using a procedure suggested by Freville and Plateau (**??**), which attempts to generate instances hard to solve. The number of constraints m varies among 5, 10 and 30, and the number of variables n varies among 100, 250 and 500.

The w_{ij} were integer numbers drawn from the discrete uniform distribution $U(0, 1000)$. The capacity coefficient c_i were set using $b_i = \alpha \sum_{j=1}^{n} w_{ij}$ where α is a tightness ratio and varies among 0.25, 0.5 and 0.75. For each combination of (m, n, α) parameters, 10 random problems was generated, totaling 270 problems. The profit p_j of the items were correlated

to w_{ij} and generated as follows:

$$p_j = \sum_{i=1}^{m} \frac{w_{ij}}{m} + 500q_j \qquad j = 1, \ldots, n \tag{4.15}$$

All the experiments were run on a Intel^R Core i5-3570 CPU @3.40GHz computer with 4GB of RAM. The original SCE and SCEcr was both implemented in C programming language.

For the variable fixing procedure used on SCEcr , the range size of the approximate core was $|C| = m + \frac{n}{10}$ for all instances. In all instances the parameters used for SCE and SCEcr were the same recommended in (**??**) which was found after a batch test using Chu and Beasley instances:

- $N = 20$: number of complexes;
- $M = 20$: number of individuals in each complex;
- $P = 5$: number of individuals in each subcomplex;
- $K = 300$: number of algorithm iterations;
- $K' = 20$: number of iterations used in the complex evolving process;
- $c = n/5$: number of genes carried from parent in crossing process.

Table **??** shows the performance of the SCE and SCEcr on the Chu-Beasley set of instances. Each instance in the set was executed 10 times for each algorithm. Columns n, m and α shows the parameters used on each instance generation. The *time* column shows the average execution time of the algorithms (lower is better). The *quality* column shows the average ratio of the solution found and the best known solution from literature ((**????**)) of each instance (higher is better). Best values are in bold.

It can be observed that the SCEcr had faster convergence speed, achieving higher quality solutions in all cases, achieving at least 97.96% of best known, in less than 2 seconds for every instance.

It can be also noticed that SCEcr executed in much less processing time than original SCE. This is due the variable fixing procedure which reduced the problem size, resulting in less genes operations during the evolving procedures. The variable fixing procedure also brought robustness for the method, as the quality of the solution found increased in case of larger instances while on original SCE the quality decreased considerably.

Table **??** shows the performance of SCE and SCEcr on the Glover-Kochenberger set of instances. Each instance in the set was executed 10 times for each algorithm. Columns n and m indicate the size of each instance. The *time* column shows the average execution (lower is better). The *quality* column shows the average ratio of the solution found and the best known solution of each instance. Best values are in bold. It can be noticed that

SCEcr achieved high quality solutions, at least 98.22% of best known solution, spending small amount of processing time, compared to the time taken to find the best known solutions.

n	**m**	α	**time** (s)		**quality**(%)	
			SCE	**SCEcr**	**SCE**	**SCEcr**
100	5	0.25	1.22	**0.17**	96.51	**99.73**
		0.50	1.34	**0.18**	97.42	**99.86**
		0.75	1.37	**0.17**	98.87	**99.91**
	10	0.25	1.32	**0.25**	95.68	**99.53**
		0.50	1.51	**0.25**	96.65	**99.76**
		0.75	1.46	**0.27**	98.54	**99.96**
	30	0.25	1.74	**1.20**	95.38	**97.96**
		0.50	1.79	**0.89**	96.41	**99.18**
		0.75	1.72	**0.95**	98.18	**99.52**
250	5	0.25	2.87	**0.69**	93.22	**99.86**
		0.50	2.82	**0.70**	94.88	**99.94**
		0.75	2.93	**0.69**	97.57	**99.96**
	10	0.25	3.08	**0.87**	93.14	**99.58**
		0.50	3.03	**0.79**	94.55	**99.79**
		0.75	3.12	**0.84**	97.16	**99.88**
	30	0.25	3.74	**1.52**	93.10	**98.42**
		0.50	3.74	**1.36**	94.20	**99.33**
		0.75	3.99	**1.48**	96.64	**99.59**
500	5	0.25	5.62	**1.25**	91.37	**99.77**
		0.50	5.72	**1.24**	93.39	**99.88**
		0.75	5.88	**1.20**	96.42	**99.92**
	10	0.25	5.97	**1.41**	91.62	**99.51**
		0.50	6.11	**1.36**	93.09	**99.77**
		0.75	5.47	**1.21**	96.24	**99.84**
	30	0.25	6.20	**1.96**	91.37	**98.76**
		0.50	6.26	**1.82**	92.56	**99.42**
		0.75	6.05	**1.73**	95.97	**99.67**

Table 3: SCE and SCEcr performance on Chu-Beasley problems.

#	**n**	**m**	**time**(s)		**quality**(%)	
			SCE	**SCEcr**	**SCE**	**SCEcr**
01	100	15	1.47	**0.08**	97.66	**99.24**
02	100	25	1.61	**0.09**	97.94	**98.94**
03	150	25	2.51	**0.09**	97.22	**99.09**
04	150	50	3.56	**0.09**	97.40	**98.52**
05	200	25	3.55	**0.09**	96.88	**99.28**
06	200	50	4.81	**0.10**	97.68	**98.90**
07	500	25	7.30	**0.10**	97.12	**99.54**
08	500	50	12.20	**0.11**	97.27	**99.33**
09	1500	25	24.61	**0.12**	95.40	**98.22**
10	1500	50	33.79	**0.13**	97.50	**99.64**
11	2500	100	121.28	**0.15**	97.95	**99.70**

Table 4: SCEcr performance on Glover-Kochenberger problems.

4.2.3 Conclusions

The SCE algorithm, which combines the ideas of a controlled random search with the concepts of competitive evolution, proved to be able to achieve fast convergence ratio, finding good quality near optimal solutions, demanding small amount of computational time.

The application of the core concept as a variable fixing procedure for MKP proved to be efficient to reduce the size of the problems which provided fast execution time, producing higher quality solutions.

SCEcr algorithm presented faster convergence speed, achieving higher quality solutions in all cases, achieving at least 99.02% of best known, in less than 2 seconds for every instance. The variable fixing procedure also brought robustness for the method, as the quality of the solution found increased in case of larger instances.

The heuristic presented in this chapter is very well suitable if it is necessary to compute a good solution in a small processing time. The heuristic could achieve 99.61% on average of quality of the best known solution for the 270 Chu-Beasley instances and 99.46% on average for the Glover-Kochenberger instances.

5 A SCE for the MOKP

As exposed in previous chapters, exact methods have difficulties solving several MOKP instances, especially with more than 2 objectives. This motivates the development of heuristic methods. This chapter will discuss the development of the proposed heuristic algorithm, how tests will be conducted and analyzed. A final schedule is presented in the last section.

5.1 Algorithm Proposal

For designing a multi-objective heuristic, one still take into account two important criteria: (a) how to allow the diversity of solutions and (b) how to ensure the convergence towards the Pareto optimal set. The SCE algorithm (a) allows diversity of solutions by evolving independent sets of solutions which are constructed by shuffling the population and (b) ensures convergence by prioritizing the crossing of individuals with better fitness, which indicates the potential to compute a good approximate Pareto optimal set.

As seen in previous sections the SCE is easily applied to any optimization problem. The fact that the MKP shares the same solution representation as the MOKP – set of items included in the knapsack – allow us to use the same procedures presented in Section **??** for (a) creating a new random solution and (b) crossing two solutions.

A crucial point for the successful application of a metaheuristic on a multi-objective problem is the definition of the fitness function. One of the most popular multi-objective heuristic algorithms successfully uses sorts based on the dominance relation for measuring the quality of a solution. The method is called *non-dominated sort* (**??**): given a set Q of solution, it first selects those which are not dominated by any other solution in Q to compose the set S_1, denoted as *first non-dominated front*. Now, all select solutions are removed from Q set and the process is repeated to make set S_2. The process continues until all fronts are identified. Solutions in the first fronts are considered the ones with the best fitness. For tie breaking the aggregation method can be used, which consist of using a weight vector to compute a scalar over the multiple objective values of a solution.

A performance issue on the non-dominated sort procedure is checking if a solutions is dominated by another. Our propose is to minimize this issue with the assistance of the indexing strategy proposed in Chapter **??**.

5.2 Computational Experiments and Analyzes

The proposed algorithm will be tested on the same types of instances presented in Chapter **??**, this time considering 2, 3 and 4 objective cases and compared with the results that will be obtained from the MOFPA proposed in (**??**).

Two performance metrics will be used to evaluate the approximation quality of the generated solutions: (a) set coverage metric (**??**) and (b) spacing metric (**??**):

(a) Set coverage metric (SC): This metric is intended to be used for comparing two non-dominated Pareto sets. It defines a semi-distance between two solutions set A and B by:

$$SC(A,B) = \frac{\left|\{b \in B \mid \exists a \in A : dom(a,b)\}\right|}{|B|}$$

This quantity corresponds to the proportion of the solutions of the set B, that are dominated by at least one solution of A. It should be noted that $C(A,B)$ is not necessarily equal to $1 - C(B,A)$. To compare A and B sets, both values of $C(A,B)$ and $C(B,A)$ must be calculated.

(b) Spacing metric (SP): This metric evaluates the uniformity of the distribution of non-dominated solutions:

$$SP(A) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\bar{d} - d_i)^2}$$

where d_i is defined as:

$$d_i = min \sum_{m=1}^{M} |f_m^i - f_m^k| \qquad k \in A, k \neq i$$

and $\bar{d}$ denotes the average of the distances d_i for $i = 1, \ldots, n$. SP represents the standard deviation of distance values between two consecutive solutions of the set A. The smaller SP, the better distribution of the solutions. Moreover, a null value of SP indicates that the solutions are equidistant.

The computational results will be reported on a paper to be submitted to IEEE Congress on Evolutionary Computation 2018 (CEC 2018) that will be held on July 80-14 in Rio de Janeiro, Brazil. Its paper submission deadline is January 15.

5.3 Final Doctoral Schedule

Table **??** presents the final doctoral schedule.

Activities	**Weeks**										
	November			**December**				**January**			
	2º	3º	4º	1º	2º	3º	4º	1º	2º	3º	4º
Literature review	●	●	●								
Implementation and adjustments	●	●	●	●							
Computational experiments			●	●	●						
CEC Paper writing						●	●	●			
CEC Paper submission									●		
Thesis writing				●	●	●	●	●	●	●	

Table 5: Final doctoral schedule.