Shadoks Approach to Knapsack Polygonal Packing

Guilherme D. da Fonseca
LIS, Aix-Marseille Université

Yan Gerard
LIMOS, Université Clermont Auvergne

Abstract

We describe the heuristics used by the Shadoks team in the CG:SHOP 2024 Challenge. Each instance consists of a convex polygon called *container* and a multiset of *items*, where each item is a simple polygon and has an associated *value*. The goal is to pack some of the items inside the container using translations, in order to maximize the sum of their values. Our strategy consists of obtaining good initial solutions and improving them with local search. To obtain the initial solutions we used integer programming and a carefully designed greedy approach.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Packing, polygons, heuristics, integer programming, computational geometry

Category CG Challenge

Related Version arxiv.org/abs/

Supplementary Material *(Source Code)*: `https://github.com/gfonsecabr/shadoks-CGSHOP2024`

Funding Work supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005).

Acknowledgements We would like to thank the Challenge organizers and other competitors for their time, feedback, and making this whole event possible. We would like to thank Hélène Toussaint, Raphaël Amato, Boris Lonjon, and William Guyot-Lénat from LIMOS, as well as the Qarma and TALEP teams and Manuel Bertrand from LIS, who continue to make the computational resources of the LIMOS and LIS clusters available to our research. We would also like to thank Aldo Gonzalez-Lorenzo and the undergraduate students Aymeric Beck, Houssam Boufarachan, Marine Izoulet, and Carla Scardigli for coding viewers for the solutions.

1 Introduction

CG:SHOP Challenge is an annual geometric optimization challenge. The sixth edition in 2024 considers a 2-dimensional knapsack packing problem. The team Shadoks won first place with the best solution (among the 14 participating teams) to 75 instances out of 180. In this

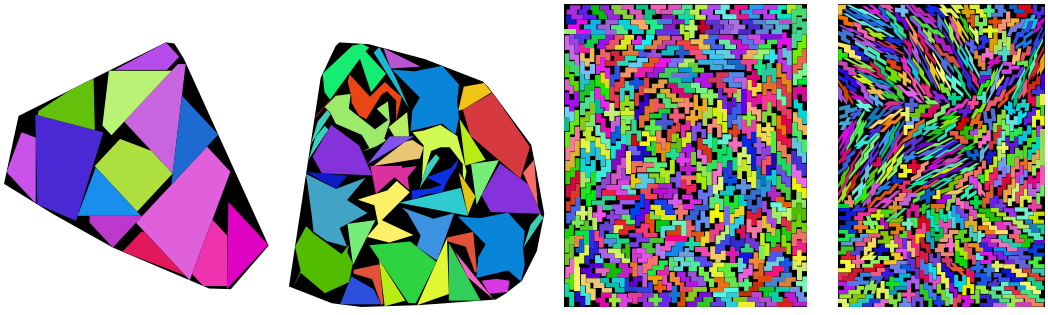

Figure 1 Our best solutions to `jigsaw_cf2_5db5d75a_34`, `random_rcf4_6e323d40_100`, `atris1240`, and `satris1786` instances.

paper, we describe the heuristics we used. We start by briefly describing the problem.

An input instance consists of a convex polygon called *container* and a multiset of *items*. Each item is a simple polygon with an associated integer *value*. The goal is to pack some of the instance items inside the container using integer translations in order to maximize the sum of their values. In total, 180 instances have been given, ranging from 28 to 50,000 items. The instances are of several different types according to the shape and values of the items. Some instances have mostly convex items, while other instances have many non-convex items such as polyominoes. In terms of item values, some items have only unit value, some have values proportionals to the area, and other have random values, for example. Some solutions are presented in Figure 1 and more details about the challenge are available in the organizers' survey paper [3].

Our general strategy consists of finding a good initial solutions (using integer programming or a greedy heuristic) and subsequently optimizing them with local search. Our strategy shares many common elements with the second place [5], but they did not use integer programming to obtain initial solutions and their optimization phase is more sophisticated than ours. The third place [4] uses an hierarchical grid approach. The fourth place [1] used a completely different integer programming model and a genetic algorithm.

We describe the algorithms in Section 2 and experimentally analyze their performance using different parameters in Section 3. Our solvers were coded in Python and C++ and executed on several desktop laptop computers, as well as the LIMOS and LIS clusters.

2 Algorithms

We used two different algorithms to compute initial solutions, a preprocessing phase that can be executed beforehand, and a local search phase to improve the solutions.

2.1 Integer Programming Approach

A simple idea to solve the challenge problem is to produce a set V of random translations of each item inside the container and then reduce the problem to a kind of maximum weight independent set problem in a graph $G = (V, E)$. Each translated item is a vertex and there are two types of edges: (1) an edge between two translations that overlap and (2) translations of the same item i form a clique C_i. If all item have quantity one, then this is a traditional maximum weight independent set problem. However, if items have non-unit quantities, then each clique C_i is associated with the quantity q_i of item i and at most q_i vertices of the clique are allowed in the solution.

This combinatorial problem can easily be modeled as integer programming with one binary variable per vertex. A type-1 edge uv is modeled as $u + v \leq 1$ and each clique C_i is modeled as $\sum_{v \in C_i} v \leq q_i$. The CPLEX solver [2] can optimally solve graphs with a few thousand vertices obtained from the challenge instances, which is not enough to obtain good solutions using uniformly random placements.

To obtain better solutions, we start from a solutions S obtained with the aforementioned method and build a new graph $G = (V, E)$ as follows. Let $\sigma > 0$ be a parameter and N be a set of the zero vector and random vectors where each random vector has x and y coordinates as Gaussian random variables of average 0 and standard deviation σ. We create a translation in V for each item that is placed in S and for each translation vector in N if the translation is inside the container. We also create vertices in V using uniform random translations for all items. Edges and cliques are created as before, and the new combinatorial problem is

solved with CPLEX. We repeat this procedure multiple times using the previous solution S and reducing the value of σ at each step.

This method works well for instances with up to 200 items. To handle larger instances, we partition the container using a square grid and partition the items equally among the cells. The partition is such that items are grouped by the slope of the longest edge, breaking ties by the slope of the diameter. Each cell is then solved independently. The intuition to group items of similar slope together is that they can often be placed in a way that minimizes the wasted space. Since the values of the items do not seem to be related to the slopes, this approach works well for the challenge instances.

2.2 Greedy Heuristic

The greedy heuristic starts by creating an initial list L of n grid points inside the container (typically $n = 1000$). The list L is shuffled and we compute its centroid c rounded to integer coordinates. The point c is inserted in the beginning of L. The input items are placed into a list I ordered by decreasing utility, where the utility function is described next. Different utility functions may be used (more details in Section 3). The goal is that items of small area and high value have high utility while large items with small value have low utility.

We then build the packing by considering the items of the list I one by one in order. At each step, we have a current packing and a new item i to pack. We first try to pack i at the first grid position $g \in L$ and at a number of random positions around g. If one of these positions is available, i.e the item is fully contained in the container and does not cross any currently packed item, then we pack item i at the corresponding position and proceed to the next item in I. Otherwise, we go to the next position. If all the positions have been tried without success, then we go to the next item in I.

Immediately after an item $i \in I$ is packed, its position is modified as follows. We choose a direction u in which i is *pushed* until it hits another item or the container boundary. Pushing an item means that we translate it in the direction u and in directions v where the dot product of u and v is positive until no improvement in any direction u or v can be achieved.

The greedy algorithms ends when all items in I have either been packed or tested at all the positions of the list L.

2.3 Slates Preprocessing

The goal of the preprocessing step is to precompute some sets of items that assemble together efficiently. We call *slate* a set of items with a fixed relative position. A slate plays the same role as an item, but it is a polygonal set instead of a simple polygon. As computing slates takes a lot of time, we control it by building graphs of items. The goal of these graphs is to tell the pairs of items that we want to try to assemble together. A first graph contains for instance all the thin items and connects the items with similar diameter directions. Another graph contains the non convex items and connects them with items that can fill their concavities. We also compute graphs containing hundreds of random items and connecting items having a close pair of edges uv and $u'v'$ where the Euclidean distance $d(v - u, u' - v')$ is small compared to $d(u, v)$. For each graph, we try to assemble adjacent items by searching for a relative position minimizing the area of the convex hull of their union. Each slate composed by two items has a utility and we only keep the slates of high utility. We take the slates composed by two items and try to assemble a third adjacent item of the graph. We get a new generation of slates with 3 items and continue the growing process until a fixed level. Again at each generation, we keep only the slates of high utility.

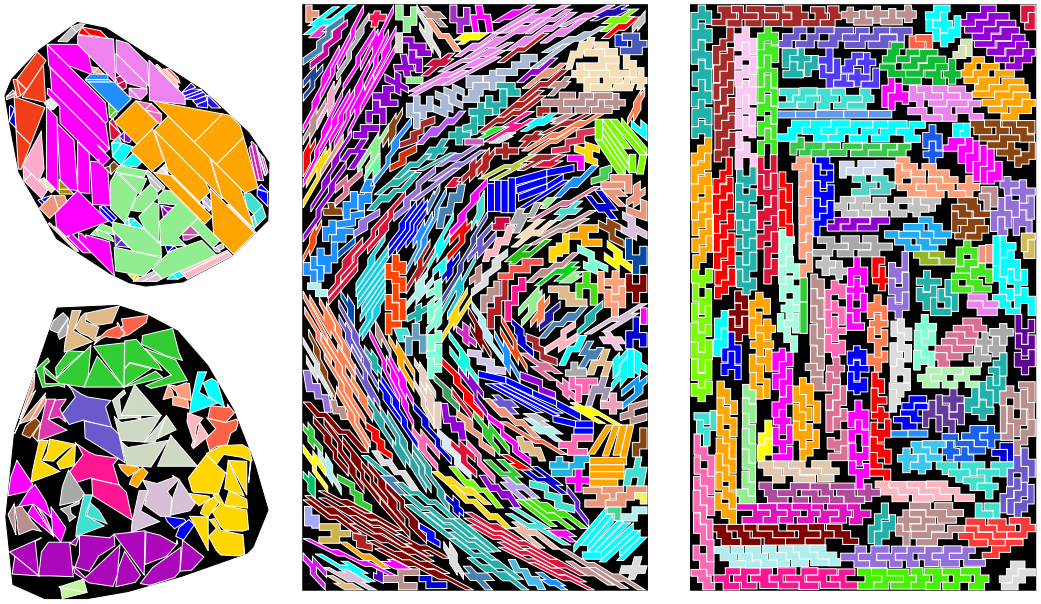

Figure 2 Slates for the instances `jigsaw_cf2_xf42cb20_670`, `random_cf3_x21f5def_200`, `satris1685`, and `atris1660`. The items of a slate are drawn with the same color.

Slates work particularly well for the `atris` instances where the items resemble polyominoes.

2.4 Local Search

To improve the quality of the solutions obtained with the previous approaches, we use a local search optimization scheme by repeating the following routine until a time limit is reached. We randomly choose between executing two routines, either a *fill* routine or a *push* routine. The fill routine is similar to the greedy heuristic. We try to pack all the unpacked items at several grid and random positions.

The push routine first randomly chooses either a point v in the interior of the container or a vertex of the container. The packed items are sorted by the distance of their centroids to the point v, from the farthest to the closest. Then, each packed item i with centroid c_i is pushed in the direction $c_i - v$. The goal is to free the space around v. Once all packed items have been pushed, we try to pack the unpacked items around v by using the grid points g in L close to v and some random integer points around each point g. It may be the case that an item can be packed if we remove some other items of the packing that intersect it. In this case, we compute the benefit of removing the conflicting items in order to pack the new item, replacing the item if the benefit is non-negative. To accelerate the procedure for large instances, we add a parameter which allows us to push only the items that are close to v.

3 Parameter Analysis

In order to compare the different parameters, we consider 18 instances of various sizes, as shown in Table 1. Throughout, we refer to the ratio between the value of the solution in question and the value of the best solution in the challenge as the *value ratio* (notice that the competition score is the square of the value ratio). The best solutions to the first 15 table instances are shown in Figure 3.

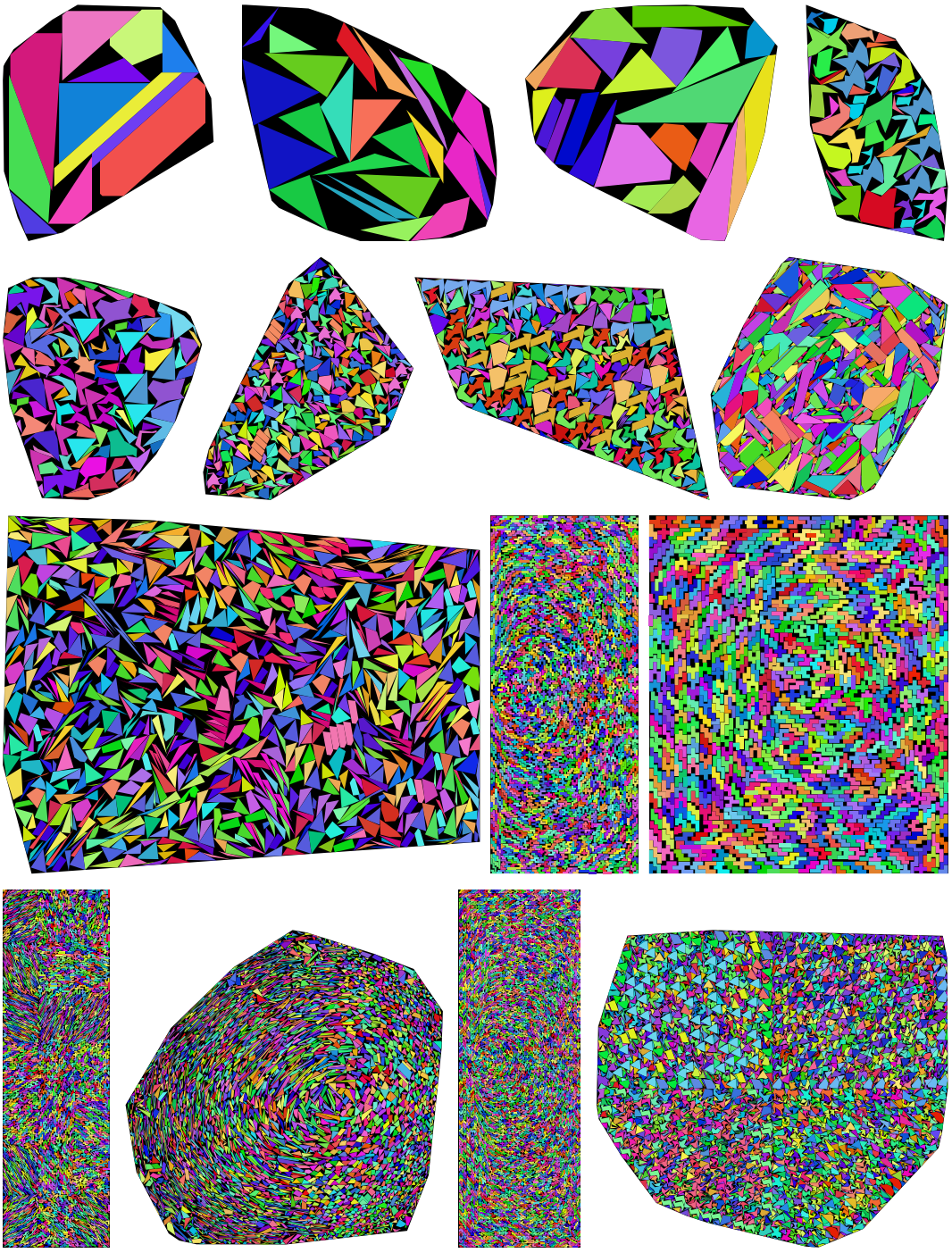

Figure 3 Our best solutions to the first 15 rows of Table 1, in order.

Instance	IP	IP + LS	Gr	Gr + LS	Our best
jigsaw_cf4_273db689_28	**1**	**1**	.900	.950	1
random_cf1_64ac4991_50	**.926**	**.926**	.851	.925	.926
jigsaw_rcf4_7702a097_70	.959	**.982**	.867	.924	1
random_cf4_50e0d4d9_100	.943	**.972**	.877	.941	1
random_cf3_x21f5def_200	.950	**.967**	.893	**.967**	1
random_rcf1_340f4443_500	.952	.960	.932	**.981**	.984
random_rcf3_x7651267_1000	.927	.970	.926	**.980**	1
jigsaw_rcf4_6de1b3b7_1363	.934	.948	.928	**.980**	1
random_cf1_x51ab828_2000	.937	.945	.892	**.950**	.961
atris2986	.910	.931	.881	**.948**	.988
atris3323	.893	.918	.866	**.951**	1
satris4681	.919	.930	.886	**.937**	1
jigsaw_cf1_4fd4c46e_6548	.944	.944	.955	**.963**	.975
atris7260	.886	.902	.872	**.920**	.974
random_cf3_x4b49fe2_10000	.924	.938	.889	**.963**	1
satris15666	.923	**.951**	.877	.910	1
jigsaw_cf1_203072aa_32622	.732	.876	.972	**.973**	.985
atris41643	.513	.803	.854	**.877**	.911

Table 1 Value ratio of several instances using integer programming (IP) or greedy (Gr), both before and after 24 hours of local search (LS).

Figure 4 shows the evolution of the value ratio by using the slate preprocessing step, the greedy algorithm followed by local search until it becomes stable with standard parameters for three instances with 100, 200 and 1000 items, respectively. It shows that sometimes the slates provide a benefit, while sometimes it has a negative impact on the results.

We now analyze the parameters of our algorithms. The greedy algorithm depends on several parameters:

- The number of grid positions and the number of random integer positions to consider around each of these points. We typically used numbers from 500 to 3000 grid positions plus from 3 to 10 random positions around each grid point. Figure 5 compares different choices of the parameter and shows that there is no major improvement to wait for by using more than 1000 grid positions.
- The utility function used to sort L may be the value of the item, the value of the item divided by its area, the value of the item to a power 1.5 divided by its area with or without complementary weights. For the Table 1 instances, we show in Figure 6 the value ratio first after greedy and second after optimization for different utility functions.
- The direction u to push an item may be chosen randomly (strategy 1), or as a normal to the item diameter (with a random choice to determine which side, strategy 2). However, we mainly used the following direction choice. If the item is thin (ratio between its diameter and its width normal to the diameter above 3), we push it in the normal direction to its diameter but to the left while if it is fat (ratio below 3) we push it to the right (strategy 3). Finally, we may push the items in a direction normal to their longest edge if they are fat (strategy 4). The value of the average for 10 runs on each Table 1 instance is shown in Figure 7.

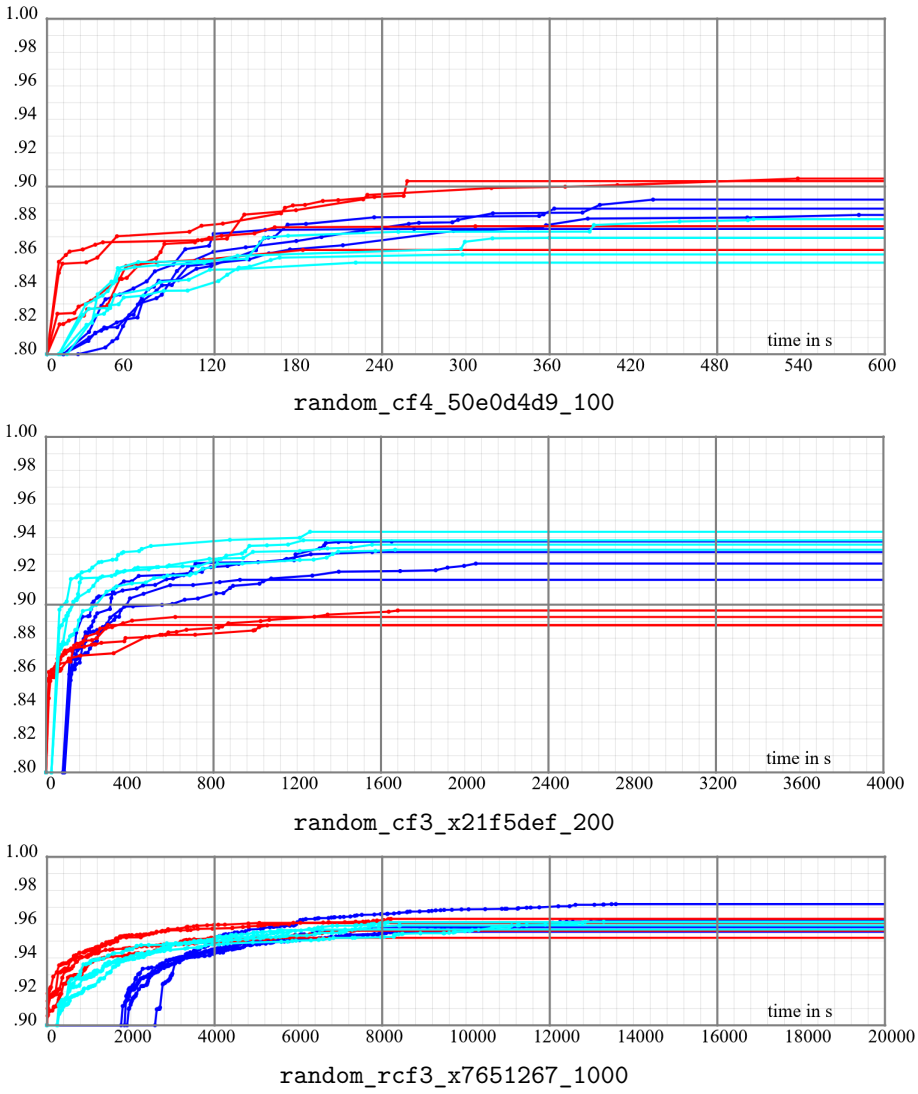

Figure 4 The value ratio over time for the instances `random_cf4_50e0d4d9_100`, `random_cf3_x21f5def_200`, and `random_rcf3_x7651267_1000` for 12 independent executions of the greedy algorithm and a subsequent local search. Red curves are computed without slate preprocessing, while the light blue and dark blue curves respectively include a preprocessing of slates with at most 5 and 10 items.

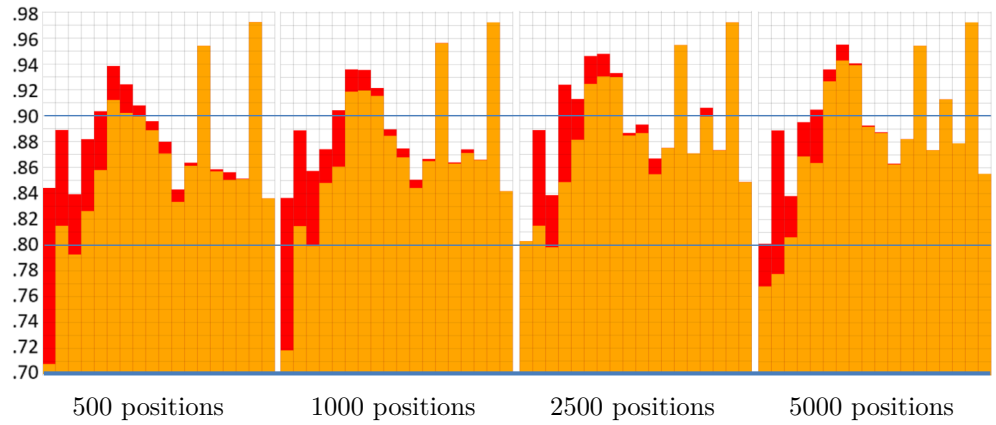

Figure 5 Value ratio of the greedy algorithm (orange) for 500, 1000, 2500 and 5000 grid positions, for the 18 instances in Table 1. In red, we show the value attained after 1000 seconds of local search.

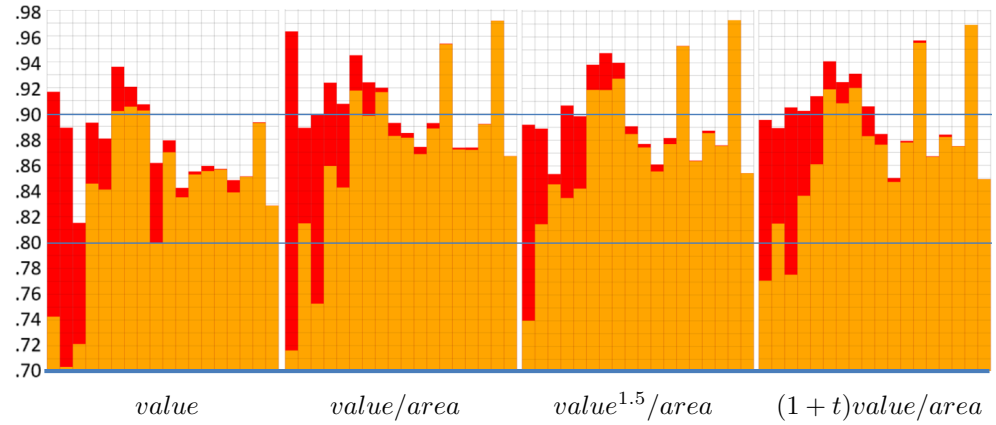

Figure 6 Value ratio of the greedy algorithm (orange) using different utility functions, for the 18 instances in Table 1. t is the ratio between the length of the item diameter and the width of the item in the direction normal to the diameter. In red, we show the value attained after 1000 seconds of local search.

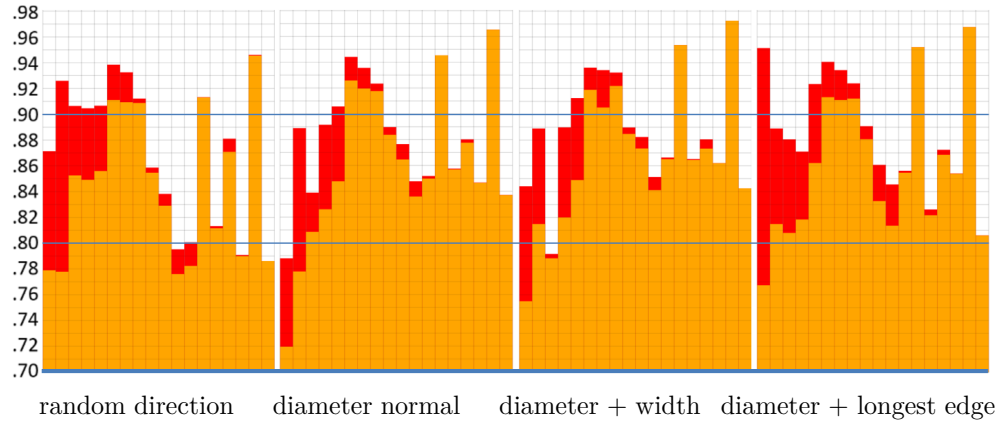

Figure 7 Value ratio of the greedy algorithm (orange) using different push strategies, for the 18 instances in Table 1. In red, we show the value attained after 1 hour of local search.

References

1 Alkan Atak, Kevin Buchin, Mart Hagedoorn, Jona Heinrichs, Karsten Hogreve, Guangping Li, and Patrick Pawelczyk. Computing maximum polygonal packings in convex polygons using best-fit, genetic algorithms and ilps. In *Symposium on Computational Geometry (SoCG)*, volume 293 of *LIPIcs*, pages 86:1–86:9, 2024.

2 IBM ILOG CPLEX. V22.1: User's manual for CPLEX. *International Business Machines Corporation*, 2023.

3 Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. Maximum polygon packing: The cg:shop challenge 2024, 2024. `arXiv:2403.16203`.

4 Martin Held. Priority-driven nesting of irregular polygonal shapes within a convex polygonal container based on a hierarchical integer grid. In *Symposium on Computational Geometry (SoCG)*, volume 293 of *LIPIcs*, pages 85:1–85:6, 2024.

5 Canhui Luo, Zhouxing Su, and Zhipeng Lü. A general heuristic approach for maximum polygon packing. In *Symposium on Computational Geometry (SoCG)*, volume 293 of *LIPIcs*, pages 84:1–84:9, 2024.