



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH



UNIVERSITAT DE
BARCELONA



UNIVERSITAT
ROVIRA i VIRGILI

Facultat d'Informàtica de Barcelona

Facultat de
Matemàtiques

Escola Tècnica
Superior d'Enginyeria

An Evolutionary Algorithm for Solving the Two-Dimensional Irregular Shape Packing Problem Combined with the Knapsack Problem

Albert Espín Román

Advisor: René Alquézar Mancho
Department of Computer Science, UPC

Master Thesis
Master in Artificial Intelligence

January 31, 2020

Abstract

The Packing Problem and the Knapsack Problem are two well-known optimization problems focused on placing items in a container. In this work, a Joint Problem was defined to combine the geometric shapes associated to the items and the container in the Packing Problem, with the value and weight notions of the Knapsack Problem. In the Joint Problem, the objective is the maximization of the value sum of items placed in the container, respecting a maximum weight constraint imposed by the container, and without causing any geometric intersection. In particular, this work is focused on the two-dimensional version of the Joint Problem, with the potential applicability of refining the way in which objectives and constraints are expressed when packing and cutting objects in the textile and furniture industries, among others. Motivated by the ability of evolutionary algorithms to obtain high-quality results for both the Packing Problem and the Knapsack Problem in the literature, a novel evolutionary algorithm was designed for the Joint Problem. The method is characterized by a mutation operator that can add items to the container, move or rotate placed items using multiple strategies, and remove them on certain occasions. A crossover strategy was also designed, but it did not lead to better solutions; it swaps in the offspring the items located in randomly determined regions of the container of the progenitors. The elitism strategy is used to preserve the best individuals, while parent selection and population update employ the tournament mechanism to favour the survival of the fittest chromosomes, where the fitness is equivalent to the value of the items in the container. The evolutionary algorithm was compared with two other methods: a greedy approach that prioritizes profitable placements, and a reversible algorithm that can occasionally remove items, modify placements and revert a solution to a previous state. The three methods were tested on the Joint Problem Dataset, a new collection of problems with manually obtained optimal solutions, to asses the solution quality of the algorithms, as well as their computational efficiency. The evolutionary algorithm significantly outperformed the other methods in terms of solution quality, obtaining optimal solutions in 7 of the 10 problems, and close-to-optimal solutions in other 2, although the greedy and reversible algorithms proved to be much faster. The implemented algorithms were also tested in an existing variant of the Packing Problem that aims to maximize the number of items placed in the container. It was observed that the evolutionary algorithm was the most effective of the three for this task too, obtaining optimal solutions in 5 of the 9 tested problems, without any modification of its original formulation, devised for the Joint Problem. The flexibility of the evolutionary algorithm leads to think that it would be possible to adapt it to solve the three-dimensional version of the Joint Problem, applicable in logistics to improve the optimization and prioritization of transportation processes.

Keywords: Packing Problem, Irregular Shape Packing Problem, Two-Dimensional Packing Problem, Knapsack Problem, 0/1 Knapsack Problem, Packing Knapsack Problem, Knapsack Packing Problem, Joint Problem, Optimization, Evolutionary algorithm, Genetic algorithm

Acknowledgements

I would like to thank Professor René Alquézar for his suggestions and the insightful discussions.

Contents

1	Introduction	10
2	Previous Work	13
2.1	Evolutionary algorithms to solve the 0/1 Knapsack Problem	13
2.2	Evolutionary algorithms to solve the Two-Dimensional Irregular Shape Packing Problem	13
2.3	Combination of discrete and continuous variables in evolutionary algorithms	15
3	Proposed Methods	17
3.1	Greedy algorithm	17
3.2	Reversible algorithm	18
3.3	Evolutionary algorithm	20
3.3.1	Chromosome representation	20
3.3.2	Fitness function	21
3.3.3	Solution feasibility	22
3.3.4	Generation of the initial population	23
3.3.5	Parent selection and offspring generation	25
3.3.6	Crossover	25
3.3.7	Mutation	31
3.3.8	Population update	33
3.3.9	Termination criteria	33
4	Implementation	37
4.1	Technology and geometric checks	37
4.2	Visualization	38
4.3	Code availability	41
5	Joint Problem Dataset	42
5.1	Context, Goals and Design Principles	42
5.2	Description and Analysis of the problems	44
6	Parameter Configuration and Optimization	57
6.1	Motivation and Goals	57
6.2	Greedy algorithm	57
6.2.1	Experimental Methodology	57
6.2.2	Results and Discussion	58
6.3	Reversible algorithm	60
6.3.1	Experimental Methodology	60
6.3.2	Results and Discussion	61
6.4	Evolutionary algorithm	62
6.4.1	Experimental Methodology	62
6.4.1.1	General comments	62
6.4.1.2	Generation of the initial population	63
6.4.1.3	Parent selection and offspring generation	64
6.4.1.4	Crossover	65
6.4.1.5	Mutation	65
6.4.1.6	Population update	66
6.4.1.7	Termination criteria	66

6.4.2	Results and Discussion	67
7	Experimental Comparative Analysis of the Proposed Methods with the Joint Problem Dataset	70
7.1	Experimental Methodology	70
7.2	Results and Discussion	70
7.2.1	Solution Quality Analysis	70
7.2.2	Time Analysis	73
8	Experimental Analysis of the Applicability of the Proposed Methods to solve a traditional Packing Problem	78
8.1	Contextualization and Experimental Methodology	78
8.2	Analysis of the Packing Problem Dataset	79
8.3	Results and Discussion	79
9	Conclusions and Future Work	85
9.1	Main Conclusions	85
9.2	Possible modifications for the evolutionary algorithm applied to the Joint Problem	86
9.2.1	Multi-item placement specialization in the generation of the initial population	86
9.2.2	Nested movement and rotation of compound polygons and the items placed in their holes	87
9.2.3	Self-evolution of parameters	88
9.3	Possible modifications for the evolutionary algorithm to apply it to other problems	88
9.3.1	Area Minimization Packing Problem	88
9.3.2	Three-Dimensional Irregular Shape Packing Problem combined with the Knapsack Problem	89
9.4	The Optimality Goal	90
A	Appendix	96
A.1	Pseudo-code of the reversible algorithm	96
A.2	List of parameters of the evolutionary algorithm	99
A.3	Secondary tables of Parameter Optimization	100
A.3.1	Greedy algorithm	100
A.3.2	Reversible algorithm	103
A.3.3	Evolutionary algorithm	105
A.4	Visualization of the best solutions of the evolutionary algorithm for the Joint Problem Dataset	107
A.5	Visualization of the best solutions of the algorithms for the Packing Problem Dataset	112

List of Algorithms

Greedy algorithm	17
1 Greedy algorithm	19
Evolutionary algorithm	20
2 Initial population generation	24
3 Parent selection	26
4 Offspring generation	27
5 Crossover	30
6 Mutation	34
7 Surviving population selection	35
8 Evolutionary algorithm	36
Reversible algorithm	96
9 Reversible algorithm	96

List of Figures

Implementation	37
1 Example problem in its initial state.	39
2 Possible solution for an example problem.	40
Joint Problem Dataset	42
3 Initial state of Problem 1.	45
4 Optimal solution of Problem 1.	45
5 Initial state of Problem 2.	46
6 Optimal solution of Problem 2.	46
7 Initial state of Problem 3.	47
8 Optimal solution of Problem 3.	47
9 Initial state of Problem 4.	48
10 Optimal solution of Problem 4.	48
11 Initial state of Problem 5.	49
12 Optimal solution of Problem 5.	49
13 Initial state of Problem 6.	50
14 Optimal solution of Problem 6.	50
15 Initial state of Problem 7.	51
16 Optimal solution of Problem 7.	51
17 Initial state of Problem 8.	52
18 Optimal solution of Problem 8.	52
19 Initial state of Problem 9.	53
20 Optimal solution of Problem 9.	54
21 Initial state of Problem 10.	54
22 Optimal solution of Problem 10.	55

Experimental Comparative Analysis of the Proposed Methods with the Joint Problem Dataset	70
23 Fitness evolution in the evolutionary algorithm.	73
24 Value evolution in the greedy algorithm.	74
25 Value evolution in the reversible algorithm.	75
26 Task time division in the evolutionary algorithm.	76
27 Task time division in the greedy algorithm.	76
28 Task time division in the reversible algorithm.	77
Experimental Analysis of the Applicability of the Proposed Methods to solve a traditional Packing Problem	78
29 Task time division in the greedy algorithm.	82
30 Task time division in the reversible algorithm.	83
Appendix	96
31 Best algorithmic solution for Problem 1.	107
32 Best algorithmic solution for Problem 2.	107
33 Best algorithmic solution for Problem 3.	108
34 Best algorithmic solution for Problem 4.	108
35 Best algorithmic solution for Problem 5.	109
36 Best algorithmic solution for Problem 6.	109
37 Best algorithmic solution for Problem 7.	110
38 Best algorithmic solution for Problem 8.	110
39 Best algorithmic solution for Problem 9.	111
40 Best algorithmic solution for Problem 10.	111
41 Best algorithmic solution for “Circles in circle”.	112
42 Best algorithmic solution for “Triangles in circle”.	112
43 Best algorithmic solution for “Squares in circle”.	113
44 Best algorithmic solution for “Circles in triangle”.	113
45 Best algorithmic solution for “Triangles in triangle”.	114
46 Best algorithmic solution for “Squares in triangle”.	114
47 Best algorithmic solution for “Circles in square”.	115
48 Best algorithmic solution for “Triangles in square”.	115
49 Best algorithmic solution for “Squares in square”.	116

List of Tables

Joint Problem Dataset	42
1 Statistics of the Joint Problem Dataset.	56
Parameter Configuration and Optimization	57
2 Solution value of the iteration configurations of the greedy algorithm.	60
3 Solution value of the iteration configurations of the reversible algorithm.	62
4 Solution value of the evolutionary algorithm variants.	69
5 Execution time of the evolutionary algorithm variants.	69

Experimental Comparative Analysis of the Proposed Methods with the Joint Problem Dataset	70
6 Solution value of algorithms in the Joint Problem Dataset.	71
7 Execution time of algorithms in the Joint Problem Dataset.	74
Experimental Analysis of the Applicability of the Proposed Methods to solve a traditional Packing Problem	78
8 Statistics of the Packing Problem Dataset.	79
9 Solution value of algorithms in the Packing Problem Dataset.	80
10 Execution time of algorithms in the Packing Problem Dataset.	82
Appendix	96
11 Solution value of the weight configurations of the greedy algorithm.	101
12 Execution time of the weight configurations of the greedy algorithm.	102
13 Execution time of the iteration configurations of the greedy algorithm.	103
14 Solution value of the probability configurations of the reversible algorithm.	103
15 Execution time of the probability configurations of the reversible algorithm.	104
16 Execution time of the iteration configurations of the reversible algorithm.	104
17 Solution value of the population size configurations of the evolutionary algorithm.	105
18 Execution time of the population size configurations of the evolutionary algorithm.	105
19 Solution value of the offspring size configurations of the evolutionary algorithm. .	106
20 Execution time of the offspring size configurations of the evolutionary algorithm.	106

1 Introduction

The Knapsack Problem and the Packing Problem are two well-known optimization problems ([BV04](#)). In both cases, different objects (also called items) must be selected to be placed in a container, among a set of possible objects. In the Knapsack Problem, each item is described by its weight and value, and the objective is to maximize the value of the objects placed inside the container, without exceeding its weight capacity. The most common version of the problem is the 0/1 Knapsack Problem ([MPT99](#)), in which each object can be added to the container just one time or none, as opposed to the Bounded Knapsack Problem ([Pis00](#)) and the Unbounded Knapsack Problem ([APR00](#)), that allow items to have, respectively, a fixed maximum or unlimited number of occurrences in the container. In the Packing Problem, both the objects and the container are defined by their geometric shape, and the goal can differ depending on the particular application, with the common constraint that objects should not overlap with each other nor with the container. In some cases, the objective is to place the items maximizing the percentage of the area (or volume) of the container covered with objects ([JG98](#)). Alternatively, fitting a certain set of objects in the smallest possible area or volume can be the intended goal ([HK13](#)). Theoretically, the objective can also be the maximization of the number of items placed in the container.

In real world situations, for instance in the field of logistics, it can be very useful to consider all the characteristics of the two problems (value, weight and shape of the objects and the container) in a single problem. The following example illustrates a feasible scenario in which the constraints and objectives of both problems intersect in a natural way. Consider a transportation company which needs to freight a set of objects from one location to another inside the container of a lorry. The container has a particular shape that cannot be altered, and a maximum weight; surpassing this limit would make the vehicle's movement more difficult or even impossible. The objects to transport are solid, and therefore cannot be placed in the container overlapping with each other, nor with the container's bounds. Each item has a measurable weight and can be assigned a value using a certain function chosen by the company, e.g. considering only the monetary value of the item, adding a weight based on the scheduled delivery quickness, or taking into account any other strategic variables. The objective of the company is to fill the container with objects to maximize the transported value, respecting all the mentioned conditions. In this context the items are defined with three-dimensional shapes, and gravity is an additional constraint, since objects cannot float. Further constraints may be considered, such as vehicle stability or object fragility, e.g. limiting the weight that can be placed on top of an object. When attempting to solve this problem, and particularly to check if object overlapping takes place in a candidate solution (which would make it invalid) it is required to apply an intersection check between three-dimensional models ([Ros08](#)), considering that models are represented as polygon meshes, usually as triangle meshes for efficiency. This operation is substantially complex in computational terms, especially if there are many objects made up of many polygons, and this calculation may need to be repeated many times when attempting to solve the Three-Dimensional Packing Problem, and any variation that introduces weight and value constraints from the Knapsack Problem. For simplicity, many attempts to solve the Packing Problem simplify the shapes of the objects and the container to some that allow trivial intersection checks, such as rectangular cuboids ([MPV00](#)) or spheres ([SWM08](#)).

In other scenarios, the objects can be treated as two-dimensional shapes. Some works assume regular shaped objects for simplicity or domain-specific convenience, such as rectangles ([HK13](#)) and circles ([CS03](#)). A less restrictive subclass of the Packing Problem is the Two-Dimensional Irregular Shape Packing Problem ([CFLR03](#)), in which both the objects and the container are two-dimensional geometric shapes, which may comprise not only regular shaped objects, but

also polygons of arbitrary shape, either convex or concave. There are many applications of this problem (Gom13), such as layout design and marker making in the textile and garment industry (DM96; DRG19), furniture and shoe designs robust to defective outlines of pieces (HL95), or the design of high precision tools (GO02; MOGF07). In fields of application including but not limited to the mentioned ones, it seems reasonable to think that being able to add a notion of weight and value to the packed objects may allow to enrich the definition of the optimal objectives to be satisfied. For instance, when designing a tool in a factory, some of the pieces that compose the tool may be essential to make it functional, and therefore they would deserve higher value than other pieces. At the same time, each piece contributes with its weight to the total weight of the tool, that should be upper-limited to make feasible its intended use, e.g. in hands of a human or manipulated by a machine.

A modified version of the Packing Problem which incorporates the weight constraints and value maximization objective of the Knapsack Problem seems to have potential applicability to help refine some industrial processes, as explained in the previous paragraph. These circumstances have motivated to explore the problem in this work, where it is referred to as the joint problem of the Packing Problem and the 0/1 Knapsack Problem. Particularly, this work is focused on solving the variant of the joint problem in which the shapes of the objects are two-dimensional and irregular polygons are admitted, that is, the joint problem of the Two-Dimensional Irregular Shape Packing Problem and the 0/1 Knapsack Problem (Joint Problem hereinafter). Nevertheless, the aim of this work is not limited to design and implement an algorithm for solving the problem, but also to analyze the time performance and the quality of the solutions produced by the new methods, in a set of meaningful problem instances partially extracted from real-world challenges. An initial hypothesis of this work was that being able to solve the two-dimensional version of the joint problem may facilitate, in future work, the design of a solution for three-dimensional variant, which is potentially much more computationally expensive, but would add applications of use, e.g. in the aforementioned logistics scenarios.

The Joint Problem is defined as follows. Consider a set of n objects, each of them defined by their geometric shape s , weight w and value v . The objective is to place a subset of those objects in a position p with a rotation angle r within a container, defined by its shape S and highest acceptable weight W , maximizing the container's value V (defined as the sum of the value of the contained objects) and without any overlap (geometric intersection between different objects or items and the container). Let x be binary variable representing the presence (1) or absence (0) of an item in the container, and consider *intersect* to be a binary function that returns 1 if two items intersect and 0 otherwise, given the parameter values of shape, position and rotation angle (an unspecified position value is treated as a (0, 0) coordinate while an omitted angle is replaced with 0; both circumstances are applied for the container, that does not ever need to move or rotate). The value function V to maximize and the problem constraints are formalized as follows:

$$\begin{aligned} \text{maximize } V(x, p, r) = & \sum_{i=1}^n v_i x_i : \sum_{i=1}^n w_i x_i \leq W, x_i \in \{0, 1\}, \sum_{i=1}^n x_i \cdot \text{intersect}(s_i, p_i, r_i, S) = 0, \\ & \sum_{i=1}^n \sum_{j=1, j \neq i}^n x_i \cdot x_j \cdot \text{intersect}(s_i, p_i, r_i, s_j, p_j, r_j) = 0 \end{aligned} \quad (1)$$

Since the objective function does not have a derivative (neither in the two separate problems nor in the joint one), gradient-based approaches should be discarded for solving the Joint Problem. Instead, a black-box function optimization method can be useful (BB13), and evolutionary algorithms (ES⁺03), e.g. genetic algorithms, are well-known to be very competitive in derivative-free optimization problems (OD94; HQL94), with greater capabilities to surpass local optima

than alternative methods ([SD08](#)). Therefore, the goal of this work is to define an evolutionary algorithm for solving the Two-Dimensional Irregular Shape Packing Problem combined with the 0/1 Knapsack Problem (the Joint Problem). Previous works have shown that promising results can be obtained when using evolutionary algorithms, especially genetic algorithms, in both the Packing Problem ([CFLR03; JPSP14](#)) and the Knapsack Problem ([ZT98; Jas02](#)) separately, but to the knowledge of the author of this work there is no existing approach that solves the Joint Problem. The new problem should not be confused with the Two-Dimensional 0/1 Knapsack Problem ([DVDQMX12](#)), solved with a greedy algorithm with local search, where objects defined by certain irregular shape and value need to be placed into a container, to maximize the total value. Such problem does not present a notion of weight for the container or the items, while the Joint Problem incorporates weights and the container's capacity constraint. Furthermore, the value of objects in the preexisting problem is simply the area of the items, while in the Joint Problem the value is a completely independent variable. Combining both the Knapsack Problem and the Packing Problem leads to a rather uncommon situation in evolutionary algorithms: there are both discrete and continuous variables in the same problem. The discrete data is the binary presence or absence of items in the container, whilst their position coordinates and rotation angle are real-valued, continuous data, represented with limited-precision floating-point numbers.

Prior to proposing an algorithm to solve the Joint Problem, previous works were studied, namely those using evolutionary algorithms to solve each problem separately, i.e. the Two-Dimensional Packing Problem and the 0/1 Knapsack Problem. The approaches used by other works to handle the presence of both discrete and continuous variables in evolutionary algorithms were also analyzed. The study of previous work is presented in Chapter [2](#). Since the Joint Problem contains variables that operate non-trivially in the continuous space (position and rotation of non-overlapping geometric shapes), finding a global optimum solution of the problem seems computationally unfeasible. Based on this fact, and considering the lack of literature about the Joint Problem, other algorithms were designed to be able to assess the solution quality and the time efficiency of the evolutionary algorithm, in a comparative way. In particular, the additional methods are a greedy algorithm, formulated with the will to approximate solutions easily following a best-first approach, and a reversible algorithm, which combines random decisions with the possibility to restore the solution to a previous state if experimental changes do not lead to improving the quality of the solution after some time. These techniques constitute, with the evolutionary algorithm, the proposed methods analyzed in Chapter [3](#), while an overview of the implementation can be found in Chapter [4](#). The techniques are applied to solve a set of problem instances that constitute the Joint Problem Dataset, described in Chapter [5](#), and used to optimize the configuration of parameters of the algorithms, as detailed in Chapter [6](#). The final configuration of each algorithm is applied to the problems of the created dataset, and the quality of the solution produced by each algorithm, as well as the execution time, is compared for each problem. The methodology and results of the experiments are explained in Chapter [7](#). Chapter [8](#) studies the applicability of the designed algorithms to solve a traditional Packing Problem. Chapter [9](#) expounds the conclusions of the work, and outlines the directions in which future work could be undertaken. The [Appendix](#) contains a list with the numerous parameters of the evolutionary algorithm, the pseudo-code for the reversible algorithm (due to its considerable length), secondary tables of the parameter optimization phases and visualizations of the best solutions obtained for both the Joint Problem and the Packing Problem.

2 Previous Work

2.1 Evolutionary algorithms to solve the 0/1 Knapsack Problem

Many evolutionary algorithms have been designed to solve the 0/1 Knapsack Problem. One of the most extensively used approaches is a genetic algorithm where each chromosome is encoded as a bit string that has as many bits as items in the problem, representing the presence of an item in the container with 1 and its absence with 0 ([MA94](#)). This approach can lead to unfeasible solutions that exceed the capacity of the container. Therefore, it is common to use a repair operator to correct solutions after mutation or crossover, discarding items that were added to the container, usually by means of a greedy algorithm, until the sum of item weights is equal to or lower than the capacity bound. Normally, the items are removed in opposite order of profitability ratio, which is the result of dividing the item's value by its weight ([ZT98](#)). The objective of the 0/1 Knapsack Problem is to maximize the sum of values of the items in the knapsack (subject to the container's weight limit), and this can precisely be used as the fitness function to solve the standard version of the problem ([HLM96](#)). Alternative problems with multiple knapsacks to fill belong to the multi-objective optimization problems, as opposed to the original 0/1 Knapsack Problem. In such cases, Pareto dominance-based fitness functions are commonplace, with the interpretation that items belong to different knapsack niches ([ZT98](#)).

Multiple approaches use standard versions of the crossover operator, especially one-point crossover ([HLM96; ZT98; Jas02](#)), where a random split point is selected and offspring is produced by combining the part before the split point of the first parent with the section after the split of the second progenitor. Two-point crossover operates in a similar way with two split points, and has also some precedents of use for this problem ([HK02](#)). When it comes to the mutation operator, is it usual to apply the standard binomial formulation that states that, for every bit of the bit string, there is a certain probability that the value of the bit will be mutated, i.e. changed from 0 to 1 or vice versa ([ZT98; HK02](#)). Some other approaches select a single bit randomly and perform a mutation on it ([HLM96](#)), i.e. they always change one bit, instead of performing a variable number of bit modifications (following probabilities). The technique can be easily extended to selecting any number or proportion of bits.

The selection of parent chromosomes for recombination is usually performed using a tournament strategy ([ZT98](#)), where two individuals are chosen as progenitors for new offspring, being the ones with the highest fitness value of two pools (one per parent) of a fixed number of randomly checked individuals of the population. In conjunction with tournament selection, it is commonplace to use a partial replacement strategy to update the population ([ZT98](#)), in which part of the old population is replaced by some of the new offspring, making the survival chance proportional to fitness, e.g. by performing inverse tournament selection to discard the worst individuals gathered in a pool, considering both the whole old population and the offspring for selection. Regardless of the method, the fittest individual is commonly preserved by default, applying the elitism strategy ([DPAM02](#)). A typical termination criterion in evolutionary algorithms is to stop after a certain number of iterations ([ZZZ06](#)), but other criteria can be used, such as a maximum number of recombinations ([Jas02](#)).

2.2 Evolutionary algorithms to solve the Two-Dimensional Irregular Shape Packing Problem

In the Two-Dimensional Irregular Shape Packing Problem, items can have very different shapes ([JG98](#)), ranging from regular-shaped objects (including but not limited to squares, rectangles, circles and ellipses) to polygons of any convex or concave shape, and in some occasions even compound shapes with holes. Using the original shapes in the packing solver algorithm would

require to define geometric intersection functions for every pair of shape types (e.g. circle with circle, circle with rectangle, circle with polygon, and many others). The operations with the most complex shapes are much more computationally expensive than those involving only simple regular shapes, and for this reason it is commonplace to transform all items to an approximate simple shape, such as a rectangle (FS75), or a set of adjacent shapes of a simple type, such as nested rectangles (AA76), or equidistributed equal-size squares (JG98). Following the latter scheme, if the container shape is also discretized into squares, a chromosome can be encoded as a two-dimensional grid or matrix, where a certain value can indicate that a cell is out of the bounds of the container (e.g. -1), another value can represent that a square is empty within the container (e.g. 0), and other integer values, from 1 up to the number of items, can represent that a cell is occupied by an item with a given index (JG98). This approach eliminates the continuous nature of the position and the shape of the items, and in some scenarios a limited number of orientations is used, e.g. 4, by allowing only 90-degree rotations (JG98), so that the problem does not depend on any real-valued variables and is instead simplified as a combinatorial optimization problem with finite sets of states and solutions. An alternative option is to represent shapes with a boundary polygon (CFLR03), with the drawback that holes contained in complex shapes will not be filled. Other approaches avoid discretizing the shapes of objects altogether, but still discretize the position space (DRG19), by generating a set of equidistributed grid points in the container's space, that can be set as an object's reference position at placement time. The density of the grid, i.e. the distance between the points, can be dynamic, as part of the representation of the individual. Instead of being represented by the coordinates and rotations of objects, a chromosome can also be represented by the order in which a latter algorithm will place the objects (JPSP14), in occasions also including as a variable the specific type of the placement algorithm (DRG19). It is relevant to note how in these latter scenarios, the algorithms do not explicitly describe the actual placement in the chromosome encoding, which can anyway be processed and taken into account at a later step, e.g. during fitness evaluation.

The crossover and mutation operators can be adapted to be effective in the packing context. When it comes to crossover, if the container space and shapes have been discretized into grid cells (squares), a common policy is to preserve in the children the common placements of the parents and swap the remaining regions (CFLR03), believing that the coincidence of information in both progenitors, especially in advanced generations, is a sign of potentially useful genetic data that should be passed to the offspring. This technique requires further adjustments to preserve solution feasibility, such as changing the position of items that are cut in the process, removing duplicate items and placing back the removed ones, if possible. Another possible approach for the grid-based chromosome scenario is to randomly choose a rectangular region for swapping (JG98), expanded to cover the area of shapes that would be cut by the initial rectangle. Then, one of the offspring inherits the items contained in the swap region in the first parent, and the items lying in the external zone and the second parent (except those that would cause an overlap). Afterwards, the opposite procedure is done to generate a second offspring. As in the previous case, this technique also requires to check for duplicate and missing items. For chromosome representations that encode the order in which another algorithm will place items, the usage of discrete recombination (choosing the value of one parent or another for every gene of an offspring) has been reported (DRG19), after being used in evolutionary techniques such as the Breeder genetic algorithm (SVM93).

Explored options for mutation, in those cases in which chromosomes directly encode placement information, include randomly selecting a placed item and moving it to an arbitrary position (JG98; CFLR03), randomly altering the rotation of an item (JG98), or swapping the positions of two randomly selected items (JG98). In some cases, mutations are only applied (confirmed) after ensuring that the new configuration does not cause any overlap, and attempt-

ing to solve intersecting cases with further changes in the position or rotation angle of items ([JG98](#)). Some works which use a representation of chromosomes that encode the order of item placement instead of actual positions report the usage of integer mutation ([DRG19](#)) to change the order of items, i.e. modify the integer index of a gene.

New operators have been designed in the context of the Packing Problem to attempt to make algorithms converge faster to high-quality solutions. A noteworthy case is the compaction operator ([JG98](#)), that aims to reduce the empty space between objects by moving all of them towards a corner of the container, until they become in contact with one another. In discrete, e.g. grid-based representations, there are a finite number of displacements of an object, but when the position space is continuous some optimizations are used, such as using binary search to check intervals of displacement when moving an item in a direction ([CFLR03](#)), to delimit the zone in which the moved object would intersect with other items; both ends of the interval need to be checked to overcome situations in which there is an intersection at one of them, but not in the other one.

The usage of fitness proportional parent selection and tournament-based population update is widespread ([JG98](#); [JPSP14](#)), as in the case of the Knapsack Problem, and many other problems approached with evolutionary algorithms, mainly genetic algorithms. Elitism is commonly used ([JG98](#); [JPSP14](#)), to guarantee that the best solution (in terms of the fitness function) found up to any time is preserved until a better one is found, promoting exploitation. The algorithm is stopped at a termination condition, e.g. after a maximum running time ([JPSP14](#)) or a certain number of generations without improvement in the fittest individual ([CFLR03](#)).

2.3 Combination of discrete and continuous variables in evolutionary algorithms

In most scenarios covered in the literature, evolutionary algorithms are applied to problems where all the variables, codified as genes of the population's chromosomes, belong to the same data type. Some of the most common data types used to represent genetic information are sequences of binary values (also known as bit strings), discrete (integer, ordinal or nominal) values, or continuous (real, floating-point) numbers. Nevertheless, some problems solved with evolutionary algorithms have been reported to contain variables of different data types. There are two main ways in which this situation has been handled. The first option is to convert all data into a common representation type prior to using the evolutionary algorithm, whilst the second option is to design an evolutionary algorithm that is capable of processing a genome with multiple data types, using the appropriate (different) operators and adjustments in a specialized way for every variable type.

In the data conversion scenario, the more complex types are usually transformed into the simplest type among all the data types in the problem, or even the simplest possible representation, that of bit strings ([For93](#)). Nominal data is usually converted into one-hot-encoding bit strings, with the same length as possible categories, having a single bit with the 1 value, in the index associated to a specific category. Ordinal data is usually converted into integers. Integer values are often transformed into bit strings by shifting from the decimal to the binary base. Such method can increase the range of values in the binary version of the number, which may require additional validation checks and a repair operator to ensure that a candidate solution is feasible ([MBM03](#)). It is common to see how real-valued variables are discretized into integer values or directly transformed into bit strings. A range mapping operation is applied to floating-point values, such as fixed-point integer encoding ([BGB10](#)), that defines a lower and higher bound of the floating point range and an exponent that determines the number of bits to use and the maximum attainable precision, which should be chosen carefully to ensure that the

algorithm is able to successfully obtain an optimal solution, or an approximation that is very close. Another issue related with the conversion of numeric values into bit strings is that the difference between adjacent numeric values is magnified or reduced in a non-linear way in their bit representation, which can make mutation less effective, by causing much larger or smaller changes than desired. Instead of converting numeric values to their standard bit representation, many approaches use Gray coding (Wri91) instead, a technique that produces bit strings with the property that a single variation in one bit yields a 1-unit integer difference. Alternatively, direct truncation or rounding of real values can be applied to obtain integers, which imply an obvious loss of precision, but it has been shown that it can be effective to use the output individuals of the genetic algorithm that used the rough discretization of variables as a starting (initialization) point for the population of a second variation of the algorithm that only considers the real-valued variables, and can eventually find a more precise, closer-to-optimal solution (SNB98). Of course, such an approach implies the execution of two algorithm runs, which may increase the total computation time, even though the overhead is mitigated by the fact that only some variables are used in each of the two runs. When real-valued variables are discretized into integers, or when variables are integers themselves, some approaches use local search to explore in more detail areas of the search space that seem promising according to the fitness function value, simulating Lamarck's evolution ideas (Ros99). Other approaches, also for numeric variables, use binary trees as a memory that stores past results that may be expensive to obtain (e.g. some fitness function calculations), so that each newly generated chromosome is compared with those in the memory tree (in a very inexpensive way due to tree balancing); if a very similar individual is found (according to a threshold of difference), its calculations are reused or extrapolated for the new chromosome (GACGW03).

When the different variable types are kept at genome level, it is the task of the evolutionary algorithm to process each type of variable in the appropriate way, by defining operators that are specific for each of them. For instance, when evolving a population of fuzzy rule-based systems, two different crossover operator should be applied for the recombination of membership functions and that of fuzzy rules (RG^P01), since their data representation is potentially very different; while the former can be stored as a function type with some real-valued parameters, the latter may be defined as a matrix structure. Other works contain a great variety of variable types as genes and perform different mutations for each of them (CJW00), including uniform random variation for binary variables, single-unit or multi-unit displacement for integers and Gaussian noise addition for continuous variables. Nevertheless, the possibility of using different versions of an operator, e.g. mutation, is not limited to the multi-type genome scenario. Even with genes of the same data type, multiple mutations, recombinations or other operators can be considered. A very intuitive case is that of having a different probability distribution or strength (magnitude) of mutation for different variables, depending on their domain. Other approaches consider different versions of an operator for a same gene, or artificially lead to modify the worst gene (from the point of view of the fitness function or another specific evaluation) with higher likelihood, in some cases estimating which is the best mutation among different alternatives (HAAN⁺¹⁶).

3 Proposed Methods

3.1 Greedy algorithm

The greedy algorithm adds items to the container, one by one, until any of the following termination conditions are met:

- All the items have been placed, if that is feasible.
- The capacity of the container has been exactly reached, or all the remaining items would lead to exceeding the capacity.
- It has not been possible to place any item (without overlapping) for a certain number of consecutive iterations.
- A maximum total number of iterations have been performed.

In each iteration, the algorithm selects an item from outside the container in a stochastic way. In particular, the selection probability of an item is proportional to a function that takes as parameters the value, weight and area of the item, to calculate a numeric score representing the suitability of choosing the item for placement: the higher the score, the more suitable the item. Any function respecting the definition can be used, and changing from one to another would affect the greedy policy of selection. Intuitively, the function should reward items with high value, since maximizing the value of the container is the ultimate objective, and penalize the weight and area, that are two finite resources in the container that constraint the feasibility of candidate solutions. In some contexts of application, it may be interesting to give much more importance to the value than anything else, while in others the relation of the value, weight and area should be a priority. For this reason, a flexible function to calculate the greedy score of an item is the weighted sum of the value and the ratio defined by dividing the value by a weighted version of the product of the item's weight and the area of its shape. Let v , w and a be the value, weight and area of the item, respectively. Consider K_v as a constant that expresses the importance of the value of the item alone compared to the previously mentioned ratio. Given the constraint $0 \leq K_v \leq 1$, a value of K_v close to 0 implies that the ratio is more relevant to calculate the score, while a value of K_v close to 1 gives more importance to the value exclusively, decreasing the influence of both weight and area. Let K_a be a constant specifying the importance of the area in the ratio, such that $0 \leq K_a \leq 1$. The specific value of K_v and K_a can be fixed in the specific context of application. Then, a greedy score function G for an item can be formally defined as:

$$G(v, w, a) = K_v \cdot v + (1 - K_v) \cdot (v / ((1 - K_a) \cdot w + K_a \cdot a)) \quad (2)$$

The selection algorithm associates items with adjacent ranges of values with the length of their greedy score, and then a random value between 0 and the sum of ratios is chosen. Afterwards, binary search, a technique with logarithmic computational complexity, is used to find the item with the range of values that contains the random number. The item is subsequently selected, and assigned a random position and rotation to be placed in the container. If it causes any intersection with other items or the container, the addition of the object is discarded. Every time an item is validly placed in the container, and also prior to the first iteration, the objects whose weight would make the container exceed its capacity are discarded permanently, so that they are not even checked in later iterations. To make this task efficient, the items are sorted by ascending order of weight at the beginning of the algorithm, so that binary search can be used to find the first item that would cause the capacity to be exceeded, and discard it, along with

all the following ones. A pseudo-code of the greedy algorithm is represented in Algorithm 1. Section 3.3.1 explains in detail the common data structures used to represent the container, the items and the solution of a problem, in the greedy algorithm and the rest of methods presented in this work.

The described algorithm is considered to be greedy because of two reasons. Firstly, once the algorithm has placed an item in the container it will never remove it or modify its position or rotation. Secondly, the algorithm is prone to prioritize attempting to place items that provide the greatest short-term gain, in terms of the greedy score, e.g. a remarkable addition in value with a low increase in the container's weight. However, the item selection was made stochastic, without using a deterministic order for item placement, because as soon as one of the items (the next one given the order) is very difficult to place, many iterations would take place without any success in placements (therefore not improving the solution), even if a maximum number of attempts per item is defined. Placing some items can be even impossible in some problems, due to their shape and the container's shape, and it is not trivial to detect these cases beforehand, unless the item's area is larger than the container's area. Therefore, it was considered that it was more appropriate to make the item selection process stochastic. Nevertheless, the greedy score formula is flexible enough to intensify the importance of the value of the items alone (using a weight), to make the algorithm more likely to place items in descending order of value.

In some problems, the solution quality of the greedy algorithm can potentially have a lot of variance from one execution to another, due to the non-deterministic order in which items are placed. It may frequently happen that the algorithm is unable to place some items because there is no space left for them due to how previous objects were arranged. Therefore, it can be interesting to apply an iterated greedy algorithm, i.e. run the base algorithm for a certain number of iterations, and keep the solution with the highest value as the final one. For greater efficiency, the iterated version of the greedy algorithm performs some preliminary operations just once, to later reuse the result in each iteration. These tasks include the calculation of the greedy score for all items, the weight-based sorting and the discarding of items whose weight is greater than the capacity of the empty container. In hypothetical contexts of application where it is possible to estimate the optimal value for a problem beforehand, it may be a good idea to stop the algorithm as soon as the optimal or a close-to-optimal solution is obtained.

3.2 Reversible algorithm

The reversible algorithm has the same termination conditions as the greedy algorithm, but it has different criteria to select which object is added into the container in a certain iteration: it performs a uniformly (non-weighted) random item selection. This choice can diversify the exploration in the search space of placements, at the cost of not exploiting the short-term benefits of a greedy score-based criterion which, conversely, can constraint the long-term quality of the solution. The reversible algorithm introduces the possibility of removing a randomly selected item in an iteration with a certain (small) probability. It is obvious that the removal operation decreases the total value of the container, but it brings the opportunity to explore different regions of the search space, hoping to improve the value after some iterations. However, to ensure that there is not a global, long-term decrease in the value of the container, the solution prior to removing an item is reverted after a certain number of iterations if the removal has not led to an eventual improvement in the container's value. Before a recent removal is reverted or permanently accepted, more items can be removed, but with a significantly lower probability; otherwise it can be unlikely that enough placements will be done to increase the value to a sufficient extent as to improve the container's value and preserve the changes. To mitigate the possibility that an item becomes placed again soon after being removed, it can be ignored until the latter examination of the removal (to revert or accept it), with a certain probability. The

Algorithm 1: Greedy algorithm

Input : container /*Container, described by its shape and maximum weight*/,
items /*List of items, described by their value, weight and shape*/,
max_iter_num /*Maximum number of iterations*/,
converge_iter_num /*Number of consecutive iterations to stop the
algorithm if there are no placements*/

Output: solution /*Solution, describing a placement of items in the container*/

```
1 /*Create an initial solution with no placements, to be modified*/
2 solution ← create_solution(container, items);
3 /*Calculate the greedy score for all items*/
4 items_with_score ← calculate_greedy_score(items);
5 /*Sort the items by weight, to make filtering faster*/
6 items_by_weight ← sort_by_weight(items_with_score);
7 /*Discard the items whose weight is greater than the container's capacity*/
8 items_by_weight ← filter_items(items_by_weight, container.max_weight);
9 unchanged_iter_count ← 0;
10 /*Placements can only be possible with capacity and valid items*/
11 if container.max_weight > 0 and items_by_weight ≠ ∅ then
12   /*Try to add items to the container, for a maximum number of iterations*/
13   for i ← 1 to max_iter_num do
14     /*Perform a weighted random choice of the next item to try to add*/
15     item ← select_item(items_by_weight);
16     /*Try to place the item in the container, with random position and rotation*/
17     if solution.add_item(item.index, get_random_position(container.shape),
18       get_random_rotation()) then
19       /*Determine the weight that can still be added*/
20       remaining_weight ← container.max_weight - solution.weight;
21       /*Stop early if the capacity has been exactly reached*/
22       if remaining_weight == 0 then
23         break;
24       /*Remove the placed item from the list of pending items*/
25       items_by_weight.pop(item.index);
26       /*Discard the items whose weight is greater than the capacity*/
27       items_by_weight ← filter_items(items_by_weight, remaining_weight);
28       /* Stop early if there are no items remaining or they would cause the capacity
          to be exceeded*/
29       if items_by_weight == ∅ then
30         break;
31       /*Reset the potential convergence counter, since an item has been added*/
32       unchanged_iter_count ← 0;
33     else
34       /*Register the fact of being unable to place an item this iteration*/
35       unchanged_iter_count ← unchanged_iter_count + 1;
36       /*Stop early if there have been too many iterations without changes*/
37       if unchanged_iter_count == converge_iter_num then
38         break;
```

reversible algorithm can also randomly rotate or move (to another point of the container) a randomly selected object according to a specific probability. Such kind of changes are only applied if they do not generate shape overlapping. Analogously to the removal case, these modifications are motivated on the hypothesis that they may lead to an improvement of the value after some iterations, but they do not directly affect the container's value, that remains unchanged, and therefore it is unnecessary to define any reverting operation for movements or rotations. A pseudo-code for the reversible algorithm is represented in Algorithm 9, split in consecutive pages due to its considerable length, that has motivated placing the pseudo-code in the Appendix, to avoid introducing a long break in the sections of this chapter.

The reversible algorithm's strategy of conditional changes that can be reverted if proven useless is very different to that of the greedy algorithm, that cannot change poor decisions after making them. Nevertheless, it can be argued that when a solution is reverted, all the work done since the last removal has been in vain, or only to confirm that a certain direction for exploration was not convenient. Conversely, all the steps made during the execution of the greedy algorithm contribute to the final solution, excluding the cancelled additions that would have caused intersections. The reversible algorithm can be seen as an attempt to emulate (in a non-literal way) some mechanisms of an evolutionary algorithm in a single individual, without any notion of crossover, and understanding mutation as all the possible actions that change a solution: addition of items, removal and modification of placements. There is no actual competition for survival among a population of candidate solutions, but some time after removing an item, there is a small selection among two solutions, the one before the removal and the current one: the one with the highest container's value (which can be seen as the fitness value) is preserved.

3.3 Evolutionary algorithm

3.3.1 Chromosome representation

The function to maximize in the problem, V , is the sum of the values of the items placed in the container. The shape, value and weight of all items, as well as the capacity and the shape of the container, are constant values, i.e. they do not change from one solution to another in the same problem. Therefore, it would be redundant to encode such data in the chromosomes of the evolutionary algorithm. Instead, the set of items which have been placed in the container can be used to describe a candidate solution; their placement information, namely the position and rotation of every placed item, is also essential to fully specify an individual.

Therefore, a chromosome is represented as a variable-length data structure which contains one data unit associated to every item placed in the container. For simplicity the data structure may be thought of as a list, but in practise it is a hash table indexed with an integer value, the object's identifier, comprised between 1 and the total number of items. An identifier is assigned to every item before any main action of the algorithm is performed. The hash function, that receives an identifier as a parameter, simply returns that identifier, so it cannot generate collisions, because identifiers are unique. Every element in the hash table represents the placement information of the associated placed object. This information can be represented in two ways. For most objects, it is a pair of values describing the following aspects of the placement:

1. Reference position of the item in the container, expressed as a vector of two real values, the width and height coordinate, respectively.
2. Rotation angle of the item in the container, greater than (or equal to) 0 degrees and lesser than 360.

For some object shapes, namely circles, the rotation is irrelevant, because it does not change the covered space in the container. In these cases, the placement information of an object stored

in the hash table contains a single real number, the reference position.

The reference position coincides with the center of the object for simple regular shapes (e.g. rectangles and circles). In the case of more complex shapes (e.g. polygons), the reference position is the center of the bounding rectangle of the shape for the 0-degree rotation angle, and it is used as the origin point for rotations. The motivation to fix this point as the bounding rectangle's center for the 0-degree case and not recalculate it for every rotation guarantees that all rotations keep the reference point in the same position, even if the bounding rectangle does not coincide for different angles of rotation, therefore altering its center. For the shapes in which the reference position is inside the geometry of the shape, the point should always be inside the container. To avoid calculating if the container contains the point, this requirement can be simplified to ensuring that, when placing or moving an item, the reference position lies in the range defined by the container's rectangle bounds, i.e. the coordinates of its leftmost and rightmost points for the horizontal axis, and the lowest and highest values for the vertical axis. However, for some shapes, such as compound polygons with holes, the reference point may not be part of the geometry, and it should not be discarded to place the shape in a way that makes the reference position be outside the container. As a limitation for the values of the reference point in the latter scenario, it is fixed that, if the reference position lies outside of the container in the potential placement of a shape in the container, at least one point of the bounding rectangle of the shape should be inside the container. In any case, and regardless of the reference point, no placement of a shape in the container is confirmed until a validation check guarantees that there is no intersection between the new item and the container or any other placed item, and analogous checks are used when moving an item that was already in the container.

The reason to represent the chromosome with a hash table instead of using another data structure, e.g. a list, is based on performance optimization. The algorithmic complexity of adding or removing an element from a hash table is $\mathcal{O}(1)$, which is a significant improvement over the linear or logarithmic complexity of other data structures. Searching an item or checking whether it is already placed in the container has $\mathcal{O}(1)$ complexity too. Due to the trivial nature of the hashing function, it does not introduce overhead. Although it is not part of the chromosome, an immutable hash table which associates all the object identifiers to their constant information (shape, weight and value) is also generated at the beginning of the problem, to allow immediate retrieval of the data of any item of the problem.

The chromosome representation defined for this algorithm does not encode candidate solutions using approaches specific of some types of evolutionary algorithms, such as the bit-strings commonly used in genetic algorithms. Instead, it directly operates in the domain level of the problem, and avoids the computational cost of encoding and decoding individuals. The representation is therefore flexible enough to be easy to adopt by non-evolutionary algorithms. Regardless of the fact that it was originally conceived for this evolutionary algorithm, the described representation of candidate solutions was also used in the greedy algorithm presented in Section 3.1, and also in the reversible algorithm, explained in Section 3.2.

3.3.2 Fitness function

The fitness function yields a value of 0 if any of the Joint Problem constraints is not met, i.e. if the capacity is exceeded or there is an overlap between items or between an item and the container. Otherwise, when all constraints are respected, the fitness function coincides with the objective function maximized in the Joint Problem, defined in Equation 1. The two possibilities are formalized as follows:

$$F(x, p, r) = \begin{cases} V(x, p, r) & \sum_{i=1}^n w_i x_i \leq W, x_i \in \{0, 1\}, \\ & \sum_{i=1}^n x_i \cdot \text{intersect}(s_i, p_i, r_i, S) = 0, \\ & \sum_{i=1}^n \sum_{j=1, j \neq i}^n x_i \cdot x_j \cdot \text{intersect}(s_i, p_i, r_i, s_j, p_j, r_j) = 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

When the fitness value of two individuals is compared in the evolutionary algorithm (e.g. during parent selection or population update, as explained in Sections 3.3.5 and 3.3.8, respectively), it may happen that both chromosomes have the same fitness because their container's value coincides. Despite of the equality in the fitness value, it may be interesting to estimate which of the two chromosomes is more likely to lead to better solutions in future generations, and avoid selecting one of them in a uniformly random way. For this reason, a tie-breaking strategy is used, based on the hypothesis that the best solution is the one with the smallest total area (computed as the sum of the areas of the items placed in the container), since it keeps more space free for items that have not been placed yet. The hypothesis may not be true in all cases, depending on the characteristics of the container, the placed items, and those which are outside of the container. Nevertheless, the criterion serves as an intuitive guideline assumed to be true most of the times. In situations in which the total area of two compared chromosomes is exactly equal, it may mean that the same objects have been placed in the container in the two individuals. In such cases, the area of the tightest bounding rectangle that contains all items is computed for both chromosomes, since an additional tie-breaking criterion sets that the best individual is the one with the bounding rectangle with the smallest area. Even though this idea is not guaranteed to be true in every context, it seems intuitive that a greater compaction of objects can facilitate leaving larger free spaces for new items. If two solutions are still equal for both tie-breaking criteria, they are considered equal in selection operations, and one of them is chosen randomly.

It is interesting to notice that the tie-breaking criteria are not incorporated into the definition of the fitness function, but only calculated when it is necessary to break a tie in tournament selection, which is explained in Section 3.3.5. Calculating the areas involved in the tie-breaking criteria only for those individuals of a tournament pool which are in a tie (instead of following the process for all individuals that opt to be selected), can be seen as an optimization of the algorithm. Regardless of this design choice, it may be possible, if still wanted, to numerically incorporate the tie-breaking information into the fitness formula, if some consistency requirements were met. Namely, a real value inversely proportional to the sum of areas mentioned in the first tie-breaking criterion should be defined in a range of values with a minimum value greater than 0 (even if the totality of the area of the container was filled by items) and a maximum value below the value of the less valuable item of the problem, so that the first tie-breaking criterion, by means of its numerical contribution to the fitness value, would never be more important than having an additional item placed in the container. Analogously, the value of the second tie-breaking criterion should be inversely proportional to the area of the bounding rectangle of all the placed items; the real value would be 0 if the inversely proportional area was greater than (or equal to) the area of the bounding rectangle of the container, and otherwise it would be lower than the minimum value of the first tie-breaking criterion, to avoid being treated as more relevant than the previous finding.

3.3.3 Solution feasibility

The formalization of the fitness function (3) considers the theoretical possibility of having unfeasible individuals. However, in practise and for convenience, the evolutionary algorithm is never

allowed to create (from scratch or by modification) a solution which does not respect all the constraints of the problem. In other words, solutions are either created feasible or not created. The motivation of this policy is to avoid the need of designing a repair operator to transform invalid solutions into feasible ones. Imagine, for instance, the hypothetical case in which the algorithm would tolerate overlapping of items in the container, as a result of placing an object. It would be required to perform a latter modification operation, that may include movements and rotations for intersecting items, and additional intersection checks to determine the validity of the solution. Such computationally expensive operations would be performed without any guarantee of eventually finding a valid solution, unless it was possible to remove items as part of the repair process, which would reduce the fitness value, in some cases not compensating the value added with the placement that caused the unfeasible situation. Since the whole search space of solutions can be eventually explored without producing invalid solutions at any time, unfeasible situations are simply prevented.

3.3.4 Generation of the initial population

The initial population of the problem is generated executing a random placement algorithm for each individual to create. Specifically, the method selects, in a uniformly random way (except for special situations explained in next paragraph), an item that is not in the container and would not cause to exceed the capacity, and assigns it a random position and rotation angle. Subsequently, the algorithm checks whether the object, taking its shape into account, can be placed into the container (with the selected position and rotation) without causing any overlapping. If the non-intersecting condition is met, the placement is confirmed, since it preserves the feasibility of the solution; otherwise, it is discarded. The process is repeated until a maximum number of iterations I_{max} is reached, or an early termination condition is met: there are no more items to place (discarding those that would cause the container's capacity to be exceeded) or the sum of item weights is equal to the container's capacity, or a fixed number of iterations I_{conv} has been performed without being able to add an object without overlapping. It is relevant to note that the algorithm used to generate each of the initial solutions of the population is similar to the greedy algorithm presented in Section 3.1, with the exception of not using a greedy score to weight the probability of selecting an item (in this case the selection is uniformly random), and also incorporating a novel specialization mechanism, explained in the next paragraph.

The population has a prefixed size μ , which remains constant throughout generations. Generating a suitable initial population is a key step in an evolutionary algorithm. Having a wide variety of initial solutions can boost the exploration of extensive regions of the search space, and therefore facilitate the creation of solutions of high quality via mutations and other operators, in the subsequent generations. Therefore, it is wanted that all items whose weight is not higher than the container's capacity appear at least in some of the initial solutions, to have diversity for later exploration. Furthermore, it can be intuitively understood that if an item does not appear in any initial solution because it is very difficult to place it (all the initial placement attempts fail), it will be even less likely to place after some generations, when more items have been added and there is less free space. If placing such item is essential to obtain an optimal or close-to-optimal solution, the algorithm may not find it and converge to local optima. This problematic scenario can be common, for instance, in problems with very large items (in terms of area) that only fit if their reference position is placed within a very small region of the container, and constrained to have a very restricted range of rotation angles to avoid intersections: it is unlikely that a random placement algorithm can meet these strict requirements in a small number of attempts. To ensure that all items have a high chance of being present in at least one or few of the initial solutions, it is set that each of these solutions should try to place a

certain item (which can be called specialization item) before trying to place the rest of items in a uniformly random way. The algorithm tries to place the specialization item for at most a proportion S of the maximum number iterations, such that $0 < S \leq 1$; convergence is disabled while that item is not placed. All items whose weight is within the capacity limit are used in these specialized initialization procedures for the same number of initial solutions, calculated as the truncated result of dividing the population size by the number of items whose weight does not exceed the empty container's capacity. If the division has a non-zero residual, the remaining solutions to reach the initial population size μ are generated with the uniformly random scheme for placement selection described in the previous paragraph.

Algorithm 2 presents the pseudo-code of the generation of the initial population, which is used, calling the function "generate_population", in the pseudo-code of the main evolutionary algorithm, introduced in Section 3.3.9. The next sections of the evolutionary algorithm present other parts of the main algorithm, which are also invoked via function calls in the pseudo-code of Section 3.3.9, or in other secondary pseudo-codes.

Algorithm 2: Initial population generation (generate_population)

Input : container /*Container, described by its shape and maximum weight*/, items /*List of items, described by their value, weight and shape*/, population_size /*Size of the new population*/, item_specialization_iter_proportion /*Maximum proportion of iterations to specialize in trying to place a specific item in a solution, at start*/

Output: population /*Generated population*/

- 1 /*Find the items whose weight does not exceed the container's capacity*/
- 2 feasible_item_indices \leftarrow get_feasible_item_indices(items, problem.container.max_weight);
- 3 /*Limit the initial number of items to the population size (so that each of the kept items can have at least one specialization solution)*/
- 4 if feasible_item_indices.num() > population_size then
 - 5 feasible_item_indices \leftarrow get_shuffled_sublist(feasible_item_indices, population_size);
- 6 /*To promote the diversification of solutions in the search space, each feasible item will have the same number of solutions where it is tried to be insistently placed first, before any other item, to promote that all items appear at least in some initial solutions*/
- 7 solution_num_per_item_specialization \leftarrow truncate(population_size / feasible_item_indices.num());
- 8 population \leftarrow list();
- 9 /*For each feasible item, initialize a certain number of solutions with that item placed first (if possible), then completed with other items*/
- 10 for each item_index in feasible_item_indices do
 - 11 for i \leftarrow 1 to solution_num_per_item_specialization do
 - 12 population.append(generate_initial_solution(container, items, item_index, item_specialization_iter_proportion));
- 13 /*Create as many solutions with standard initialization as needed to reach the wanted population size*/
 - 14 remaining_solution_num \leftarrow population_size - population.num();
 - 15 for i \leftarrow 1 to remaining_solution_num do
 - 16 population.append(generate_initial_solution(container, items));

3.3.5 Parent selection and offspring generation

In every generation, a variable number of offspring λ' is generated. Traditionally in evolutionary algorithms, λ' is equal to a constant parameter λ . Conversely, in this algorithm, it is true that $\lambda' \geq \lambda$, i.e. λ fixes the minimum number of offspring generated (or $\lambda + 1$ individuals if λ is odd and crossover is used, as explained later in this section). The variability is rooted in the proposed crossover operator, which can generate additional offspring in special circumstances, as explained in Section 3.3.6.

There are two variants of the algorithm, that generate offspring in different ways. However, both techniques have in common that, at some point, they select individuals from the population proportionally to their fitness, to be progenitors of new offspring. Specifically, the tournament strategy is used, in which a pool with a fixed number T_p of individuals is randomly considered as candidates, and the one with the highest fitness value is chosen as the tournament winner. If there is a tie in which all the tie-breaking methods explained in section 3.3.2 fail to determine the best individual, a chromosome is randomly selected among all the ones involved in the tie.

A Boolean parameter of the algorithm, U_c , fixes whether the usage of crossover is disabled (which leads to the first variant of the algorithm) or enabled (which corresponds to the second variant). The first variant of the algorithm uses mutation as the only method to generate offspring. A total of λ individuals is chosen using the tournament strategy described in the previous paragraph. Subsequently, each of the selected chromosomes is duplicated and the resulting offspring undergoes mutation. The idea behind this variant of the evolutionary algorithm is that mutation alone can promote all the changes needed to explore the whole search space from an initial individual to an optimal (or close-to-optimal) solution, avoiding the additional complexity of the crossover operator. Conversely, the second variant of the algorithm selects $\lambda/2$ pairs of individuals if λ is even, or $\lfloor \lambda/2 \rfloor + 1$ pairs if λ is odd. Crossover is applied to each pair to create two offspring in each operation, that subsequently undergo mutation. It is interesting to observe how the choice for the offspring generation method helps to define which type of evolutionary algorithm is being used. On one hand, the first offspring generation technique is focused on mutation as done in evolution strategies (BS02) and in the Breeder genetic algorithm (SVM93), where mutation is the only (or at least main) source of variation. On the other hand, the second variant of the algorithm is aligned with the most common formulation of genetic algorithms (BBM93), where crossover is the main driving force of variation, while mutation is a secondary one.

The pseudo-codes of parent selection and offspring generation can be seen in Algorithm 3 and Algorithm 4, respectively. In both cases, the two mentioned variants of the evolutionary algorithm are considered: a parameter fixes whether crossover is used or only mutation.

3.3.6 Crossover

New offspring can be generated from a recombination of two parent individuals. In previous works where the container space and the items were discretized into grid cells, it was reasonable to attempt to identify broad common regions of placed items between the parent chromosomes to pass them to offspring, assuming that they may contain useful information since it was present in both individuals, given that they had survived and preserved that information until then. However, in the currently proposed algorithm, the space of positions is continuous, so trying to find strict coincidences is meaningless, even though it would be possible to attempt to work with approximations. Nevertheless, the hypothesis that common or very similar information is potentially beneficial is not always true, especially in the early stages of the algorithm, where similar placements may be caused by randomness.

A different approach at recombination, proposed in this work, is to partition the container in

Algorithm 3: Parent selection (select_parents)

Input : population /*Individuals that opt to be parents*/,
offspring_size /*Number of offspring to produce*/,
pool_size /*Number of individuals per tournament pool*/,
can_use_crossover /*Whether crossover will be used to generate
offspring*/

Output: parents /*Selected parents*/

1 /*If crossover will be used, pairs of parents will be selected; otherwise just one offspring
will be generated per parent, using mutation*/
2 *selection_num* \leftarrow *offspring_size*;
3 **if** *can_use_crossover* **then**
4 *selection_num* \leftarrow *truncate(selection_num/2)*;
5 **if** *offspring_size* % 2 \neq 0 **then**
6 *selection_num* \leftarrow *selection_num* + 1;
7 *parents* \leftarrow *list()*;
8 /*Perform parent selection*/
9 **for** *i* \leftarrow 1 **to** *selection_num* **do**
10 /*For crossover, parents are represented as pairs, and they cannot be the same
individual*/
11 **if** *can_use_crossover* **then**
12 *parent0* \leftarrow *get_tournament_winner(population, pool_size)*;
13 *parent1* \leftarrow *get_tournament_winner(population.except(parent0), pool_size)*;
14 *parents.append((parent0, parent1))*;
15 /*When there is no crossover, only mutation, parents are selected and represented
individually*/
16 **else**
17 *parents.append(get_tournament_winner(population, pool_size))*;

Algorithm 4: Offspring generation (generate_offspring)

Input : parents /*Individuals to use as parents of new offspring*/,
params /*Structure containing the numerous parameters of mutation
and crossover (not listed individually for readability)*/

Output: offspring /*Generated offspring*/

```
1 /*If there is no crossover, the parents will be the base for mutations*/
2 individuals_to_mutate ← list() if can_use_crossover else parents;
3 /*Use crossover to generate (at least) two individuals per pair, to be mutated later*/
4 if can_use_crossover then
5   for each (parent0, parent1) in parents do
6     individuals_to_mutate.extend(get_crossover(parent0, parent1,
7       params.subset('crossover')));
8 offspring ← list();
9 for each individual in individuals_to_mutate do
10  /*If the individual has been generated with crossover, ignore mutation altogether
   with a certain probability*/
11  if can_use_crossover and uniform_float(0, 1) <
12    params.crossover_ignore_mutation_prob then
13    offspring.append(individual);
14  else
15    /*Create a mutated copy of the individual*/
16    mutated_individual ← get_mutation(individual,
17      params.subset('mutation'));
18    /*If crossover was used and the mutated individual is less fit (or same but loses
   the tie-break), keep the pre-mutation individual with a certain probability*/
19    if can_use_crossover and is_solution_better_than(individual,
20      mutated_individual)) then
21      offspring.append(individual);
22    /*In normal conditions, keep the mutated individual*/
23    else
24      offspring.append(mutated_individual);
```

two regions to generate at least two offspring based on the items placed in the regions for different progenitors, with the possibility of producing additional individuals under certain conditions, as explained later in this section. Different geometric objects can be used as partitioning shapes, among the following options:

1. Segment with its two endpoints placed at random points on top of two different sides of the boundary rectangle of the container. The sides of the rectangle are randomly selected. For each endpoint of the partitioning segment, one of the coordinates is imposed by the chosen side of the rectangle (width if vertical, height if horizontal), while the other coordinate is generated in a uniformly random way, lying within the range of width or height of the container's bounding rectangle. In practise, a segment alone is not used, but a polygon that contains the segment and provides the same partitioning as the segment alone. If the two selected sides of the bounding rectangle are adjacent, a right-angle triangle is used; if the two sides are parallel, a quadrilateral is used instead. Both cases are given the same probability to be randomly selected. A partitioning triangle has the chosen segment as one of its three segments, while the other two lie on top of two segments of the container's bounding rectangle. The endpoints of the initial segment are two of the vertices of the triangle, while the third one coincides with one of the vertices of the rectangle (the one where the two related segments of the rectangle meet). In the quadrilateral case, the shape also contains the partitioning segment; another segment is a randomly selected side of the bounding rectangle of the container that is not touched by the partitioning segment, and two additional segments (lying in two parallel sides of the bounding rectangle) connect the already mentioned ones. The described shapes (triangle or quadrilateral) allow to define the same partitioning regions of the container in two regions as the segment they derives from. The motivation to use these polygons is related to the implementation of the algorithm, since the check of whether a polygon intersects with other shapes was already implemented, as well as the containing check; both checks are needed to determine the separation of placed items in two regions (or intersecting between both). Using the segment alone would have required to implement checks of whether another shape is left or right to (or above or below of) the segment, to determine the partitioning region of the items, so the polygon approach was preferred.
2. Geometric shape placed partially or completely within the container, with the center and all points lying within the boundary rectangle of the container (or being on the segments that separate the rectangle with the exterior). One of the newly separated regions is comprised of the area that lies inside the shape, while the other region is located in the external area. The shape is randomly selected among the following ones:
 - (a) Ellipse, whose width and height are randomly calculated values, in the ranges of a minimum and maximum width and height (respectively). The minimum and maximum length in each axis is calculated using proportions of the side length of the bounding rectangle of the container, \bar{L}_{min} and \bar{L}_{max} , respectively, such that $\bar{L}_{min} > 0$, $\bar{L}_{max} > 0$, $\bar{L}_{min} \leq \bar{L}_{max}$.
 - (b) Circle with a randomly determined radius that lies between a proportion of the horizontal or vertical side length of the bounding rectangle of the container (the particular axis is randomly chosen). The proportion of length of the selected side that determines the radius is in the range $[\bar{L}_{min}, \bar{L}_{max}]$.
 - (c) Polygon with a random number of vertices, not lower than 3 (which is the triangle case) and not greater than a fixed integer \bar{V}_c , such that $\bar{V}_c \geq 3$. A polygon is generated in such way that it does not have self-crossings or any degenerate property, and with all points within the area of a bounding rectangle, whose dimensions are calculated

proportionally to the container's bounding rectangle, using \bar{L}_{min} and \bar{L}_{max} , as done for the previously mentioned shapes.

If the container is not rectangular (i.e. different in shape than its boundary rectangle) it may happen that the whole partitioning shape lies outside of the actual container (or touching its boundary but not entering in its area). If such situation is detected, an alternative shape is generated, and the same condition is tested again; the process can be performed up to M_c times, with $M_c \geq 1$. If the final shape is still outside of the container, the crossover operator creates a duplicate of each parent as the two generated offspring, that can be subsequently modified with the mutation operator. Nevertheless, the situation of generating a partitioning shape that lies completely outside of the container only has a high likelihood to happen if the area of the container is considerably smaller than that of its boundary rectangle, or if the container is a compound polygon with holes of large area. A shape can only be accepted if it does not have a very small area, but large enough to make it likely to find items within that region. For this reason, the area of the partitioning shape needs to be greater or equal than $A \cdot A_c$, where A is a fixed proportion of the container's area (such that $0 < A < 1$), and A_c is the area of the container. If the area is smaller, the shape is considered invalid, and another attempt is performed (if still possible). In other words, in each shape generation attempt, the algorithm only validates a shape if it intersects with the container (it is partially inside of it) and it has an area that is large enough.

For simplicity, the two listed partitioning alternatives have the same probability to be chosen at a given time. Once the two regions have been defined, two offspring are generated. The first newly created individual preserves in the first partition region all the items that were completely contained in that region in the first parent, while placing in the other region all the items completely contained in the analogous area of the second progenitor. The second offspring applies the same principle but in the opposite mapping of parents with regions. The new offspring are completely feasible solutions after removing duplicate items, at the expense of having lost those items that were placed in the progenitors but lied partially in both regions of partition. To tackle the issue, it is attempted to restore the missing items in their former locations (if present in both parents, they can be only restored for one of them).

Some permutations of data pairs (representing an item and the parent from which it should be restored) are generated, based on the cut items and the parents where they were placed in. The number of permutations is limited to a maximum number R , to avoid a significant increase in the computational time. These permutations are randomly selected among all the possible ones, and describe the order in which restoring operations should be attempted. For each of the chosen permutations, every item is attempted to be placed in the selected parent's position (with the rotation it had there), something that is only applied for an item if it causes no intersection. Each permutation is applied only to one of the two base offspring, generating an alternative version of it (keeping a copy of the old one for other permutations). Among all the placement configurations obtained from the permutations (which can be considered as candidates to become new individuals), the one with the highest fitness value (for each of the two base offspring, the ones that were derived in permutations) is chosen and preserved as the definitive offspring. If other candidates have a fitness value which is among the best in the most recently updated population, they are made available as additional offspring, which makes them eligible in the survival selection of the next population update. Specifically, the candidates need to have a fitness value that is better than that of the $C \cdot 100\%$ individuals of the previous population, where $0 < C \leq 1$; the proportion C is a parameter of the algorithm. For instance, a value of 1 implies that any additional offspring can only be accepted if its fitness value is higher than that of the best solution found in the current population. The crossover pseudo-code is shown in Algorithm 5.

Algorithm 5: Crossover (get_crossover)

Input : parent0 /*First parent*/,
parent1 /*Second parent*/,
params /*Structure containing the numerous parameters of crossover
(not listed individually for readability)*/

Output: offspring /*Generated offspring*/

1 /*Initially, each offspring is generated as the copy of one parent, before recombination*/
2 offspring $\leftarrow [parent0.copy(), parent1.copy()];$
3 /*Try to create a shape that fully or partially lies within the container, and has at least a
minimum area*/
4 shape $\leftarrow create_partitioning_shape(parent0.container, params.subset('shape'));$
5 /*Only proceed if a valid shape has been obtained*/
6 **if** shape $\neq \emptyset$ **then**
7 /*Use the shape to divide the placed items of each parent in three lists: items in the
first region (inside the shape), items in the second region (outside the shape), and
shape-intersecting items*/
8 parent0_region0, parent0_region1, parent0_intersec \leftarrow
 get_shape_separation(parent0, shape);
9 parent1_region0, parent1_region1, parent1_intersec \leftarrow
 get_shape_separation(parent1, shape);
10 /*For the first offspring, keep only the items placed in the first region in the first
parent, then try to place the items of the second region of the second parent
(duplication is prevented); use the opposite parent-region mapping for the second
offspring*/
11 offspring[0].remove_items(parent0_region1 + parent0_intersec);
12 offspring[0].add_items(parent1_region1);
13 offspring[1].remove_items(parent0_region0 + parent0_intersec);
14 offspring[1].add_items(parent1_region0);
15 /*Find item-parent pairs, one for each item that intersected with the partitioning
shape in a parent*/
16 item_parent_pairs $\leftarrow get_item_parent_pairs(parent0_intersec, parent1_intersec);$
17 /*Further actions are only needed if there are intersected items*/
18 **if** item_parent_pairs $\neq \emptyset$ **then**
19 /*Generate the permutations of all (or many of) the possible orders of placement
attempts for the items that intersected with the shape*/
20 placement_permutations $\leftarrow get_permutations(item_parent_pairs,$
 params.max_permutation_num);
21 /*Try to add items following the order of permutations, keeping the best solution
derived from each of the two base offspring as final (and any alternative solutions
with very high fitness)*/
22 offspring $\leftarrow get_best_solutions_after_placements(offspring,$
 placement_permutations, params.min_fitness_for_non_best);

3.3.7 Mutation

Understanding mutation as a resource of evolutionary algorithms to explore the search space of solutions and escape local optima, different ways of altering candidate solutions are defined to promote variability among the individuals of the population. Each type of alteration (mutation step) has an associated weight proportional to its probability of being selected. A single, atomic execution of the mutation operation can be composed of a set of different mutation steps, that can be divided in three types:

1. Addition of a randomly selected item into the container, at a random position and with a random rotation angle. More specifically, the only items that opt to be selected are those whose weight would not cause the container's capacity to be exceeded. Adding an item to the container increases the total value and therefore the fitness function, which is a desirable property. The operation has a probability weight, P_a .
2. Removal of a randomly selected item from the container. Since the removal of an item decreases the fitness value of an individual, executing this operation alone would decrease the chance of survival of the individual, i.e. removal alone is detrimental. For this reason, the mutation operator has the constraint that any removal step must be compensated by a later addition step in the same execution of the mutation operator, even if it is not immediate, i.e. other steps can take place before. Nevertheless, it is still possible for a removal to cause a decrease in fitness even if an item is added later as a compensation, if the new item has less value than the previous one. The removal action can take place with a certain probability, proportional to a weight, P_r .
3. Modification of existing placements, i.e. the position or the rotation of items that are present in the container. This operation can take place with a specific probability weight, P_m , and does not affect the value of the container. Therefore, the core fitness value itself is not modified, but some of the tie-breaking criteria can be affected by the changes. Different ways to modify placements exist, related to position changes and rotations; the latter cannot be applied to items for which the rotation angle is irrelevant, namely circles. Each of the possible actions has the same probability to be selected (for simplicity, to avoid a considerable increase the number of parameters):
 - (a) Movement (i.e. position change) of a randomly selected item in a randomly determined direction, as far as possible without intersecting, using a finite number of tested positions. Namely, a two-dimensional ray (with the mentioned direction and origin in the reference point of the selected item) is used to find intersection points with the container or other placed items. The intersection point with the lowest distance to the reference point of the item to move is fixed as the furthest point of displacement. For convenience, the operation is not done if the two points are already very near, namely at a distance $M = \|(D \cdot L_x, D \cdot L_y)\|$ or smaller, where D is a parameter defining a proportion of distance to use as the minimum threshold (such that $0 < D < 1$), while L_x and L_y are, the horizontal and vertical length of the bounding area of the container, respectively. A parameter B_p determines the number of equidistributed points to check for a valid placement of the item through the half-line, with limit in the mentioned intersection point; such point is the only checked point if B_p is equal to 1, but other points between the item's reference point and the nearest intersection point are checked if B_p is greater than 1, which is the recommended setting. To find the furthest of the point to check in which the item can be placed without causing intersections, binary search is used, trying to place the item in subsequent positions,

only confirming the movements (and progressing further in distance from the initial reference point) when they succeed. This operation can act as a compaction method, since it attempts to place the object nearer to other objects or to the boundary of the container.

- (b) Rotation of a randomly selected item in a randomly determined rotation direction (clockwise or counter-clockwise), until an intersection would take place, checking a finite number of angles with uniform separation from each other, and with the first and last of those angles being at the same distance to the original angle: the 0-to-360 rotation space is divided in equal sized bins. Therefore, if no intersections are found after the last rotation, the item is not restored to its original rotation, but left in the last changed one, to produce a variation. The number of angles to check, B_r , is a parameter of the algorithm, that applies rotations as incremental constant-angle changes, stopping as soon as one of the rotations cannot be successfully applied because it would cause an overlapping situation. As the previously described movement action, this rotation operation can be interpreted as a potential compaction technique.
- (c) Change in the position of a randomly selected item, to any other position within the container.
- (d) Small change in the position of a randomly selected item, without surpassing a threshold. The small offset in position is a randomly determined point in the range $[(-\Delta p_x, -\Delta p_y), (\Delta p_x, \Delta p_y)]$, with $\Delta p_x = L_x \cdot S_p$ and $\Delta p_y = L_y \cdot S_p$, where L_x and L_y are the horizontal and vertical length of the container's bounding rectangle, respectively, and S_p is a parameter that sets which proportion of length is considered to be small, such that $0 < S_p < 1$.
- (e) Swap of the position of two items, without changing their rotation.
- (f) Change in the rotation angle of a randomly selected item, to any other angle.
- (g) Small change in the rotation angle of a randomly selected item, without surpassing a threshold. The small offset in rotation is a randomly determined value, expressed in degrees, in the range $[-\Delta r, \Delta r]$, with $\Delta r = 360 \cdot S_r$, where S_r is a parameter that sets which proportion of a complete 360-degree rotation is considered to be small, such that $0 < S_r < 1$.
- (h) Swap of the rotation of two items, preserving their position.
- (i) Change in both the position and rotation angle of a randomly selected item, to any other position (in the container) and rotation.
- (j) Small change in both the position and rotation angle of a randomly selected item, without surpassing the previously specified thresholds for both position and rotation.
- (k) Swap in the position and rotation of two items.

The mutation operation is comprised of a minimum number of iterations I_{mut} . More iterations of the item addition type are performed at the end if there are removal steps that were not compensated with a subsequent placement attempt. In each iteration, a single mutation step is performed, selected proportionally to their probability weights. The result of a mutation step is only confirmed and effectively applied to the individual if the feasibility of the solution is preserved, i.e. no intersections are generated and the weight of the container is not exceeded. If any of these constraints is not respected, different corrections are attempted to make it valid, by randomly changing position or rotation (for modifications that alter one of them), or both properties (for item additions that would have intersections), for a maximum number of attempts:

M_a for item additions and M_m for placement modifications. The only placement modification action that is not repeated multiple iterations is the one that rotates an item until it intersects: there are only two possible directions, clockwise and counter-clockwise, so they are both checked a single time, one after the other (in random order). Removal actions cannot make a solution unfeasible, so they do not need multiple attempts, but they are only applicable if the container is not empty, a constraint that is shared with the modification actions. In a single mutation step, even if multiple attempts are performed (with parameter variations), the same (primary) item is used, to encourage that difficult operations (e.g. the placement or movement of very large items when there is not much room in the container) are tried multiple times, to promote their success, instead of being discarded fast. If the action involves a secondary item, in swap actions, the second item changes from one attempt to another, while the position and rotation of the items is kept fixed, and the attempts are stopped early if all items other than the primary have been checked.

Since a single execution of the mutation operator can contain multiple mutation steps, it may happen that the configuration with the highest fitness is found at an intermediate step, and not at the end. If that is the case, the intermediate point with the highest fitness is restored with a certain probability P_i . If the mutation has been applied to an individual that had just been generated as offspring by the crossover operator, the original individual (in the state prior to mutation) can be chosen as the final offspring only if it has a greater fitness value (or same but it is better according to tie-breaking criteria) than the mutated one, with a fixed probability P_{cm} . Otherwise, if the mutated individual has better fitness (or exactly same quality, also according to tie-breaking checks), the individual resulting from mutation is selected to be the final offspring. The mutation pseudo-code is presented in Algorithm 6.

3.3.8 Population update

In every generation, the population is updated using a steady-state, partial replacement strategy $(\mu + \lambda)$. Namely, tournament selection with a fixed pool size T_u is applied to the joint population formed by the individuals of the previous generation and their offspring, to select surviving individuals proportionally to fitness, until the previous population size is restored. First of all, though, the elitism strategy is applied, leading to the fittest (elite) E individuals to have a reserved place in the new generation, and ignored in the later selection process. Afterwards, the aforementioned tournament selection process is performed $\mu - E$ times to restore the population size μ . To promote variability, once a solution has been assigned in the new population, it cannot be selected again as a duplicate. A pseudo-code with for selection of survivors is presented in Algorithm 7.

The partial replacement approach was preferred over truncation selection (i.e. keeping the best μ individuals in a deterministic way at the end of every generation) to promote variability, by giving a small chance to lower-fitness solutions to survive and improve. Some of the solutions with less fitness may be the result of removing one or more items from the container (therefore decreasing fitness), that may allow new interesting possibilities in forthcoming placements. Nevertheless, since selection is proportional to fitness, low-fitness solutions are less likely to survive than those that already have a solid fitness value, which is a desirable situation.

3.3.9 Termination criteria

The evolutionary algorithm is run for a maximum number of generations, G_{max} . After that, the fittest individual is returned as the best solution found for the problem. Early stopping is applied if the fittest individual is not replaced by a better chromosome for a certain consecutive number of generations, G_{conv} , assuming that early convergence has taken place.

Algorithm 6: Mutation (get_mutation)

Input : solution /*Base solution*/,
params /*Structure containing the numerous parameters of mutation
(not listed individually for readability)*/

Output: mutated_solution /*Mutated copy of the base solution*/

1 /*All changes will be applied to a copy of the individual*/
2 mutated_solution \leftarrow solution.copy();
3 /*Temporarily keep the original solution as the best one*/
4 best_solution \leftarrow solution;
5 removal_num_to_compensate \leftarrow 0, iter_count \leftarrow 0;
6 action_indices \leftarrow [add_index, remove_index, modify_index] \leftarrow [0, 1, 2];
7 action_weights \leftarrow [params.add_weight, params.remove_weight, params.modify_weight];
8 /*Perform at least a minimum number of iterations, and continue if removals need to be
compensated*/

9 **while** iter_count $<$ min_iter_num **or** removal_num_to_compensate $>$ 0 **do**

10 /*After the minimum number of iterations, add items to compensate removals; before,
perform a weighted random choice of the next action to do*/
11 action_index \leftarrow add_index;
12 **if** iter_count $<$ min_iter_num **then**
13 action_index \leftarrow weighted_random_choice(action_indices, action_weights);
14 /*Mutate with the selected action type*/
15 has_mutated \leftarrow False;
16 **if** action_index == add_index **then**
17 has_mutated \leftarrow mutate_with_addition(mutated_solution,
 params.max_add_attempt_num);
18 /*If the addition is successful, it can compensate a previous removal; also count as
compensated any failed additions after the base iteration limit, to avoid an eventual
infinite loop*/
19 **if** has_mutated **or** iter_count \geq min_iter_num **then**
20 removal_num_to_compensate \leftarrow max(removal_num_to_compensate - 1, 0);

21 **else if** action_index == remove_index **then**
22 has_mutated \leftarrow mutate_with_removal(mutated_solution);
23 /*If the removal is successful, it will need to be compensated*/
24 **if** has_mutated **then**
25 removal_num_to_compensate \leftarrow removal_num_to_compensate + 1;

26 **else**
27 has_mutated \leftarrow mutate_with_placement_modification(mutated_solution,
 params.subset('modify'));

28 /*If a mutation was applied and there is any possibility to select intermediate solutions as
the final ones, and the current intermediate solution is better than any previous one, keep a
copy of it*/
29 **if** has_mutated **and** params.intermediate_selection_prob $>$ 0 **and**
 get_fitness(mutated_solution) $>$ get_fitness(best_solution) **then**
30 best_solution \leftarrow mutated_solution.copy();
31 iter_count \leftarrow iter_count + 1;

32 /*If possible, select an intermediate solution as final with a certain probability if it is better than
the last mutation's solution*/
33 **if** params.intermediate_selection_prob $>$ 0 **and** mutated_solution \neq
 best_solution **and** best_solution \neq , solution **and** uniform_float(0, 1) $<$
 params.intermediate_selection_prob **then**
34 mutated_solution \leftarrow best_solution;

Algorithm 7: Surviving population selection (get_surviving_population)

Input : population /*Base population*/,
survivor_num /*Number of individuals that can survive*/,
pool_size /*Number of individuals per tournament pool*/

Output: surviving_population /*List of the individuals that survive*/

```
1 /*If the number of survivors is equal to or higher than the current population size, the whole
   population survives*/
2 surviving_population ← population if survivor_num ≥ population.num() else list();
3 /*Otherwise, select as many survivors as needed*/
4 if surviving_population.is_empty() then
5   for i ← 1 to survivor_num do
6     /*A tournament winner can survive*/
7     survivor ← get_tournament_winner(population, pool_size);
8     /*Add the survivor to the new population*/
9     surviving_population.append(survivor);
10    /*Remove the survivor from the old population, to avoid duplicity*/
11    population.remove(survivor);
```

It would be straightforward to incorporate complementary stopping criteria to the algorithm if it was needed for a certain context of usage. For instance, if used in a time-critical scenario, it would be interesting to add a constraint to stop the algorithm at the end of a generation if a maximum running time T_{max} had been elapsed since the start of the execution. Alternative additional stopping criteria may include a maximum number of fitness function evaluations, F_{max} , or stopping the algorithm as soon as one chromosome describes a solution with all the items placed in the container. However, not including the latter early stopping mechanism may eventually lead to a scenario that can be desirable in some contexts of usage: improving the tie-breaking properties of solutions with the same fitness value, by reducing the area usage or compacting items in a rectangular region with smaller area.

The general structure of the evolutionary algorithm is described in the pseudo-code of Algorithm 8. Some of the functions of the pseudo-code correspond to the algorithms presented in the previous pseudo-codes explained in other sections (e.g. for the generation of the initial population), and the recently introduced termination criteria is shown as well, as an early stopping check. It should be noted that all the numerous parameters that configure how the evolutionary algorithm works have been compressed as a single parameter structure (with each unique parameter as an accessible member) to preserve the readability of the pseudo-code.

Algorithm 8: Evolutionary algorithm

Input : container /*Container, described by its shape and maximum weight*/,
items /*List of items, described by their value, weight and shape*/,
params /*Structure containing the numerous parameters of the algorithm
(not listed individually for readability)*/

Output: solution /*Solution, describing a placement of items in the container*/

```
1 /*Generate the initial population*/
2 population ← generate_population(container, items, params.population_size,
3                                     params.initial_generation_item_specialization_iter_proportion);
4 max_fitness ← -∞;
5 iter_count_without_improvement ← 0;
6 /*Update the population up to a maximum number of generations*/
7 for i ← 1 to params.max_generation_num do
8     /*Select as parents the individuals that will generate offspring*/
9     parents ← select_parents(population, params.offspring_size,
10                            params.parent_selection_pool_size, params.can_use_crossover);
11    /*Calculate the minimum fitness for crossover non-best (alternative) solutions to be
12     accepted as additional offspring*/
13    crossover_min_fitness_for_non_best ←
14        get_crossover_min_fitness_for_non_best(population, params.can_use_crossover,
15                                              params.min_fitness_for_non_best_proportion);
16    /*Generate offspring with crossover and/or mutation*/
17    offspring ← generate_offspring(parents, params.subset(['mutation',
18                                              'crossover']) + [crossover_min_fitness_for_non_best]);
19    /*Define a temporary extended population by joining the original population and
20     their offspring*/
21    extended_population ← population + offspring;
22    /*Find the elite individuals among the extended population, sorted by descending
23     fitness value*/
24    elite ← get_fittest_solutions(extended_population, params.elite_size);
25    /*Check if the elite fitness has improved in this generation, to discard or contribute to
26     confirm the assumption of convergence*/
27    elite_fitness ← get_fitness(elite[0]);
28    if elite_fitness > max_fitness then
29        max_fitness ← elite_fitness;
30        iter_count_without_improvement ← 0;
31    else
32        iter_count_without_improvement ← iter_count_without_improvement + 1;
33    /*Stop early if the elite fitness is assumed to have converged*/
34    if iter_count_without_improvement ≥ params.converge_generation_num then
35        break;
36    /*Update the population to restore the standard population size (reserving places for
37     the elite)*/
38    population ← elite + get_surviving_population(extended_population -
39                                              elite, population_size - elite.num(), params.population_update_pool_size);
40    /*The final solution is the fittest one*/
41    solution ← elite[0];
```

4 Implementation

4.1 Technology and geometric checks

The algorithms described in Chapter 3 have been implemented in Python (version 3.6). Some features of this high-level interpreted programming language, such as dynamic typing and simple syntax, make the implementation of algorithms in Python fast and easily extensible, which was considered as a key point to be able to productively create different variants of the algorithm and hypothetically refine them with new ideas that may appear after applying the algorithms in different experiments. Nevertheless, other programming languages may offer significantly higher efficiency for some algorithms (Pre03), in terms of computational time, such as C++, that also supports the object-oriented paradigm. If the algorithms described in this work are used in the future in industrial applications, it may be recommendable to write a C++ implementation to obtain an efficiency gain.

Regardless of the fact that the algorithms designed to solve the Joint Problem have been implemented in Python, the most computationally expensive tasks, which are the geometric checks, rely on efficient implementations. The Python package Shapely is used to perform the geometric checks, e.g. determine whether two shapes intersect or a shape contains another one, internally using the Java Topology Suite, according to Shapely's user manual (Gil10). Shapely supports shapes that can be represented by a finite set of points, including polygons (triangles, squares, rectangles and any other polygon that is either regular or irregular, and convex or concave) and compound polygons, here called multi-polygons, described by an external polygon and a set of internal polygons acting as holes. For these shapes, Shapely provides all the geometric operations needed for this work, namely intersection checks and queries of whether a shape contains another one.

However, in this work it was planned to have other types of shapes that Shapely does not directly support, namely circles and ellipses. Custom shapes inheriting from the base geometry class of Shapely were created. To avoid defining the intersection and containing checks for the ellipse with any other shape, the ellipse was represented as a polygon with 66 equidistant points of the real ellipse, a number of points that the Shapely user manual claims to be sufficient to cover 99.8% of the area of a circle, and a similar (or at least not very different) figure is assumed for the ellipse. Nevertheless, this decision makes the usage of the ellipse slightly more limited in terms of precision than the other shapes, since the shape used for geometric calculations is an approximation of the real one.

The properties of a circle allow to define the intersection and containing checks with the rest of shapes in a straightforward way, so those operations were implemented for the actual circle, without discretizing it, as opposed to the case of the ellipse. The intersection check between a circle and any other shape, required to guarantee the validity of placements that include circles, is defined as follows:

1. Two circles intersect if the distance between their centers is not greater than the sum of their radii.
2. A circle intersects with a polygon if the minimum distance from the polygon to the center of the circle is not greater than the circle's radius.
3. A circle intersects with a multi-polygon if it intersects with any of the composing polygons, either the bounding one or a hole.
4. A circle intersects with an ellipse if it intersects with its approximation polygon.

The check of whether a circle is within another shape is needed when at least one item to place is a circle, and is defined as follows:

1. A circle is contained by another circle if the second one has a greater radius and the center-to-center distance plus the sum of the radii is lesser than the diameter of the second circle.
2. A circle is inside a polygon if the center of the circle is within the polygon and the minimum distance to the boundary of the polygon is greater than the radius of the circle.
3. A circle is inside a multi-polygon if it is inside of the boundary polygon and not intersecting with any hole.
4. A circle is within an ellipse if the circle is inside the polygon that approximates the ellipse.

For the situations in which a circle is the container of a problem, it is also required to define a check of whether a circle contains another shape:

1. A circle contains another circle if the previously explained circle-within-circle check is true, using opposite the order of container and content for the circles.
2. A circle contains a polygon if the distance between the center of the circle and all the vertices of the polygon is lower than the radius.
3. A circle contains a multi-polygon if the circle contains the boundary polygon of the multi-polygon.
4. A circle contains an ellipse if the circle contains the approximation polygon of the ellipse.

There is a single scenario in which a circle is discretized into a polygon, composed of points that are equidistributed throughout the circle's circumference, lying at the radius-matching distance from the center. Such approximation is used when searching the intersection points between shapes, with at least one of them being a circle, to avoid the need of implementing custom functions to find those intersection points with every shape (including polygons and multi-polygons). The intersection points are not needed in the Boolean check of whether two shapes intersect (which is commonly used in the algorithm), but in a less frequent operation, the type of mutation action that modifies the placement of an item by moving it in a random direction, detecting when an intersection would happen to prevent shape overlapping. Since it is an approximate operation, it is not required to have a very high number discretization points; 25 was the chosen value.

The area of the shapes is also required for the greedy algorithm, that takes it into account when choosing an item for placement. The area of polygons and multi-polygons is provided as a property of the geometric shapes in Shapely, while the area of the circle and the ellipse were defined using well-known formulae: the area of a circle is $\pi \cdot r^2$, where r is the radius; the area of an ellipse is $\pi \cdot a \cdot b$, where a and b are the horizontal and vertical radii of the ellipse, respectively.

4.2 Visualization

The visualization of a possible solution to an instance of the Joint Problem is comprised of two horizontally adjacent plots. The left plot depicts the placement of items that have been added to the container, which is represented in gray color, while holes and the outside are white. The right plot shows the items that have not been placed in the container, specifying the value and weight of each item. In both plots, the total value and weight of the depicted items (inside or outside of the container) is shown on the top, and the left plot also adds the

maximum weight (capacity) of the container. The items are shown in the same scale in both plots, allowing to attempt to visually detect whether an algorithm was able to place all items that it was geometrically possible to add (respecting the capacity), or there are some items that could have been placed in visually obvious spaces of the container. Another common feature of both plots is the fact that the items are colorized based on their profitability ratio, i.e. the result of dividing the value by the weight, in the color range depicted by the bottom color bar, that shows that the items with the highest ratio have an intense orange color, while the objects with the lowest ratio have a light gray color (close to light pink). In other words, the higher the ratio, the higher the orange color saturation. An example visualization of a problem in its initial state, with an empty container, can be seen in Figure 1, while a possible solution with some items in the container is depicted in Figure 2. The solution visualizations, and other plots of this document, were built using the Matplotlib plotting library ([Hun07](#)).

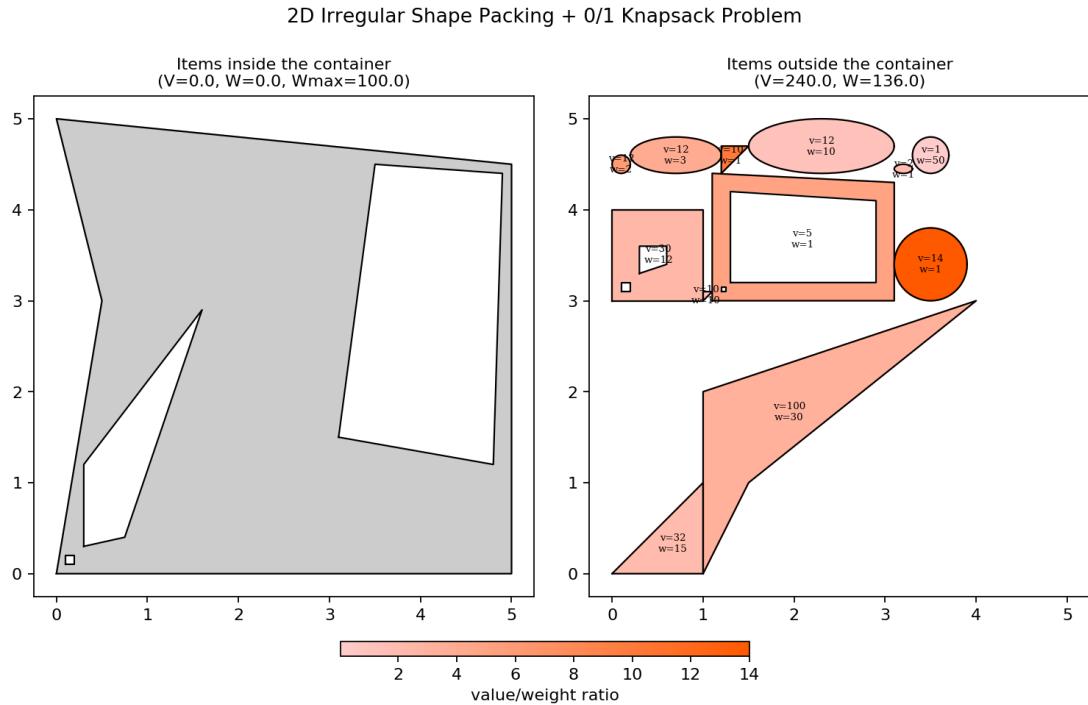


Figure 1: Example problem in its initial state, with all the items outside of the container.

The items within the container are represented at the position and rotation in which they were placed (or last moved or rotated to), only considering actions that were confirmed, to avoid generating to invalid solutions. The items which are outside of the container are arranged for visualization using a simple algorithm, that considers the bounding rectangle of items in the order in which they were added to the list of items. The space is divided in a set of rows of dynamic height, with the number of rows increasing in a logarithmic scale with respect to the total number of items in the problem, to achieve a reasonable distribution for visualization. The maximum width of the rows is defined as the result of dividing the sum of width of the bounding rectangles of all items outside of the container by the number of rows. The start position is (0, 0), where the bottom-left corner of the bounding rectangle of the first item outside of the container is placed, with the item itself lying within the bounding rectangle. Every remaining item is then

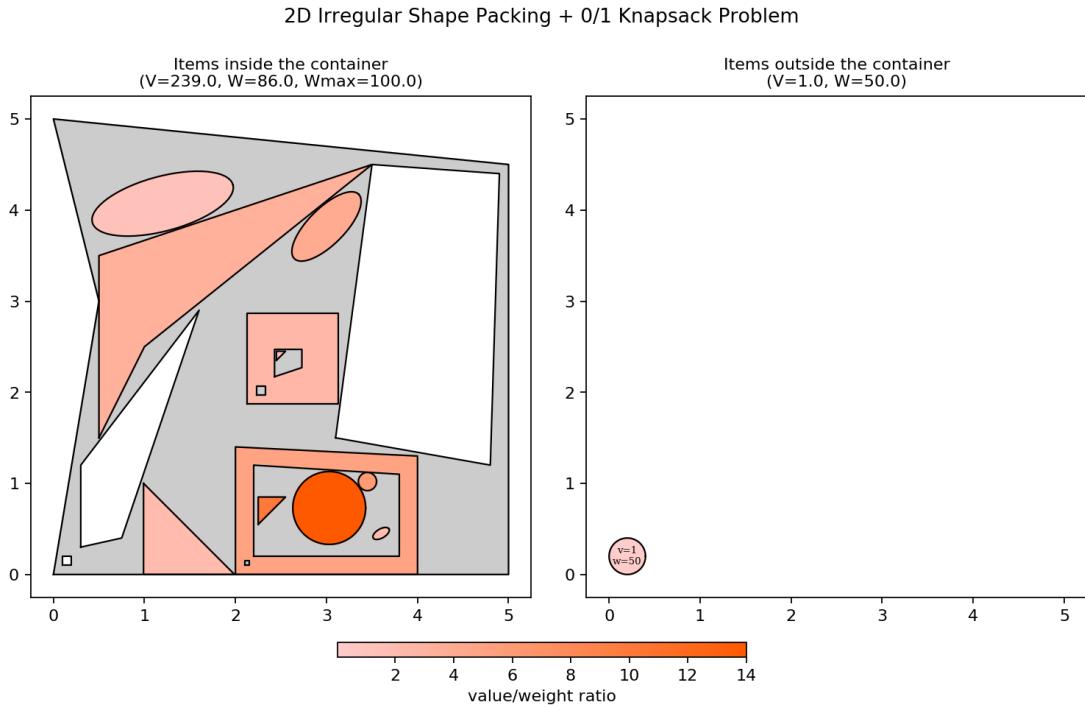


Figure 2: A possible solution to a problem, where all the items have been placed in the container except one, that would have caused the weight within the container to exceed the capacity.

placed adjacently after the horizontal end-point of the bounding rectangle of the previous one, until the maximum width of the row is reached. After that, the process continues in the row above the previous one, with the new row's minimum-height point being the maximum-height point among the bounding rectangles of the items from the previous row. It may be interesting to outline the fact that, in some problems of low complexity, particularly with a container much larger in area than the sum of all the items, the minimum-area bounding rectangle containing all the items in the visualization may eventually fit in the container, which would mean that it would be possible to perform a direct position mapping for the items from the outside visualization space to the container space, without any extra operation, as long as the total weight does not exceed the capacity. Items may be placed in descending order of profitability ratio, to have a greater total value. However, this trivial method is not expected to be useful (i.e. not causing intersections) in most real-world scenarios of certain complexity where the Joint Problem can be applied. Otherwise, it would be possible to design a specific algorithm to decide the way in which items are arranged as well as appropriate space distribution rules (e.g. to determine the number and width of rows), to try to follow the shape of the container to decrease the likelihood of causing intersections that would invalidate the attempt, and possibly considering bounding shapes more complex than a rectangle (e.g. a regular or irregular polygon) for greater compactness. However, increasing the complexity of the arrangement strategy may easily lead to a slower algorithm. Unless significant additional modifications would be added, the arrangement technique, that does not even take into account the position space of the container, would not be based in reasonably solid foundations, unlike the methods proposed in Chapter 3, that aim to be applicable to any instance of the Joint Problem.

4.3 Code availability

The code of this work is available in a public [repository](#)¹. It contains the implementation of the greedy, reversible and evolutionary algorithms, as well as the problem instances explained in Chapter 5 and Section 8.2, and the configuration to perform the experiments. In fact, data structures summarizing the results of the experiments described in Chapters 6, 7 and 8 are also present, as well as plots and figures, some of which were not shown in this document.

¹<https://github.com/albert-espin/knapsack-packing>

5 Joint Problem Dataset

5.1 Context, Goals and Design Principles

The joint problem of the Two-Dimensional Irregular Shape Packing Problem and the 0/1 Knapsack Problem (shortened in this document as Joint Problem for the sake of brevity) has not been explored previously in the literature, to the knowledge of the author of this work. Therefore, there is a lack of existing problem examples to which the new proposed algorithms can be applied, which would allow to evaluate the quality of their solutions and their time efficiency.

According to the definition of the Joint Problem, a problem instance must be defined with a container (specifying its maximum weight and shape) and a set of items (with the value, weight and shape of each of them). Different datasets exist for the 0/1 Knapsack Problem and for the Two-Dimensional Irregular Packing, but they lack part of the information needed for the Joint Problem: the former do not have shapes (for the container and the items), while the latter do not include any notion of value or weight. It would be possible to add the missing information to existing problems, either by manual annotation or through algorithmic imputation. If the programmatic option was chosen, a possible approach would be to define the capacity of the container randomly within a fixed range, and then generate the weight of each item following a chosen distribution, scaling it to any desired range, e.g. a combination of distribution and scaling that on average would produce items with a weight sum roughly equal to the container's capacity, with a certain standard deviation fixing how far the weight sum can be from the capacity (either exceeding it or not reaching it). Subsequently, it would be possible to determine the value of the items, either by using a completely independent distribution and scaling method or using a mathematical function that takes the weight as a parameter and outputs the item's value.

When evaluating the quality of the solutions produced by an algorithm for a specific problem, it is very useful to know the optimal solution beforehand, because comparing it to the solution of the algorithm allows to empirically determine how precise and effective the method is. Unfortunately, transforming Packing Problem examples into instances of the Joint Problem has the additional effect of changing the optimal solution of each problem, which is unavoidable, since the two Problems have different criteria that define what makes a solution to be optimal. In the Joint Problem, the optimum solution maximizes the sum of item values in the container (respecting non-intersection and weight constraints), while the Packing Problem does not have an explicit notion of value. It would be possible to assimilate an existing property as the value of items, such as the area of their shape, but such approach would clearly introduce a bias, because both fields can be completely unrelated in real problems, and therefore the experimental results would not represent a general scenario. Alternatively, in some simple problems it may be easy to manually determine the optimal solution by observing the properties (shape, weight, etc.) of the items and the container, but the same task can be very time-consuming for a human for complex problems, especially if they have many items.

A manually designed dataset, the Joint Problem Dataset, was created for this work. The motivation behind this decision was to have a set of problem examples specifically designed with the particularities of the Joint Problem in mind (namely the interaction of features such as weights, values and shapes), with manually obtained optimal solutions. Creating a dataset for a new problem from scratch allows to represent examples that showcase a variety of situations that are expected to be challenging for the algorithms, and therefore it is interesting to discover how the methods tackle these circumstances. It would be less likely to find some of the truly interesting cases if the problems were randomly generated, or even if some features were automatically imputed, such as value and weight for examples adapted from the Packing Problem.

Therefore, the Joint Problem Dataset contains examples that have been considered as hypo-

thetically challenging for the algorithms, and useful to detect potential weaknesses, or confirm suspected strengths. For this reason, the level of difficulty of most of the designed problems is not trivial, to avoid that all algorithms find optimal solutions very easily without showing relevant differences in the quality of results. These characteristics of the problems in the Joint Problem Dataset, along with the presence of a manually found optimal solution, make these examples suitable to test the algorithms and compare the quality of their results (between the proposed techniques and the optimal solution), as well as their time performance. These experiments are studied in Chapter 7, using the final configuration of parameters for each algorithm, which are chosen after different (potentially interesting) configurations are compared and the best is chosen. The experiments and results of the parameter optimization phases are presented and analyzed in Chapter 6.

Manually determining the optimal solution for each problem was possible by constraining the number of items (that ranges from 5 to 20 depending on the problem), since each item needs to be observed. Firstly, it is important to determine if the item can fit in any region of the container, and some problems have very large items that can be easily discarded, while most of the other items can be placed, although the ranges of positions and rotations that are feasible for each item are not always trivial to visually approximate. The maximum weight accepted by the container was adapted to ensure that some items would be necessarily left outside in optimal solutions, either because of having a very high weight (with very low profitability ratio, obtained by dividing the value by the weight), or the lowest value (as well as the lowest value-weight ratio). Since the Joint Problem uses the continuous space for some features (such as position and rotation) it is not feasible to demonstrate that the proposed solutions are optimal, but it should be visually intuitive to understand that they are indeed optimal after carefully analyzing each example. It was ensured that all the proposed solutions respect the constraints of the Joint Problem, by not presenting any intersection between items or between an item and the container, and not exceeding the capacity of the container. In fact, every placement position and rotation that was thought of while constructing solutions was tested in the implementation, and, if failed, other values were tried or eventually the placement strategy was planned again. Defining the problems manually and finding an optimal solution for them was expected to be (and proved to be) time-consuming. In consequence, the planned number of problems was 10, which is not very large.

Since the number of problems is relatively small, the diversity of the examples was a priority, to ensure that many interesting scenarios were represented. There are different notions and properties that define problems that, by being different from one example to another, make the dataset diverse:

1. Problem difficulty, which is the (subjective) perception of the author of how much human effort and time is required to obtain an optimal solution for a problem, being convinced that it is optimal.
2. Number of items in the problem.
3. Number (or percentage) of items within the container in an optimal solution.
4. Percentage of the value sum of all items that can be placed in the container in an optimal solution.
5. Percentage of the weight sum of all items needed to reach the capacity of the container.
6. Percentage of the area sum of all items needed to reach the area of the container.
7. Distribution of the value of items.

8. Distribution of the weight of items, and its similarity or distance to the container's capacity.
9. Distribution of the value-weight ratio of items.
10. Distribution of the area of items, and its similarity or distance to the container's area.
11. Distribution of the types of shapes of items, and the homogeneity or diversity when the type coincides.
12. Container's capacity.
13. Container's area.
14. Container's shape.
15. Container's capacity saturation, i.e. the percentage of the maximum weight that is used in an optimal solution.
16. Container's space saturation, i.e. the coverage of the container's area by items in an optimal solution.

It should be noted that problems can theoretically have infinite optimal solutions with a same set of items placed in the container, with different values of position coordinates and rotation angle, which are continuous numbers. For each of the designed problems, only one optimal solution is presented, with a specific set of items placed in the container, but it is plausible that optimal solutions with different placed items can exist (by providing the same value in the container), namely in problems with cloned items where not all the clones can be placed in the container, due to space or weight constraints. Therefore, it is important to know that the information provided about the optimal solutions of the examples in Section 5.2, only reflects the reality of a single optimal solution, with a set of placed items, that is not necessarily the only plausible set.

5.2 Description and Analysis of the problems

The particular problems of the Joint Problem Dataset are presented in ascending order of item number, and named with the ordinal of such sequence, starting by 1 and ending in 10. For each example, a visualization of the initial state of the problem (with the container still empty) is shown, as well as the configuration of placements of an optimal solution. At the end of this section (as explained later) one can find a table with information of every problem and the presented solution, that supports and complements the problem descriptions.

Problem 1, shown in Figure 3 defines a circle-shaped container, whose capacity (120), matches the sum of weights of the 5 items. In this problem, it is visually intuitive that all items fit in the container, as long as the largest item, which is a square (with the highest value and profitability ratio), is placed approximately in the center of the container, and the other smaller items, circles of different profitability ratios (ranging from 0.25 to 1), are placed next to the center of each side of the square. Such configuration of placements leads to the optimal solution, shown in Figure 4.

In the case of Problem 2, depicted in Figure 5, the container is an hexadecagon (a regular polygon of 16 sides), while 4 of the items are multi-polygons whose exterior and hole also have hexadecagonal shape, with different sizes and value-weight ratios, with the largest of the items having the highest value (25), while the other objects have smaller value and value-weight ratio. The fifth and smallest item has the lowest value, and it is a circle. The capacity of the container is 100, while the sum of item weights is 70, meaning that the capacity constraint

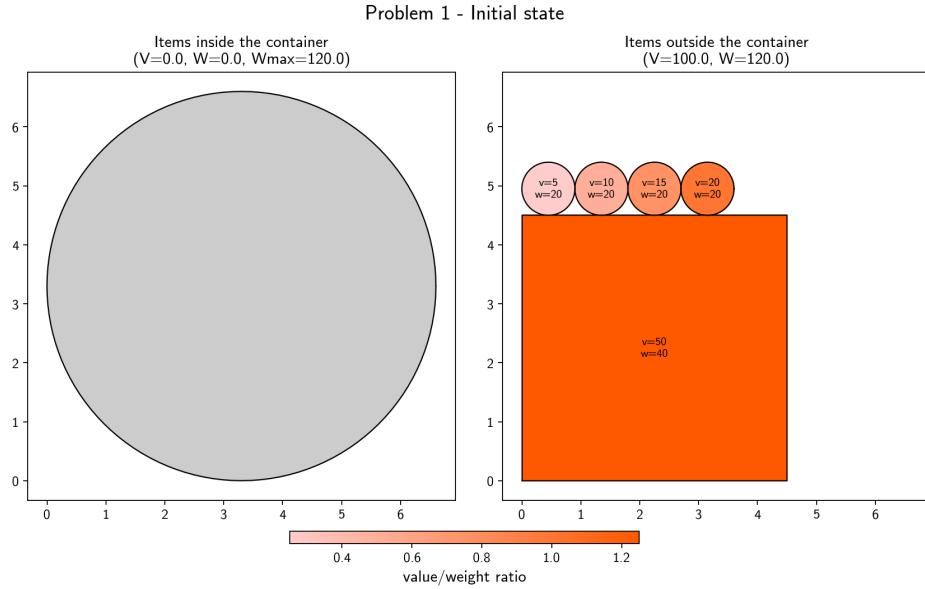


Figure 3: Initial state of Problem 1.

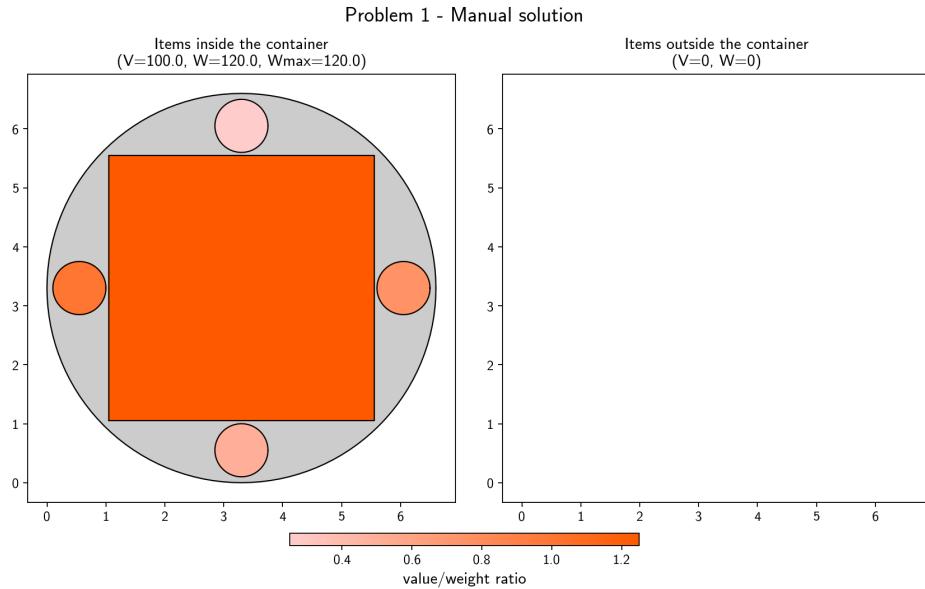


Figure 4: Optimal solution of Problem 1.

cannot invalidate any solution attempt in this example. Moreover, it is visually obvious that all items can geometrically fit in the container, if they are placed approximately in the center of the circle-shaped container, in such way that each multi-polygon contains the one with immediate smaller size inside its hole, and the small circle item is placed inside the hole of the smallest multi-polygon, as exemplified in Figure 6.

Problem 3, depicted in Figure 7, has a square-shaped container whose capacity (32) is smaller

Problem 2 - Initial state

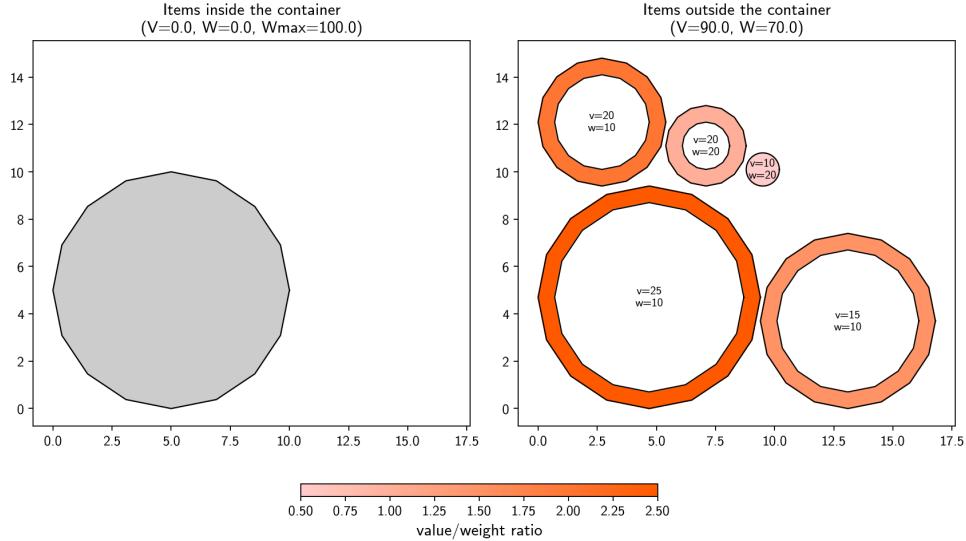


Figure 5: Initial state of Problem 2.

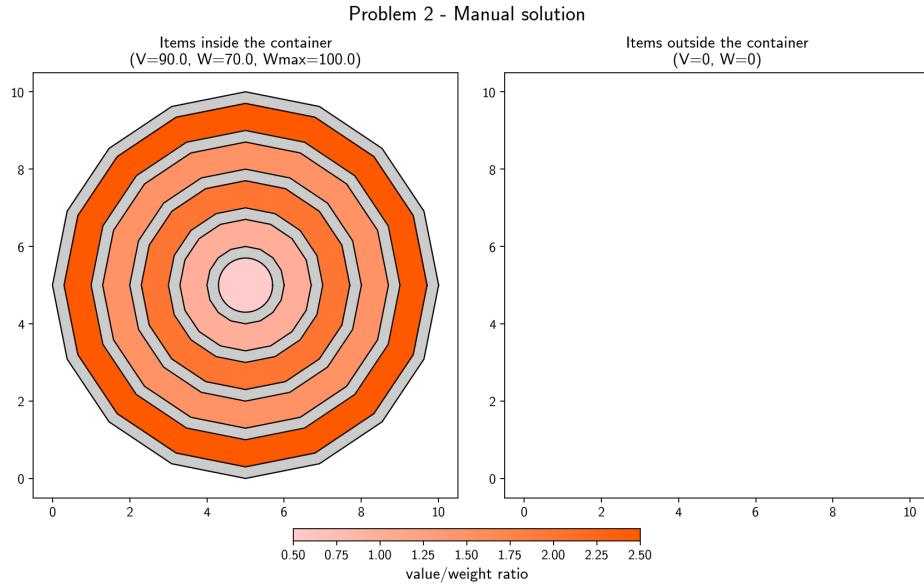


Figure 6: Optimal solution of Problem 2.

than the sum of item weights (50), so not all items can be accepted in the container in the optimal solution, shown in Figure 8. Specifically, there are two right-angle triangles with the highest value and profitability ratio (one of them tied with another item) that should be placed in the container, contributing with a combined value of 30 and a weight of 20. From the remaining items, which are 4 ellipses, it is visually clear that all would geometrically fit in the container,

but only the two with greater value and profitability ratio can be selected in the optimal solution, since their weight is 5 for each of them, making the total weight in the container 30, with none of the other items having a weight smaller or equal than 2, which would make the capacity to be reached.

Problem 3 - Initial state

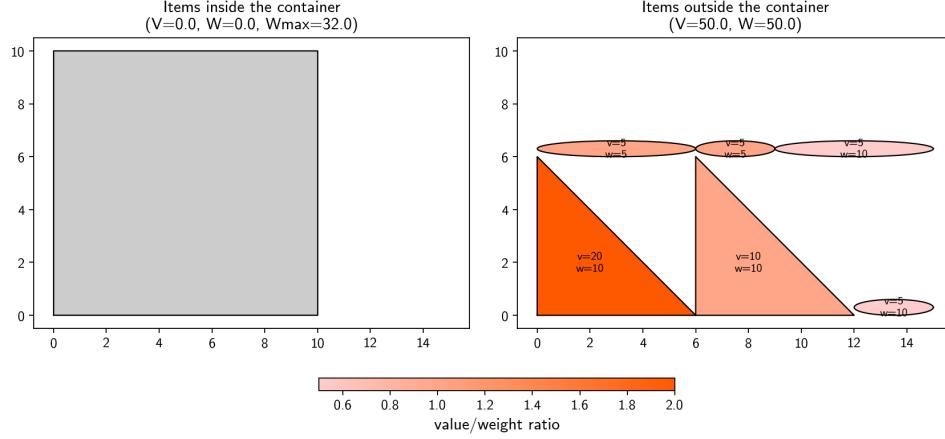


Figure 7: Initial state of Problem 3.

Problem 3 - Manual solution

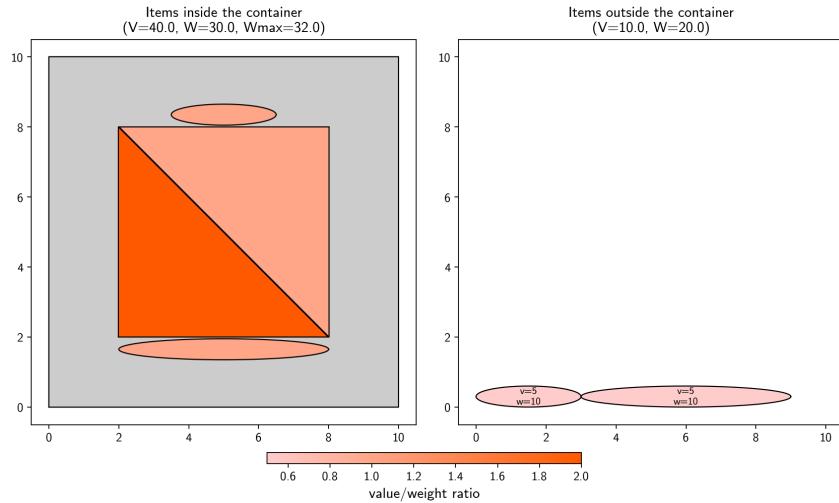


Figure 8: Optimal solution of Problem 3.

Figure 9 represents Problem 4, where the container is an ellipse, and there are 4 ellipse-shaped items and 5 circle-shaped items. Despite of the large amount of space in the container, not all items can be added inside, since the capacity (50) would be exceeded, since the sum of item weights is higher (52). It is possible to see that if items are placed in descending order of value one can obtain the optimal solution, by only leaving out an ellipse with a value of 2 and a

weight of 7. Therefore, the weight in the container in the optimal solution (45) is smaller than the capacity, as shown in Figure 10.

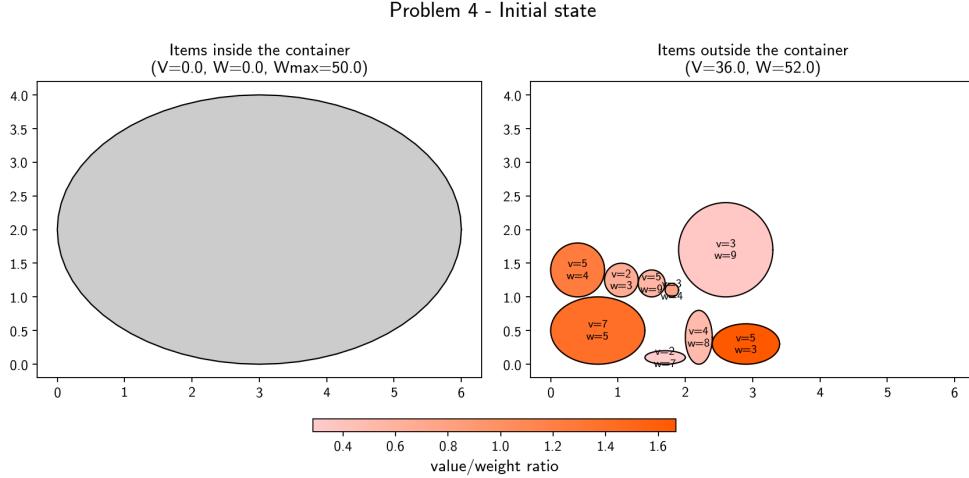


Figure 9: Initial state of Problem 4.

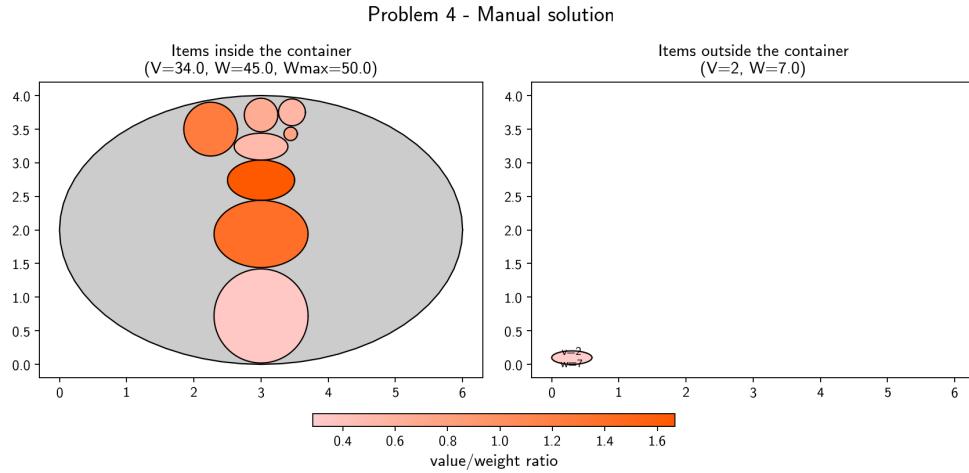


Figure 10: Optimal solution of Problem 4.

In the case of Problem 5, depicted in Figure 11, the container is a multi-polygon, whose exterior shape is a irregular polygon of five sides, with three holes, two of them of considerable area, reducing the usable space and making it more difficult to place items. This example has a great diversity of item shapes, namely all the accepted types: there are 3 circles, 3 ellipses, 4 polygons (3 right-angle triangles and a 4-side irregular polygon) and 2 multi-polygons (with two four-sided holes each). After observing the geometry of the container and the items, it can be seen that all items would fit in the container, and there is enough space to make it not compulsory to place the smaller items inside multi-polygons, even if it is plausible. It is easy to understand that the two largest items, a polygon and a multi-polygon, only fit if their reference

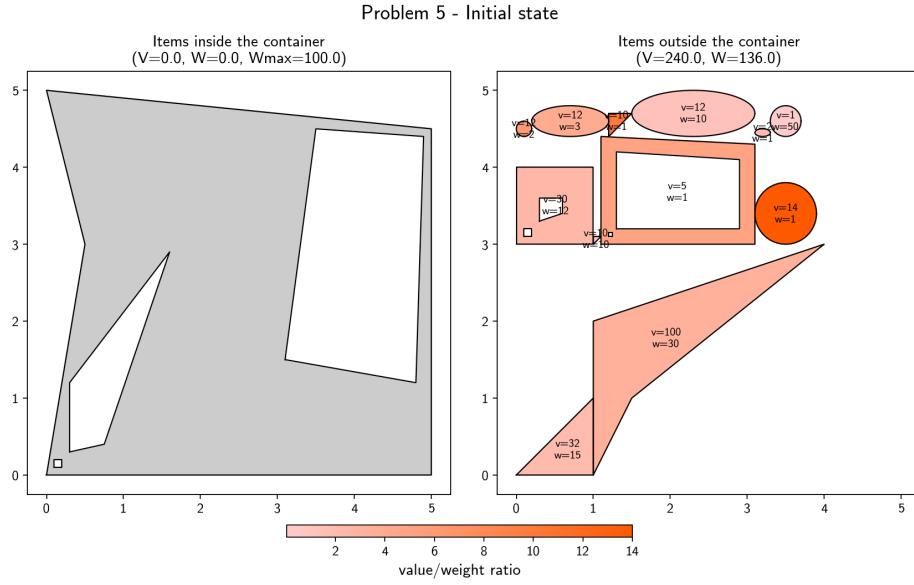


Figure 11: Initial state of Problem 5.

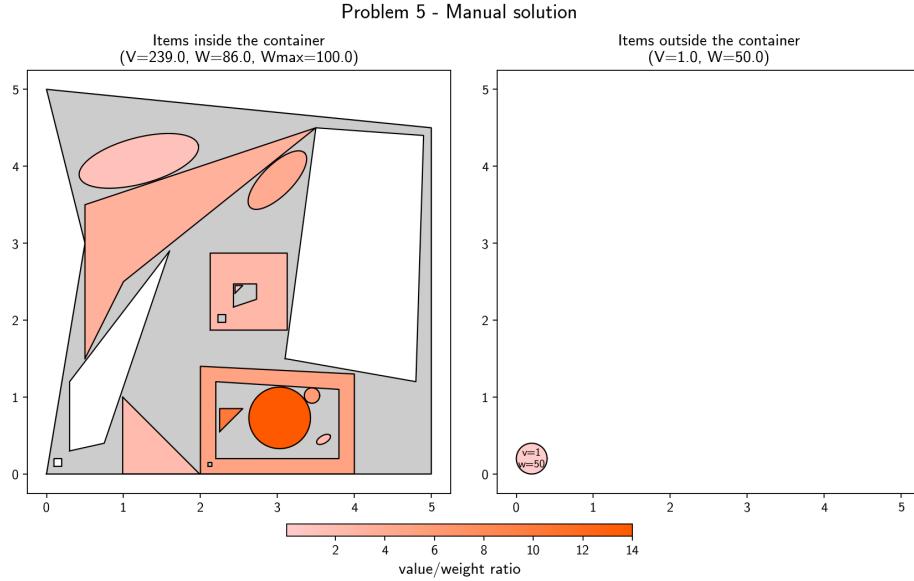


Figure 12: Optimal solution of Problem 5.

position lies within limited zones of the container's space, with specific ranges of rotation angle. Despite of the existence of enough space, the weight limit of the container is 100, while the items have a weight sum of 136, so not all items can be placed inside the container. Since there is an item that has both the lowest value (1) and highest weight (50), it is clear that it is the one to leave outside to obtain an optimal solution, as shown in Figure 12.

Problem 6, shown in Figure 13, has a multi-polygon container where the external boundary

and the 3 inner holes are all regular quadrilaterals (3 squares and 1 rectangle). One can see that there is a diverse set of 13 items: 8 polygons (all irregular, with side number ranging between 3 and 5), 3 circles and 2 ellipses. It is visually obvious that 2 of the polygons do not fit in any position of the container, so their weight (16) can be ignored, and makes it possible to place all the other items in the container, with a weight sum (145) that stays below the capacity (150). The optimal solution is shown in Figure 14.

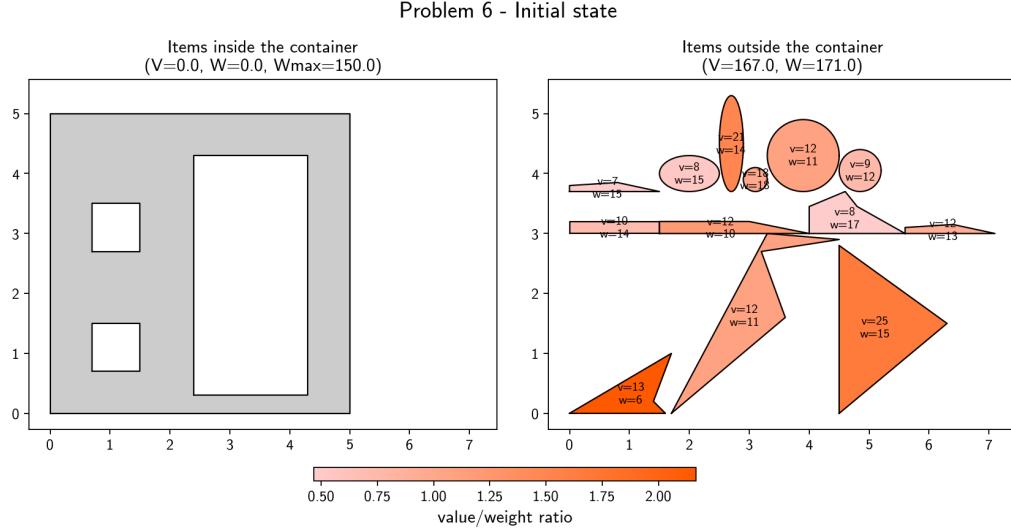


Figure 13: Initial state of Problem 6.

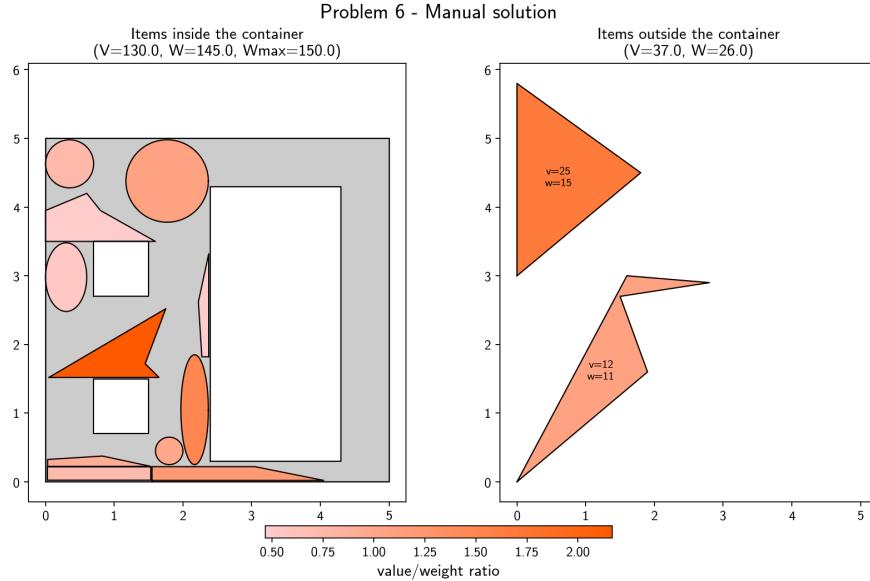


Figure 14: Optimal solution of Problem 6.

Problem 7, shown in Figure 15, features a twenty-sided irregular polygon as its container,

and a total of 14 items, including 2 single-hole multi-polygons, 6 irregular polygons, 3 ellipses and 3 circles. It is intuitive to see that the largest ellipse cannot fit in the container, so that the total item sum, ignoring it, is 140, while the capacity of the container is 122. In this problem an optimal solution can be obtained by placing items in descending order, which leads to leave outside the least valuable item, a circle, as shown in Figure 16.

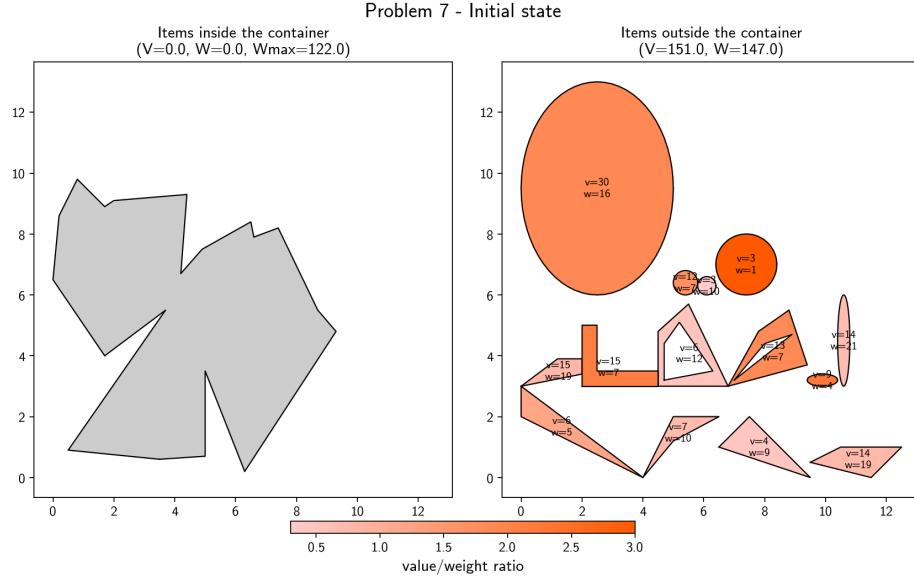


Figure 15: Initial state of Problem 7.

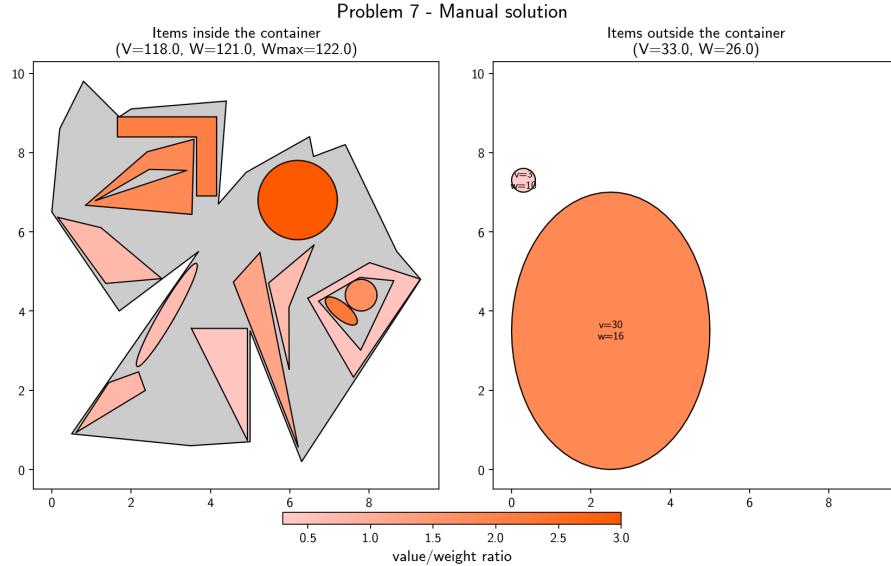


Figure 16: Optimal solution of Problem 7.

Problem 8, as one can see in Figure 17, has a symmetric eight-sided irregular polygon as

its container, and a total of 15 items, including 8 irregular polygons, 4 ellipses and 3 circles. The interesting aspect of this problem lies in the distribution of value and weight among items. Namely, a single item, a polygon resembling the "W" letter (or "M" if rotated 180 degrees), is more valuable than all other items together. Therefore, in the optimal solution shown in Figure 18 the most valuable item is placed first, but then the weight limit is reached, so no more objects can be added, but the solution is obviously optimal.

Problem 8 - Initial state

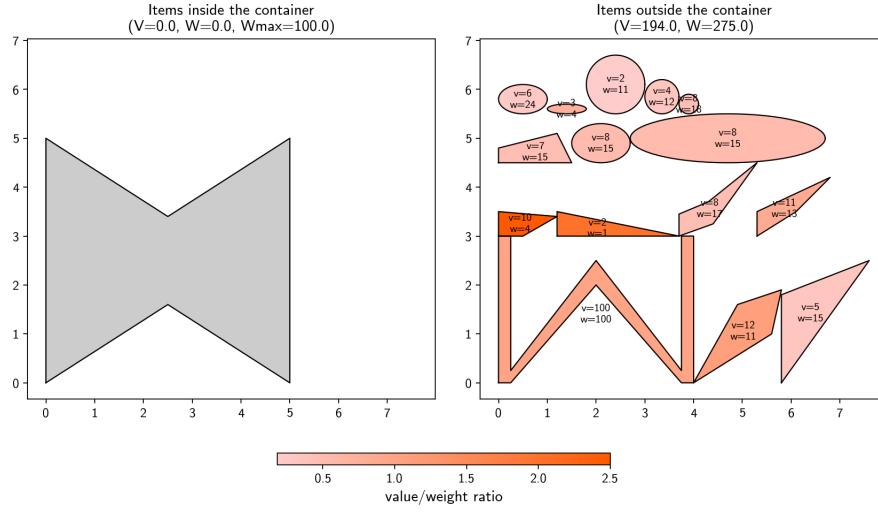


Figure 17: Initial state of Problem 8.

Problem 8 - Manual solution

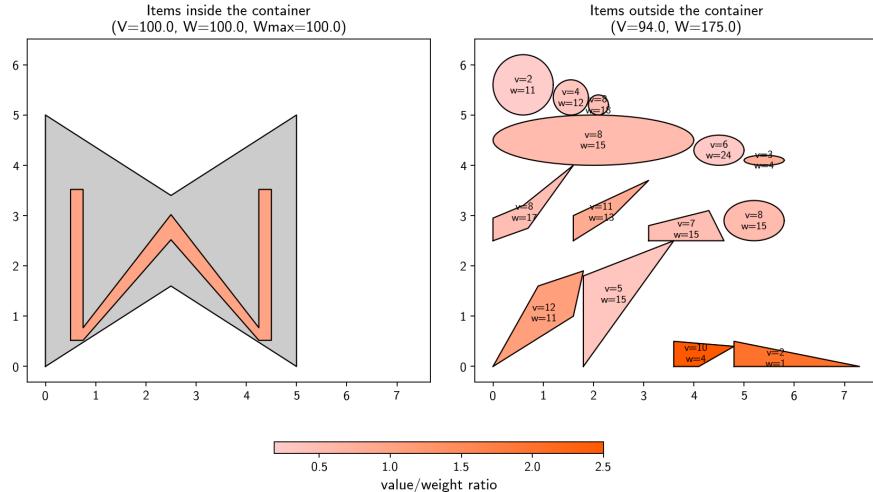


Figure 18: Optimal solution of Problem 8.

In Problem 9, shown in Figure 19, the container is a regular dodecagon, that is not large

enough to make it possible to place all the available items, some of which have a considerable area, such as a square and a multi-polygon whose exterior shape is an octagon while its hole is square-shaped. These two items have values which are significantly higher than the rest of items (80 and 110, while the next one has a value of 15). When they are placed in the container, none of the other items have room left, even if the weight limit is not reached. None of the combinations of the rest items (13 in total, 6 irregular polygons, 4 circles and 3 ellipses) would provide a higher total value, and if only of the two most valuable items are placed, there is no space for other items to sum more value than with the hypothetically removed item. The optimal solution is presented in Figure 20.

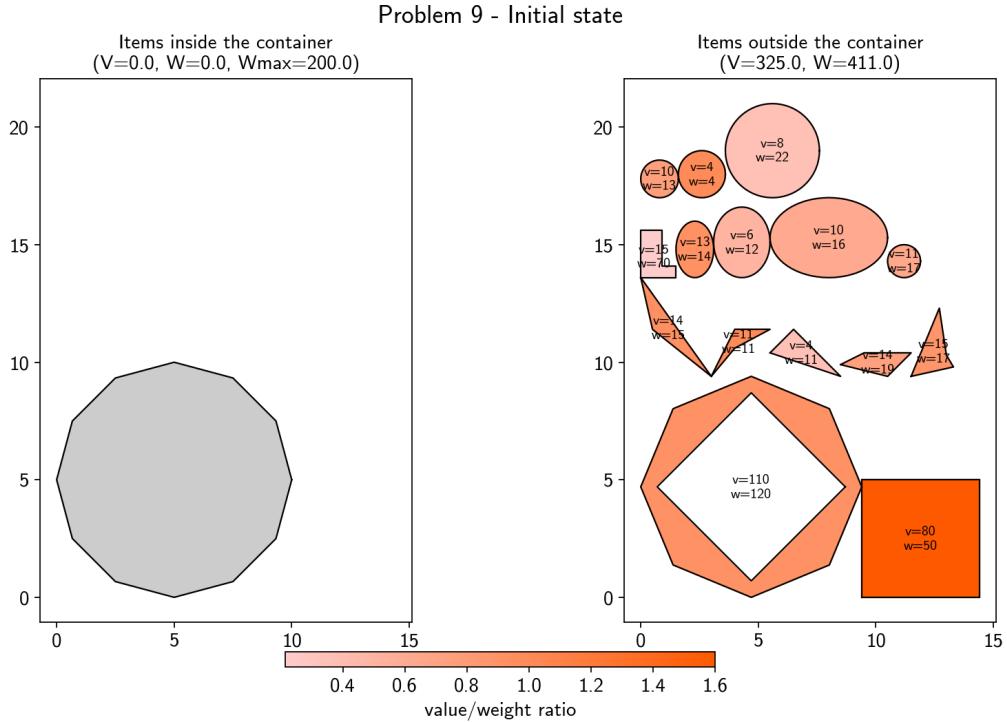


Figure 19: Initial state of Problem 9.

In the case of Problem 10, depicted in Figure 21, the container is a polygon whose shape resembles a cross. There is a large number of items (20), but they are actually repetitions of 4 different item templates. 4 of the rectangles, belonging to two different templates, are the most valuable objects and have the smallest area. The rest of items are 10 right-angle triangles and 5 rectangles, and they have interesting properties: the area of two of the triangles is approximately equal to one of the rectangles, but since the triangles have the same value and half of the weight of the rectangles, the optimal solution is obtained using the triangles (put together in pairs resembling a rectangle when combined) instead of the mentioned 5 rectangles, since there is no space for them, and the capacity of the container is exactly reached. In the optimal solution, shown in Figure 22, the distances between the items of this problem, and between the items and the container, are much smaller than in any other example. Due to these circumstances, it was expected to be extremely difficult (or even impossible) for the designed algorithms to find an optimal solution.

Table 1 presents statistical information of the problems, providing values for each problem

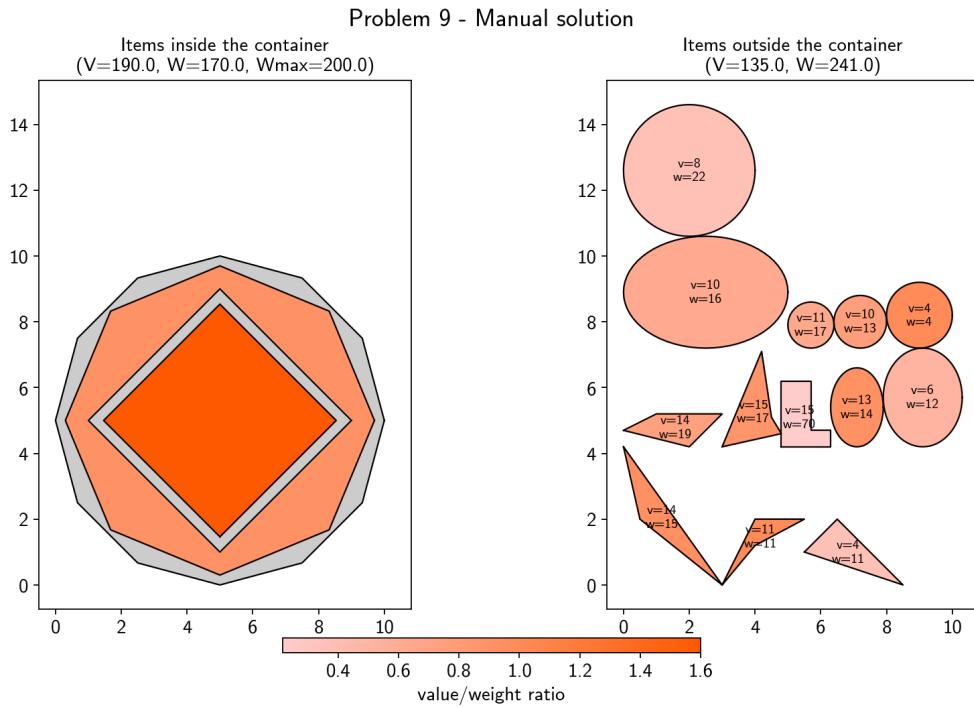


Figure 20: Optimal solution of Problem 9.

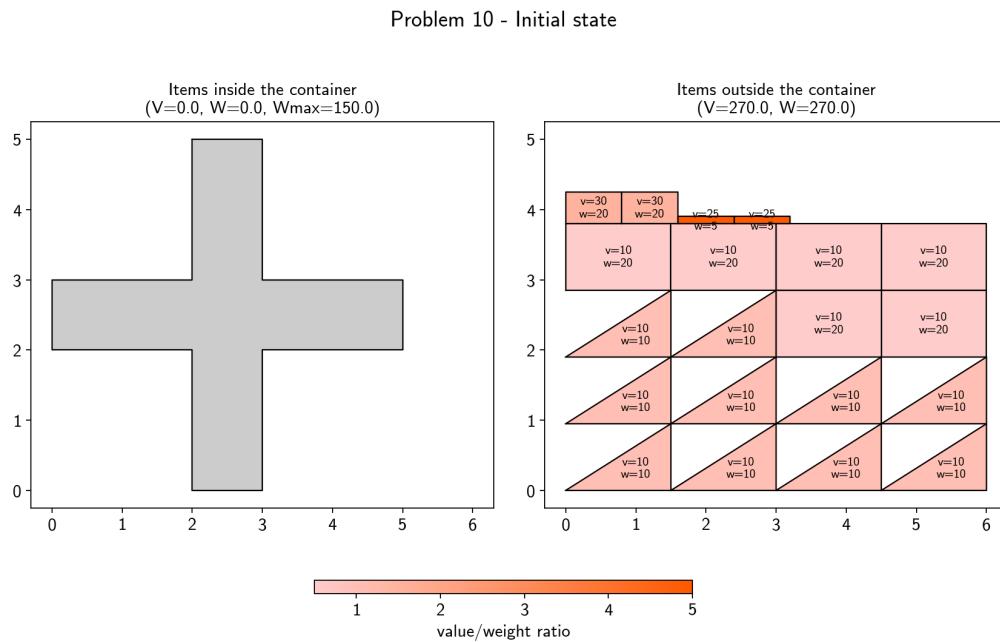


Figure 21: Initial state of Problem 10.

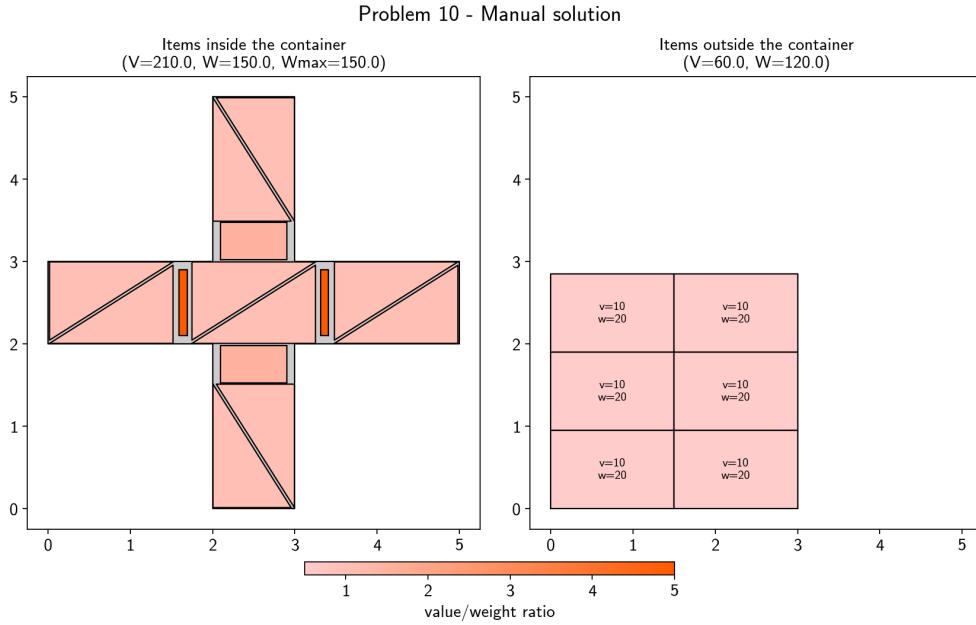


Figure 22: Optimal solution of Problem 10.

individually, as well as the minimum, maximum, mean and standard deviation considering all the problems. The last row corresponds to the percentage of the length of the range of values (between the minimum and the maximum) of each field represented by the standard deviation, i.e. it represents the standard deviation in the same range for all fields (between 0% and 100%). The calculation is aimed to be an illustrative measure of the variability of each of the fields, that are conceptually linked to some of the properties explained in Section 5.1, that the author wanted to make considerably diverse among problems. The percentage is near to 30% or higher than such value for all fields, which implies that a moderately high level of variability was effectively obtained, as wanted. The specific fields shown in the table are:

- Item number. It ranges between a small number (5) and a quite higher number of items (20), while the average (11.4) represents a medium number of items.
- Percentage of items placed in the container in the presented optimal solution. In some problems all the items can be placed in the container (100%) while in others only one or few of the many items can be placed (such as Problem 8 and Problem 9, with 6.67% and 13.33% of the items placed, respectively), while on average more than half of the items are placed (70.76%).
- Percentage of the value sum of all items that can be placed in the container in the optimal solution. In the problems where all items can be placed the value is maximum (100%), but in some cases most of the value needs to be left outside of the container, due to weight or geometric constraints. In Problem 8, for instance, only a bit more than half of the total value, 51.55%, can be placed. On average, a high majority of the items (81.78%) can be placed in the container.
- Percentage of the weight sum of all items needed to reach the capacity of the container, which exceeds 100% if the total weight is higher than the capacity. There are problems where the sum of the weight of all items is quite smaller than the capacity (70% in Problem

2 is the minimum), while in others the capacity is significantly surpassed (275% in Problem 8 is the maximum). In most examples, the capacity constraint needs to be considered when solving the problem (with an average of 146.12%).

- Percentage of the area sum of all items needed to reach the area of the container (it can also exceed 100%). The percentage ranges from 22.88% in Problem 4, where the area of the container is much larger than the sum of the area of every item, to 183.94% in Problem 10, where the presence of many items increases the total area, greater than that of the quite narrow container. In other cases, such as Problem 9, with 144.01%, the presence of few very large items is enough to surpass the area of the container. On average, the area of all items is quite close to that of the container (79.67%).
- Percentage of container's weight saturation in the optimal solution. In most problems, the optimal solution uses most of the weight allowed by the container (92.06%), in some cases using the full capacity (100% in Problems 1, 8 and 10), even though there are cases with significantly lower weight pressure (such as 70% in Problem 2).
- Percentage of container's space saturation in the optimal solution. There is no example where the exact whole area of the container is used, which would be (almost) practically impossible for any of the proposed algorithms to solve, due to the random determination of position coordinate values, where the valid options would be potentially reduced to a single value instead of a range. Nevertheless, a container saturation of 88.04%, as seen in Problem 10, means that very little space is left in the container, and it is expected to be very difficult for the algorithms to solve it optimally. The opposite case is that of Problem 8, where most of the space (all but 16.54%) is empty in the optimal solution. On average, almost half of the area (46.06%) is covered by items in the presented optimal solutions.

Table 1: Statistics of the problems of the Joint Problem Dataset and their optimal solutions.

	Item num.	Opt. % item num. in cont.	Opt. % item value in cont.	Item weight % of max weight	Item area % of max area	Cont. weight saturation %	Cont. area saturation %
Problem 1	5	100	100	100	66.63	100	66.63
Problem 2	5	100	100	70	65.85	70	65.85
Problem 3	6	66.67	80	156.25	44.48	93.75	40.24
Problem 4	9	88.89	94.44	104	22.88	90	22.38
Problem 5	12	91.67	99.58	136	40.15	86	39.39
Problem 6	13	84.62	77.84	114	56.52	96.67	30.71
Problem 7	14	85.71	78.15	120.49	87.1	99.18	35.91
Problem 8	15	6.67	51.55	275	85.16	100	16.54
Problem 9	15	13.33	58.46	205.5	144.01	85	73.97
Problem 10	20	70	77.78	180	183.94	100	88.94
Min	5	6.67	51.55	70	22.88	70	16.54
Max	20	100	100	275	183.94	100	88.94
Mean	11.4	70.76	81.78	146.12	79.67	92.06	48.06
Std	5.02	33.85	17.15	60.44	49.46	9.67	24.13
Std/(max-min)%	33.44	36.27	35.39	29.48	30.71	32.23	33.33

6 Parameter Configuration and Optimization

6.1 Motivation and Goals

The objective of this chapter is twofold. Firstly, to justify the default values for parameters selected for the proposed algorithms. Secondly, to perform an optimization of those values, to the possible extent, in the attempt of making all algorithms to compete in fair conditions in the experiments of Chapter 7. It should be noted, however, that it is not feasible in time to try a very extensive number of parameter values, especially in the case of the evolutionary algorithm, that has a significantly greater number than the others, and it is also slower. In the case of the greedy and reversible algorithms, which are faster, it is actually feasible to test multiple values for most parameters. The experiments produced a considerable number of tables; to facilitate readability, only the most relevant ones are shown in this chapter, while all the other ones are present in Section A.3, which belongs to the Appendix.

In all the optimization tests, the experiments were run 10 times to ensure that a reasonable level of statistical significance was obtained in the results. To make the experimentation faster, multi-processing was used, in such way that each of the 10 runs was executed in a different process. The experiments were performed in a machine with 6 CPU's, so each of the last 4 runs did not start until one of the initial ones had finished. The usage of multi-processing does not affect the quality of the solutions, but it was observed to increase the average time of a run, even if the total time among all runs decreases in a significant way due to the parallelism strategy, due to the overhead of creating and managing processes. Therefore, the execution times reported in this section should be given less confidence than those of Chapter 7, where the final configuration of each algorithm is tested in a single process. Nevertheless, the order of magnitude of the execution times do not change by using multi-processing, so if an algorithm is much faster than another when not dividing runs in multiple processes, the difference should be still qualitatively clear when using multi-processing, even if the reported quantities are not reliable if interpreted as exact values. To avoid increasing the time variability of using multi-processing, no external computationally expensive tasks were performed while running the experiments.

6.2 Greedy algorithm

6.2.1 Experimental Methodology

The behaviour of the greedy algorithm when it comes to selecting items in a weighted random manner is determined by the parameters of the greedy score function defined in Equation 2. One of these parameters is the value weight of the formula, K_v . If it is high, it gives a higher score to items with a higher value, independently of their area and weight; if it is low, it gives more importance to the other term of the formula, with the profitability ratio and the area of the item. The other parameter of the function is the area weight, K_a : if high, it penalizes more severely items with high area; if low, it mainly focuses in the profitability ratio of an item to determine the score. Both weights are expected to be between 0 and 1, and it is interesting to see how values in the interval affect the quality of the solutions found by the algorithm. Therefore, in the first phase of parameter optimization of the greedy algorithm, grid search was applied to find the best combination of weights among the following equidistributed values: 0, 0.25, 0.5, 0.75 and 1. Regarding the number of iterations, the initial configuration was used, with a maximum of 1000 iterations and 300 iterations to assume convergence. Such configuration provided better results in preliminary tests than other alternatives with values in the same order of magnitude. Smaller values brought a significant decrease in the solution quality, measured as the value in the container.

Additionally, as a second phase of parameter optimization, using the weights of the greedy score function resulting from the first phase, it is interesting to determine if different orders of magnitude for the maximum number of iterations and the convergence number of iterations produce significantly different results. The original setup was compared with iteration numbers 10 and 100 times greater, keeping the same proportion between the convergence number and the maximum number of iterations. In other words, a maximum of 10000 iterations and 3000 convergence iterations were checked, as well as using 100000 iterations as maximum and 30000 to assume convergence. If the difference in terms of solution quality between two of the tested configurations was not significant, the fastest one should be selected as final; since the iteration number is proportional to time, the total execution time was expected to be significantly different from one configuration to another (longer for those with more iterations). It is not expected that the iteration values and the greedy score parameters have strong common influence or interactions, since they affect different notions of the algorithm, so it was considered a valid approach to first determine the value of the greedy score function weights and then the iteration values. Furthermore, the presence of a configuration with 100 times more iterations than the original setup would have made a combined test much more slow, due to the number of combinations that would have been tested.

6.2.2 Results and Discussion

Table 11 (placed in Section A.3.1 of the Appendix due to its large size) presents the solution quality results (expressed with a mean over 10 runs and the standard deviation) of the 25 tested combinations of values for the value and area weights of the greedy score formula. A maximum of 1000 iterations was used, and 300 to assume convergence, as defined for the first optimization phase of the greedy algorithm. Observing the results, it is obvious that none of the combinations obtains the best average results in a significant number of problems (the best is shown in bold, the second best is underlined with a continuous line, and the third best has a dashed underline). It should be noted that Problem 1 was completely ignored when it comes to finding the best combination because all combinations were tied in the best value. Some combinations are the best to solve 1 of the problems, and only one combination yielded the best average value in 2 of the 9 problems (they are actually 10 but Problem 1 was ignored due to its total tie), when $K_v = 0$ and $K_a = 0.5$, which does not represent a significant difference from the rest of combinations. Furthermore, this configuration obtained very low values in some problems (e.g. the worst results among all the tested combinations in Problem 10), therefore it did not seem appropriate to select it as the best.

Table 12 (in the Appendix) shows the average time elapsed by each parameter configuration, in milliseconds. Here, the bold notation is used to identify the fastest configuration for each problem. It can be seen that, for a same problem, there are not massive differences in time, they all belong to the same order of magnitude. The considerable variability in time values is considered to be normal when running a problem that only takes few hundreds (or even tens) of milliseconds to run, caused by the executing system in ways that are not related to the algorithm. Since the times are small, it was considered unnecessary to use the time as a tie-breaking factor when solution values are similar, since it would have led to losing sight of solid configurations. For instance, using $K_v = 0$ and $K_a = 1$ is the fastest way to obtain a solution in 4 of the 10 Problems, which is significantly better in terms of time than the other configurations (the next ones are the fastest in just 1 problem), but the value results of this configuration are poor, being the best in no problem, never being the second, and just the third in a single problem. It makes sense that this configuration is very fast, though, because it gives no importance to the value of the item individually and all importance to the area as a penalizing factor in the ratio, which leads to place items in increasing order of area (from small to large), so that first items are very

easy to place successfully (they are smaller), limiting the space for larger ones later, leading to the algorithm to eventually converge faster, but not necessarily to better solutions (often worse).

To shed some light on finding a better configuration, it is interesting to observe which are the combinations of parameters that make the algorithm obtain a very good result (within the top 3 among the 25 combinations) for a higher number of problems. It can be observed that using $K_v = 0.5$ and $K_a = 0.5$ allowed to obtain top-3 results in 4 of the 9 problems of the Joint Problem Dataset (ignoring Problem 1), which doubled the results of the next configurations, that only achieved it on 2 problems. This configuration is also reasonably solid (or at least not within the top 5 worst), for the problems in which it is outside of the top 3 best results. Therefore, due to the solid results among all problems, and being among the top 3 best more than any other configuration, $K_v = 0.5$ and $K_a = 0.5$ were selected as the final weights for the value and area (respectively) in the greedy score formula. It should be noted that this configuration was not selected because it is clearly and significantly better than any other configuration: the similarity of averages in general and the considerable standard deviations do not allow to assure that any of the tested parameter configurations is the best; performing more runs in a larger dataset of problems may be needed to determine that.

The chosen configuration of weights implies that when determining the suitability of selecting an item for placement, the greedy algorithm gives 50% of weight to the value of the item, and the remaining 50% is determined by the ratio resulting from dividing the value by the a weighted sum of two penalizing elements, the item's weight and the item's area, both with same (50%) contribution. One can see that this configuration balances all the involved elements (value, weight and area) quite evenly, which seemed to work better than using extreme options, where one of the elements was ignored (which happens if the area weight is 0 or 1, or when the value weight is 0) or has all the importance (which takes place if the value weight is 1).

The configuration of weights selected in the first optimization phase ($K_v = 0.5$ and $K_a = 0.5$) is used in the second phase. In this case, 3 configurations of values for the maximum and convergence number of iterations were tested: one using 1000 and 300 (respectively), a configuration with 10000 and 3000, and the combination of 10000 and 30000. The mean value results are presented in Table 2. It can be seen that the last configuration, with up to 100 more iterations than the first one (and 10 more than the second one), was able to obtain results of higher quality most of the times. The justification lies in the fact that more iterations allow more opportunities to try to place items. In some scenarios, once the algorithm had placed a set of items, it is very difficult to place some other items, since only a very specific range of positions and rotations may make the placement valid; with more iterations (both as a maximum and before assuming convergence), the chance of eventually succeeding increased. Those situations are particularly common in complex problems such as Problem 6 and Problem 10, where the configuration with more iterations clearly outperformed the others.

The time (in milliseconds) elapsed by each configuration is shown in Table 13 (located in the Appendix). As one would expect, the more iterations that are performed, the higher the computational time required to finish the execution, therefore the configuration with the smallest limit of iterations was significantly faster than the others. The exception to this phenomenon is only visible in problems where the algorithm is prone to place a set of items in very few iterations (less than 100), and after that the capacity constraint does not allow to place any more items, so the algorithm can stop. In these cases, the time among different configurations does not have significant differences. This circumstance takes place in Problems 1, 3 and 4, and to a smaller extent in Problem 8 (in which the initial iterations to place items are more).

Despite of the massive increase in execution time when using a maximum of 100000 iterations and 30000 convergence iterations, this option was preferred as the final configuration of the algorithm due to the significant improvement in solution quality in complex problems. There is

Table 2: Solution value of the iteration configurations tested in the second optimization phase of the greedy algorithm.

	1000 max iter., 300 conv. iter.	10000 max iter., 3000 conv. iter.	100000 max iter., 30000 conv. iter.
Problem 1	50±0	50±0	50±0
Problem 2	39.5±9.86	47.5±10.06	52.5±8.14
Problem 3	28.5±5.94	21±2	24±5.83
Problem 4	32.7±1.35	32.5±1.36	32.9±1.37
Problem 5	118.7±17.12	128.3±8.91	123.2±9.25
Problem 6	90.8±10.93	102.2±5.38	112.4±5.75
Problem 7	89.8±7.33	100±8.22	108.3±6.81
Problem 8	50.4±5.54	52.4±7.57	56.4±4.9
Problem 9	107.2±15.98	108.1±5.15	122.5±25.65
Problem 10	113±6.4	123±6.4	135±6.71

no evidence that the algorithm would not be able to obtain even better solutions if many more iterations were used (e.g. 10 times more), which may be tested in future work. Nevertheless, a significant improvement would not be expected in the examples of the Joint Problem Dataset. Observing the solutions obtained with the configuration with 100000 maximum iterations, there are not many cases where it is visually clear that one of the items outside of the container in the final state would fit inside if tried with a suitable position and rotation (respecting the capacity limit), therefore improving the final value.

6.3 Reversible algorithm

6.3.1 Experimental Methodology

In the case of the reversible algorithm, the first optimization phase was dedicated to find the best combination of action probabilities, among a small set. The probability of removing an item in a certain iteration was set to be 0.005, 0.01 or 0.02, depending on the experiment. The probability should always be moderately low, since removing an item decreases the value of a solution, so it should not be continuously used. The probability to remove an item while another removal is still recent (i.e. pending to be confirmed or dismissed after some iterations) should be even smaller, because the higher the removal number, the less likely that the algorithm will be able to add enough value to compensate the losses. To be consistent with this idea, a value of 0.001 was chosen, which is clearly smaller than any of the tested values for the base removal probability. On the one hand, ignoring the most recently removed item for the next placements (until the removal is accepted or rejected) may facilitate avoiding the undesired scenario of that item being placed again soon, having no long-term impact in the solution's value. On the other hand, if the item is the last one once the rest of objects are already placed, it would be inefficient to always ignore it. Due to the confluence of both thoughts, the probability to ignore a recently removed for placement was fixed to be 0.5. The probability to modify the placement of an item does not need to be very high, since it does not directly improve the value of a solution, but it should be high enough to take place from time to time, to increase variability; the tested values

were 0.03, 0.05 and 0.1.

The second parameter optimization phase of the reversible algorithm is very similar to that of the greedy algorithm. In preliminary tests, it was observed that a set of values for the iteration parameters gave results of more quality than others of the same order of magnitude. Namely, that was setting the maximum number of iterations to 1000 (the same as for the greedy algorithm), as well as 300 iterations to assume convergence and 200 to evaluate a removal (confirming it or reverting the solution to the one that preceded the removal). It was observed that setting the removal evaluation iterations to a smaller number was not enough to make a reasonable number of removals to be accepted (since there was not enough time to compensate the value lost with new placements), but a very high number would imply that removal would play an insignificant role in the algorithm; 200 iterations was seen as a reasonable trade-off. As done in the case of the greedy algorithm, the initial value for the maximum number of iterations was compared with a figure 10 times larger (10000 iterations) and one 100 times larger (100000), and the same factor of growth is applied to the convergence iterations: 3000 and 30000 were compared to using just 300. The 200 iterations to evaluate removal stayed unchanged, to allow more removals to take place when the maximum number of iterations is longer, and study if such aspect has any noticeable impact when finding solutions.

6.3.2 Results and Discussion

The value results of the first optimization phase of the reversible algorithm can be seen in Table 14 (placed in Section A.3.2 of the Appendix). The probability to remove an item in an iteration is noted as P_r , while the probability to modify an existing placement is represented as P_m (they should not be confused with parameters of the evolutionary algorithm that use the same notation). If one observes the results, it is clear that none of the tested configurations of parameters always yields significantly better results than any other. Following the same principle applied in the optimization of the greedy algorithm (for consistency), the most suitable configuration was considered to be the one the highest number of problems where its result is among the top 3. Since there were two configurations with the same number, it was reasonable to check in Table 15 (in the Appendix) if there were significant differences between the execution times, as a potential tie-breaking criterion. However, that is not the case: when observing the table it is unclear that the time elapsed to solve a problem depends on the values for the probabilities. A plausible explanation is that the heaviest computation (which outnumbers the rest) is the placement of items, which eventually takes place a similar number of times regardless of how probable it is to remove an item or modify a placement. Therefore, to select the most suitable configuration, the number of wins among problems was used as a tie-breaking criterion, and one of the configurations tied in the number of top-3 positions won in 5 of the 10 problems (the second in only 3). This configuration was selected as the final one. It uses the smallest tested removal probability, 0.005, implying that since removal decreases the value of solutions, it should be used scarcely. The configuration sets the probability to modify a placement to be 0.05, the intermediate value among the tested ones. However, the similarity of the average values among different configurations is too similar to claim that these probabilities are actually the best for any problem, and it would be a good idea (as future work) to repeat the experiments in a larger dataset of problems before extracting further conclusions, such as to which extent the modification of placements is a key (useful) factor to obtain a solution.

The value results of the second optimization phase of the reversible algorithm are presented in Table 3. As it happened with the greedy algorithm, the configuration with the highest number of iterations (a maximum of 100000 and 30000 to assume convergence) yielded the highest quality. Namely, this configuration was the best one in 6 of the 9 problems (Problem 1 is ignored, since all configurations obtained the same result), and it was very close to the

Table 3: Solution value of the iteration configurations tested in the second optimization phase of the reversible algorithm.

	1000 max iter., 300 conv. iter.	10000 max iter., 3000 conv. iter.	100000 max iter., 30000 conv. iter.
Problem 1	50.0±0.0	50.0±0.0	50.0±0.0
Problem 2	46.0±8.0	50.5±5.22	51.0±8.6
Problem 3	22.5±4.61	20.0±0.0	22.5±5.12
Problem 4	31.3±1.79	31.3±1.68	31.0±1.79
Problem 5	110.2±8.4	114.8±10.52	109.2±6.71
Problem 6	76.0±9.75	99.2±6.95	111.6±7.05
Problem 7	81.0±9.31	102.3±6.36	103.3±4.73
Problem 8	48.7±5.78	48.0±4.69	51.3±4.69
Problem 9	99.5±7.77	103.3±5.71	106.7±4.08
Problem 10	96.0±16.4	123.0±14.18	121.5±14.67

best solution in other 2 problems, and not very far in the remaining problem. Therefore, it was selected as the final configuration of the algorithm, despite of being massively slower (except in the case of the simplest problems), as shown in Table 16 (in the Appendix). Such results were expected, since more iterations imply that actions are performed more times, including computationally expensive checks used to determine the validity of a candidate placement. It is of no surprise that performing a higher number of iterations produced better results. As it happened with the greedy algorithm, this circumstance makes difficult placements more likely to eventually take place, and, in this case, the usage of item removal and placement modification can lead to new opportunities to place items that would otherwise not fit in the container.

6.4 Evolutionary algorithm

6.4.1 Experimental Methodology

6.4.1.1 General comments

The evolutionary algorithm has a large number of parameters compared to the other methods, but its lower speed made it unfeasible to test many configurations with different parameter values. Instead, most parameters used only a fixed value, chosen based on a combination of common sense and standard values of analogous parameters in the literature of evolutionary computation (when applicable). Additionally, in most cases, a small number of values was tested at least in one problem, and the one that produced the highest solution quality on an average of some iterations (e.g. 3 to 5) was chosen. These small experiments are referred to as preliminary tests, and due to the specificity and limitations of their experimental conditions, the ideas reached through them should be given much less confidence than those of formal experiments. It is known that such methodology to set parameter values has room for improvement, and it is clear that testing an exhaustive number of values for many of the parameters of the algorithm in a diverse set of problems (e.g. in the Joint Problem Dataset presented in Chapter 5) would potentially lead to finding better configurations. Such options would be an interesting direction for future work, if enough time and computational resources were at disposal.

Nevertheless, there was a small number of parameters for which testing multiple values had a noteworthy research interest and were considered to potentially lead to significant changes in the quality of the solutions (and the execution time of the algorithm). Despite of this, there may be other parameters of considerable research interest for which multiple values were not (formally) tested, which might be explored in future work. Parameter optimization was performed in a set of sequential phases: in each of them, a single parameter is optimized, using grid search, i.e. selecting the value yielding the best solution quality among the 10 problems of the Joint Problem Dataset, executed 10 times each for statistic significance. It is clear that the sequential method is more limited since less combinations are tested than in a grid search done for all combinations of the values of each tested parameter. However, to avoid the exponential growth in execution time derived from the latter option, the sequential phase-by-phase approach was preferred. The parameter optimization phases that were performed are listed as follows, in which each phase fixed the best value obtained with grid search in the previous phase for the tested parameter:

- Population size μ , testing the values 30, 50 and 100, as explained in Section 6.4.1.2.
- Minimum offspring size λ , tested as different factors of the population size ($0.5 \cdot \mu$, μ and $2 \cdot \mu$), as explained in Section 6.4.1.3.
- Algorithm variant, defined by the combination of two parameters: whether crossover is used, U_c (true or false), and whether placement modification mutations can be used (with a fixed non-zero probability weight for these mutations, $P_m = 0.3$) or not ($P_m = 0$, in such case only placement and removal mutations are possible). This experiment was motivated by the fact that, theoretically, an evolutionary algorithm based exclusively on mutation can obtain any possible solution (if run an endless number of times), without making crossover essential at functional level, raising the question of whether it provides an effective improvement of the quality of solutions by diversifying exploration, or faster convergence, or, on the contrary, has negative effects in terms of quality, time, or both aspects. As crossover, placement modification mutations are not needed to allow the algorithm to reach any possible solution (if run for an infinite number of generations), but it is interesting to see if the incorporation of such types of mutations bring any significant positive or negative changes in quality and time. It should be noted that when placement modifications are not used, the probability of the other two types of mutation actions, item addition and removal, is implicitly changed, even if their probability weights are not changed, because the probability weight of placement modifications is reduced to 0, so each of the other two weights represents a higher proportion of the sum of all weights. The experiments were combined in a single optimization phase because the two tested parameters configure the type of variant of the algorithm that is used: a crossover-mutation variant or mutation-only variant; and with modifications of existing placements or only using addition and removal.

For better readability, the justifications of the values selected for each parameter are divided in sections that group parameters that belong to the same facets of the evolutionary algorithm, with the same naming conventions for sections as used when explaining the evolutionary technique, in Section 3.3.

6.4.1.2 Generation of the initial population

In the process of generating the initial population, up to a maximum number of iterations $I_{max} = 300$ is performed, with early stopping taking place if no improvement in the solution's value takes place after $I_{conv} = 50$ consecutive iterations; it is assumed that the algorithm has

converged. The current value of the maximum value of iterations is a trade-off between solution quality and efficiency: smaller values (e.g. 100) were seen to produce nearly empty initial solutions in preliminary tests, failing much more often to place large or difficult items (due to the lack of enough attempts), while a much larger number of iterations (e.g. 1000) did not significantly improve the quality of solutions, and made the generation process to take more time than any of the evolutionary operators (summing the time spent in all generations), which was considered to be an undesirable property for the algorithm. Before convergence can be applied, up to a proportion $S = 0.5$ of the maximum number of iterations is devoted to try to place a specific item in the container before any other, an specialization approach that aims to achieve a high level of variability among the initial solutions. The chosen value is considerably high due to the difficulty involved in placing some items, but at the same time is not closer to 1 to avoid that if it completely impossible (or extremely difficult, i.e. unlikely) to place a certain item, at least some other items can be placed after a reasonable number of specialization attempts.

The population size, μ , is a key parameter in evolutionary algorithms. A very low value may not allow to express enough variability and feature diversity among the individuals of a population, while a high value can usually achieve the objective, but implies a linear increase in the number of times that some operations need to be performed (e.g. initial solution generation and mutation), therefore making the execution time slower. In preliminary tests, a value of 50 individuals per population was seen to be a reasonable trade-off between variability and computational efficiency. It was preferred as a default value to other conventional values in the literature, such as 100, which was expected to make the algorithm slower, yet potentially able to produce solutions of higher quality. Therefore, a parameter optimization phase was designed to determine if doubling the population size from 50 to 100 provided significantly better results, and see if the expected time penalty was worthy. To have a broader perspective of the effects of different values in the space of parameter values for μ , a smaller value, 30, was also tested. For 30 individuals, the initial hypothesis, to be validated or refuted, was that the algorithm would be both faster (due to having less operations to do per generation) and unable to reach the quality of solutions produced with higher values, due to the smaller exploration possible with less individuals, given that the number of generations remained unchanged.

6.4.1.3 Parent selection and offspring generation

In each generation of the evolutionary algorithm, at least λ individuals are generated (more in the special circumstances explained in Section 3.3.6). One of the most common ways to set this value in the literature is to make it equal to the population size, so this was opted for as a natural choice. However, motivated by a notable quality of solutions obtained when using a value of λ that was times 2 smaller than μ in preliminary tests, a parameter optimization phase was designed to determine the best option, considering computational efficiency as a potential tie-breaking criterion if the two options happened to yield results without differences of enough significance. To have a greater perspective of the value of λ , as a factor of μ , an offspring size 2 times larger than the population was also tested, expected to make the algorithm slower but capable of producing solutions of higher quality, as an initial hypothesis.

The tournament pool size used in parent selection, $T_p = 3$, was chosen to give higher value to competition than with just 2 individuals, but without making selection pressure so high that the population may be prone to become too less diverse in few generations, only keeping individuals similar to the elite, and making exploration less effective (e.g. with a pool size of 5, or especially 10).

6.4.1.4 Crossover

In crossover, when the length of a non-segment partitioning shape is defined, it is set to be in the range determined by two proportions of the side length of the container's bounding rectangle, a minimum and a maximum value. It was chosen to use a minimum proportion $\bar{L}_{min} = 0.25$ because it can make a shape to be small yet large enough to cover a reasonable amount of the area to partition, and a maximum proportion $\bar{L}_{max} = 0.75$, which is considerably large but should not usually cover the whole usable space of the container (which would lead to fail to partition the container in two regions). For partitioning shapes that are polygons, it was fixed that the number of vertices would be at most $\bar{V}_c = 10$, and of course not smaller than 3, which produces a triangle; such range is wide enough to provide a large diversity of region shapes. When trying to create a valid partitioning shape, at most $M_c = 5$ attempts can be done in a single crossover action; with the used configuration of proportions for the size of the partitioning shape a valid one should be produced before that number of attempts in most containers, often in the first time. In problems with extremely narrow containers crossover is less likely to be useful, so mutation should have a primary role, without trying too many times to perform crossover in vain. It is considered that a valid partitioning shape should have an area representing at least $A = 0.1$ times the container's area: it is considered that smaller areas have a reduced likelihood of enclosing any item within them, which is important to make crossover useful.

In one of the last steps of crossover, up to $R = 5$ permutations of orders of placements of items intersecting with the two partitioning regions in the parents can be made. 5 is far from being an exhaustive number, but large enough to obtain a clear boost in the quality of results compared to not using any permutation, according to preliminary tests, since the placements derived of permutations increase the value of solutions, and still not representing a large impact in efficiency (which may happen if many more permutations were tried). Crossover can produce additional offspring (more than two) as individuals resulting from permutations which, even if they are not the best (which represent the two default offspring), have a fitness value that is better than a proportion $C = 0.95$ of the most recently updated population. Smaller values (e.g. 0.8 or below) generated a very high number of additional offspring in preliminary tests, which was considered an undesirable property, because the surplus offspring are, in general, very similar to the base ones, and should be only accepted if they are elite or almost elite, therefore a very high proportion was fixed.

6.4.1.5 Mutation

In mutation, one of the three high-level types of mutation is selected at a given mutation step, proportionally to their probability weights: $P_a = 0.6$ for addition, $P_r = 0.1$ for removal and $P_m = 0.3$ for placement modification. The weights were fixed taking into account that addition is the only type of operation that increases the fitness value, therefore it has a clearly higher value than the others, while placement modifications do not directly change the value of a solution (but can promote exploration and hypothetical improvements), and removal should be much lower because it decreases the fitness value and can only be useful for an eventual, non-guaranteed long-term improvement of the value after more placements take place. A single execution of the mutation operator performs a minimum number of iterations $I_{mut} = 5$, which seemed enough to promote that in most cases at least one change is made (i.e. making mutation effective), since some of the operations may fail. Nevertheless, to increase the likelihood that mutation is effective, the action chosen in an iteration is not tried only once (except for removal, that cannot fail if the container has at least one item), but up to a maximum number of attempts, $M_a = 10$ for addition and $M_m = 3$ for placement modification, giving more importance to the former since it is the only operation increasing fitness when it is successful, and it is potentially difficult

to place items after some generations, when the container is almost full in some problems. The intermediate solution after each mutation step is not lost if it is better than later ones. Actually, if an intermediate solution has higher fitness than the individual at the end of mutation, the intermediate solution is selected as final with a probability $P_i = 1$, i.e. always choosing the intermediate option if better. In preliminary tests, using smaller values for the probability (e.g. 0.5) brought a decrease in the quality of the final solution of the algorithm, so they were discarded. If crossover is used and the offspring generated by crossover has higher fitness than its subsequently mutated version, the individual before mutation is selected as final with a probability $P_{cm} = 0.5$, which represents a trade-off between exploiting the best option and exploring alternatives; in preliminary tests it was not observed the same effect in the final solution as seen for the previous probability.

In the case of placement modification mutations, multiple parameters are used to determine how the different actions work. In the case of moving an item in a random direction, in each iteration of movement the displacement should have at least a minimum distance, calculated using a proportion $D = 0.03$ of the length of non-parallel sides of the bounding rectangle of the container. The chosen proportion is considerably small because in containers with many items, even such proportion can provide a significant gain in terms of compacting items in the container's space. In the same modification action, the displacement segment is divided in $B_p = 15$ equidistributed positions, providing the targeted level of precision in terms of dividing the space in small sections, and taking into account that since binary search is used, the total number of iterations (tested positions) is smaller (in a logarithmic way). In the action that rotates an item in either clockwise or counter-clockwise direction, a maximum of $B_r = 8$ angles are tested, which provides an exploration that is detailed enough; the tested angles are calculated with respect to the original rotation of the item, so they are not a constant set of absolute angles that is same for all items (which would reduce variability). In the action that moves an item with a small change in position, a proportion $S_p = 0.2$ of the length of the sides of the container is used to determine what makes a distance change from being small to starting to be non-small; in the action that performs a small change in rotation, a proportion $S_r = 0.2$ of 360 degrees is used to define what is the first angle that is not considered small. In both cases, the thresholds represent a value still relatively small, to guarantee that the changes in position or rotation that are supposed to be small are indeed not large, but still significant on average.

6.4.1.6 Population update

When updating the population, tournament selection is used to determine which individuals survive, and a pool size $T_u = 3$ is used, equal to that of parent selection, for the same reason: it promotes a certain level of selection pressure, but not so high to easily jeopardize variability. In each population update, the $E = 5$ elite individuals among the population and their offspring are preserved, and ignored in tournaments. In preliminary tests it was observed that some of the best solutions are very unlikely to be generated (e.g. by being able to place a very large, high-value item that does not fit in most attempts), and with a small elite size (e.g. 1 or 2) it was more likely that they were lost (if they are not in the elite, but still having the potential to lead to be better after applying operators) in few generations (despite of the fact that they would win in most tournament pools, they need to be randomly pre-selected first). These observations motivated the choice of a considerably high elite size.

6.4.1.7 Termination criteria

The algorithm is run for a maximum number of generations, $G_{max} = 30$. Traditionally, in evolutionary algorithms, increasing the number of generations has the effect of improving the

quality of solutions (up to a certain extent, obviously the optimal value cannot be exceeded), as well as making the algorithm slower, since more operations are performed (e.g. mutation, crossover and population update). In preliminary tests, it was observed that a smaller maximum number of generations (e.g. 20) yielded a significant decrease in the quality of final solutions, while larger values (e.g. 50) did not provide a significant improvement, at least in the tested problems. Nevertheless, it would be interesting to check much larger numbers of generations (e.g. 100 or 300) in many problems (e.g. in the full Joint Problem Dataset) to determine in which range of generations the algorithm stops improving in a significant way; this is a possible direction for future work. It is considered that the algorithm has converged if the fitness value does not improve in $G_{conv} = 12$ generations. It was observed in preliminary tests that a lower value (e.g. 5 or 7, and even 10 in a much subtler way) often stopped too early, when the elite fitness still had room for improvement in the population.

6.4.2 Results and Discussion

The first optimization phase of the evolutionary algorithm compares three different values of population size: 30, 50 and 100. As expected and shown in Table 17 (placed in Section A.3.3 of the Appendix), using greater population sizes increases the likelihood of finding high-value solutions, either at the initial population generation and as a result of applying operators such as mutation or crossover. These operators are used more times since the population is larger, and the offspring size was set to be proportional to population size, namely half of it for the first optimization phase. Therefore, it is not surprising that the population size is empirically found to be proportional to the execution time, as shown in Table 18 (in the Appendix). Although the speed of the algorithm decreases in a significant way when using a population size of 100 individuals, this option was selected as the final one due to its clear increase in the quality of solutions. Namely, the three configurations were tied in Problem 4, but a population size of 100 individuals allowed to obtain better results in 6 of the 9 remaining examples of the Joint Problem Dataset (and it was especially close to the best in one of the other problems, and not far in the case of the remaining one; the best solution was within the standard deviation).

In the second optimization phase of the evolutionary algorithm, multiple values of the minimum offspring size, λ , were tested: half of the population size (50 individuals), equal to the population size (100), or twice larger (200). The value results are presented in Table 19 (located in the Appendix). As one may expect, generating a larger number of offspring increases the chance of finding better solutions, since the mutation and crossover operators are applied more time, which can lead to increasingly challenging placements to be eventually possible. However, because of the same reason (applying operators more times), the execution time increases proportionally to the offspring size, as shown in Table 20 (in the Appendix). The configuration with the largest λ obtained solutions of higher quality than that with the lowest λ in 5 of the 10 problems, while the configuration with 50 offspring is the best only 3 times. However, the difference between them is only slight, since the best solution in 9 of the problems is within reach of standard deviation for the 50-offspring configuration. The only case where the 200-offspring configuration was clearly out of reach for the others was in Problem 10, a problem where it is specially difficult to place all items due to the small space margin that allows to obtain an optimal solution. Because of this apparent advantage for particularly difficult problems, the configuration with $\lambda = 200$ was chosen as the final one. Nevertheless, due to the small overall difference with the other configurations, it should be noted that in applications where time is critical, a configuration with $\lambda = 50$ would be a very reasonable (and much faster) choice.

In the third (and last) optimization phase of the evolutionary algorithm, different variants of the method were tested, depending on whether crossover was used (which happens when the Boolean parameter U_c is true) and whether placement modifications were used (which requires

the probability weight P_m to be non-zero, in this case fixed to 0.3; if the weight is 0, modifications are not used). The value of each configuration is shown in Table 4. Analyzing the mean values and the standard deviations, one can see that the former are usually greater than the differences between the different configurations. Therefore, it cannot be said that any of the 4 variants of the algorithm is significantly better or worse than any other. The most interesting conclusion that can be extracted from this table is that crossover does not have a clearly positive nor negative effect in the quality of a solution, i.e. its presence or absence does not seem to affect the quality of solutions. A potential explanation to this circumstance is the fact that mutation alone can theoretically lead to obtain every possible solution for the problem, in other words, there is no solution that can be generated exclusively via crossover. The execution time (in milliseconds) of the configurations, presented in Table 5, also shows that the presence of crossover does not make the algorithm significantly slower. It may be surprising at first, but it has a plausible explanation: one of the parameter of the algorithms makes mutation to be avoided with a certain probability only if crossover has been used first, and its default value is 0.5, meaning that if crossover is not used, approximately twice of mutations are performed instead, which balances the overall execution time.

Another interesting conclusion that can be extracted from Table 4 is that placement modifications do not seem to have a clear impact in the results, that are similar regardless of whether they are enabled or not, as it happened with crossover. It has a similar explanation: the remaining actions, which are the placement (with random position and rotation) of items and the removal, can eventually lead to any possible solution of the algorithm, even if the usage of placement modifications can help to change the position or rotation of an already placed item without removing it and then placing it again.

From the number of wins depicted in Table 4, it seems (if one ignores the lack of significant differences) that the two best configurations are completely opposite: one does not use crossover but does use placement modifications (this configuration wins individually in 3 problems, wins in a tie in 1 problem and has the same solution as all the other configurations in 2 other problems); the other one, with results of very similar quality, uses crossover but ignores placement modifications (wins alone in 4 problems). To choose a final configuration among the two, the execution times of Table 5 were used as a tie-breaking criterion. It seems that the configuration that does not use crossover (but uses placement modifications) is somewhat faster: at least, it was the fastest in 5 of 10 problems, while the one using crossover (and ignoring placement modifications) only has the smallest time in 1 problem. Therefore, the configuration that ignores crossover altogether but still uses placement modifications was selected as final. The results do not point out clearly significant differences. However, it would not be surprising that the absence of crossover produces a slightly faster algorithm (despite of the mentioned counter-effect of performing more mutations), since crossover tasks can be avoided, some of which are somewhat computationally expensive, such as the placement permutations for items that intersect with the partitioning regions.

Table 4: Solution value of the variants of the evolutionary algorithm tested in the third optimization phase.

	$U_c = False,$ $P_m = 0$	$U_c = False,$ $P_m = 0.3$	$U_c = True,$ $P_m = 0$	$U_c = True,$ $P_m = 0.3$
Problem 1	85.0±22.91	95.0±15.0	100.0±0.0	90.0±20.0
Problem 2	76.0±4.9	75.5±5.22	77.5±6.42	74.5±1.5
Problem 3	37.0±2.45	38.5±2.29	37.5±2.5	37.5±2.5
Problem 4	34.0±0.0	34.0±0.0	34.0±0.0	34.0±0.0
Problem 5	234.0±0.0	234.0±0.0	234.0±0.0	234.0±0.0
Problem 6	123.6±6.97	122.4±5.2	124.4±7.2	122.6±6.8
Problem 7	114.6±2.94	115.1±0.7	113.3±3.0	111.1±3.11
Problem 8	90.7±14.21	90.7±14.21	84.5±15.5	87.6±15.19
Problem 9	172.0±5.35	173.8±4.94	167.4±6.51	171.1±6.09
Problem 10	137.0±4.58	139.0±7.0	141.0±5.39	138.0±4.0

Table 5: Execution time (in milliseconds) of the variants of the evolutionary algorithm tested in the third optimization phase.

	$U_c = False,$ $P_m = 0$	$U_c = False,$ $P_m = 0.3$	$U_c = True,$ $P_m = 0$	$U_c = True,$ $P_m = 0.3$
Problem 1	42111±12410	34102±5842	44694±3910	36686±5094
Problem 2	136216±14039	104512±9635	98899±16142	78979±10992
Problem 3	19354±2135	18028±1478	31242±6950	32718±7760
Problem 4	15047±2125	21729±2762	37573±4874	41418±3273
Problem 5	167462±17170	135710±8043	104630±17227	108805±8054
Problem 6	104561±15955	94134±24423	97867±19276	86834±14033
Problem 7	107666±11471	92025±12284	112887±10410	97541±18470
Problem 8	31213±7149	26282±7424	42182±20400	33355±14721
Problem 9	90454±18089	70057±13575	80248±14920	85538±19405
Problem 10	108453±15147	87828±17802	74580±16915	60702±11671

7 Experimental Comparative Analysis of the Proposed Methods with the Joint Problem Dataset

7.1 Experimental Methodology

The aim of this chapter is to compare the three algorithms that were designed and implemented in this work (greedy, reversible and evolutionary), in terms of solution quality (understood as the sum of the value of all items in the container) and execution time (reported in milliseconds). Each algorithm was run in each of the 10 problems of the Joint Problem Dataset, introduced in Chapter 5. Since at least one optimal solution is known for each of the problems (it was obtained manually), the algorithms can be compared with this gold standard in terms of quality, to assess how close to optimal they are.

The three algorithms used the parameter configurations that resulted from the optimization steps described in Chapter 6, which also used the examples of the Joint Problem Dataset. In general, it is advisable to optimize the parameters of algorithms in a set of problems that is different from the one where they are tested later, to promote generalization and avoid over-specialization biases. However, such approach was discarded for this work. It must be taken into account that there were no preexisting datasets for the Joint Problem, so the Joint Problem Dataset was created for this work (with a reduced number of problems, 10), and the author wanted to apply the noteworthy diversity achieved for the problems to both the parameter optimization phases and the algorithm comparison, to extract richer conclusions. Furthermore, since all the algorithms were optimized using the same examples, it cannot be said that any of them was given an unfair advantage over the others. Moreover, in the optimization it was found that many of the parameters did not provide significant improvements for certain values (with respect to the initial configurations), and others confirmed common-sense hypothesis that are expected to be valid for most problems, such as the fact that increasing the population size and offspring size improves the final solution, to a certain extent. However, if a greater dataset was created in future work, it would be advisable to split it in optimization and testing partitions, if performing parameter optimization was desired.

To guarantee a reasonable level of statistical significance of the results, each experiment was run for 10 times, and statistical information summarizing them (mean, standard deviation, minimum value, median and maximum value) is presented and analyzed in the next section. To ensure that the execution times were not significantly affected by external causes, the experiments were carried out in a single process, in a machine with no user interaction during the experiments, and without other programs running (other than the essential processes of the operating system). Since the machine has multiple CPU's (6) it is not expected that operative system factors affected the execution times presented in the next section in a significant way.

7.2 Results and Discussion

7.2.1 Solution Quality Analysis

The results of solution quality are presented in Table 6. For each of the three algorithms (greedy, reversible and evolutionary), the following statistics are shown, in this order: mean, standard deviation, minimum value, median and maximum value. The last column corresponds to the value of the (manually obtained) optimal solution. For each problem, the bold notation is used to identify which algorithm obtain the best mean, minimum value, median and maximum value. When one of these numbers is equal to the value of the optimal solution, an underline is used, to put focus in the fact that optimality was reached. An obvious conclusion that can be extracted from the results of Table 6 is that the evolutionary algorithm produces results of significantly

higher quality than those of the greedy and reversible algorithms. This fact is clearly consistent, not a matter of chance, since the evolutionary algorithm obtains the highest mean, minimum value, median and maximum value in absolutely all the problems (even if the other algorithms also reach that value in a small minority of cases).

Table 6: Statistics of the solution value results for each algorithm in the problems of the Joint Problem Dataset, compared to the manual optimal solutions.

Algorithm	Greedy					Reversible					Evolutionary					Manual	
	Statistic	mean	std	min	med	max	mean	std	min	med	max	mean	std	min	med	max	
Problem 1	50	0	50	50	50	50	50	0	50	50	50	100	0	100	100	100	100
Problem 2	50	10.49	30	50	75	50	3.16	45	50	55	77.5	6.42	70	75	90	90	
Problem 3	23	5.1	20	20	35	22	5.1	15	20	35	37.5	2.5	35	37.5	40	40	
Problem 4	32.7	0.78	31	33	34	30.1	1.81	29	29	34	34	0	34	34	34	34	
Problem 5	130.1	4.97	123	134	134	111.6	11.73	103	105	139	234	0	234	234	234	239	
Problem 6	104.9	6.41	97	105	118	110	10.1	97	109	130	119.9	5.77	110	118	130	130	
Problem 7	110.3	6.39	96	113.5	115	103.4	6.89	91	105	115	114.7	1.95	111	115	118	118	
Problem 8	53.8	5.93	45	55	60	49.9	7.63	42	48.5	65	84.5	15.5	69	84.5	100	100	
Problem 9	109.3	2.61	105	111	113	104.4	5.62	94	103.5	113	172.4	4.43	166	174	177	190	
Problem 10	136	4.9	130	140	140	132	8.72	120	130	150	140	6.32	130	140	150	210	

The evolutionary algorithm is capable of obtaining the optimal value in at least one of the 10 executed runs in 7 of the 10 problems of the Joint Problem Dataset, which is a positive quality. However, it is obvious that the algorithm does not guarantee that perfect results will be obtained. To understand why, it can be useful to check the best evolutionary solutions for each problem, whose visualization is shown in Section A.4 (in the Appendix), which can be compared with the manual optimal solutions of Section 5.2. In both Problem 5 and Problem 9, two of the largest items need to be placed in the container to achieve optimality, but in evolutionary algorithm only manages to place one of them (in the visualized solutions). This is caused by the fact that the item specialization in the generation of the initial generation (which is essential for large items to be featured in solutions of complex problems) is performed on a single item per initial solution; if the specialization was performed for more than one item, for sufficient iterations, the optimal solutions would be much easier to obtain. Problem 10 was devised as a very difficult problem where the optimal solution requires all items to be placed within a very limited range of reference positions and rotations: the algorithm would have been much more effective if it had known that using multiples of 90 degree was sufficient (but that is not beneficial for all problems). As soon as one item does not meet these requirements, it becomes impossible to find the optimal solution, unless it gets modified with a sequence of very specific changes, which is unlikely. In Chapter 9, different ways to improve the algorithm are explored as potential future work, to opt to find optimal solutions for the mentioned problems, and new ones. Conversely, in the majority of problems (7 of the 10), as mentioned, the evolutionary algorithm is able to obtain optimal solutions (either in all runs, some of them or at least one).

A key aspect of the success of the evolutionary algorithm is the creation of a large population of candidate solutions that evolve through time, by the usage of operators, namely mutation, that can place items in the container and perform removals and placement modifications; the

crossover operator was discarded in the last parameter optimization phase of Section 6.4. The fitness-proportional selection of individuals in the population update of each generation promote the progressive refinement of the quality of the solutions, until the best one remains intact for enough generations to consider that the algorithm has converged. The fitness evolution process is depicted in Figure 23 for one of the tested problems. It is easy to observe how the variability of the problem is large by the time that the initial population is generated, and it is progressively decreased (and improved) throughout generations; once a very good solution is found, it is interesting to observe how it is preserved (by using elitism), while the population in general are still refined to try to become even better.

This paragraph can be skipped by the reader if wanted, since it is exclusively devoted to explain in more detail how statistical information is represented in Figure 23's sequence of boxplots to depict the evolution of the fitness value of the population through the generations of the evolutionary algorithm. The first boxplot corresponds to the population after its original creation, before any operator is applied, and each of the next boxplots corresponds to the population after being updated in a generation. If early stopping takes place, the last generation does not update the population (choosing the survivors among the old group and the offspring), since it stops just after realizing that the elite has not improved with the new offspring, and has kept unchanged for a certain number of convergence iterations. In such case, and with the only motivation of performing a boxplot based on the same number of fitness values as in the previous generations, the population is updated to restore the normal population size. The described option of updating the population of an early-stopped generation for visualization purposes has been preferred over duplicating the boxplot of the previous generation, which would have also been a valid option (since the most recently confirmed population would be the previous one), but it would not reflect the influence in the population's fitness made by the last-generated offspring, which may be interesting to see. In the boxplots, the rectangular shape (box) represents the data between the first quartile (25th percentile) and the third quartile (75th percentile); the median (or second quartile) is shown as an orange line, while the mean is depicted in green. The lower whisker indicates the 5th percentile, whilst the upper whisker indicates the 95th percentile. The values outside of these ranges are represented with plus symbols.

One of the main limitations of the greedy and reversible algorithms is that a single solution is managed at a time (not a whole population as done in the evolutionary algorithm), which massively limits the space for exploration. Furthermore, in the case of the greedy algorithm, item additions to the container are permanent, i.e. they are never changed. This has the consequence of easily limiting the algorithm to fall into local optima, but since it can focus in a fixed set of circumstances, it can eventually be able to place an item for which there is still enough free room but a high placement difficulty (e.g. a small range of reference positions where it is feasible), after a long number of attempts. An example of this scenario can be seen in Figure 24, where a smaller number of iterations to assume convergence would not have allowed to place a valuable item, that here is eventually placed after a large number of attempts.

In the case of the reversible algorithm, placements can be reverted via item removal, or modified with the movement or rotation of placed items. While this circumstance helps to escape potential local optima by providing new possibilities for placements, it can be detrimental in some ways, even if an older state of the solution is restored if a removal does not lead to higher value after a certain number of iterations. In some occasions a removal paves the way to a placement which increases the value, as shown in Figure 25. The negative point is that the algorithm sometimes places many efforts in trying to improve a solution after a removal in vain, since it is unable to place enough items to compensate the removal before the evaluation period ends. Conversely, the greedy algorithm is focused on improving a more limited exploration space

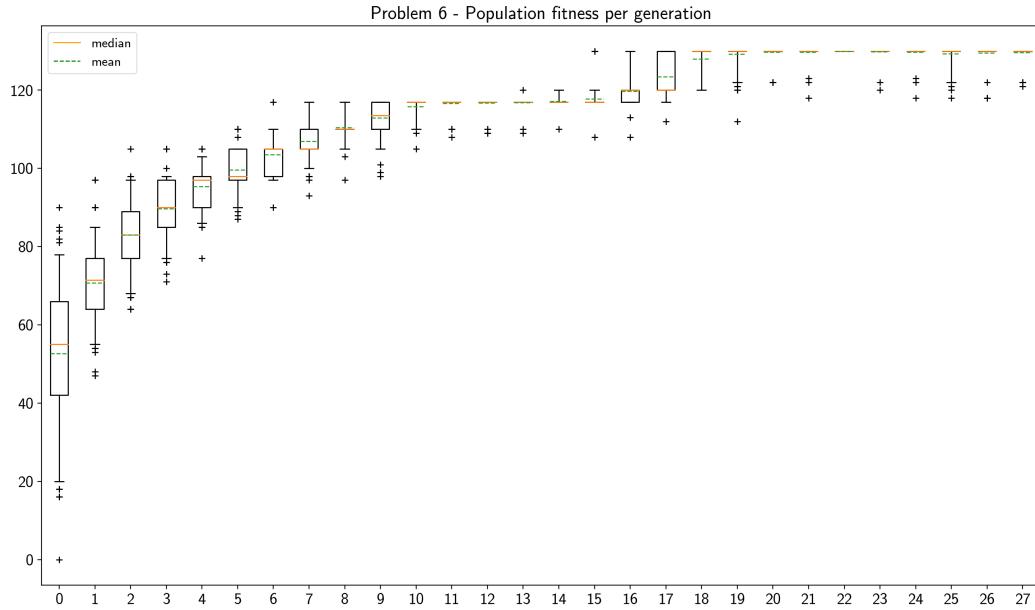


Figure 23: Evolution of the fitness of a population of the evolutionary algorithm in one problem, throughout generations.

(since it has no removals or placement modifications), which can be a key factor in some runs.

The explained advantages and drawbacks of both the greedy and reversible algorithm make it easier to understand the fact that both obtain solutions of similar quality, even though a detailed observation of the statistics of Table 6 gives some advantage to the greedy algorithm. The greedy algorithm has a higher average in 7 problems, the reversible one has a better mean in 1, and are tied in 2. A similar result can be seen for the median, where the greedy wins 6 times, the greedy 1 and there are 3 ties. The greedy algorithm also has better minimum values: it wins in 7 problems, with 1 win for the reversible algorithm and 2 ties. However, the reversible algorithm obtains better maximum values: it wins 4 times, while the greedy algorithm has only 1 greater maximum, and there are 5 ties. Therefore, it seems that the the specialization of the greedy algorithm in fixing placements is, in general, more effective than the conditional removal policy of the reversible algorithm, which can still be useful sometimes to obtain better maximum values, thanks to performing a greater exploration via removals and placement modifications.

7.2.2 Time Analysis

Table 7 presents the execution time (in milliseconds) for each algorithm, showing the same statistics used for solution quality. In this case, the bold notation is used to identify the lowest times, which are considered to be more computationally efficient. It is easy to observe that the evolutionary algorithm is always the slowest one, while the other two, greedy and reversible, both have a balanced number of wins. The reversible has the best mean in 7 problems. The median results are almost even, with 4 wins for the greedy, 5 for the reversible and 1 tie. However, some of the largest maximum times of the reversible algorithm are twice (or more times) greater than those of the greedy algorithm (e.g. in Problems 2, 5 and 9). Therefore, it is not clear that any of the two is significantly faster. Instead, both algorithms are in general in the same order of magnitude in terms of time, which is massively smaller than the time of the evolutionary

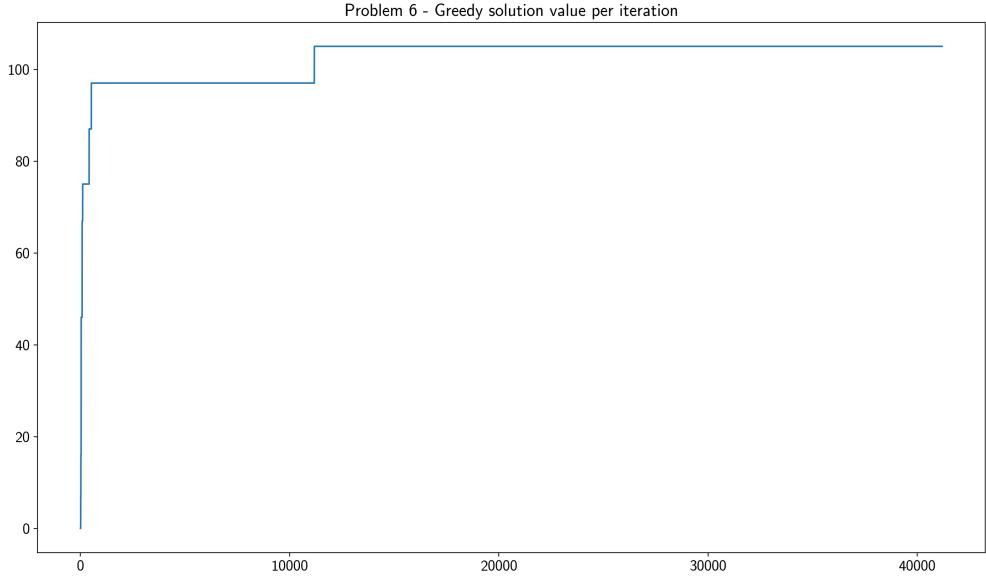


Figure 24: Evolution of the value of a solution of the greedy algorithm in one problem, throughout iterations.

algorithm. This fact is unsurprising, since the evolutionary algorithm keeps a large population of solution, that are generated and updated throughout generations, while the iterations of the greedy and reversible algorithm perform a single or small number of tasks in a single solution. The tasks performed by the greedy and reversible algorithms are simpler than the mutation operator, which is used many times in the evolutionary algorithm; for instance, a placement in the greedy or reversible algorithm is attempted just once, while the mutation operator has multiple attempts, and it is applied many times for each generation.

Table 7: Statistics of the execution time (in milliseconds) for each algorithm in the problems of the Joint Problem Dataset.

Algorithm	Greedy					Reversible					Evolutionary				
	Statistic	mean	std	min	med	max	mean	std	min	med	max	mean	std	min	med
Problem 1	15	13	2	13	48	4	2	2	4	7	20399	1854	18103	20462	23525
Problem 2	17899	526	17439	17713	19054	30902	15166	2981	33202	50392	80866	12294	66535	75567	107385
Problem 3	19	10	9	17	47	13	8	5	11	34	13380	3619	9967	11068	19299
Problem 4	8	1	6	8	11	8	2	6	8	10	13223	637	12347	12850	14239
Problem 5	13970	4470	9893	11263	21198	20804	18130	3382	16192	61936	86815	8486	74924	84015	100684
Problem 6	14145	3106	11420	12622	21785	9662	1954	7102	9638	13730	59861	7535	49838	57350	76820
Problem 7	9897	5045	2428	9309	19342	6441	3649	1223	6227	13989	66235	18088	42913	59258	96525
Problem 8	145	96	37	128	330	120	93	34	104	354	23117	9808	13161	19901	44300
Problem 9	343	283	103	249	1100	1892	3174	104	294	9501	50271	12260	35532	46005	71366
Problem 10	13807	2566	11448	12828	19797	13113	909	12028	12877	14680	62213	13849	42430	61981	93049

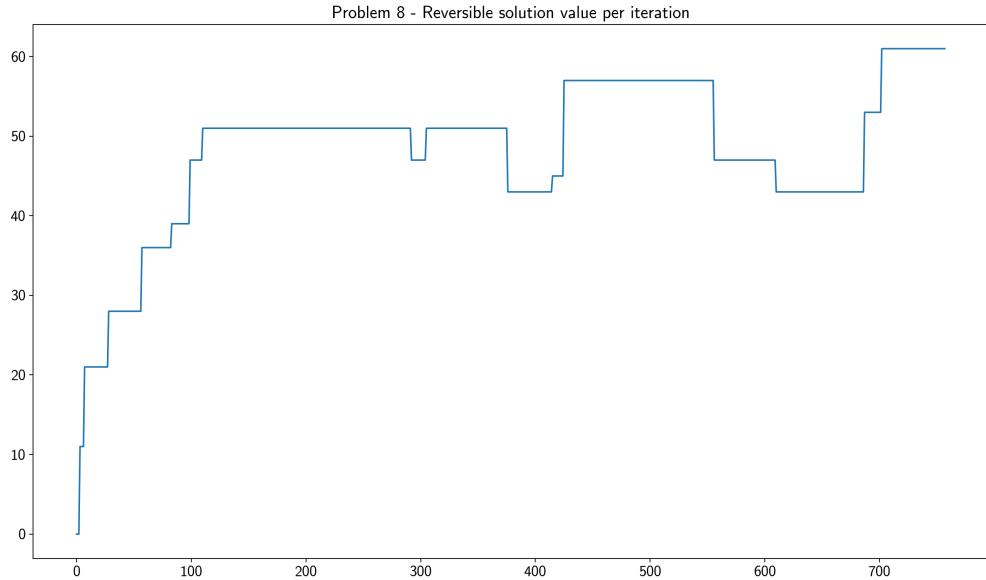


Figure 25: Evolution of the value of a solution of the reversible algorithm in the start of one problem, throughout iterations.

The already explained role of mutation in making the evolutionary algorithm much slower (and capable to obtain solutions of higher quality, too) is supported by the division of tasks by time, shown in Figure 26, where it is obvious that mutation takes about 90% of the time. In previous experiments where crossover was still applied, it also used to take a considerable amount of time, but a bit less than mutation (when both were used mutation took less absolute time, due to a probability of crossover to avoid performing mutation afterwards, set to 50%). Currently, the next heaviest task after mutation is the generation of the initial population, due to the considerable number of iterations devoted to obtaining some reasonably fit solutions from the very beginning. The time of parent selection, elite fitness and population update is significantly smaller, and the rest of tasks have a negligible impact.

In the case of the greedy algorithm, the task that takes almost the totality of the running time is the addition of items, as one can see in Figure 27 (in the same problem as shown for the evolutionary algorithm, for consistency). This happens because the computations that check whether a candidate placement is geometrically valid require are not trivial, especially for items with complex shapes. The stochastic selection of items has a certain impact in time, but about 100 times smaller than geometric checks; this circumstance is also present in the case of the reversible algorithm, as shown in Figure 28. In the reversible case, geometric checks are performed not only in item additions but also in placement modifications (which are attempted less frequently); removals have a very small impact, since they do not require geometric checks. In both algorithms, sorting the items by weight and discarding those that are outside of the capacity is negligible in terms of time.

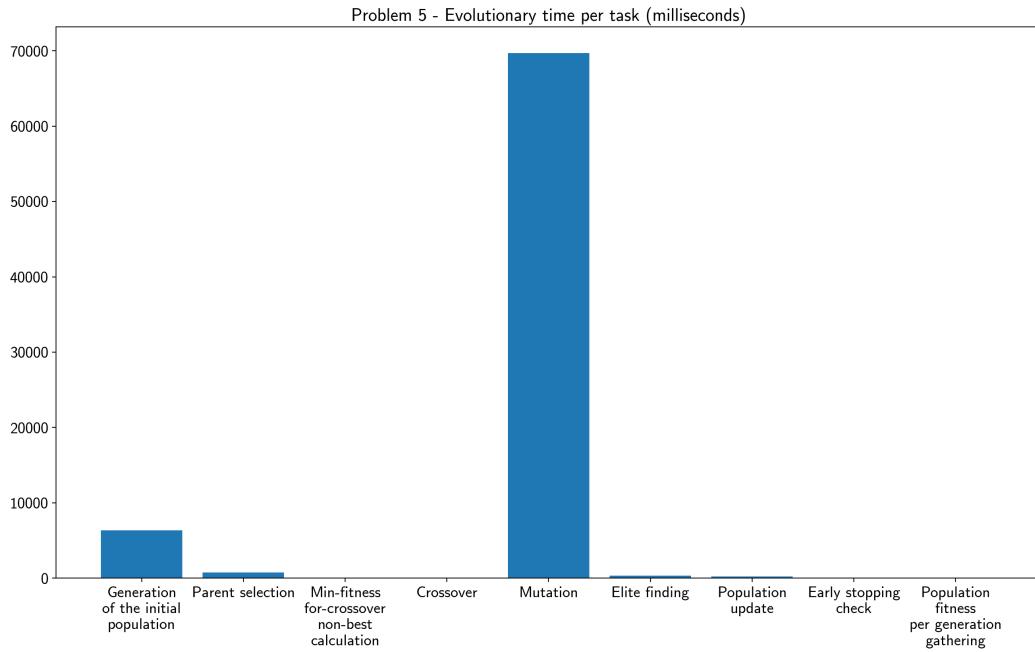


Figure 26: Time division of the tasks of the evolutionary algorithm in one of the tested problems.

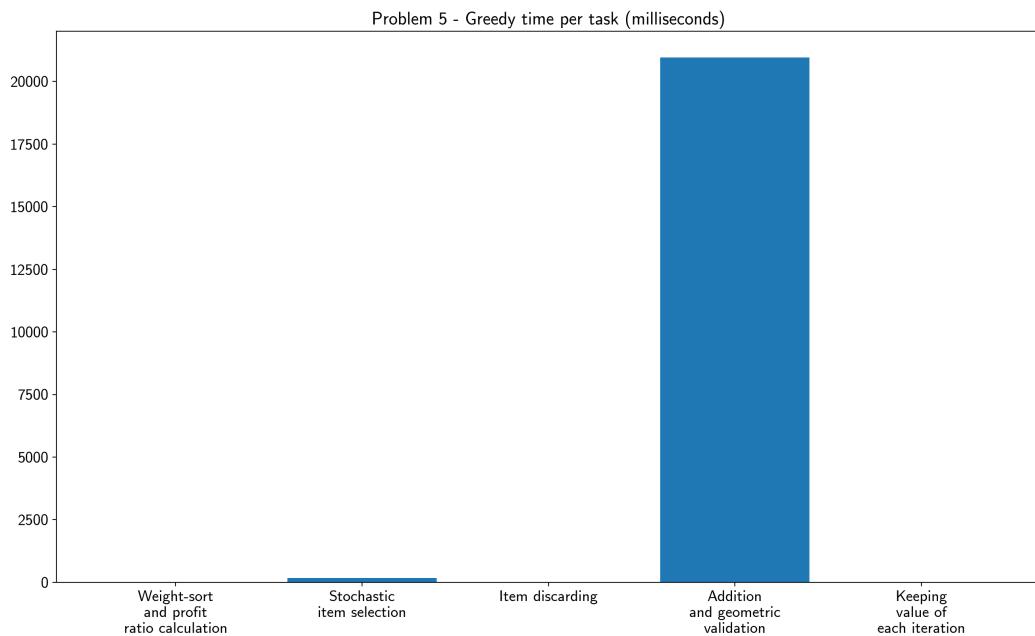


Figure 27: Time division of the tasks of the greedy algorithm in one of the tested problems.

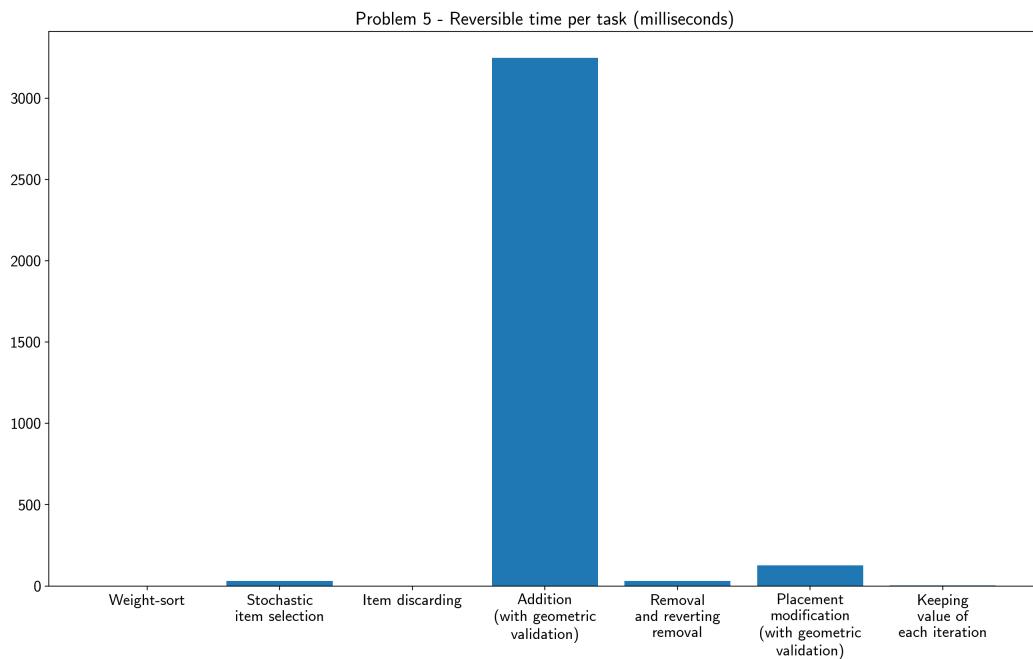


Figure 28: Time division of the tasks of the reversible algorithm in one of the tested problems.

8 Experimental Analysis of the Applicability of the Proposed Methods to solve a traditional Packing Problem

8.1 Contextualization and Experimental Methodology

The aim of this chapter is to determine whether the proposed algorithms (greedy, reversible and evolutionary) can be successfully applied to obtain results of a reasonably high quality in the domain of the Packing Problem. Specifically, the author decided to explore the type of Packing Problem whose objective is to maximize the number of items (eligible from a finite set) placed in a container. It was considered that the developed algorithms can be naturally oriented to such objective than to more distant problems, such as those that do not specify a fixed shape for the container, and instead, try to minimize the area or side length of the bounding rectangle of the region resulting from placing items in a compacted way. Many publicly available datasets exist for this latter type of Packing Problem ([AS80](#); [DJ95](#); [DDB98](#); [Hop00](#)), which also constraint the possible rotation angles (e.g. to 0, 90, 180 and 270 degrees), unlike the proposed methods, designed to support arbitrary rotation angles.

Unfortunately, the author did not find any public dataset for the type of Two-Dimensional Irregular Packing Problem in which the item number must be maximized. A feasible alternative would have been to convert the examples of the Joint Problem to this Packing Problem for testing, but it was discarded, since it would require to manually determine the optimal solution, and it would have been a bit repetitive to use the same problems again. Instead, the author decided to use the two-dimensional packing solver ([WR19](#)) of the Wolfram Alpha knowledge engine to create a set of problems and find the highest (optimal) number of items that can be placed in the container. The solver is limited to items and containers whose shape is a circle, triangle or rectangle, which excludes some of the shapes explored in the Joint Problem Dataset: ellipses, polygons with more than 4 sides and compound polygons. Furthermore, all items are restricted to have the same shape and dimensions. Despite of these drawbacks, an illustrative variety of problems was created, along with their optimal solution, used to assess the solution quality obtained by each of the algorithms. Since the Wolfram Alpha solver places all the items in the container in all the cases, checking if the proposed algorithms reach optimality is as simple as seeing if it has been able to place all items, without leaving any outside the container. Apart from this intuitive analysis, another positive point of using Wolfram Alpha is that the solver does not simplify or discretize the allowed ranges of positions and rotation angles (at least explicitly), which is aligned with the designed methods (unlike many well-known, generally old, Packing datasets, which imposed constrained rotations). Section [8.2](#) provides more details of the created set of problems. As done in Chapters [6](#) and [7](#), each experiment was run 10 times to obtain a reasonable level of statistical significance. To make the total time of executing the experiments significantly faster, multi-processing was used, in the same terms (and with the same effects in time stability) as explained in Chapter [6](#).

To switch from the Joint Problem to the Packing problem, the value and weight notions need to disappear, or at least become irrelevant. The latter option was chosen, since it only requires to change the data of the problems, while keeping the algorithms intact. All the items were given the same value (1 unit), so that increasing the number of items makes the solution value higher (and the fitness, in the case of the evolutionary algorithm), without preference (value bias) for any item in particular. All items were assigned a common weight (1 unit), and the capacity constraint is never applied, since the container was given infinite capacity.

8.2 Analysis of the Packing Problem Dataset

Hereinafter, the set of problems created for the experiments of this Chapter is referred to as Packing Problem Dataset. In the problems, the items are a homogeneous set of circles, triangles or squares, and the container is also either a circle, triangle or square; there are 9 combinations of item-container types, and one problem instance was created for each of them. The used item types are simple if compared to polygons of many sides or compound polygons with holes. Therefore, the geometric intersection and containing checks are less computationally expensive, so it was feasible to have problems with a considerably larger number of items (with a maximum of 100) than in the Joint Problem Dataset, where some items have more complex shapes. However, to have a greater diversity of problems, there are also instances with fewer number of items (with a minimum of 7).

In the created problems, the optimal solutions have in common that all the items can be placed in the container. The characteristics of the optimal solution were checked in Wolfram Alpha's packing solver ([WR19](#)), namely the percentage of coverage of the container's area when all items are placed (in an optimal solution). Such percentage, and other statistical information of each instance of the Packing Problem Dataset (including the dimensions of the items and the container), are shown in Table 8. One can see that the problems have a moderately high area pressure (with coverage percentages of optimal solutions ranging from 47.56% to 63%), though it is lower than in some of the problems of the Joint Problem Dataset, e.g. in Problem 10 it reached 88.94%. One may think that this makes the new problems easier, but the large number of items in some problems is a new difficulty, added to the fact that the used algorithms were not originally devised to solve the Packing Problem.

Table 8: Statistics of the problems of the Packing Problem Dataset and their optimal solutions.

	Item num.	Item side/ radius	Cont. side/ radius	Opt. % item num. in cont.	Items' area % of cont. area
Circles in circle	7	12	3.9	100	63
Triangles in circle	20	4	9.5	100	48.87
Squares in circle	12	3	7.8	100	56.5
Circles in triangle	10	1	9.5	100	59.96
Triangles in triangle	18	3.5	20	100	55.13
Squares in triangle	30	7.5	80	100	60.89
Circles in square	50	17	300	100	50.44
Triangles in square	15	4	14	100	53.02
Squares in square	100	4	58	100	47.56

8.3 Results and Discussion

A statistical overview of the solution quality results obtained by the algorithms in each of the problems of the Packing Problem Dataset is shown in Table 9, while a visualization of the best solution for each problem is present in Section A.5 of the Appendix. In these problems, the value of a solution is equivalent to the number of items placed in the container, and a solution is optimal if all the items have been placed. The bold notation is used in Table 9 to indicate which algorithm obtains the highest mean, minimum, median or maximum value in each problem,

while an underscore implies that the optimal value (checked using Wolfram Alpha’s packing solver ([WR19](#))) was reached. It is easy to observe that the evolutionary algorithm obtains the best results, achieving optimality in 5 of the 9 tested problems (at least in one of the 10 performed runs).

Table 9: Statistics of the solution value results for each algorithm in the problems of the Packing Problem Dataset, compared to the optimal solutions obtained with Wolfram Alpha’s packing solver.

Algorithm	Greedy					Reversible					Evolutionary					W. Alpha
	mean	std	min	med	max	mean	std	min	med	max	mean	std	min	med	max	
Statistic																optimum
Circles in circle	4.7	0.64	4	5	6	4.8	0.6	4	5	6	6.8	0.4	6	<u>7</u>	<u>7</u>	7
Triangles in circle	17.9	1.04	16	18	20	15.4	0.92	14	15.5	17	18.7	0.9	17	19	20	20
Squares in circle	8.9	0.7	8	9	10	8	0.63	7	8	9	10.3	0.46	10	10	11	12
Circles in triangle	7.4	0.8	6	7	9	6.4	0.49	6	6	7	9.8	0.4	9	<u>10</u>	<u>10</u>	10
Triangles in triangle	13.6	1.2	11	14	15	11.4	0.8	10	11	13	15.5	0.67	14	16	16	18
Squares in triangle	21.4	1.11	19	22	23	19.1	1.3	18	18.5	21	23	0.77	22	23	24	30
Circles in square	48.5	1.2	46	49	<u>50</u>	45.5	1.5	44	45.5	48	<u>50</u>	0	<u>50</u>	<u>50</u>	<u>50</u>	50
Triangles in square	11.9	0.94	10	12	13	10.1	0.7	9	10	11	13.9	0.7	13	14	<u>15</u>	15
Squares in square	97.6	2.46	94	98.5	<u>100</u>	90	2.32	86	90.5	93	95.5	1.28	93	95.5	97	100

As it happened in the experiments of the Joint Problem (in Chapter [7](#)), the evolutionary algorithm has the advantage of creating and managing not only a single solution at a time (as done in the greedy and reversible algorithms), but a whole population. This approach allows the evolutionary algorithm to expand the exploration of the search space, and progressively refine solutions by mutation, generating offspring and exploiting the highest-value solutions via elite keeping and fitness-proportional population update. This combination of exploration and exploitation leads the evolutionary algorithm to solutions of higher quality.

Conversely, the greedy and reversible algorithm manage a single solution at a time, even though the reversible algorithm can restore a solution to a pre-removal state if removing an item does not lead to an improvement in the solution value after some time. The quality of the results of the greedy algorithm is significantly higher than those of the reversible algorithm: in 8 of the 9 problems, the greedy algorithm obtains a higher mean, median and maximum value than the reversible algorithm. The removal strategy of the reversible algorithm seems more potentially useful in the Joint Problem, since items have different value and weight (and the container had a maximum weight acting as a constraint); in such context, removing an item may give an opportunity to place an item with higher value and lower weight, which would be potentially useful. However, in the Packing Problem all items have the same value, and the weight is ignored, so removing an item implies decreasing the value of the solution with a smaller likelihood of eventually achieving a better solution than in some scenarios of the Joint Problem. Since all the items have the same value, it is unsurprising that the greedy algorithm’s strategy of focusing only in adding items to the container (keeping the placements of the already added items unchanged) is more appropriate.

In fact, the addition approach of the greedy algorithm is effective enough to achieve better results than the evolutionary algorithm in the “Squares in square” problem, which has the largest number of items (100). Since the evolutionary algorithm does not focus exclusively on

adding items (but also occasionally removing items or modifying existing placements), it is plausible that when the number of items is large, it can be outperformed by an algorithm that focuses on getting items placed, at least in the Packing Problem (where all objects have the same value), in problems with a considerable percentage of free space. Specifically, one should think of problems in which the random-position accumulative item placement strategy of the greedy algorithm does not easily lead to situations in which a certain position or rotation of one or multiple items makes it impossible to place additional objects. Nevertheless, it is expected that if the evolutionary algorithm performed a higher number of epochs (e.g. 100) in problems with a very large number of items, it would be eventually able to obtain results of equivalent or higher quality than those of the greedy algorithm.

While maintaining and updating a whole population of solutions tends to lead the evolutionary algorithm to results of higher quality, it implies performing certain operations (namely placement attempts and the required geometric checks to ensure solution validity) many more times, which significantly increases the time of computation. Therefore, as shown in Table 10, the evolutionary algorithm is the slowest of the three algorithms, while the reversible algorithm obtains the fastest times (in bold) for all the problems. This is an interesting circumstance, since the time results of the Joint Problem experiments did not show significant differences in time between the greedy and the reversible algorithm. In the Packing Problem, at least in the tested cases, the reversible algorithm is significantly faster than the greedy method. To understand the reason that causes this phenomenon, it is useful to analyze, in a same problem, the time spent by the greedy algorithm (shown Figure 29) and the reversible algorithm (as seen in Figure 30). One can observe that since the greedy algorithm is solely dedicated to item addition, it spends the vast majority of the time attempting to place items, and geometrically validating (or discarding) the placements. Conversely the reversible algorithm tries to perform removals in a considerable percentage of iterations, even if most iterations are also devoted to perform addition attempts, or placement modifications; in both cases, geometric checks are required. The usage of removals can lead to facilitate further placements: since there is more space available, less attempts (i.e. less time) are needed to try to make new placements effective. This situation is not in conflict with the fact that the removal strategy is usually counter-productive for the solution quality (as seen previously in this section when comparing the algorithms), since the placements made by the greedy algorithm always help improve the final solution, while those of the reversible algorithm can be cancelled via removal, and eventually lead to poorer final solutions.

Despite of the ability of the proposed methods to obtain solutions of considerable quality in the tested problems, if a geometric optimization tool that guarantees finding optimal solutions is available, it should be preferred to solve the Packing Problem, especially when it is also faster. In the Packing Problem Dataset, Wolfram Alpha's solver returned a solution in less time than the proposed algorithms. The only (possible) exception may be in the fastest executions of the greedy and reversible algorithms (those from less than 1 second to about 3 seconds), but since the solver is used via website the author was not able to measure the exact time taken by Wolfram Alpha to perform the geometric optimization (i.e. separating the calculation from communication and page-loading times). Therefore, when packing equal circles, equilateral triangles or squares in containers chosen among those same shapes, it is more practical to use a solver that always provides an optimal solution. In some of the problems of the Packing Problem Dataset in which the items are triangles or squares, the optimal solution can be obtained even if using only one rotation angle (0 degrees), or just two (adding 180 degrees). Hence, it is easy to understand that the proposed algorithms, that do not assume or constraint orientations (since they are prepared for arbitrary problems that may require greater diversity) face an increased difficulty to solve the problems, compared to methods that apply angle constraints. Of course,

Table 10: Statistics of the execution time (in milliseconds) for each algorithm in the problems of the Packing Problem Dataset.

Algorithm	Greedy					Reversible					Evolutionary				
	Statistic	mean	std	min	med	max	mean	std	min	med	max	mean	std	min	med
Circles in circle	11718	2924	7837	11262	19155	1188	1219	104	711	3612	53358	9591	39993	51557	74487
Triangles in circle	29495	10115	15888	28477	48196	6707	1155	4392	6841	8340	128983	20693	88910	128645	158268
Squares in circle	21546	3048	16039	21243	26079	5291	2356	627	5523	8309	92970	14828	69090	90720	124333
Circles in triangle	15096	2013	10779	15213	19510	3732	1857	661	4334	5813	90995	10043	73988	92274	107548
Triangles in triangle	23024	4585	14943	22939	32097	6576	894	4830	6865	7823	127117	17367	93394	134287	143723
Squares in triangle	23698	4973	16507	23752	33381	8106	810	6447	8175	8901	138361	15574	104662	147375	151629
Circles in square	20904	7311	2286	22945	30490	6646	820	4746	6920	7524	182541	32785	109134	185819	224984
Triangles in square	25013	5354	16700	22741	32321	6116	914	4094	6426	7070	108574	19371	67710	110076	132873
Squares in square	39235	13827	14417	41365	60622	10026	1565	6878	10640	11946	255571	33845	186449	266038	289289

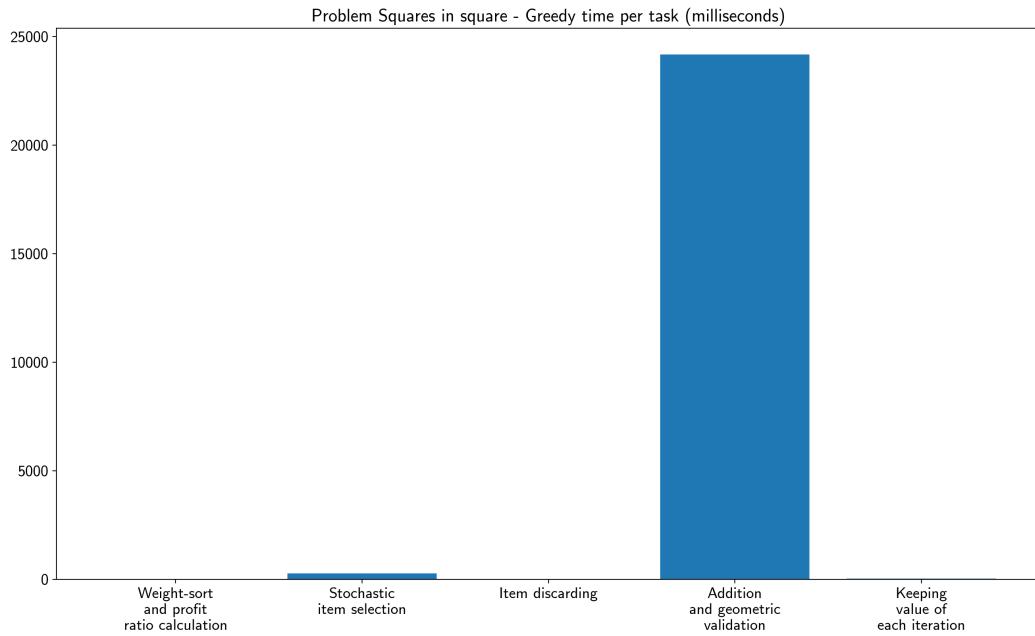


Figure 29: Time division of the tasks of the greedy algorithm in one of the tested problems.

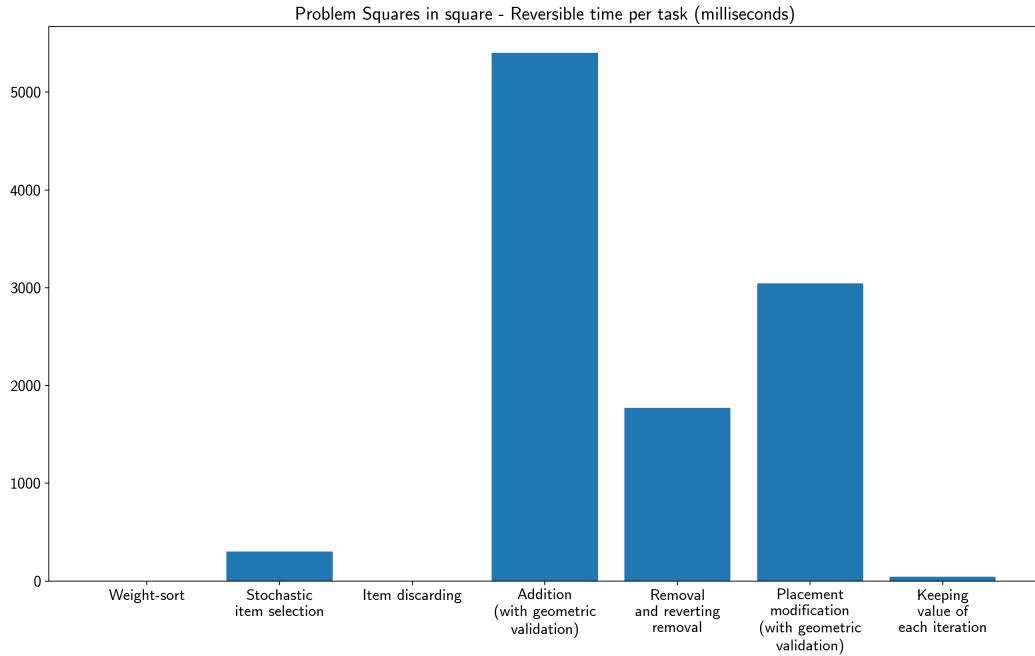


Figure 30: Time division of the tasks of the reversible algorithm in one of the tested problems.

it would be trivial to modify the proposed algorithms to use rotation angle constraints, and if such action was done, results of higher quality can be expected for these problems, but not for problems of more complex shapes. However, the aim of this chapter was to test the Packing Problem applicability of the proposed methods without changing anything from their original definition, devised for the Joint Problem.

In some of the tested problems, it is especially obvious that the proposed algorithms are not the most effective approach. Namely, in problems such as “Circles in square” or “Squares in squares”, a better (and actually optimal) solution can be obtained by simply using the algorithm implemented for visualizing the items that are outside the container (as explained in Section 4.2). Such algorithm arranges the items with an adjacency criteria, using bounding rectangles as size references, and it would just be required to move the set of items with that arrangement into the container, e.g. placing the center of the bounding rectangle formed by all the items in the center of the container’s shape. However, the situation is only expected to happen when using simple shapes in problems where all or most items have the same or very similar dimensions.

The author of this work was not able to find geometric optimization solvers that guarantee optimal results for problems that include some of the shapes that the proposed algorithms accept (e.g. irregular polygons of any side number, either concave or convex, or compound polygons with holes), even though previous methods that solve the Packing Problem have been reported to obtain optimal results in some problems (AS80; TG95), with methods that include evolutionary algorithms and simulated annealing, that generally cannot guarantee universal optimality of solutions. In other words, to the knowledge of the author, existing Packing algorithms are unable to guarantee optimal solutions for arbitrary problems with complex shapes, and in some cases the methods are even unable to accept those shapes, i.e. they are not only unable to ensure that optimal results are obtained, but even to find any (non-optimal) solution. In such cases, the proposed methods have the potential of being useful, especially the evolutionary algorithm, and

even the greedy algorithm if the problem has a very large number of items. These algorithms have shown the ability to obtain a notable quality of solutions in instances of Packing Problem with simple shapes, and using more complex shapes in problems of the Joint Problem Dataset. It would be interesting to test the proposed algorithms in a larger dataset of the Packing Problem, including problems whose items have complex shapes, as future work. Ideally, to be able to fully contextualize the quality of the solutions, it would be required to determine the optimal solutions, which may be difficult if no other method is able to obtain them (ensuring that the solutions are optimal), since that would require to try to solve the problems manually.

9 Conclusions and Future Work

9.1 Main Conclusions

The experiments of the Joint Problem performed in Chapter 7 showed that the designed evolutionary algorithm is capable of obtaining solutions of high quality in the tested set of problems (the Joint Problem Dataset, presented in Chapter 5). Namely, the evolutionary algorithm obtained optimal solutions in 7 of the 10 problems in at least one of the 10 performed runs, and in every run for 2 of the problems. In most of the other problems the solutions were close-to-optimal, except for Problem 10, an especially difficult problem where items need to be placed in very restricted ranges of positions (and rotations) to reach optimality.

In all the problems, the evolutionary algorithm obtained better results than the other two developed methods, greedy and reversible, except for a small number of runs in which they were in a tie; the evolutionary algorithm was not outperformed in any case. The success of the evolutionary algorithm can be explained by the fact that it creates and maintains a population of many candidate solutions in parallel, while the other algorithms focus on just one at a time. The evolutionary algorithm progressively refines the population throughout generations, by creating modified solutions using the mutation operator (adding new items to the container with high probability, or, with lower probability, removing placed items or modifying their position or rotation), selecting surviving individuals in each generation proportionally to their fitness. The crossover operator was not used in the final experiments: in a parameter optimization phase (explained in Chapter 6) it was seen that the presence of crossover did not improve solutions, even if not making them worse neither; using crossover made the execution slightly slower. Keeping and updating a population of individuals makes the evolutionary algorithm significantly slower than the greedy and reversible methods (in some problems the times are in a different order of magnitude). In the experiments of the Joint Problem, the advantages and limitations of both the greedy and reversible algorithms led them to obtain results of similar quality (which is acceptable, yet generally not very good), in a similar time.

In the experiments of Chapter 8, it was shown that the applicability of the proposed algorithms is not limited to the Joint Problem. Instead, they can be applied to a traditional Packing Problem (devoted to maximize the number of items placed in the container) without having to change any aspect of the methods, just adding minor changes in the input data to adapt it to the expected format. The evolutionary algorithm obtained optimal results in at least one of 10 runs in 5 of the 9 tested problems, and close-to-optimal results in the majority of other problems. The evolutionary algorithm outperformed the greedy and reversible algorithm in most problems, due to the same reasons that made it effective for the Joint Problem, as explained in the previous paragraph. However, an external method, Wolfram Alpha's packing solver ([WR19](#)), is able to obtain optimal results in all the tested problems, which use simple shapes. Therefore, when using simple shapes, the proposed methods are not the most competitive option. Nevertheless, the proposed algorithms are capable of solving problems with more complex shapes, e.g. irregular polygons and compound polygons, even if unable to guarantee optimality, while the mentioned mathematical solver cannot (currently) give any solution for such problems.

The following sections of this chapter examine possible future modifications in the evolutionary algorithm that may (hypothetically) lead to find even more effective ways of solving the Joint Problem, or become capable of solving alternative problems in a competitive way. In any case, it should be noted, as explained in Section 4.3, that the whole code of the original algorithm is available in a public repository, which should facilitate the development of future work.

9.2 Possible modifications for the evolutionary algorithm applied to the Joint Problem

9.2.1 Multi-item placement specialization in the generation of the initial population

In the current version of the algorithm, an item specialization process is used in the generation of the initial population as an attempt to make all the items (or at least a large number of them) represented in some solutions (or at least one), by focusing on the placement of an individual item (before any other item) for a certain initial solution. This approach is useful to place some items which are either very large or have challenging shapes for the container; when not using specialization, small items are usually placed first, making the placement of the difficult-to-place items less likely in later attempts. If one of the latter items helps to make a solution more valuable than the average, the solution has a higher likelihood of being preserved and mutated throughout generations, and additional items can be added to the container. However, if the optimal solution requires adding to the container more than one difficult-to-place item, it is considerably unlikely that the evolutionary algorithm reaches optimality, especially if there are many other items which are much easier to place (e.g. small ones), that block potential positions where the ideal items would be placed.

To tackle this problem, it would be possible to extend the item specialization operation to more than one item, such as pairs and triples. It would be advisable to study the specific context of application (e.g. a set of representative problems) to choose an appropriate number; keeping just one item might be the best option in some cases (e.g. if it is known that all problems will just have a very large item, and many tiny ones). If the multi-item approach was used, each item in the group would be specialized in a certain sequential order, or using a random selection in each placement attempt. However, to be able to represent all the possible combinations of items, one would need to have either a small number of items or a large population size, to be able to represent each combination in at least one of the initial solutions. Alternatively, it would be possible to define a heuristic method to determine which items should be considered as difficult-to-place (and optionally even choose only the most valuable ones, or those with a higher value-weight ratio), so that only combinations of those items are used in specialization: the easy-to-place items would be naturally added via the normal mutation operator throughout generations. However, if specialization is applied only to a finite subset of items, a certain number of initial solutions should be generated without any specialization, to be consistent with the fact that the optimal solution may not need any difficult-to-place item in some problems.

Some of the elements that may be worth considering to define the heuristic include the area of the item and the largest-extent side of its boundary rectangle, which are, intuitively, proportional to the placement difficulty. It is also worth to empirically determine whether making the random selection of items to place weighted (instead of uniform) may improve the solution quality. Namely, it would be possible to give a higher chance of selection to those items that are considered more difficult to place (using a heuristic estimation score), to compensate the fact that, on average, they are expected to be less likely to be (successfully) added to the container. Such weighting selection should be used carefully (with a well-defined heuristic score) and not too aggressively, to avoid restricting too much the order in which items get placed in the container, since it would reduce the diversity of solutions, and therefore the possibilities of exploration. If a heuristic score was used for weighted selection of items, considering the item's value and weight (as done in the greedy algorithm) may also be interesting, since the algorithm seeks the maximization of the value given the weight constraint of the container.

9.2.2 Nested movement and rotation of compound polygons and the items placed in their holes

In the mutation operator, when performing modification actions on an already placed item, namely movement or rotation, it may be interesting (in future work) to consider a special scenario involving compound polygons which have holes, since they can contain other shapes in the holes. In some circumstances, when a modification is applied to a compound shape, it may be suitable to move or rotate not only the shape but also all the shapes that have been placed inside the compound object (recursively, i.e. also for contained compound shapes with their content shapes), to preserve the relative position and rotation of the contained shapes, with respect to the compound shape. The movement case would require to apply the same displacement to the shapes within the compound shape as to that shape (calculating the relative position when the movement consists in moving the compound shape reference point to a specific position). It is therefore conceptually simple, but would require to do more intersection checks, which would make the operations slower. Particularly, in the validation of the candidate solution resulting from the action, it would be necessary to check if all the moved shapes intersect with any other non-moved shape or the container, while it would be not needed to check for intersections between the moved shapes since their relative position and rotation is preserved. The additional intersection computations are also required in the compound rotation case. Furthermore, in such scenario, to keep the alignment with the compound shape, the internal shapes would need to be rotated with the same rotation origin as the compound shape. The drawback of this operation is that it would require to recalculate the absolute reference position and rotation of the internal shapes to be consistent with their definition and guarantee that the placements of items in the container of the final solution would be trivially reproducible. In the case of the reference position, it should be updated to match again the bounding rectangle center that the shapes would have with 0-degree rotation, which may not be trivial to calculate in a very efficient way after the performed indirect-origin rotation, which would also be the case for the new angle of rotation, knowing that it would not be the rotation angle of the compound shape, but another one to be calculated based on the distance between the internal and the compound shape, and also taking into account the angle that the internal shape already had.

However, in some scenarios, moving or rotating only the compound shape and not the internal ones may lead to a placement setting that facilitates the addition of new items to the container, and this was the approach used in this work. As mentioned, the compound (or nested) approach, involving the movement or rotation of the shapes contained in a compound shape, would involve performing geometric checks for every other placed item to see if one of the holes of the compound shape contains it, which may add a considerable computational overhead, which would be only valuable if this method would lead to faster convergence to solutions of the same (or higher) quality. The overhead may be reduced by storing the shapes contained by each compound shape, but would still require to perform the geometric checks every time an item is added to the container or its placement is modified, and updating the lists when removing items. For research purposes, it may be interesting to test the two types of scenarios, performing experiments comparing the usage of one method alone or both combined. It would be possible to have both variants of movement or rotation in a same problem by means of the following approach: when a movement or rotation action would have to be performed in a compound shape in the context of the mutation operator, a probability P_{comp} would be used to determine whether a compound action should be applied, i.e. moving or rotating also the shapes found within the compound shape, or otherwise a standard single-shape rotation should be selected.

9.2.3 Self-evolution of parameters

When designing the algorithm, the possibility to encode self-evolution parameters in solutions was considered. Such parameters may include mutation-related probabilities and weights, along with their standard derivation and covariance. Self-evolution is used in evolution strategies to tune certain parameters of an evolutionary algorithm and guide it towards solutions of higher quality (BS02). It is important to outline, though, that incorporating self-evolution would initially increase the complexity of the algorithm and its computational time, a drawback that would be only compensated if it would eventually converge faster to results of similar or higher quality. However, there is no theoretical guarantee that self-evolution would improve the quality of solutions. Based on the hypothesis that the benefits of its addition may not compensate the drawbacks, self-evolution was not incorporated into the algorithm. Nevertheless, it may be an interesting direction for future work to study and eventually design and implement a specific self-evolution scheme for an evolutionary algorithm to solve the Joint Problem, if a preliminary analysis makes it seem a potentially useful improvement.

Alternative approaches to let the evolutionary algorithm update the value of parameters include the usage of adaptive strategies (PK96), which have been applied in previous works to both the Packing Problem (HWK99) and the Knapsack Problem (CJ07). Parameter adaptation can be achieved by repeating the execution of the evolutionary algorithm multiple runs, varying the value of parameters (e.g. population size) with a certain criterion; for instance, increasing the value of the parameter if the elite fitness has increased since the previous run, or decreasing the value otherwise. Elitism can be extended from a generational context to an inter-run scenario, i.e. preserving the best solution found at any run. The drawback of using multiple runs is that the computational time of the algorithm increases (although early stopping can be performed if the elite fitness does not improve after some runs), so it was discarded for this work. Nevertheless, adaptive strategies may help to converge to higher-quality solutions (PK96), so exploring their integration in the current evolutionary algorithm would be an interesting direction for future work. However, it may not be appropriate to apply the adaptation of multiple (or at least not many) parameters at the same time, since then it would be difficult to assess the influence of each of the concurrently modified parameters in the solution quality of new runs.

9.3 Possible modifications for the evolutionary algorithm to apply it to other problems

9.3.1 Area Minimization Packing Problem

In the literature of the Two-Dimensional Packing Problem, a commonly approached problem sub-type is the Area Minimization Packing Problem (AS80; Hop00), in which the items must be placed in a region with the smallest possible area; alternatively, some algorithms minimize the smallest possible extent in one dimension. In many cases, the dimensions of the container are not fixed, even though a maximum value for the extents is frequently given. The container is usually expected to be rectangular, optimized as the minimum bounding rectangle that contains all items in the final solution. Some existing algorithms (Jak96) compact the items by placing the first one in a corner of the rectangle (e.g. in the bottom left corner), and then sliding the following items (e.g. starting in the opposite corner) in an iterative way until they are next to already placed ones. While these approaches have been widely used in the past with considerably successful results (Hop00), this section is devoted to examining how the proposed algorithm can be changed to be effective in the area minimization problem, without completely changing its definition.

In its current formulation, the evolutionary algorithm does not try to minimize the space in which items are placed, since that was not the main aim of the Joint Problem, where value maximization is the priority. It would be possible to promote the minimization of the area by defining a certain position as the center of placements (e.g. the center of the container's shape), and calculating the position of a placement attempt with a new formula: instead of uniformly selecting a point within the container's bounding rectangle, selecting coordinates proportionally to their closeness to the center, i.e. making near-to-center points be used more often in item placement attempts. This way, there would be a tendency to place items near to a certain point (the center), but with enough flexibility to progressively go further from there as the nearby space becomes full. Additionally, when selecting the item to perform a removal in mutation, it should be more likely to select items far away from the center than those lying near to it, inverting the recently mentioned formula. Furthermore, placement modifications would need to be adapted too: all the operations that select a new position for an item, should make it more likely to move the item near to the center (using the first formula). In the operation of moving an item in a random direction until it intersects, the most likely direction of movement of the item should be the one of the vector that goes from the reference position of the item to the center of placements: the probability of choosing a direction should be inversely proportional to its angle with that vector, to promote the progressive process of compacting items together (but being flexible enough to escape local optima).

9.3.2 Three-Dimensional Irregular Shape Packing Problem combined with the Knapsack Problem

The Three-Dimensional Irregular Shape Packing Problem combined with the Knapsack Problem, or Three-Dimensional Joint Problem for short, can be defined analogously to its two-dimensional counterpart, with the exception that both the items and the container have three-dimensional shapes. The objective is still the maximization of the value of items in the container, given the same constraints of not exceeding the container's capacity and not causing intersections between items or between an item and the boundaries of the container. Given this formulation, the current evolutionary algorithm should be directly applicable to the three-dimensional problem, with the only requirement of providing an appropriate representations for the shapes, and a method for checking the intersection of each pair of shape types, and whether an item of one of the shapes is contained in the other one. It would be reasonable to support simple shapes (e.g. spheres, rectangular cuboids, cylinders, prisms, cones, pyramids and tori) and complex ones, such as three-dimensional models (commonly defined using polygon meshes). It is well-known that geometric tests for three-dimensional shapes are significantly more expensive than for two-dimensional shapes, so it might not be feasible to solve the problem in a reasonable time if there were many items, especially if they have complex shapes, using conventional (single-machine) computing resources. For this reason, it is common to see how the algorithms that solve a Three-Dimensional Packing Problem use only simple shapes, such as making both the objects and containers to be rectangular cuboids ([HH11](#)), and simplifying the space of positions as a discrete grid can also help, as well as limiting the possible rotations to a small finite set.

One of the potential scenarios of applications of the Three-Dimensional Joint Problem is that of logistics, i.e. transporting real-world objects in a container. Items would have a certain value, expressing either a monetary cost, the priority of delivery or some other notion, as well as a weight, which can be (if it desired) the real weight of the object (e.g. expressed in kilograms), while the container's weight would be the maximum weight in which it is feasible to move the vehicle in normal conditions. In the logistics scenario, it is necessary to introduce additional constraints that a solution must satisfy to be practical when applied in a real-world situation. Firstly, the objects cannot be floating, but placed on top of the floor of the container or another

item: this requirement can be called the gravity constraint (and may not be applicable to a space vehicle once in a zero-gravity scenario). In the case of placing items on top of others, a secondary aspect of the gravity constraint may be required too: the items should be placed in such way that they would not fall, for instance centered on top of another object, or at least not lying on the edge. It may be suitable to perform physical simulations considering the mass distribution of objects in the chosen rotation (and other aspects, such as the friction coefficients of the materials of the items) to determine the validity of a placement of one object on top of another one. It would be possible to introduce other constraints, such as one related to vehicle stability, to ensure that the mass of objects is distributed in a reasonably even way in the volume of the container, according to a specified formula (which may depend on particular characteristics of the vehicle, as well as the weight of the items). Intuitively, the greater the number and complexity of the constraints, the more difficult (and time-consuming) would (potentially) be to find a high-quality solution. Despite of that, most of the core features of the evolutionary algorithm would not need to be changed; the constraint checks would be added to the process of determining the validity of solutions. Additionally, the process of selecting candidate positions (now in three-dimensional coordinates) for item additions or placement modifications would be changed. In the gravity-respecting scenario, a position cannot be any point of the container's volume, Instead, it should be selected among feasible regions of the top surfaces of items (just outside of them), and the floor surface inside the container's shape. Detecting which surfaces are on top and which zones of them would make a placement feasible may not be trivial. The mutation operation of moving an item in a direction should be performed keeping the distance to the floor surface constant, i.e. moving through the surface, to respect the gravity constraint.

9.4 The Optimality Goal

It should be noted the Joint Problem operates in the continuous space: there are theoretically infinite position coordinates and rotation angles that can be tried for each item; in practise, it is not infinite but a very large number, once the problem is brought to the computational domain, since floating point representations have a finite number of values. Therefore, an exhaustive method would hypothetically try all the possible combinations of position coordinates and rotation angles for each item (in all the required placement orders) to guarantee that the optimal solution is found, which is computationally unfeasible. Discretizing the position coordinates in a grid and constraining the possible rotations to a finite set can eliminate the continuous nature of the Joint Problem. Such approach has been used in some algorithms ([TCR⁺13](#)) to solve the Packing Problem with non-trivial shapes, including irregular polygons, and obtaining results (for some tested problems) that are optimal in the solution space of the discretizations, but which are not necessarily optimal in the unrestricted version of the problem.

For the Joint Problem, one cannot expect to devise a method that always obtain the optimal solution, unless a general geometrical method is proposed. There are formulae for cases of the Packing Problem with simple shapes for the container and the items, e.g. squares ([Wei05](#)) or circles ([Wei02](#)), but of course it would be more challenging for arbitrary shapes, or at least the types tested in this work (circles, ellipses, polygons and compound polygons with holes). Such hypothetical method would also need to incorporate some kind of optimization method for the value, and a mechanism to respect the capacity constraint. The author is not sure of whether it would be possible to create an optimal method with these characteristics, but exploring this option would arguably be an interesting direction for future work.

References

- [AA76] Michael Adamowicz and Antonio Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8(1):27–33, 1976.
- [APR00] Rumen Andonov, Vincent Poirriez, and Sanjay Rajopadhye. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2):394–407, 2000.
- [AS80] Antonio Albano and Giuseppe Sapuppo. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(5):242–248, 1980.
- [BB13] Paul Belitz and Thomas Bewley. New horizons in sphere-packing theory, part ii: lattice-based derivative-free optimization via global surrogates. *Journal of Global Optimization*, 56(1):61–91, 2013.
- [BBM93] David Beasley, David R Bull, and Ralph Robert Martin. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, 15(2):56–69, 1993.
- [BGB10] Leo Budin, Marin Golub, and Andrea Budin. Traditional techniques of genetic algorithms applied to floating-point chromosome representations. *sign*, 1(11):52, 2010.
- [BS02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies—a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.
- [BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [CFLR03] Ping Chen, Zhaohui Fu, Andrew Lim, and Brian Rodrigues. Two-dimensional packing for irregular shaped objects. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pages 10–pp. IEEE, 2003.
- [CJ07] Rung-Ching Chen and Cheng-Huei Jian. Solving unbounded knapsack problem using an adaptive genetic algorithm with elitism strategy. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 193–202. Springer, 2007.
- [CJW00] YJ Cao, L Jiang, and QH Wu. An evolutionary programming approach to mixed-variable optimization problems. *Applied Mathematical Modelling*, 24(12):931–942, 2000.
- [CS03] Charles R Collins and Kenneth Stephenson. A circle packing algorithm. *Computational Geometry*, 25(3):233–256, 2003.
- [DDB98] KA Dowsland, WB Dowsland, and JA Bennell. Jostling for position: local improvement for irregular cutting patterns. *Journal of the Operational Research Society*, 49(6):647–658, 1998.
- [DJ95] Rahul Dighe and Mark J Jakielka. Solving pattern nesting problems with genetic algorithms employing task decomposition and contact detection. *Evolutionary Computation*, 3(3):239–266, 1995.

- [DM96] Karen Daniels and Victor J Milenkovic. Column-based strip packing using ordered and compliant containment. In *Workshop on Applied Computational Geometry*, pages 91–107. Springer, 1996.
- [DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.
- [DRG19] Daniel Domović, Tomislav Rolich, and Marin Golub. Evolutionary hyper-heuristic for solving the strip-packing problem. *The Journal of The Textile Institute*, pages 1–11, 2019.
- [DVDQMX12] Aline M Del Valle, Thiago A De Queiroz, Flavio K Miyazawa, and Eduardo C Xavier. Heuristics for two-dimensional knapsack and cutting stock problems with items of irregular shape. *Expert Systems with Applications*, 39(16):12589–12598, 2012.
- [ES⁺03] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.
- [For93] Stephanie Forrest. Genetic algorithms: principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993.
- [FS75] Herbert Freeman and Ruth Shapira. Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Communications of the ACM*, 18(7):409–413, 1975.
- [GACGW03] Vladimir B Gantovnik, Christine M Anderson-Cook, Zafer Gürdal, and Layne T Watson. A genetic algorithm with memory for mixed discrete–continuous design optimization. *Computers & Structures*, 81(20), 2003.
- [Gil10] Sean Gillies. Shapely user manual, 2010. Accessed: 2019-09.
- [GO02] A Miguel Gomes and José F Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141(2):359–370, 2002.
- [Gom13] A Miguel Gomes. Irregular packing problems: Industrial applications and new directions using computational geometry. volume 46, pages 378–383. Elsevier, 2013.
- [HAAN⁺16] Ahmad Hassanat, Esra' Alkafaween, Nedal A Al-Nawaiseh, Mohammad A Abbadi, Mouhammd Alkasassbeh, and Mahmoud B Alhasanat. Enhancing genetic algorithms using multi mutations. *arXiv preprint arXiv:1602.08313*, 2016.
- [HH11] Kun He and Wenqi Huang. An efficient placement heuristic for three-dimensional rectangular packing. *Computers & Operations Research*, 38(1):227–233, 2011.
- [HK02] Kuk-Hyun Han and Jong-Hwan Kim. Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. volume 6, pages 580–593. IEEE, 2002.
- [HK13] Eric Huang and Richard E Korf. Optimal rectangle packing: An absolute placement approach. *Journal of Artificial Intelligence Research*, 46:47–87, 2013.
- [HL95] Jörg Heistermann and Thomas Lengauer. The nesting problem in the leather manufacturing industry. *Annals of Operations Research*, 57(1):147–173, 1995.

- [HLM96] Arild Hoff, Arne Løkketangen, and Ingvar Mittet. Genetic algorithms for 0/1 multidimensional knapsack problems. In *Proceedings Norsk Informatikk Konferanse*, pages 291–301. Citeseer, 1996.
- [Hop00] Eva Hopper. *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods*. PhD thesis, University of Wales. Cardiff, 2000.
- [HQL94] Abdollah Homaifar, Charlene X Qi, and Steven H Lai. Constrained optimization via genetic algorithms. *Simulation*, 62(4):242–253, 1994.
- [Hun07] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.
- [HWK99] Koichi Hatta, Shinichi Wakabayashi, and Tetsushi Koide. Solving the rectangular packing problem by an adaptive ga based on sequence-pair. In *Proceedings of the ASP-DAC'99 Asia and South Pacific Design Automation Conference 1999 (Cat. No. 99EX198)*, pages 181–184. IEEE, 1999.
- [Jak96] Stefan Jakobs. On genetic algorithms for the packing of polygons. *European journal of operational research*, 88(1):165–181, 1996.
- [Jas02] Andrzej Jaszkiewicz. On the performance of multiple-objective genetic local search on the 0/1 knapsack problem-a comparative experiment. *IEEE Transactions on Evolutionary Computation*, 6(4):402–412, 2002.
- [JG98] Sakait Jain and Hae Chang Gea. Two-dimensional packing problems using genetic algorithms. *Engineering with Computers*, 14(3):206–213, 1998.
- [JPSP14] Bonfim Amaro Júnior, Plácido Rogério Pinheiro, Rommel Dias Saraiva, and Pedro Gabriel Calíope Dantas Pinheiro. Dealing with nonregular shapes packing. *Mathematical Problems in Engineering*, 2014, 2014.
- [MA94] Zbigniew Michalewicz and Jarosław Arabas. Genetic algorithms for the 0/1 knapsack problem. In *International Symposium on Methodologies for Intelligent Systems*, pages 134–143. Springer, 1994.
- [MBM03] George G. Mitchell, David Barnes, and Mark Mccarville. Generepair- a repair operator for genetic algorithms, 2003.
- [MOGF07] Luís M Moreira, José F Oliveira, A Miguel Gomes, and J Soeiro Ferreira. Heuristics for a dynamic rural postman problem. *Computers & operations research*, 34(11):3281–3294, 2007.
- [MPT99] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.
- [MPV00] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations research*, 48(2):256–267, 2000.
- [OD94] David Orvosh and Lawrence Davis. Using a genetic algorithm to optimize problems with feasibility constraints. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 548–553. IEEE, 1994.

- [Pis00] David Pisinger. A minimal algorithm for the bounded knapsack problem. *INFORMS Journal on Computing*, 12(1):75–82, 2000.
- [PK96] Zhengjun Pan and Lishan Kang. An adaptive evolutionary algorithm for numerical optimization. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 27–34. Springer, 1996.
- [Pre03] Lutz Prechelt. Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. *Advances in Computers*, 57(2003):205–270, 2003.
- [RGP⁺01] Ignacio Rojas, Jesús Gonzalez, Héctor Pomares, FJ Rojas, FJ Fernández, and Alberto Prieto. Multidimensional and multideme genetic algorithms for the construction of fuzzy systems. *International Journal of Approximate Reasoning*, 26(3):179–210, 2001.
- [Ros99] Brian J Ross. A lamarckian evolution strategy for genetic algorithms. 1999.
- [Ros08] David Rosen. Seamless intersection between triangle meshes. In *Proceedings: Senior Conference on Computational Geometry*, pages 1–8, 2008.
- [SD08] SN Sivanandam and SN Deepa. Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer, 2008.
- [SNB98] Marc Stelmack, Nari Nakashima, and Stephen Batill. Genetic algorithms for mixed discrete/continuous optimization in multidisciplinary design. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, page 4771, 1998.
- [SVM93] Dirk Schlierkamp-Voosen and H Mühlenbein. Predictive models for the breeder genetic algorithm. *Evolutionary Computation*, 1(1):25–49, 1993.
- [SWM08] Chaoming Song, Ping Wang, and Hernán A Makse. A phase diagram for jammed matter. *Nature*, 453(7195):629, 2008.
- [TCR⁺13] Franklina MB Toledo, Maria Antónia Carravilla, Cristina Ribeiro, José F Oliveira, and A Miguel Gomes. The dotted-board model: a new mip model for nesting irregular shapes. *International Journal of Production Economics*, 145(2):478–487, 2013.
- [TG95] Vassilios E Theodoracatos and James L Grimsley. The optimal packing of arbitrarily-shaped polygons using simulated annealing and polynomial-time cooling schedules. *Computer methods in applied mechanics and engineering*, 125(1–4):53–70, 1995.
- [Wei02] Eric W Weisstein. Circle packing. 2002.
- [Wei05] Eric W Weisstein. Square packing. 2005.
- [WR19] Inc. Wolfram Research. Geometric packing in 2d - wolfram—alpha examples, 2019. Accessed: 2019-12.
- [Wri91] Alden H Wright. Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms*, volume 1, pages 205–218. Elsevier, 1991.

- [ZT98] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithmsa comparative case study. In *International conference on parallel problem solving from nature*, pages 292–301. Springer, 1998.
- [ZZZ06] Peiyi Zhao, Peixin Zhao, and Xin Zhang. A new ant colony optimization for the knapsack problem. In *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design*, pages 1–3. IEEE, 2006.

A Appendix

A.1 Pseudo-code of the reversible algorithm

Algorithm 9: Reversible algorithm (I)

Input : container /*Container, described by its shape and maximum weight*/,
 items /*List of items, described by their value, weight and shape*/,
 max_iter_num /*Maximum number of iterations*/,
 converge_iter_num /*Number of consecutive iterations to stop the algorithm if
 there are no placements*/,
 revert_iter_num /*Number of iterations to revert a solution to a previous state,
 if a recent removal did not increase the value*/,
 remove_prob /*Probability to remove an item in an iteration*/,
 consec_remove_prob /*Probability to remove an item if a removal has already
 taken place recently*/,
 ignore_prob /*Probability to ignore placing an item after removing it, until the
 operation gets reverted or permanently accepted*/,
 modify_prob /*Probability to modify an existing placement (move or rotate)*/

Output: solution /*Solution, describing a placement of items in the container*/

```

1 /*Create an initial solution with no placements, to be modified*/
2 solution ← create_solution(container, items);
3 /*Sort the items by weight, to make filtering faster*/
4 items_by_weight ← sort_by_weight(items);
5 /*Discard the items whose weight is greater than the container's capacity*/
6 items_by_weight ← filter_items(items_by_weight, container.max_weight);
7 iter_count_since_addition ← 0; iter_count_since_removal ← 0;
8 solution.before_removal ← null; ignored.item_index ← -1;
9 /*Placements can only be possible with capacity and valid items*/
10 if container.max_weight > 0 and items_by_weight ≠ ∅ then
11   /*Try to add items to the container, for a maximum number of iterations*/
12   for i ← 1 to max_iter_num do
13     /*Perform a random choice of the next item to try to add*/
14     item ← random_choice(items_by_weight);
15     /*Try to place the item in the container, with random position and rotation*/
16     if solution.add_item(item.index, get_random_position(container.shape),
17       get_random_rotation()) then
18       /*Determine the weight that can still be added*/
19       remaining_weight ← container.max_weight - solution.weight;
20       /*Stop early if the capacity has been exactly reached*/
21       if remaining_weight == 0 then
22         break;
23       /*Remove the placed item from the list of pending items*/
24       items_by_weight.pop(item.index);
25       /*Discard the items whose weight is greater than the capacity*/
26       items_by_weight ← filter_items(items_by_weight, remaining_weight);
27       /*Stop early if there are no items remaining or the capacity would be exceeded*/
28       if items_by_weight == ∅ then
29         break;
30       /*Reset the potential convergence counter, since an item has been added*/
31       iter_count_since_addition ← 0;
32   /*The algorithm continues in the next page*/

```

Algorithm 9: Reversible algorithm (II)

```
/*The previous page's last line was inside the following if statement and for loop*/
if container.max_weight > 0 and items_by_weight ≠ ∅ then
    for i ← 1 to max_iter_num do
        /*The following if statement had been just used in the previous page*/
        if solution.add_item(item.index, get_random_position(container.shape),
            get_random_rotation()) then
            ... /*See previous page*/
        /*The algorithm of the previous page continues as follows*/
    else
        /*Register the fact of being unable to place an item this iteration*/
        iter_count_since_addition ← iter_count_since_addition + 1;
        /*Stop early if there have been too many iterations without placements*/
        if iter_count_since_addition == converge_iter_num then
            break;
    /*If there are items in the container, try to remove an item with a certain
    probability, which depends on whether there was a recent removal*/
    if solution.weight > 0 and uniform_float(0, 1) < get_cond_prob(remove_prob,
        consec_remove_prob, solution_before_removal == null) then
        /*If there is no solution prior to a removal with pending re-examination*/
        if solution.before_removal == null then
            /*Save the current solution before removing, just in case it needs to be
            restored later*/
            solution_before_removal ← solution.copy();
            /*Reset the counter of iterations since removal, to avoid reverting earlier
            than needed*/
            iter_count_since_removal ← 0;
        /*Get the removed item, which is randomly chosen*/
        removed_item ← solution.remove_random_item();
        /*With a certain probability, only if not ignoring an item already and if there
        are pending items, ignore placing again the removed item until the operation
        gets reverted or permanently accepted*/
        if ignored_item_index < 0 and items_by_index.num() > 0 and
            uniform_float(0, 1) < ignore_prob then
            ignored_item_index ← removed_item.index;
        /*Otherwise, add the removed item to the weight-sorted list of items to try to
        place*/
        else
            insert_with_order(items_by_weight, removed_item);
    /*The algorithm continues in the next page*/
```

Algorithm 9: Reversible algorithm (III)

```
/*The previous page's last line was inside the following if statement and for loop*/
if container.max_weight > 0 and items_by_weight ≠ ∅ then
    for i ← 1 to max_iter_num do
        /*The algorithm of the previous page continues as follows*/
        /*If there is a recent removal to be confirmed or discarded after some time*/
        if solution.before_removal ≠ null then
            /*Re-examine a removal after a certain number of iterations*/
            if iter_count_since_removal == revert_iter_num then
                /*If the value in the container has improved since removal, accept the
                 operation in a definitive way*/
                if solution.value > solution.before_removal.value then
                    /*If an item had been ignored, make it available for placement again*/
                    if ignored_item_index >= 0 then
                        insert_with_order(items_by_weight, items.get(ignored_item_index));
                /*Otherwise, revert the solution to the pre-removal state*/
                else
                    solution ← solution.before_removal;
                    /*After reverting, have some time to try to add items*/
                    iter_count_since_addition ← 0;
                /*Reset removal data*/
                solution.before_removal ← null; iter_count_since_removal ← 0;
                ignored_item_index ← -1;
            /*Otherwise, the check will be done after more iterations*/
            else
                iter_count_since_removal ← iter_count_since_removal + 1;
        /*If there are still items in the container (maybe there was a removal), modify the
         existing placements with a certain probability*/
        if solution.weight > 0 and uniform_float(0, 1) < modify_prob then
            /*Perform a random choice of the item to affect*/
            item ← random_choice(items_by_weight);
            /*Try to move the item to a random position of the container with a
             probability of 50%*/
            if uniform_float(0, 1) < 0.5 then
                solution.move_item_to(item.index, get_random_position(container.shape));
            /*Otherwise, try to perform a random rotation*/
            else
                solution.rotate_item_to(item.index, get_random_rotation());
        /*In the end, revert the last unconfirmed removal if it did not improve the container's
         value*/
        if solution.before_removal ≠ null and
            solution.value < solution.before_removal.value then
                solution ← solution.before_removal;
```

A.2 List of parameters of the evolutionary algorithm

The evolutionary algorithm has a considerable number of parameters that are independent of the data of the specific problems to be solved, which has motivated to have a complete list with all of them. It should be noted that some parameters may not be used depending on the value of others, e.g. the crossover-related probabilities are ignored if the parameter of crossover usage fixes that the operator cannot be applied. The parameters are divided in the same groups used in Chapter 6, where parameter optimization was performed, and the values for the comparative experiments of Chapter 7 were fixed; such values are mentioned here as well, as default values. The parameters within each group are presented in alphabetic order of description. The grouped parameters are as follows:

- Generation of the initial population:
 - Maximum number of iterations to generate a solution to be part of the initial population, I_{max} . Default value: 300.
 - Maximum number of iterations without improvements in the solution's value to assume convergence (and perform early stopping) in the generation of a solution that is going to be part of the initial population, I_{conv} . Default value: 50.
 - Population size, μ . Default value: 100.
 - Proportion of the maximum number iterations in which the algorithm tries to place a specialization item before any other item, S . The proportion sets the maximum iterations to try to place the item, but the specialization can stop earlier if a placement attempt succeeds. Default value: 0.5.
- Parent selection and offspring generation:
 - Boolean property describing whether crossover can be used to generate offspring (otherwise only mutation is applied), U_c . Default value: False.
 - Minimum number of offspring created in a generation, λ . Default value: 200.
 - Pool size for the tournament strategy used in parent selection, T_p . Default value: 3.
- Crossover:
 - Maximum number of attempts to try to produce a partitioning shape for crossover that intersects with the container (covering part of its area), M_c . Default value: 5.
 - Maximum number of permutations determining the possible order in which item that lied partially in both regions of a crossover partitioning can be tried to be placed again in the container, R . Default value: 5.
 - Maximum number of vertices that can be used to generate a polygon to partition the container space of parents in two regions when performing crossover, \bar{V}_c . Default value: 10.
 - Maximum proportion of the side length of the container's bounding rectangle (for the relevant axis) that can be used to define the length (in the same axis) of a non-segment partitioning shape in crossover, \bar{L}_{max} . Default value: 0.75.
 - Minimum proportion of the container's area that a closed-region shape needs to have to be accepted as a valid partitioning shape in crossover, A . Default value: 0.1.
 - Minimum proportion of the side length of the container's bounding rectangle (for the relevant axis) that can be used to define the length (in the same axis) of a non-segment partitioning shape in crossover, \bar{L}_{min} . Default value: 0.25.
 - Proportion of individuals in a population that the candidate solutions generated in crossover permutations (for items that lied in both regions of the partitioning of the container) need to outperform in fitness to be eligible in the next population update, C . Default value: 0.95.
- Mutation:

- Maximum number of attempts to try to produce a valid addition of a specific item in a step of the mutation operator, M_a . Default value: 10.
- Maximum number of attempts to try to produce a valid placement modification of a specific item in a step of the mutation operator, M_m . Default value: 3.
- Minimum number of iterations of a single execution of the mutation operator, I_{mut} . Default value: 5.
- Maximum number of positions to check in the placement modification mutation that tries to move an item in a direction until it intersects, B_p . Default value: 15.
- Maximum number of rotation angles to check in the placement modification mutation that tries to rotate an item in a direction until it intersects, B_r . Default value: 8.
- Maximum proportion of a 360-degree rotation that can be applied as a rotation offset in a mutation step of placement modification of the type small rotation change, S_r . Default value: 0.2.
- Maximum proportion of the side length of the container’s bounding rectangle (for the relevant axis) that can be applied as a position offset in a mutation step of placement modification of the type small position change, S_p . Default value: 0.2.
- Probability to select as the final mutation individual the candidate solution of the intermediate step with highest fitness instead of the final result of applying all steps, P_i . It is only considered if the final candidate solution does not have higher fitness than all the previous intermediate ones. Default value: 1.
- Probability to select as the final mutation individual the solution in the state that had prior to the mutation, P_{cm} . It is only applicable if the solution had just been generated using the crossover operator and the solution had greater fitness (or equal, but is better in the tie-break check) before mutation than after applying the operator. Default value: 0.5.
- Probability weight to add an item in the container in a step of the mutation operator, P_a . Default value: 0.6.
- Probability weight to modify the placement of an item placed in the container in a step of the mutation operator, P_m . Default value: 0.3.
- Probability weight to remove an item from the container in a step of the mutation operator, P_r . Default value: 0.1.
- Proportion of the side length of the container’s bounding rectangle used to calculate the minimum relevant distance between an item’s reference point and the first intersection point determined by the direction of a move-until-intersection mutation, D . Default value: 0.03.
- Population update:
 - Number of elite individuals in a generation, E . Default value: 5.
 - Pool size for the tournament strategy used in the population update, T_u . Default value: 3.
- Termination criteria:
 - Maximum number of consecutive generations without highest fitness improvement to assume convergence and perform early stopping, G_{conv} . Default value: 12.
 - Maximum number of generations, G_{max} . Default value: 30.

A.3 Secondary tables of Parameter Optimization

A.3.1 Greedy algorithm

This section contains the tables of the parameter optimization of the greedy algorithm that were not shown in Section 6.2.

Table 11: Solution value of the different configurations of value and area weights for the first optimization phase of the greedy algorithm.

	$K_v = 0,$ $K_a = 0$	$K_v = 0,$ $K_a = 0.25$	$K_v = 0,$ $K_a = 0.5$	$K_v = 0,$ $K_a = 0.75$	$K_v = 0,$ $K_a = 1$	$K_v = 0.25,$ $K_a = 0$	$K_v = 0.25,$ $K_a = 0.25$	$K_v = 0.25,$ $K_a = 0.5$	$K_v = 0.25,$ $K_a = 0.75$
Problem 1	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0
Problem 2	41.0±10.2	48.0±6.0	40.5±10.59	38.5±10.5	41.0±9.17	36.0±9.17	46.0±10.68	41.5±9.76	45.0±8.94
Problem 3	24.5±5.68	20.5±1.5	24.0±5.83	26.0±6.63	21.5±4.5	25.5±6.5	23.5±5.94	26.0±7.35	24.0±5.83
Problem 4	33.2±0.4	32.6±1.28	32.3±1.73	32.7±1.27	32.8±0.6	33.4±0.49	32.8±1.33	33.3±0.46	32.4±0.92
Problem 5	111.1±17.25	126.7±34.35	119.5±13.73	126.4±9.23	116.9±10.22	118.7±12.62	122.8±12.86	121.8±12.16	119.9±8.46
Problem 6	80.5±13.95	85.5±14.69	93.3±12.65	86.7±13.22	89.8±7.7	81.0±12.25	76.8±14.51	91.2±9.69	89.7±11.85
Problem 7	82.0±12.78	85.0±11.38	87.8±10.21	88.0±6.13	89.2±6.23	89.3±11.04	87.8±7.73	80.0±12.14	90.2±8.67
Problem 8	53.1±5.11	52.5±4.06	57.0±4.98	54.4±5.08	51.8±3.63	52.9±6.16	53.8±6.18	53.3±6.08	53.4±5.39
Problem 9	106.5±10.11	99.4±8.18	104.2±13.72	103.9±5.39	109.6±4.2	100.9±15.29	110.5±12.55	101.8±13.55	105.1±9.96
Problem 10	98.0±18.87	103.0±19.0	97.0±19.0	111.0±13.75	115.0±8.06	107.0±11.87	112.0±9.8	114.0±6.63	116.0±12.0
# Top 3	1	1	2	0	1	1	0	2	1

	$K_v = 0.25,$ $K_a = 1$	$K_v = 0.5,$ $K_a = 0$	$K_v = 0.5,$ $K_a = 0.25$	$K_v = 0.5,$ $K_a = 0.5$	$K_v = 0.5,$ $K_a = 0.75$	$K_v = 0.5,$ $K_a = 1$	$K_v = 0.75,$ $K_a = 0$	$K_v = 0.75,$ $K_a = 0.25$
Problem 1	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0
Problem 2	41.5±10.74	44.0±9.17	41.5±9.5	48.0±11.0	43.5±9.5	45.5±7.89	38.5±10.5	43.5±8.96
Problem 3	21.5±4.5	24.5±4.72	21.5±3.2	27.5±7.5	23.5±5.94	26.5±5.94	22.0±4.58	24.5±6.87
Problem 4	32.1±1.7	32.5±1.8	32.9±1.37	33.0±1.41	32.2±1.6	32.4±1.28	32.7±0.64	32.1±1.58
Problem 5	116.9±10.22	121.5±12.17	125.4±8.67	120.8±11.35	124.7±28.39	121.9±14.63	123.8±10.81	132.5±23.21
Problem 6	89.8±8.12	83.9±10.78	89.9±8.99	83.0±10.99	77.5±12.26	89.7±8.97	87.7±7.47	85.8±12.35
Problem 7	90.6±6.09	84.8±10.86	92.6±8.9	91.2±8.01	87.2±7.64	89.4±7.57	93.2±10.24	90.9±7.29
Problem 8	55.2±5.64	54.5±5.04	54.5±5.14	54.3±6.81	55.6±4.65	53.0±4.52	52.3±3.03	51.2±3.31
Problem 9	104.0±6.83	106.6±9.16	110.9±17.11	112.6±13.44	110.9±12.89	102.0±13.94	111.0±11.38	112.6±16.31
Problem 10	112.0±9.8	107.0±11.87	113.0±10.05	117.0±6.4	106.5±16.74	115.0±6.71	113.0±11.87	110.0±14.14
# Top 3	1	0	1	4	1	1	1	1

	$K_v = 0.75,$ $K_a = 0.5$	$K_v = 0.75,$ $K_a = 0.75$	$K_v = 0.75,$ $K_a = 1$	$K_v = 1,$ $K_a = 0$	$K_v = 1,$ $K_a = 0.25$	$K_v = 1,$ $K_a = 0.5$	$K_v = 1,$ $K_a = 0.75$	$K_v = 1,$ $K_a = 1$
Problem 1	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0
Problem 2	37.5±9.29	48.5±8.08	42.5±10.31	42.0±10.05	46.0±8.6	44.0±9.43	35.5±9.07	40.0±10.25
Problem 3	23.5±4.5	23.5±5.94	27.0±6.78	24.5±5.68	25.5±6.1	25.0±5.92	28.0±6.4	26.5±7.09
Problem 4	32.8±0.98	31.9±1.3	33.0±0.0	32.5±1.43	32.6±1.5	32.8±1.4	33.1±0.54	31.9±1.76
Problem 5	130.7±26.42	122.4±10.41	128.4±27.92	121.8±10.84	122.0±13.3	122.0±12.02	124.2±12.98	130.3±5.25
Problem 6	89.9±12.79	86.3±9.32	92.5±9.28	84.4±8.92	81.4±21.78	83.4±11.51	84.3±9.66	84.7±10.74
Problem 7	82.0±11.33	86.2±11.36	87.0±7.64	91.6±9.1	81.5±13.38	86.4±11.98	89.4±9.79	90.5±8.71
Problem 8	50.8±4.21	51.9±4.01	54.7±5.2	53.1±7.42	52.3±5.78	54.4±4.76	52.6±4.76	53.1±3.67
Problem 9	109.9±17.05	96.7±11.92	106.1±9.65	110.1±17.48	114.7±14.46	113.9±17.25	110.9±14.72	117.6±18.11
Problem 10	112.0±8.72	108.0±15.36	112.0±9.8	99.0±19.21	106.5±12.66	107.0±14.87	114.5±13.12	113.0±11.0
# Top 3	1	1	2	0	1	1	1	2

Table 12: Execution time (in milliseconds) of the different configurations of value and area weights for the first optimization phase of the greedy algorithm.

	$K_v = 0,$ $K_a = 0$	$K_v = 0,$ $K_a = 0.25$	$K_v = 0,$ $K_a = 0.5$	$K_v = 0,$ $K_a = 0.75$	$K_v = 0,$ $K_a = 1$	$K_v = 0.25,$ $K_a = 0$	$K_v = 0.25,$ $K_a = 0.25$	$K_v = 0.25,$ $K_a = 0.5$	$K_v = 0.25,$ $K_a = 0.75$
Problem 1	13±4	16±8	15±7	11±3	8±3	20±9	20±8	21±5	21±10
Problem 2	355±57	379±75	347±101	369±63	346±81	337±71	391±86	372±85	456±73
Problem 3	28±9	18±8	25±19	21±15	16±7	25±12	30±13	33±16	26±17
Problem 4	18±2	21±9	18±5	18±4	15±5	18±4	20±6	17±4	16±4
Problem 5	403±77	391±67	396±104	368±96	405±70	370±82	364±99	374±73	356±106
Problem 6	371±69	445±127	371±83	374±110	314±73	351±70	356±80	421±99	404±101
Problem 7	412±92	408±75	343±52	402±101	425±107	353±100	413±93	320±72	394±86
Problem 8	122±40	129±41	150±83	234±93	133±56	139±81	150±61	153±56	153±75
Problem 9	189±70	247±67	204±80	244±80	124±47	292±95	210±74	212±69	211±71
Problem 10	280±64	278±53	235±57	310±90	276±66	274±43	297±54	300±59	312±74

	$K_v = 0.25,$ $K_a = 1$	$K_v = 0.5,$ $K_a = 0$	$K_v = 0.5,$ $K_a = 0.25$	$K_v = 0.5,$ $K_a = 0.5$	$K_v = 0.5,$ $K_a = 0.75$	$K_v = 0.5,$ $K_a = 1$	$K_v = 0.75,$ $K_a = 0$	$K_v = 0.75,$ $K_a = 0.25$
Problem 1	12±3	14±8	20±9	23±8	23±10	12±3	21±15	21±10
Problem 2	351±38	420±68	361±47	341±46	388±67	359±61	341±70	365±57
Problem 3	24±12	32±19	25±15	29±21	26±13	25±16	27±14	34±19
Problem 4	21±9	21±6	19±7	17±3	18±5	21±5	18±4	20±8
Problem 5	380±77	343±112	376±117	383±68	425±75	346±91	412±76	329±74
Problem 6	363±91	412±89	476±123	313±77	340±71	349±74	373±83	353±90
Problem 7	375±86	380±49	508±77	415±86	340±75	382±81	353±82	389±103
Problem 8	114±44	147±43	165±73	125±62	146±89	129±38	156±69	124±50
Problem 9	231±58	349±85	328±95	297±92	285±51	252±71	308±67	255±80
Problem 10	274±83	297±47	278±78	317±62	337±71	285±62	301±52	287±56

	$K_v = 0.75,$ $K_a = 0.5$	$K_v = 0.75,$ $K_a = 0.75$	$K_v = 0.75,$ $K_a = 1$	$K_v = 1,$ $K_a = 0$	$K_v = 1,$ $K_a = 0.25$	$K_v = 1,$ $K_a = 0.5$	$K_v = 1,$ $K_a = 0.75$	$K_v = 1,$ $K_a = 1$
Problem 1	20±7	21±11	18±6	19±13	20±5	19±10	16±6	28±10
Problem 2	359±71	379±76	351±44	367±41	379±78	361±75	341±49	344±94
Problem 3	34±17	28±17	29±14	33±13	27±16	26±15	28±13	27±10
Problem 4	20±7	18±6	15±6	17±3	19±6	16±3	17±4	22±7
Problem 5	413±90	373±44	379±76	348±86	335±85	421±84	383±83	298±64
Problem 6	379±81	369±86	381±113	338±53	364±87	344±87	361±84	363±98
Problem 7	321±42	400±88	345±88	366±79	338±51	446±83	419±92	344±91
Problem 8	107±57	154±57	131±78	133±73	107±42	133±61	148±78	122±59
Problem 9	236±82	283±92	309±74	240±72	232±100	340±85	280±103	291±94
Problem 10	282±53	318±63	271±60	298±57	282±67	274±84	277±56	309±68

Table 13: Execution time (in milliseconds) of the iteration configurations tested in the second optimization phase of the greedy algorithm.

	1000 max iter., 300 conv. iter.	10000 max iter., 3000 conv. iter.	100000 max iter., 30000 conv. iter.
Problem 1	19±7	19±9	20±7
Problem 2	340±56	2809±688	28802±2793
Problem 3	27±12	28±9	40±16
Problem 4	16±5	16±3	17±6
Problem 5	325±95	1902±629	22358±7245
Problem 6	375±77	2721±666	21109±4122
Problem 7	351±83	3215±441	14432±9070
Problem 8	168±66	109±64	298±257
Problem 9	272±83	578±501	1091±2634
Problem 10	305±52	2271±473	20331±3745

A.3.2 Reversible algorithm

This section contains the tables of the parameter optimization of the reversible algorithm that were not shown in Section 6.3.

Table 14: Solution value of the different configurations of removal and placement modification probabilities for the first optimization phase of the reversible algorithm.

	$P_r = 0.005,$ $P_m = 0.03$	$P_r = 0.005,$ $P_m = 0.05$	$P_r = 0.005,$ $P_m = 0.1$	$P_r = 0.01,$ $P_m = 0.03$	$P_r = 0.01,$ $P_m = 0.05$	$P_r = 0.01,$ $P_m = 0.1$	$P_r = 0.01,$ $P_m = 0.1$	$P_r = 0.02,$ $P_m = 0.03$	$P_r = 0.02,$ $P_m = 0.05$	$P_r = 0.02,$ $P_m = 0.1$
Problem 1	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	50.0±0.0	49.0±3.0	49.0±3.0	50.0±0.0
Problem 2	42.0±10.05	46.0±10.68	36.0±9.17	38.5±9.76	48.0±6.4	42.5±8.44	43.5±9.23	43.5±8.08	46.5±10.5	
Problem 3	22.5±6.02	24.0±5.83	23.5±5.94	19.5±1.5	20.0±0.0	22.0±4.58	19.5±1.5	24.0±5.83	20.5±2.69	
Problem 4	31.1±1.76	32.3±1.68	30.8±1.6	31.9±1.7	<u>31.9±1.7</u>	31.5±1.75	31.8±1.33	31.5±1.63	31.0±2.05	
Problem 5	111.8±9.17	106.4±5.59	110.2±8.4	111.1±10.61	<u>112.2±8.82</u>	115.6±17.91	<u>112.5±8.73</u>	112.0±9.43	108.0±7.55	
Problem 6	81.4±14.79	95.7±11.46	84.4±12.08	91.5±11.74	81.9±10.25	83.4±14.96	83.7±12.08	85.2±10.68	89.1±9.52	
Problem 7	<u>91.5±7.38</u>	86.6±11.02	87.1±12.02	83.6±10.62	91.8±10.65	88.8±12.98	87.2±7.43	85.5±6.23	<u>89.0±6.0</u>	
Problem 8	50.4±7.74	54.4±6.05	49.2±5.6	50.6±4.61	49.0±6.9	49.7±4.43	48.2±4.24	<u>52.0±6.16</u>	<u>53.5±7.63</u>	
Problem 9	98.6±9.85	103.0±5.42	103.7±6.17	<u>104.3±10.85</u>	97.2±11.07	105.4±6.67	101.6±8.18	<u>104.8±5.9</u>	103.1±4.89	
Problem 10	108.0±13.08	94.5±11.93	99.5±19.03	<u>101.0±12.81</u>	<u>103.5±16.13</u>	86.5±17.47	83.0±20.52	96.0±17.44	93.5±17.47	
# Wins	<u>2</u>	5	1	1	<u>3</u>	<u>3</u>	0	1	1	
# Top 3	3	6	2	4	6	3	1	3	<u>5</u>	

Table 15: Execution time (in milliseconds) of the different configurations of removal and placement modification for the first optimization phase of the reversible algorithm.

	$P_r = 0.005, P_m = 0.03$	$P_r = 0.005, P_m = 0.05$	$P_r = 0.005, P_m = 0.1$	$P_r = 0.01, P_m = 0.03$	$P_r = 0.01, P_m = 0.05$	$P_r = 0.01, P_m = 0.1$	$P_r = 0.02, P_m = 0.03$	$P_r = 0.02, P_m = 0.05$	$P_r = 0.02, P_m = 0.1$
Problem 1	10±4	9±4	12±4	10±3	11±4	28±35	47±108	51±127	12±4
Problem 2	461±142	523±149	571±190	603±189	690±179	704±196	643±215	643±193	878±114
Problem 3	37±34	25±15	29±18	25±13	22±8	26±20	23±19	21±11	18±7
Problem 4	15±4	18±6	20±8	17±4	20±4	17±5	26±19	23±11	22±6
Problem 5	510±171	700±187	515±218	1002±128	570±141	797±220	684±240	784±225	810±200
Problem 6	387±92	461±112	431±132	480±113	451±106	444±111	457±88	476±130	606±100
Problem 7	502±143	504±131	516±133	530±200	546±126	553±126	501±108	633±121	593±101
Problem 8	129±95	188±91	165±103	88±39	177±115	135±60	134±68	205±85	216±117
Problem 9	279±93	284±117	249±79	265±108	297±106	219±129	359±139	318±132	335±154
Problem 10	462±92	395±103	470±109	444±89	496±118	429±107	409±92	430±98	532±83

Table 16: Execution time (in milliseconds) of the iteration configurations tested in the second optimization phase of the reversible algorithm.

	1000 max iter., 300 conv. iter.	10000 max iter., 3000 conv. iter.	100000 max iter., 30000 conv. iter.
Problem 1	12±5	11±5	11±2
Problem 2	538±193	6847±1206	33884±20221
Problem 3	19±7	16±6	18±8
Problem 4	20±11	15±3	24±8
Problem 5	574±180	5756±2441	30254±22536
Problem 6	441±119	3570±613	11182±2100
Problem 7	422±81	2792±870	11457±2028
Problem 8	136±93	112±74	241±110
Problem 9	234±109	940±1034	1194±2573
Problem 10	405±100	4302±497	19181±2750

A.3.3 Evolutionary algorithm

This section contains the tables of the parameter optimization of the evolutionary algorithm that were not shown in Section 6.4.

Table 17: Solution value of the population size configurations tested in the first optimization phase of the evolutionary algorithm.

	$\mu = 30$	$\mu = 50$	$\mu = 100$
Problem 1	65.0 ± 22.91	80.0 ± 24.49	95.0 ± 15.0
Problem 2	62.5 ± 9.29	65.5 ± 12.54	73.0 ± 6.0
Problem 3	35.0 ± 0.0	35.5 ± 1.5	35.0 ± 0.0
Problem 4	34.0 ± 0.0	34.0 ± 0.0	34.0 ± 0.0
Problem 5	214.7 ± 22.45	230.4 ± 5.5	221.0 ± 30.35
Problem 6	107.5 ± 9.75	115.4 ± 8.19	116.9 ± 5.2
Problem 7	106.4 ± 5.77	106.8 ± 4.62	110.9 ± 3.62
Problem 8	73.5 ± 17.77	80.2 ± 16.34	89.9 ± 15.46
Problem 9	164.3 ± 8.85	168.2 ± 6.05	168.4 ± 9.84
Problem 10	124.0 ± 4.9	128.0 ± 4.0	129.0 ± 5.39

Table 18: Execution time (in milliseconds) of the population size configurations tested in the first optimization phase of the evolutionary algorithm.

	$\mu = 30$	$\mu = 50$	$\mu = 100$
Problem 1	3803 ± 951	7076 ± 575	13174 ± 2246
Problem 2	7939 ± 1223	14634 ± 3642	27582 ± 3995
Problem 3	2617 ± 420	4884 ± 821	8623 ± 1176
Problem 4	3557 ± 586	6412 ± 761	11233 ± 1735
Problem 5	11762 ± 2267	23009 ± 3436	37452 ± 7094
Problem 6	8531 ± 2051	16772 ± 3587	30691 ± 4982
Problem 7	10699 ± 1552	17034 ± 2318	33867 ± 8046
Problem 8	5219 ± 1760	9978 ± 2860	14305 ± 5334
Problem 9	8130 ± 1580	13924 ± 2856	25042 ± 5013
Problem 10	5861 ± 953	11290 ± 2203	20610 ± 4705

Table 19: Solution value of the offspring size configurations tested in the second optimization phase of the evolutionary algorithm.

	$\lambda = 0.5 \cdot \mu$ (50)	$\lambda = \mu$ (100)	$\lambda = 2 \cdot \mu$ (200)
Problem 1	100.0±0.0	90.0±20.0	95.0±15.0
Problem 2	71.0±5.83	74.0±8.0	74.5±9.6
Problem 3	36.0±2.0	36.0±2.0	37.5±2.5
Problem 4	34.0±0.0	34.0±0.0	34.0±0.0
Problem 5	228.6±9.47	234.0±0.0	234.0±0.0
Problem 6	120.7±9.12	119.6±6.5	119.9±5.91
Problem 7	109.5±2.77	112.6±2.62	114.0±1.55
Problem 8	90.3±14.85	87.6±15.19	87.6±15.19
Problem 9	167.9±6.44	171.0±5.73	167.5±8.59
Problem 10	131.0±5.39	132.0±4.0	138.0±6.0

Table 20: Execution time (in milliseconds) of the offspring size configurations tested in the second optimization phase of the evolutionary algorithm.

	$\lambda = 0.5 \cdot \mu$ (50)	$\lambda = \mu$ (100)	$\lambda = 2 \cdot \mu$ (200)
Problem 1	12957±2274	20188±3147	38489±3410
Problem 2	29089±5113	48845±7238	80617±11690
Problem 3	8449±1402	15653±1655	32169±6970
Problem 4	11418±1320	21783±2658	39406±5333
Problem 5	39353±5239	62929±6572	106269±10769
Problem 6	32852±5313	57462±4173	84603±13950
Problem 7	36193±5385	66445±7227	106128±23333
Problem 8	14816±4878	22325±9929	33336±17545
Problem 9	29604±3400	46536±5323	71307±9668
Problem 10	22750±3303	40928±5300	67961±13852

A.4 Visualization of the best solutions of the evolutionary algorithm for the Joint Problem Dataset

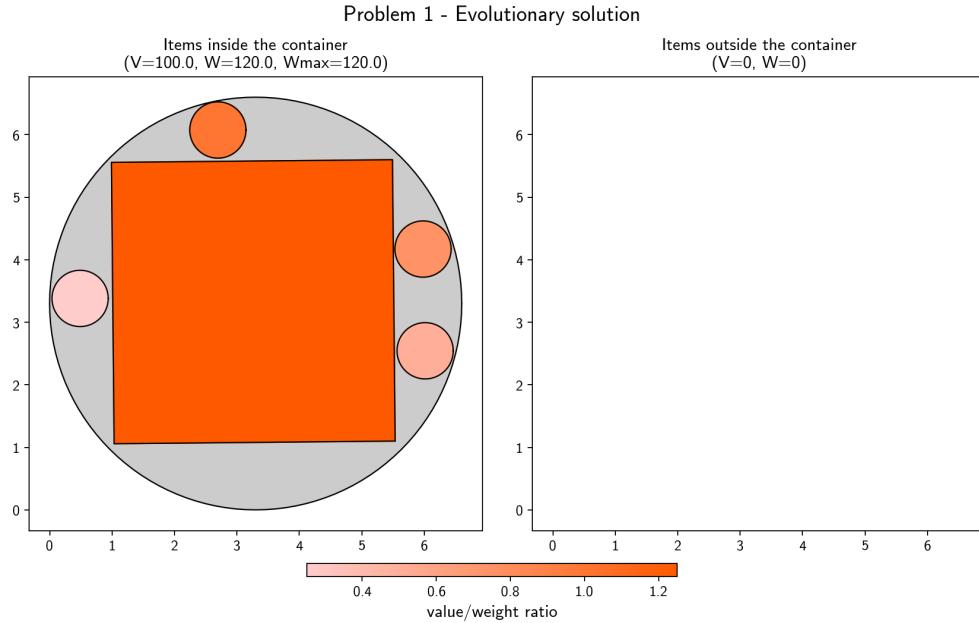


Figure 31: Best solution of the evolutionary algorithm for Problem 1, which is optimal.

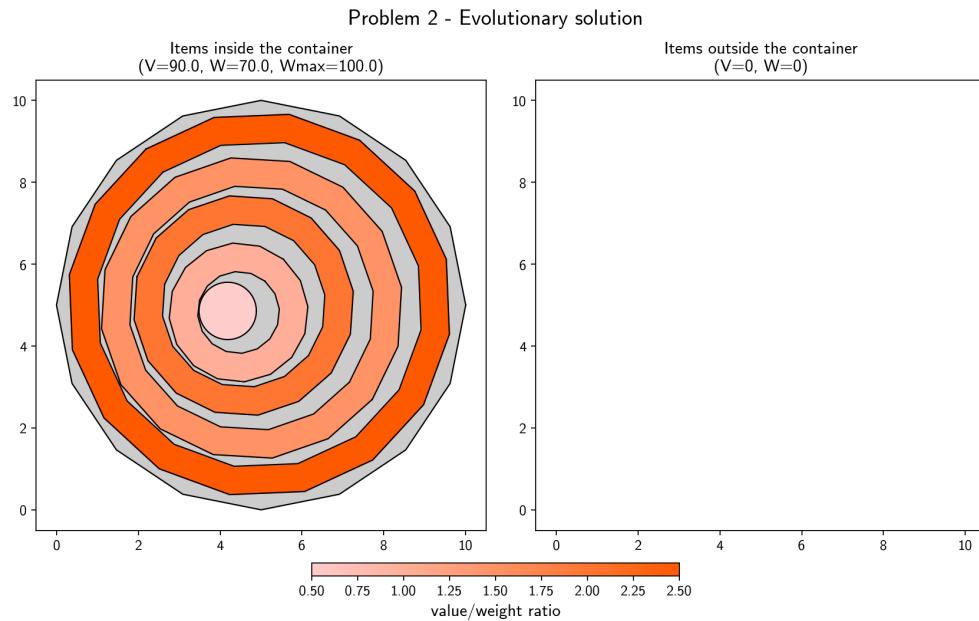


Figure 32: Best solution of the evolutionary algorithm for Problem 2, which is optimal.

Problem 3 - Evolutionary solution

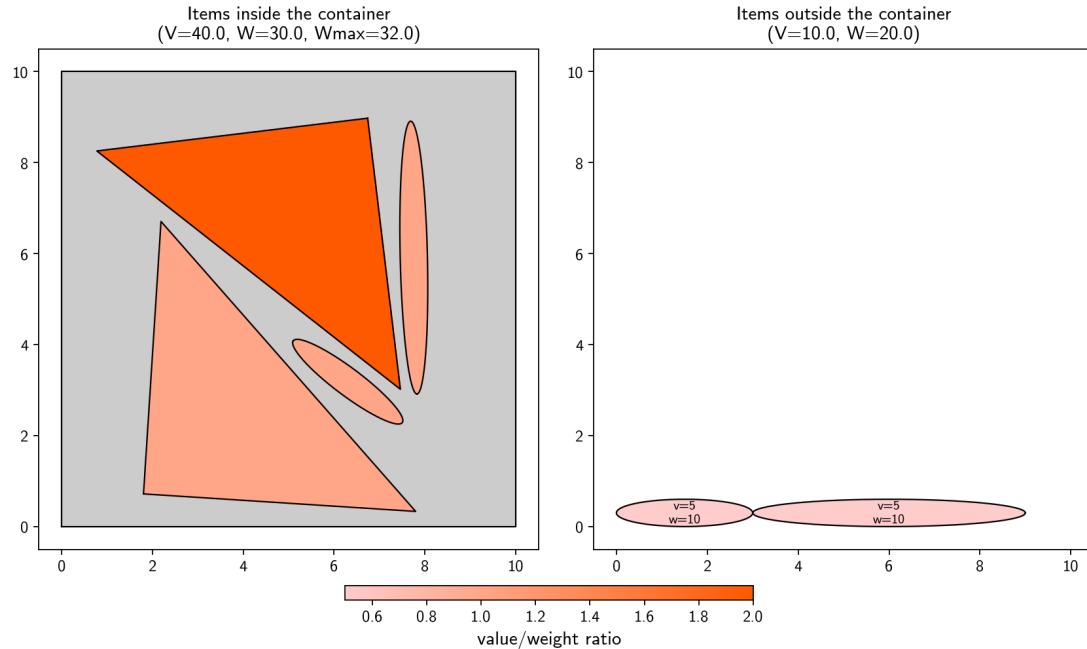


Figure 33: Best solution of the evolutionary algorithm for Problem 3, which is optimal.

Problem 4 - Evolutionary solution

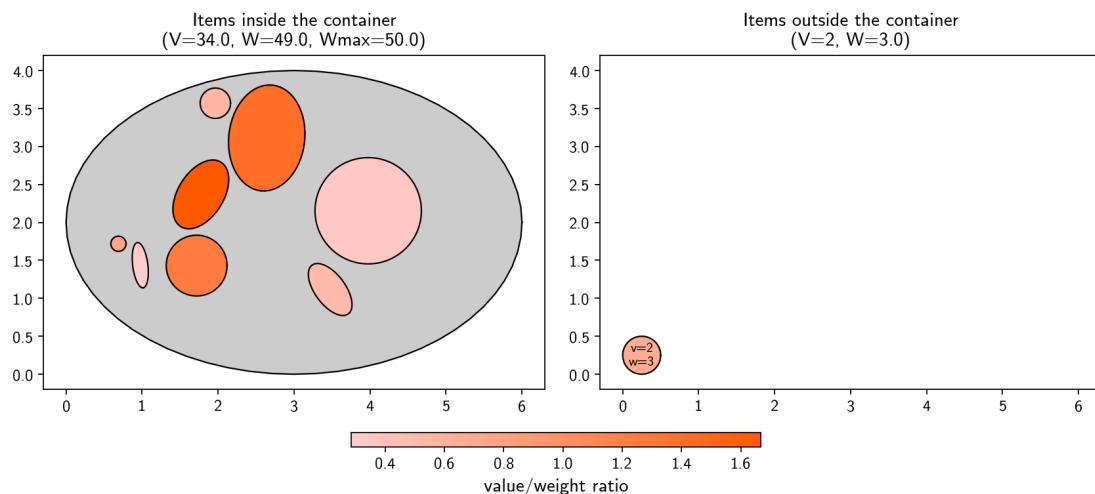


Figure 34: Best solution of the evolutionary algorithm for Problem 4, which is optimal.

Problem 5 - Evolutionary solution

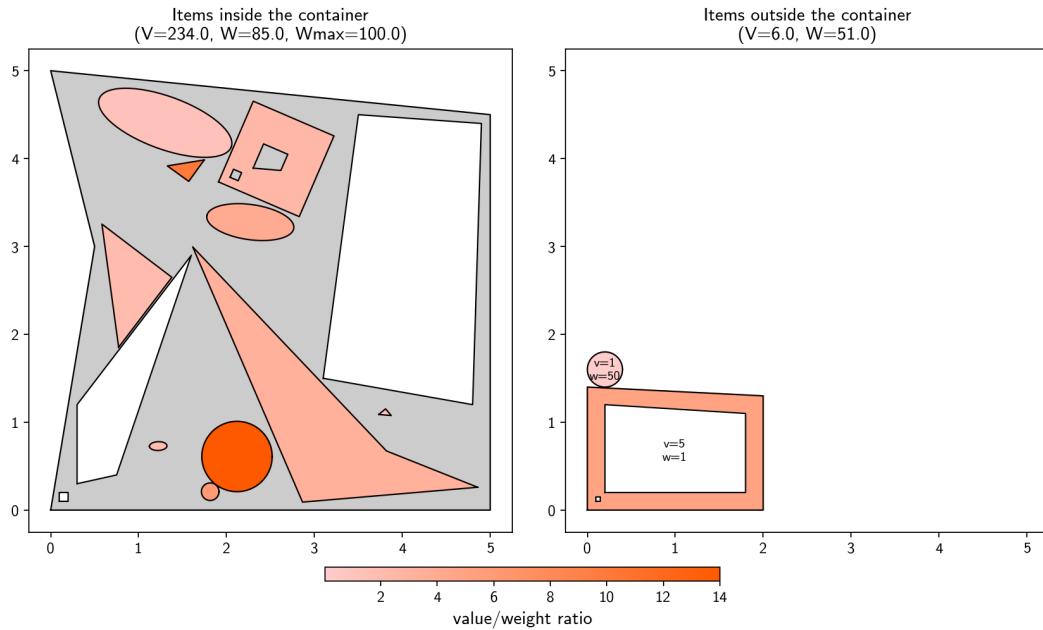


Figure 35: Best solution of the evolutionary algorithm for Problem 5, which is not optimal.

Problem 6 - Evolutionary solution

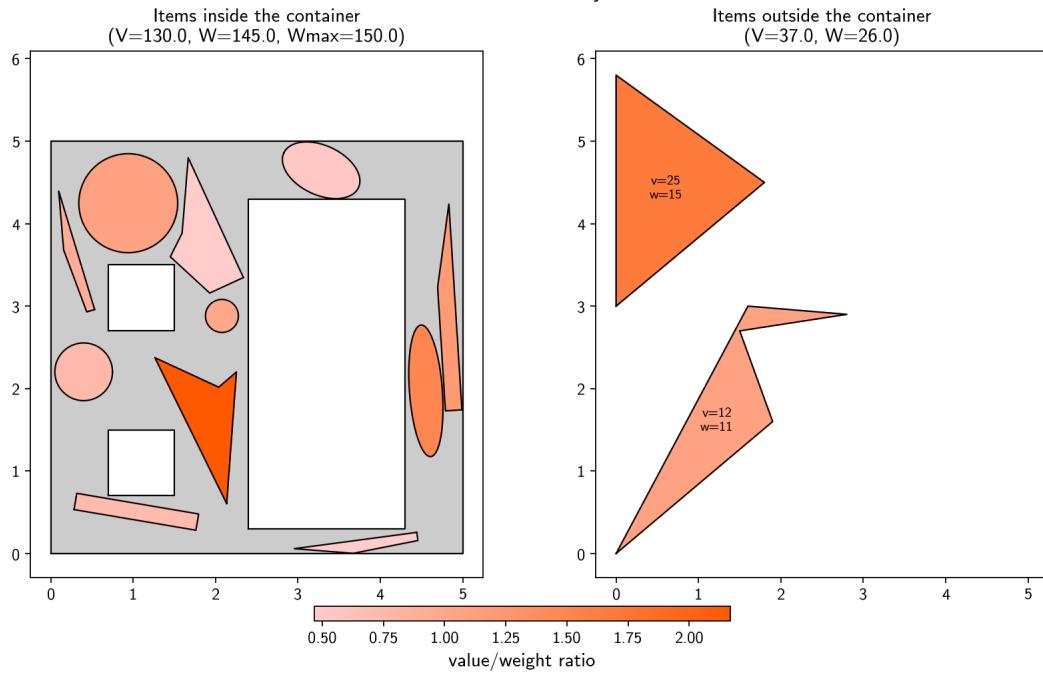


Figure 36: Best solution of the evolutionary algorithm for Problem 6, which is optimal.

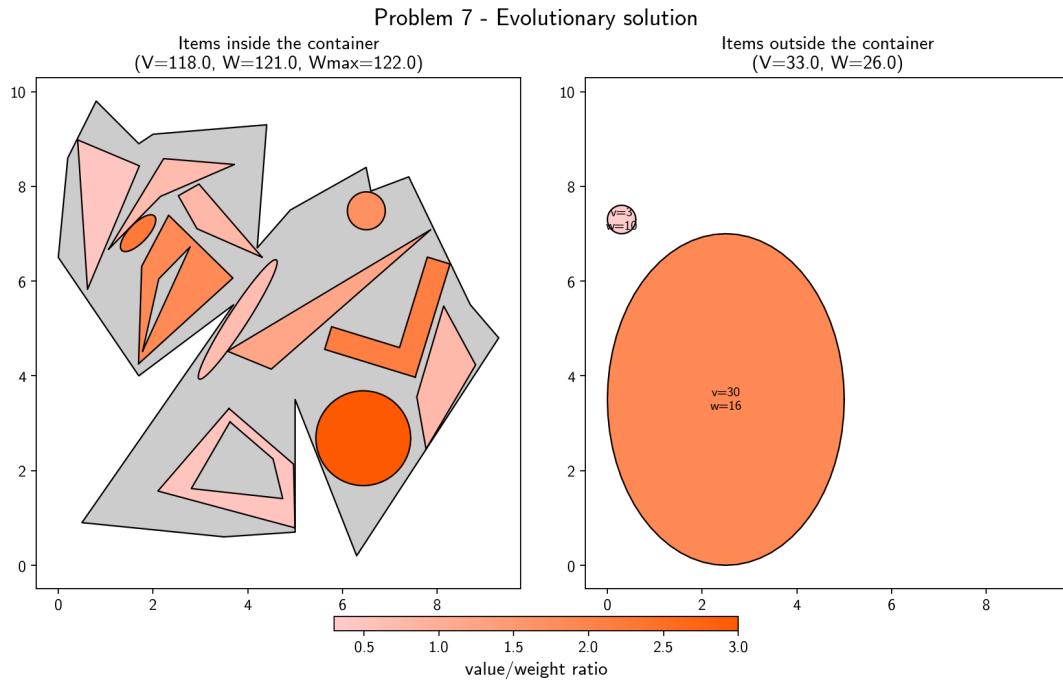


Figure 37: Best solution of the evolutionary algorithm for Problem 7, which is optimal.

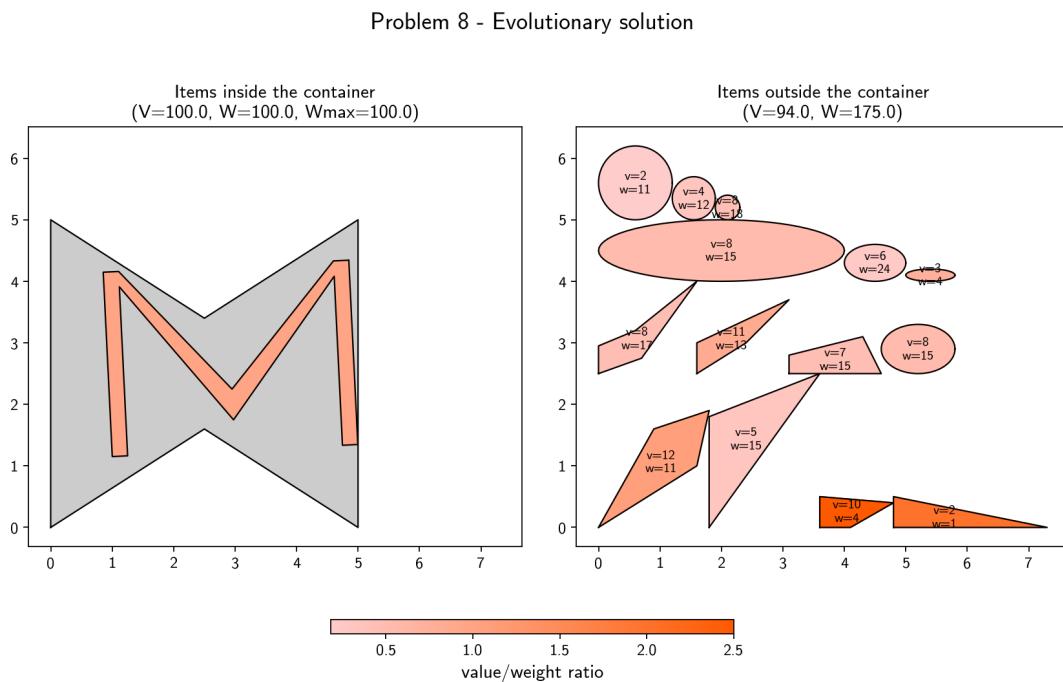


Figure 38: Best solution of the evolutionary algorithm for Problem 8, which is optimal.

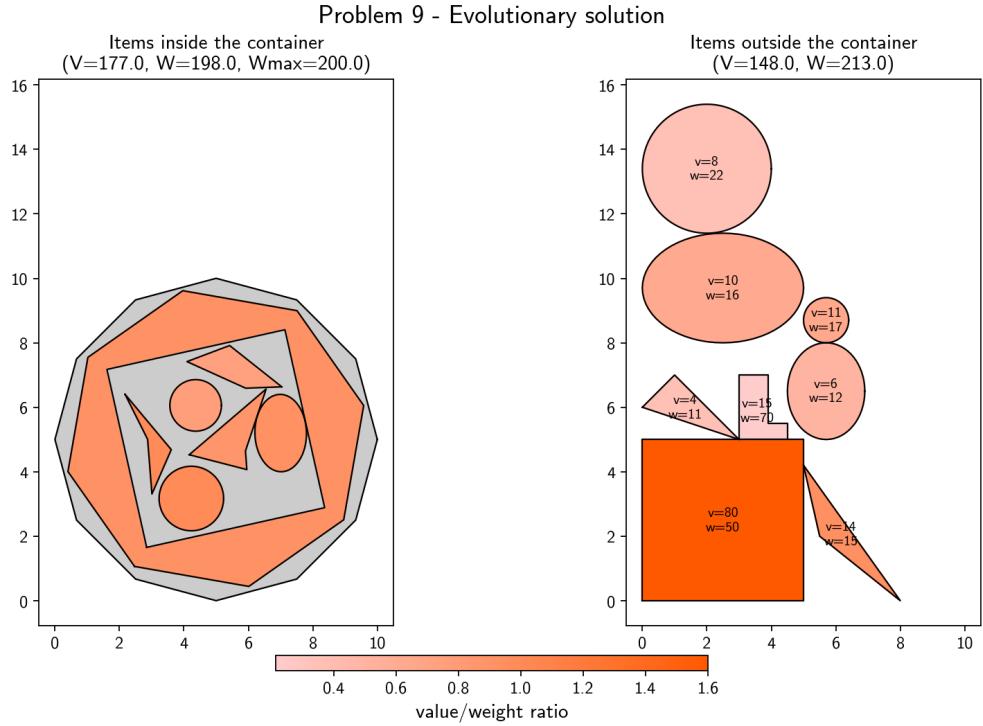


Figure 39: Best solution of the evolutionary algorithm for Problem 9, which is not optimal.

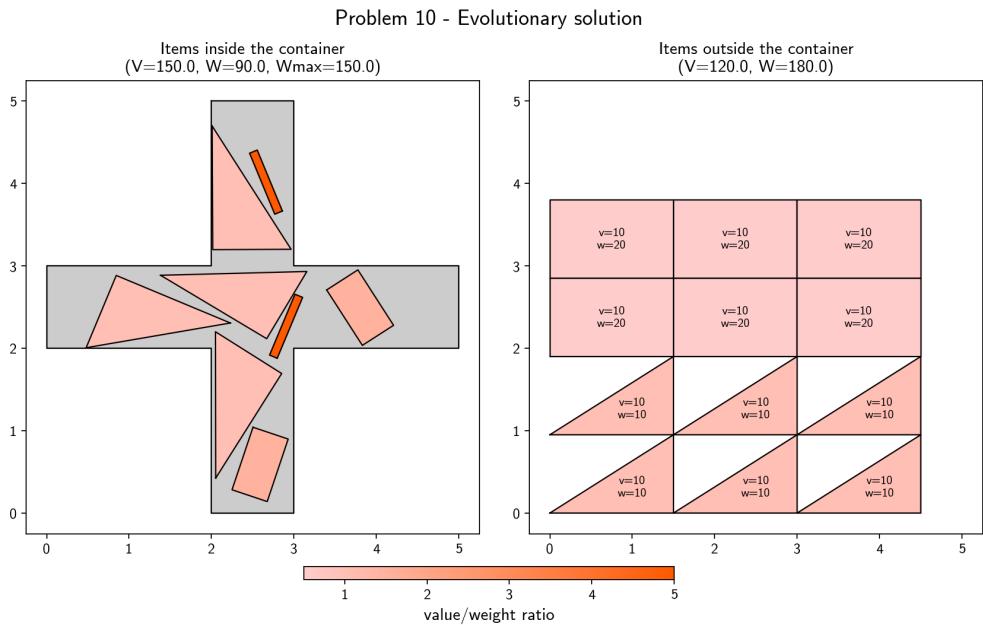


Figure 40: Best solution of the evolutionary algorithm for Problem 10, which is not optimal.

A.5 Visualization of the best solutions of the algorithms for the Packing Problem Dataset

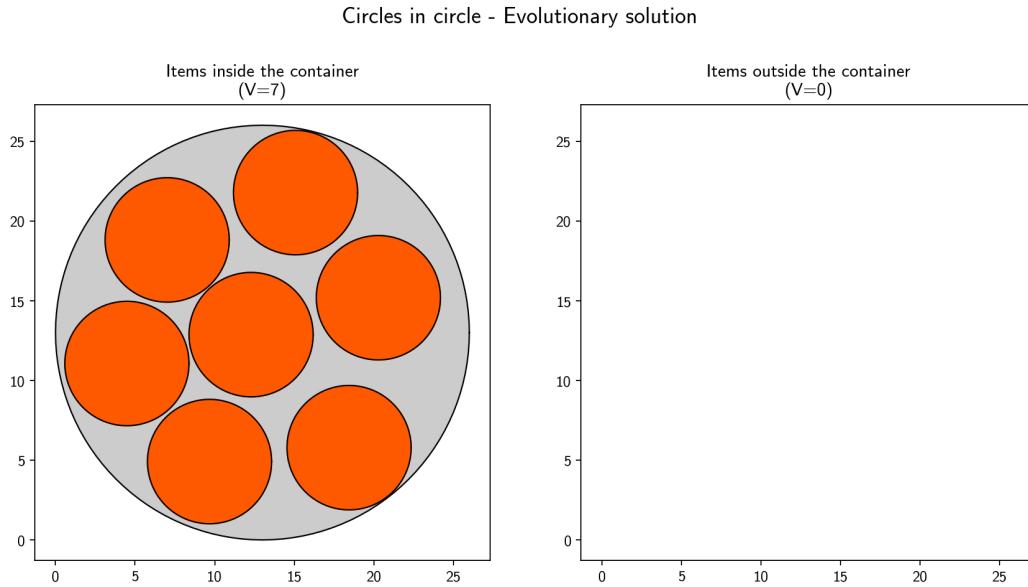


Figure 41: Best solution obtained for the “Circles in circle” problem of the Packing Problem Dataset. The solution is optimal and was produced by the evolutionary algorithm.

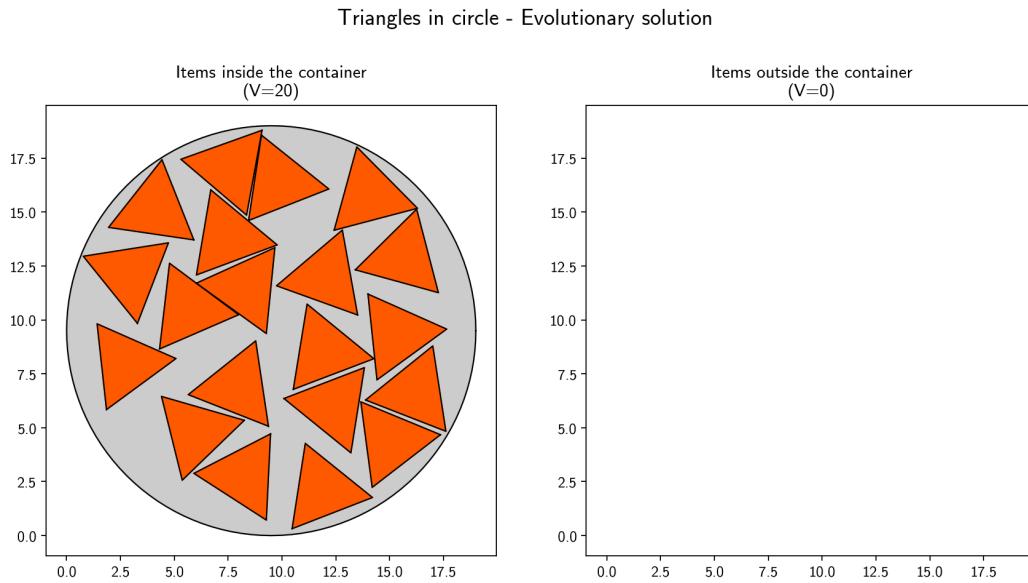


Figure 42: Best solution obtained for the “Triangles in circle” problem of the Packing Problem Dataset. The solution is optimal and was produced by the evolutionary algorithm.

Squares in circle - Evolutionary solution

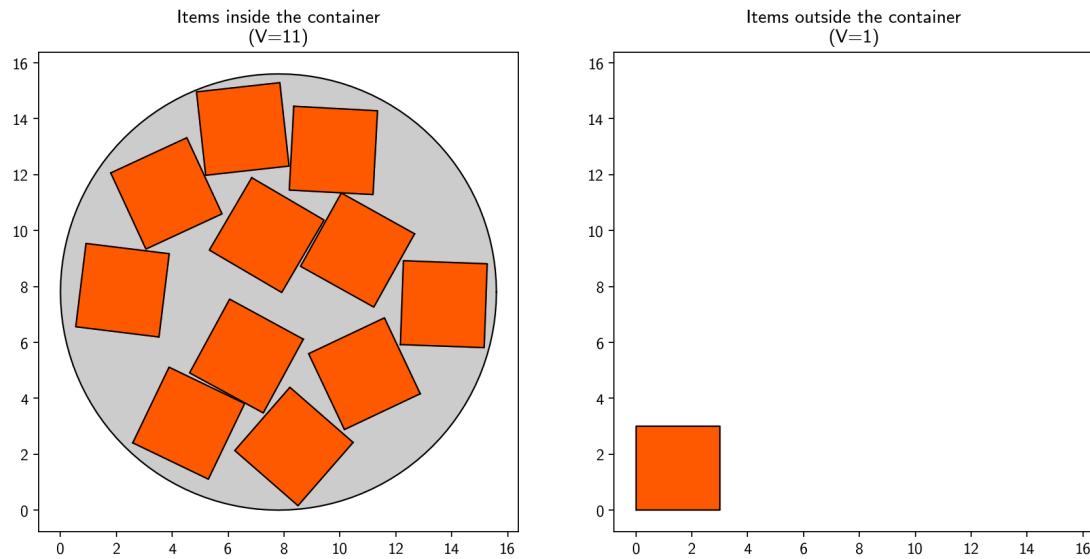


Figure 43: Best solution obtained for the “Squares in circle” problem of the Packing Problem Dataset. The solution is not optimal and was produced by the evolutionary algorithm.

Circles in triangle - Evolutionary solution

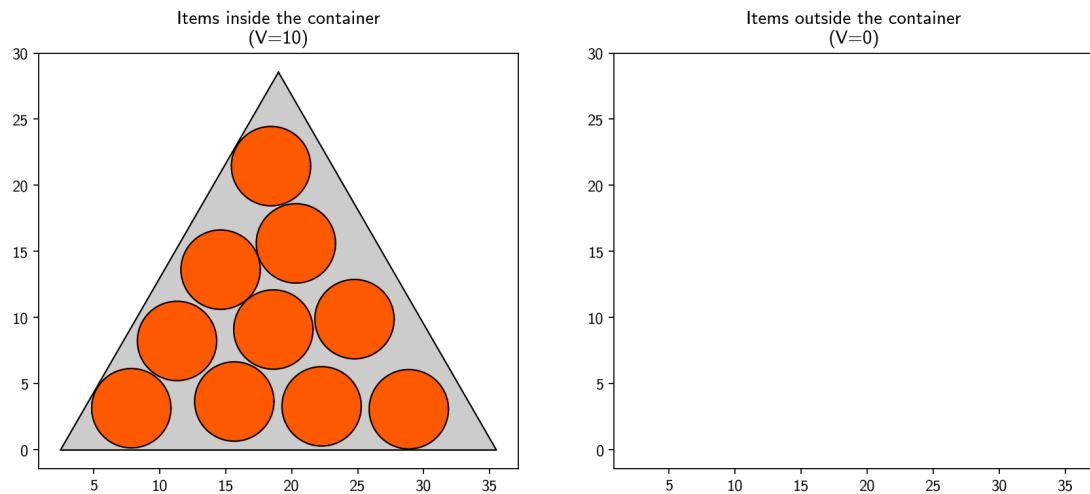


Figure 44: Best solution obtained for the “Circles in triangle” problem of the Packing Problem Dataset. The solution is optimal and was produced by the evolutionary algorithm.

Triangles in triangle - Evolutionary solution

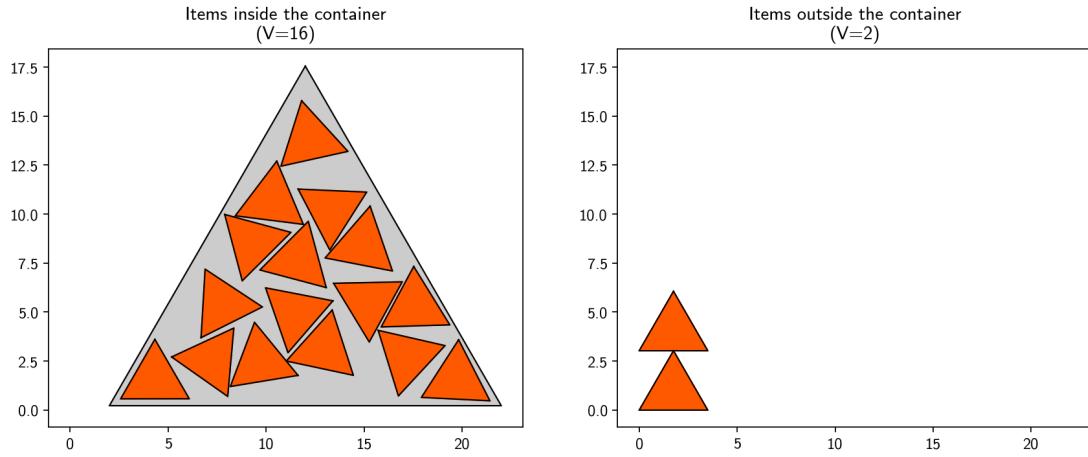


Figure 45: Best solution obtained for the “Triangles in triangle” problem of the Packing Problem Dataset. The solution is not optimal and was produced by the evolutionary algorithm.

Squares in triangle - Evolutionary solution

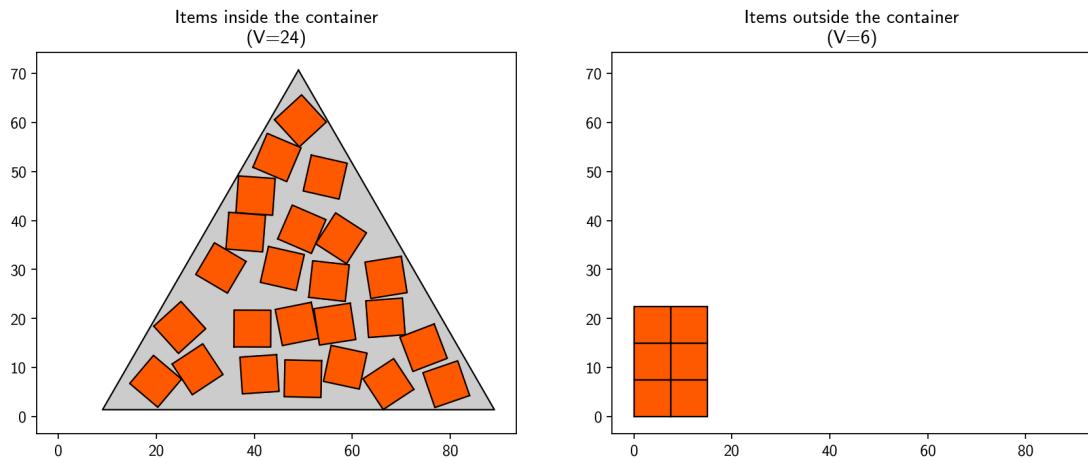


Figure 46: Best solution obtained for the “Squares in triangle” problem of the Packing Problem Dataset. The solution is not optimal and was produced by the evolutionary algorithm.

Circles in square - Evolutionary solution

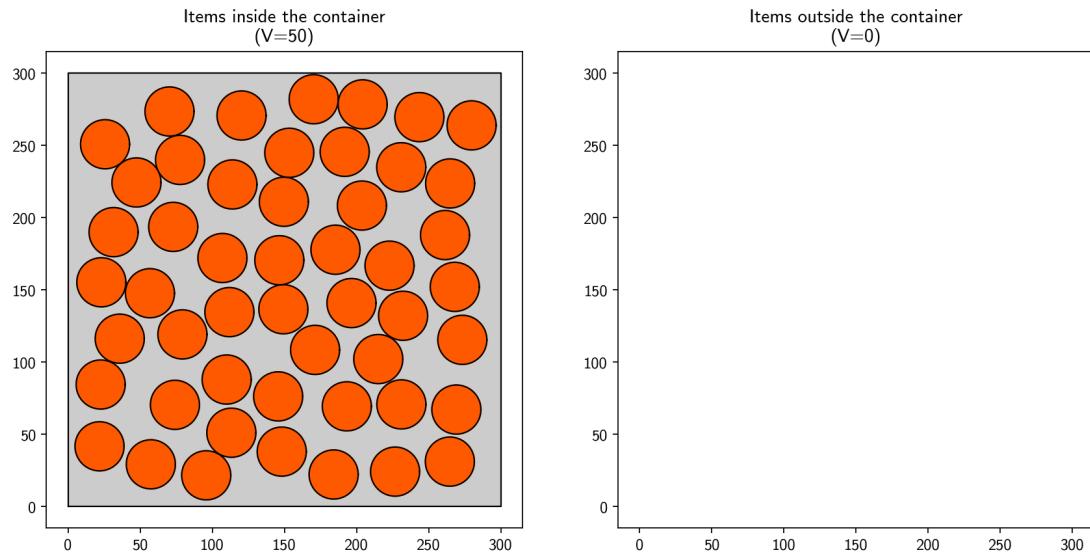


Figure 47: Best solution obtained for the “Circles in square” problem of the Packing Problem Dataset. The solution is optimal and was produced by the evolutionary algorithm.

Triangles in square - Evolutionary solution

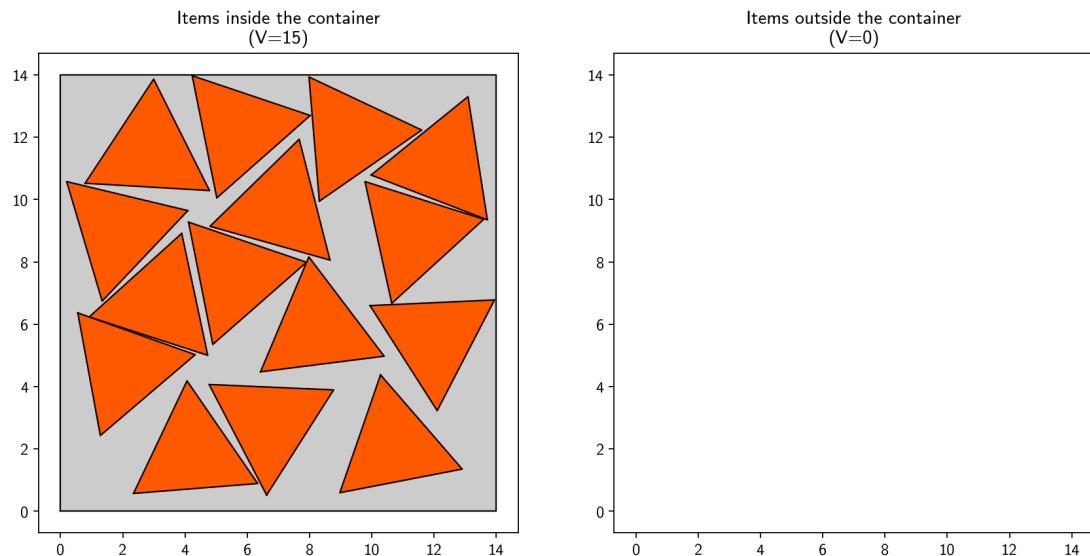


Figure 48: Best solution obtained for the “Triangles in square” problem of the Packing Problem Dataset. The solution is optimal and was produced by the evolutionary algorithm.

Squares in square - Greedy solution

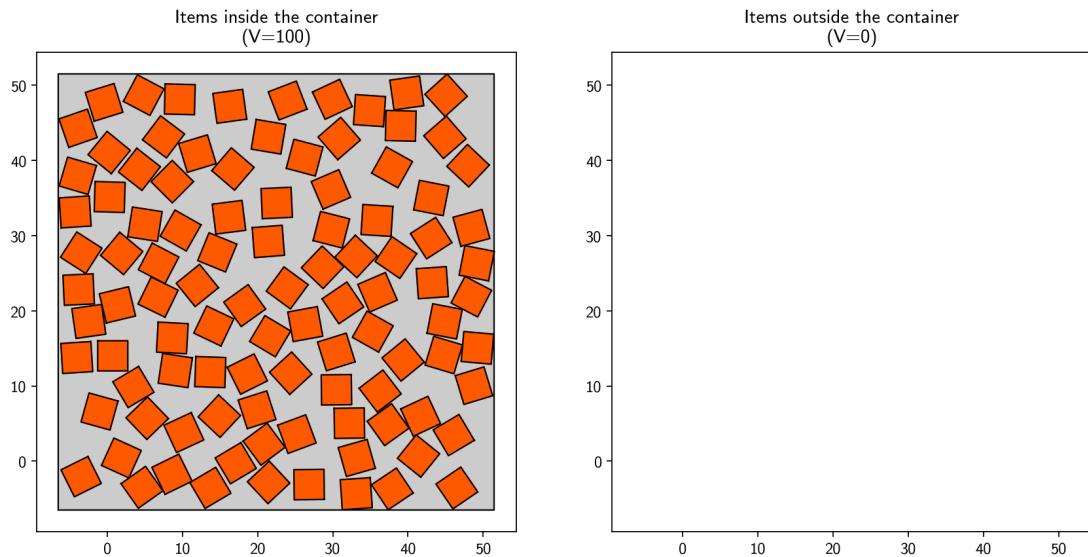


Figure 49: Best solution obtained for the “Squares in square” problem of the Packing Problem Dataset. The solution is optimal and was produced by the greedy algorithm.