Priority-Driven Nesting of Irregular Polygonal Shapes Within a Convex Polygonal Container Based on a Hierarchical Integer Grid

Martin Held
FB Informatik, Universität Salzburg, Austria

Abstract

Our work on nesting polygons is based on two key components: (1) a hierarchy of uniform integer grids for maintaining free space within the container during the nesting such that placement queries can be answered reasonably efficiently, and (2) priority heuristics for choosing the order in which the polygons are tested for placement. We discuss our approach and shed a light on the results obtained.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Computational Geometry, geometric optimization, nesting, packing, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SoCG.2024.85

Category CG Challenge

1 Introduction

The task of the 2024 Computational Geometry Challenge "Maximum Polygon Packing" – called Challenge in the sequel for the sake of brevity – was to place (a subset of) non-convex simple polygons ("*items*") within a convex polygon ("*container*") in the plane. Every item has an integer value associated with it. If not all items can be packed into the container then one is to choose a subset of the items such that the overall sum of the values achieved by the items nested is maximized. The Challenge asked to find nestings for 180 instances of containers and their items. We refer to the survey by Fekete et al. [4] for more details. The approaches of the other three top teams of the Challenge are presented in [1, 2, 3].

2 Approach

2.1 Basic algorithm

All initial and goal positions of the polygons of the Challenge instances have integer coordinates. Hence, in theory, every instance could be solved to optimality by trying all combinations of points on the integer grid as possible placements for the items. Needless to say, the enormous complexity of this naïve approach will not allow to handle anything but toy instances. In order to bring the Challenge problem down to a complexity that we can manage, we made the following simplifications that are explained in more detail below:

- We use the nodes of a coarse integer "container grid" as candidate placements of an item.
- We assign a priority to every item and sort the items in decreasing order of their priorities.
- We test placements of the items in a slightly randomized order of their priorities.

To obtain a *container grid* we lay a uniform grid of square cells over the bounding box of a container polygon. The actual number of cells of this container grid mostly depends on the memory available on a computer: We employed grids with 100 000 000 to 12 800 000 000 cells. All cells that do not lie completely within the container polygon are marked as "full"; all

40th International Symposium on Computational Geometry (SoCG 2024).
Editors: Wolfgang Mulzer and Jeff M. Phillips; Article No. 85; pp. 85:1–85:6
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

other cells are marked "empty". For every item polygon we compute its bounding box and subdivide it by a uniform grid of the same cell size. For item grids a cell is marked "empty" if it lies completely outside of the item polygon, and "full" otherwise. See Figure 1.

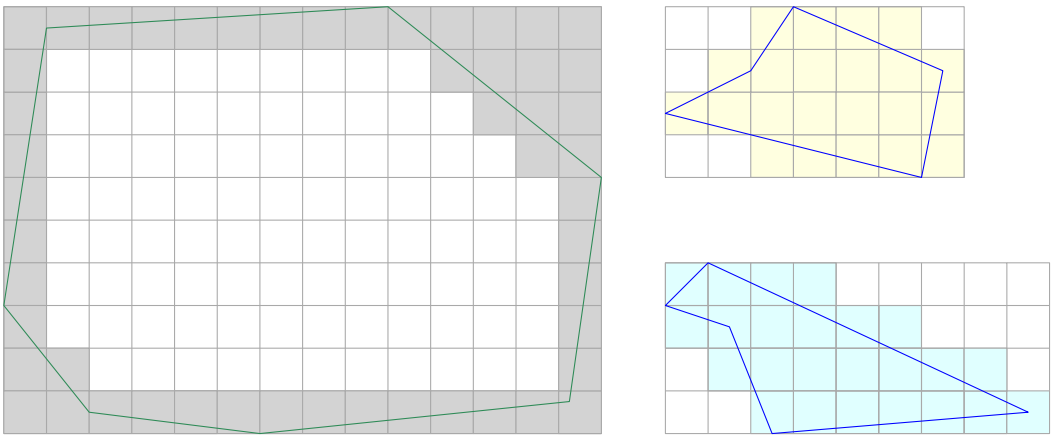

Figure 1 Grids of sample container (left) and two item polygons (right). The full cells are shaded.

We note that the exterior of the container is a subset of the union of all full container cells, while the interior of an item is a subset of the union of its full cells. A placement of the lower-left cell of an item grid on a cell of the container establishes a pairing between all item cells and some container cells. We run a cell-by-cell test: If no full item cell overlaps with a full container cell then the placement is feasible. For a feasible placement the container cell overlapped by the lower-left cell of the item grid is called an *anchor* for the item.

To find an anchor for an item, we scan all cells of the container grid from its bottom row to its top row, and from left to right within every row. We stop these rightwards/upwards scans as soon as the right-most (top-most, resp.) cells of the item overlap with the right-most (top-most, resp.) cells of the container. If an anchor is found, i.e., if the item can be packed into the container relative to some other items already packed, then the item stays at this position for the rest of the nesting process. If the bottom item of Figure 1 is to be placed prior to the top item, then this deterministic approach yields the nesting shown in Figure 2.

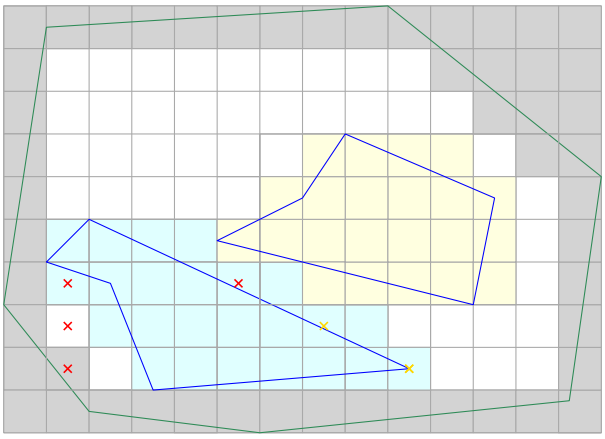

Figure 2 Sample nesting achieved by our approach for the setting of Figure 1. The crosses show the six candidate anchors tested by the improved approach that is explained in Section 2.2.

As *priority* key for the selection of the items we use one of three alternatives: (1) the value of the item specified by the organizers of the Challenge, named "value", (2) its value divided by the area of the item polygon ("area"), or (3) its value divided by the area of the union of the full item cells ("cells"). Some *randomness* in the selection of the items to be packed is introduced as follows: If the instance has k items then the array of the items (sorted in decreasing order of their priorities) is split into $\lfloor\sqrt{k}\rfloor$ many subarrays ("*buckets*") of roughly equal length. Let $0 < p < 1$ be a user-specified *probability*. Then we pick the first bucket (which contains the items with highest priorities) with probability p. If the first bucket is not picked (or already empty), then we recurse on the remaining $\lfloor\sqrt{k}\rfloor - 1$ buckets. Within the bucket picked, the actual item to be tested for placement is selected with uniform probability. Of course, every item selected for placement is deleted from its bucket.

2.2 Improvements

The advantage of the approach sketched above is that it is easy to implement. The main disadvantage is its computational inefficiency: Suppose that we try to pack an item with $w \times h$ cells into a partly filled container but that it does not fit. Then we would end up trying all cells of the container grid as anchor except for cells within the w left-most columns and the h top-most rows of the container grid. E.g., we end up running cell-by-cell tests for 17 candidate placements until an actual anchor for the second item in Figure 2 is found.

As first improvement, for every item grid we determine a *shift block*: It is a longest one-row rectangle of contiguous full cells. In Figure 1, the bottom item has two shift blocks with seven cells each, and the top item has one shift block with seven cells. Now suppose that we check whether a container cell forms an anchor for an item. In such a candidate placement the shift block of the item covers some cells of the container grid. If one of these cells is full then let c be the right-most cell of any block of contiguous full container cells partly covered by the shift block. Then we know that we need to shift the item rightwards such that its shift block starts immediately right of c. If the shift block does not cover any full container cell then a cell-by-cell test is carried out for all cells of the item grid. As a result, in order to find an anchor for the second item, in the setting of Figure 2 we test only the six candidate cells marked by crosses. (No cell-by-cell test is required for the two candidate cells marked by golden crosses since we are already too far to the right.)

As second improvement we implemented a grid hierarchy: For an initial container grid $\mathcal{G}_0$, the next coarser grid $\mathcal{G}_1$ has about one quarter of the cells of $\mathcal{G}_0$ at exactly four times their size. A cell of $\mathcal{G}_1$ is marked as full only if all four corresponding cells of $\mathcal{G}_0$ are full, and empty otherwise. By recursively computing coarser grids we end up with a hierarchy of container grids. This recursive process stops once a grid has less than ten rows or columns. (The threshold 10 was determined by quick experiments.) Same for the item grids, see Figure 3.

Summarizing, our improved nesting approach proceeds as follows, with details as outlined above: (1) Randomly determine an item to be packed. (2) For the coarsest grid of the item, scan the corresponding grid $\mathcal{G}_i$ of the container until a candidate anchor is found, thereby making use of the item's shift block. (3) Recursively test the four candidate anchors within G_{i-1} until a valid placement is found in the container grid $\mathcal{G}_0$. (4) If a placement is found, the appropriate cells in $\mathcal{G}_0$ are marked as full and this information is propagated upwards to the coarser container grids. Otherwise, we proceed with other candidate anchors within $\mathcal{G}_i$.

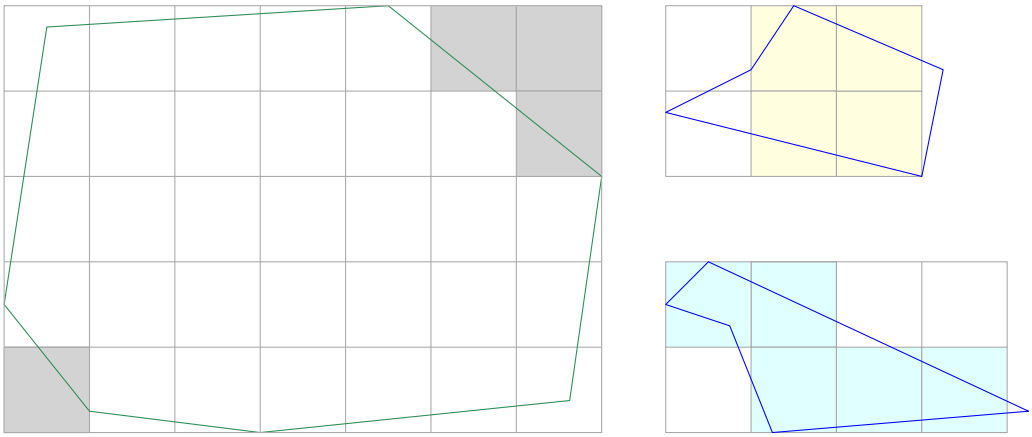

Figure 3 Coarser grids for the container and item polygons of Figure 1. The full cells are shaded.

3 Practical computation

3.1 Computational environment

We ran our nesting code on a diverse set of computers operated by our Computational Geometry and Applications Lab, by the High-Performance Computing Center and by other groups at the University of Salzburg. We used a varying number of standard PCs plus some (rather small) compute servers. One server, an Intel Core i9-10980XE CPU clocked at 3.00 GHz, with 128 GB of RAM, was used to collect statistics for the resource consumptions for four sample Challenge instances, see Table 1. The entries list the maximum process size (in GB) and the total runtime (in seconds) for the initial set-up of the grid hierarchies and for 100 nesting runs for grids with $k \cdot 100\,000\,000$ cells, for $k \in \{2, 8, 32, 128\}$, with the improvements of Section 2.2. (Without them even the fastest test runs take several days.)

Table 1 Maximum process size and runtime for the test runs for four sample instances.

process size (in GB) for k	2	8	32	128
atris19260	0.64	2.57	10.3	41.2
jigsaw_rcf4_x296c58c_70	0.57	2.28	9.15	36.6
random_rcf2_x25afb10_2000	0.59	2.37	9.51	38.0
satris1685	1.68	6.74	26.9	107.8
runtime (in sec) for k	2	8	32	128
atris19260	8431	9909	12280	19360
jigsaw_rcf4_x296c58c_70	193	493	2217	6682
random_rcf2_x25afb10_2000	2564	8003	28431	57582
satris1685	785	1717	5428	18925

As expected, the memory consumption grows linearly with the number of grid cells. The actual runtime is significantly more difficult to predict: It depends on the number of items to be packed. But it also depends on how dense a nesting is achieved and how quickly a dense nesting is achieved, thus making our heuristics for pruning the number of potential anchors more or less effective. As a result, the differences in runtime are substantial both among the instances for the same number of grid cells and when increasing the number of grid cells for one instance. The three different priority schemes have no impact on the runtimes, though.

Our low-profile way of accessing computers resulted in a non-uniform consumption of computational resources, which in turn had highly non-uniform performance levels, ranging from 15-year-old compute servers to machines acquired just a year ago. The heterogeneity of our computational resources makes it very difficult to come up with a reliable ball-park figure of the total CPU time consumed. We estimate, though, that processing all 180 data sets of the Challenge would have kept a standard desktop machine (with a decent amount of RAM) busy for a few years.

3.2 Experimental results

At the beginning of our attempts to compute good nestings we tried to understand which of the three ways to assign priority keys to the instances would be best for which group of items. Unfortunately, we could not figure out any characteristic that would allow to predict the best priority scheme for some new input. In particular, the priority schemes "area" and "cells" scored winners among all four groups of the Challenge instances, see Figure 4. However, the scheme "value", which simply sorts the items in decreasing order of their values, was not successful for the atris and satris instances. Overall, there is a winner, though: The scheme "cells", which assigns the value of an item divided by the area of the union of the full item cells as priority, scored our best result for about 44% of the instances. The two other priority schemes each scored the best result for about 28% of the instances. Considering the top three results per instance causes these percentage numbers to change only in their fractional parts.

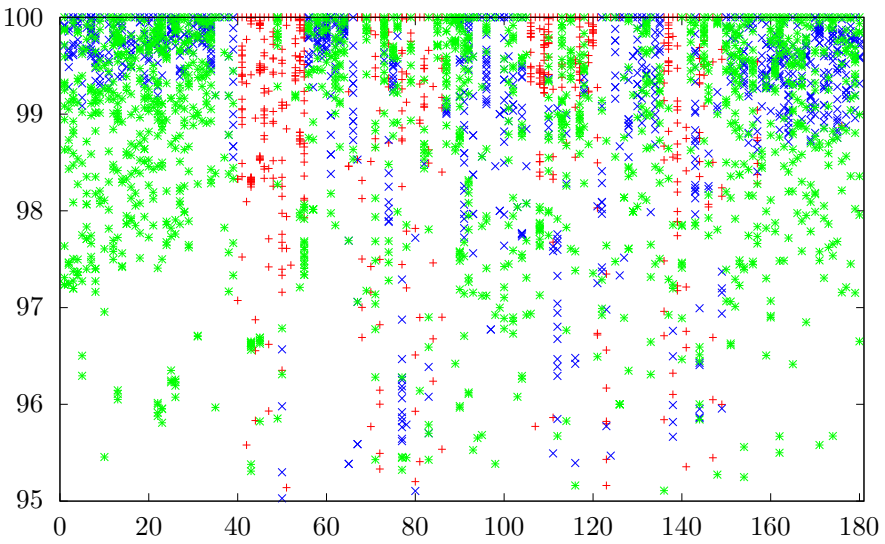

Figure 4 Percentage scores achieved by the three priority schemes for those results that ended up above 95% of the best score per instance. A red plus denotes "value", a blue cross denotes "area", and a green star denotes "cells". The instances are numbered from 1 to 180 according to the lexicographical ordering of their names; the atris instances correspond to the numbers 1–29, the jigsaw instances to 30–89, the random instances to 90–149, and the satris instances to 150–180.

Another parameter to play with is the probability p for choosing the buckets and items in decreasing order of the priorities of the items. The situation is fairly clear for the "area" scheme: Virtually all best scores were attained for $p := 0.90$. For "value", the best scores were attained for $p := 0.95$ (in about 55% of the wins) and $p := 0.90$ (in the remaining 45% of the wins). For "cells", we observed about 45% of the best scores each for $p := 0.90$ and $p := 0.80$, and just a few wins for $p := 0.70$ and $p := 0.60$.

Finally we examined the dependence of the scores achieved on the number of grid cells used. Does it pay off to increase the runtime by employing a higher-resolution grid? In general, the answer seems to be yes – except for the two instances jigsaw_rcf4_x6f71c05_3333 and random_rcf2_x4db924e_5000: For those two instances we achieved our best scores (with "cells") during our early packing runs with grids with just 100 000 000 cells.

For most other instances our best scores were achieved with grids with 6 400 000 000 cells, and with 12 800 000 000 cells for a few instances. But, in general, the gain in total value is rather moderate, as shown by taking the best value obtained by a high-resolution grid and considering the percentage achieved by a grid with $1/64$ the number of cells of that grid. Over all 180 instances of the Challenge, the best percentage value was 99.75% and the worst value was 58.97%. We get a mean percentage of 94.25%, and a median percentage of 97.14%. These numbers are likely to be biased in favor of high-resolution grids since we had considerably more packing runs for high-resolution grids than for low-resolution grids. Hence, for the Challenge instances one may expect our approach to gain just very few percent in total value if the number of grid cells is increased by a factor of 64.

4 Discussion

Our simple approach yields surprisingly decent results. Still, there are natural avenues for improvements which we did not implement due to lack of time: (1) Rather than packing the items in a right-upwards nesting direction only, randomly pick one of the four nesting directions per item to be placed. (2) Identify regions of empty cells in the container grid $\mathcal{G}_0$ that are far too small to hold an item and mark them as full in order to make our grid hierarchy more effective. (3) Weed out impossible placements of an item early in the process by using several shift blocks or shift blocks that span multiple rows. (4) Reduce the memory consumption by a more clever way of storing the grids, e.g., based on run-length encoding.

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 83:1–83:9, 2024. doi:10.4230/LIPIcs.SoCG.2024.83.

2 Guilherme Dias da Fonseca and Yan Gerard. Shadoks approach to knapsack polygonal packing. In *Symposium on Computational Geometry (SoCG)*, volume 293 of *LIPIcs*, pages 84:1–84:9, 2024. doi:10.4230/LIPIcs.SoCG.2024.84.

3 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 86:1–86:9, 2024. doi:10.4230/LIPIcs.SoCG.2024.86.

4 Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. Maximum polygon packing: The CG:SHOP Challenge 2024, 2024. arXiv:2403.16203.