Algorithms for the Lower-Left Anchored Rectangle Problem

Anton Lorenzen

Allen Freedman [Tut69, p. 345] posed the following combinatorial problem: For some finite set of points in the unit square, we must find for each point p a rectangle R^p that has p as its lower-left corner such that no two rectangles intersect. If the origin is in the point set, can we cover at least half of the unit square in such rectangles?

Surprisingly little was known about the problem just a few years ago, but it has received renewed attention in recent work. Damerius et al. [Dam+21] show that the TilePacking heuristic introduced below can always cover 39% of the unit square, but they also give an instance where this heuristic only covers 43.3%. No better ratio is known.

Rather than searching for the fastest algorithm attaining a known ratio, we therefore consider the problem of maximising the area covered. Formally:

Def. 1 (Lower-left Anchored Rectangle Packing)**.** Given a finite set of points P in the unit square $[0,1]^2$, find a rectangle $R^p = [p_x, r^p_x) \times [p_y, r^p_y) \subseteq [0,1]^2$ for each $p \in P$ such that $R^p \cap R^q = \varnothing$ for all $p \neq q \in P$ and the total area $\sum_{p \in P} \text{area}(R^p)$ is maximized (where $\text{area}(R^p) = (r^p_x - p_x)(r^p_y - p_y)$).

Note that in particular $q \notin R^p$ for $p \neq q$. Often, we will have that the origin $(0,0)$ is in P, but this is not required. Indeed we will see, that the rectangle of the origin can always be chosen last, independent of the other rectangles.

It is not known whether the problem can be solved in polynomial time, but it is widely believed that this is not the case. Antoniadis et al. [Ant+19] show that the related problem, where the points p need to be in the center of their rectangle R^p, is NP-hard. However, their analysis does not translate easily to the present problem.

So how could one solve this problem? All the algorithms we discuss below are based on a strategy Dumitrescu and Tóth [DT15] proposed. They call a feasible solution *pareto optimal* if no rectangle can be made bigger without changing the others. Then they describe how to choose for each pareto optimal solution a point p, such that removing R^p yields a pareto optimal solution for $P \setminus \{p\}$:

Given a solution, we write $p \preceq' q$ if $[p_x, 1] \times [p_y, 1]$ intersects R^q. In particular, if R^q prevents the rectangle R^p from growing, we will have $p \preceq' q$. The transitive closure $\preceq$ of $\preceq'$ induces a partial order: This is clear except in the antisymmetry case. But it can be seen by induction that if $p \preceq' q$ then there is no q' to the lower-left of $(\max\{p_x, q_x\}, \max\{p_y, q_y\})$ with $q \preceq q'$.

Therefore, we can pick any linear ordering compatible with the partial order $\preceq$ and remove the smallest element p (which prevents no rectangle from growing) to obtain a pareto optimal solution for $P \setminus \{p\}$. As the optimal solution is also pareto optimal, this implies that there is a permutation $p_1, \dots, p_n$ of the points in P such that we obtain the optimal solution by choosing maximal rectangles for each p_i in that order. We call this permutation the *optimal permutation*. We do not have direct access to it, but we can simply try all possible permutations and take the best result.

Unfortunately, this algorithm needs to consider $O(n!)$ permutations. Harder [Har19] discovered that the dynamic programming technique of the $O(2^n n^2)$ Held-Karp algorithm (for the Traveling Salesman Problem) also works for this problem. While we discuss that technique in section 3, packing according to special permutations has proven effective in practice. Several variations are possible, which we discuss in section 1 and section 2.

Our improved algorithms allow us to consider a wide range of instances. In section 4 we show some charts that display the performance of the algorithms on a dataset of random points. We also implemented a user interface that allows interactive exploration of these instances. In section 5 we give a short manual.

1 Tile Packing

How could we place the rectangles in the optimal permutation? Dumitrescu and Tóth [DT15] show that this can be done in time $O(n \log n)$ using the TilePacking algorithm: To place p_{i+1}, we iteratively keep $R_i = \bigcup_{p=p_1}^{p_i} [p_x, 1] \times [p_y, 1]$. We call $R_{i+1} \setminus R_i$ the *tile* of p_{i+1} and choose the best rectangle $R^{p_{i+1}}$ within that tile. As every R_i is a polygon consisting of at most $2(i+1)$ points, we keep all these corners (except $(1,1)$) in a binary search tree sorted by x-coordinate. For point p

we then lookup the point q in the binary search tree that is closest to the right of p_x and consider the points to the right of q iteratively while their y-coordinate is bigger than p_y. From these points $q = q^1, ..., q^m$, we choose the biggest rectangle $[p_x, q_x^i) \times [p_y, q_y^i)$ and delete them from the tree. Finally, we add p, (p_x, q_y^1) and (q_x^m, p_y) to the tree. Clearly, this preserves the invariant that the tree contains the contour of R_{i+1} and can be done in time $O(n \log n)$. It will also reach an optimal solution, as $p_i \precsim p_j$ for $i < j$ and therefore no p_{i+1} has its optimal rectangle intersecting R_i.

Figure 1: Tile rectangle for the blue rectangle

1.1 Norm-derived Permutations

Note that TilePacking requires that the permutation obeys $\preceq$ if we seek an optimal solution. However, we can run the algorithm on any permutation, as long as new points p_{i+1} are outside R_i. We say that q *dominates* p, written $p \leqslant q$ if $p_x \leqslant q_x$ and $p_y \leqslant q_y$. Clearly, $p \leqslant q$ implies $p \preceq q$. We call a set P *dominant* if for all $p \in P$ and $p \leqslant q$, we have that $q \in P$. For the TilePacking algorithm to work we need that any prefix $p_1, ..., p_i$ of the permutation is dominant. However, by the definition of $\preceq$ and $\leqslant$, every prefix of the optimal permutation is dominant.

To obtain an approximate solution, we can choose to use any permutation obtained by sorting the points according to a norm that guarantees that $\|p\| < \|q\|$ if $p < q$. In the literature, the most commonly used norm is the 1-norm (or Manhattan-norm) as this can also be seen as packing the points according to their occurrence in a diagonal sweep from $(1, 1)$ to $(0, 0)$. This is called the TilePacking *heuristic*.

Another interesting norm is the $-\infty$-'norm' (computed by taking the minimum of the coordinates). The algorithm then behaves as if we were growing the unit square towards the lower-left and packing new points as they arrive (in a form of online strategy).

2 Greedy Packing

While the TilePacking algorithm is fast and easy to implement, the result will not necessarily be pareto optimal if we do not use the optimal permutation. In this section, we discuss the GreedyPacking algorithm which is very similar to the TilePacking algorithm but always returns a pareto optimal solution. Harder [Har19] showed how this can be achieved in time $O(n^2 \log n)$ and space $O(n^2)$.

However, we can improve this to $O(n^2)$ and $O(n)$ respectively if we perform the algorithm as follows: In step 1, we find for the to-be-placed point p the *tile rectangle*: the smallest rectangle that contains any rectangle R^p that does not intersect any existing rectangle. In other terms, we choose a point q such that the rectangle T spanned by p and q intersects only rectangles R^s where s lies in T and T is maximal. This can be done in time $O(n)$ by setting q to $(1, 1)$ and then inspecting all existing rectangles: if R^s intersects T and $s_x \leqslant p_x$, we set $q_y = \min\{q_y, s_y\}$; if R^s intersects T and $s_y \leqslant p_y$ then we set $q_x = \min\{q_x, s_x\}$.

In step 2, we consider all points in T to find the rectangle R^p. For this we maintain a list of all points sorted by x-coordinate. We go through the list, and if a point s lies in T, we set $R^p = \max\{R^p, [p_x, s_x) \times [p_y, q_y)\}$ and set $q_y = s_y$.

While GreedyPacking performs slightly better than TilePacking in practice, there is no difference in the approximation guarantee as Dumitrescu and Tóth [DT15] show. Indeed, if we place for any point (x, y) two new points directly below and to the left at $(x - \epsilon, y)$ and $(x, y - \epsilon)$ and choose squares for them greedily, directly after choosing a rectangle for (x, y), their squares will add at most 2ϵ covered area and prevent any other uses of the tile of (x, y). As such, the result of Damerius et al. [Dam+21], that TilePacking can not cover more than 43.3% for any instance, applies to GreedyPacking as well.

Unlike the TilePacking algorithm above, the GreedyPacking algorithm works even for arbitrary permutations where a prefix of the permutation may not be dominant. However, even with this new capability, the difference in runtime between the two algorithms seems big. Next, we give a $O(n \log^2 n + k)$ algorithm that can perform both TilePacking and GreedyPacking on arbitrary permutations, where k is the sum of the number of points in all tile rectangles. For TilePacking, we have $k \in O(n)$, but for GreedyPacking $k \in O(n^2)$. Nonetheless, in most instances we expect k to grow smaller than $O(n^2)$ even for GreedyPacking.

We achieve this by implementing step 1 with a new datastructure that we call a *priority interval tree* and implementing step 2 with a range tree. We give descriptions of some auxiliary datastructures below, which can also be found in the computational geometry literature (see e.g. [Kle06, p. 135])

2.1 Finding all points in an interval

Given a set of n points p on the line, we want to find for any interval the set of points contained in that interval. We can construct a *range tree* for this in time $O(n \log n)$ that supports queries returning k points in time $O(\log n + k)$.

We sort the points and create a balanced binary tree that has them as leaves. In each internal node we store the smallest interval that contains all leaves reachable from this node. Clearly, this can be done in time $O(n \log n)$.

For a query we then descend into the binary tree by going left if the interval of the left internal node intersects the query interval and else right if the interval of the right internal node intersects the query interval (if none of them intersects there are no points in the query interval). This way we find the left-most point contained in the query interval in time $O(\log n)$. Then we report all points to the right of the left-most point that are contained in the query interval in time $O(k)$. This can for example be done by connecting all leaves in a linked list. Alternatively we can also walk through the tree in time $O(\log n + k)$.

2.2 Finding all points in a rectangle

Given a set of n points p in the plane, we want to find for any rectangle the set of points contained in that rectangle. We can construct a *2-dimensional range tree* for this in time $O(n \log n)$ that supports queries returning k points in time $O(\log^2 n + k)$.

First we take the set of x-coordinates of the points and construct a one-dimensional range tree for them as above. For each leaf and internal node, we then construct another one-dimensional range tree that contains all points with an x-coordinate in the interval associated to the node.

To do this efficiently, we first construct the x-coordinate range tree in time $O(n \log n)$ time as above. Then we compute a list L_2 of all points sorted by y-coordinate and assign it to the root of the x-coordinate range tree. Then we create such lists for the children of the root by sending a point in L_2 to the left child if its x-coordinate is in the interval of the left child and sending it to the right child else. We apply this step recursively and obtain at each node and leaf a list of all points in the associated x-interval. Clearly, every point is only contained in $\lceil \log n \rceil$ many lists, so this can be done in time $O(n \log n)$. Finally, we construct a range tree out of each list. Since the lists are already sorted, we can construct a balanced binary tree for one in linear time. In total, this construction takes $O(n \log n)$ time.

For a given rectangle, we consider the maximal nodes in the x-coordinate tree whose associated interval is completely contained in the x-interval of the rectangle. As there are only two of them in any level of the tree, there are at most $O(\log n)$ in total which can be found in time $O(\log^2 n)$. Then we start a y-interval lookup as above in each of the x-intervals. In total, a query thus takes $O(\log^2 n + k)$ time.

2.3 Finding all points in a half segment

Given a set of n points p in the plane, we want to find for any half segment $([x_1, x_2] \times [y_1, \infty))$, the set of points contained in that half segment. We can construct a *priority search tree* for this in time $O(n \log n)$ that supports queries returning k points in time $O(\log n + k)$.

First, we construct a range tree on the x-coordinates of the points (we keep duplicates sorted by the respective y-coordinate). Then we sort the points by y-coordinates non-ascending and insert them into the tree by placing them in the first empty internal node on the path to their x-coordinate. This can be done in time $O(n \log n)$.

We can perform a query for a given half segment, similar to how we would look its interval up in the range tree. However, we report the points along the way as we find them and stop descending when the y-coordinate of the stored point becomes too small.

2.4 Finding all intervals containing a point

Given a set of n intervals in the line, we want to find for any point the set of intervals that contain this point. We can construct an *interval tree* in time $O(n \log n)$ that supports queries returning k intervals in time $O(\log^2 n + k)$.

First, we create two lists of the intervals sorted by the left and right endpoint respectively. Then we create a balanced binary tree of the left endpoints where we store points in *both* the leaves and internal nodes. Recursively, we partition the intervals over the nodes of the tree. We start by assigning all intervals to the root. Then we assign a interval to the left child if it is completely to the left of the point at the current node, assign it to the right child if it is completely to the right of the point at the current node and else keep it at the node. In the end, we have at every node the intervals that contain the associated point of this node but no point associated to a parent. At every node, we keep the intervals in two lists, still sorted by left and right endpoint respectively. As any interval is contained in only one node, this takes $O(n \log n)$ time.

For a given query point, we check if the query point is equal to the root and return the list of intervals at the root in that case. Else we check if it is to the left or the right of the root. If it is to the left, we return all intervals stored at the root that contain this point (which we can find by binary search on the list sorted by left endpoints) and recurse into the left subtree. Else, we perform binary search on the list sorted by right endpoints and recurse into the right subtree. In total, we have $O(\log n)$ applications of binary search, or $O(\log^2 n + k)$ time in total.

2.5 Finding the y-smallest interval containing a point

Below we are not interested in finding all intervals, but rather we want to find the 'smallest' one. To do this, we will need a datastructure which we call a *priority interval tree* as we have not seen described before (although it likely exists already).

Given a finite set of n x-axis-parallel line segments in the plane, we want to find for any point q the line that is hit first when we shoot a y-axis parallel ray from q to the top. In other words, we look for the interval $I = [i_1, i_2] \times \{i_y\}$ such that $i_1 \leqslant q_x \leqslant i_2$, $i_y \geqslant q_y$ with i_y smallest among such intervals. Our datastructure can be constructed in time $O(n \log^2 n)$ and supports queries in time $O(\log^2 n)$.

We construct an interval tree as above, but instead of two lists, we store two priority search trees at each node, that contain the left/right endpoints of the intervals. At each node we have to solve the following problem:

Assume that we are given n points in the plane, each shooting an axis-parallel ray to the right. For a query point q, we want to find out which ray we hit first, if we shoot an axis-parallel ray out from the query point to the top. In other words, we are looking for the point p such that $p_x \leqslant q_x$, $p_y \geqslant q_y$ and p_y is smallest among all such points (or the message that no such point exists). This problem can be solved by a priority search tree, as we can simply search for the half segment $(-\infty, q_x] \times [q_y, \infty)$ and take the top-most node we encounter on the search path.

When constructing the priority search we also have to sort the points by y-coordinate. As a result the total time to construct a priority interval tree is $O(n \log^2 n)$. Otherwise the analysis remains the same. Later, in our algorithm, we will insert intervals dynamically. It is possible to support this in time $O(\log n)$ with no impact on query times [WO93], but one might chose instead to use a simpler approach based on binary structures and occasional rebuilding (see e.g. [Kle06, p. 110]).

2.6 Faster Algorithm

To implement GreedyPacking, we will reimplement steps one and two above using the advanced datastructures we just presented. For step one, we keep two priority interval trees: one for x-axis parallel lines and one for y-axis parallel lines. Whenever we place a rectangle, we insert its left and bottom border into the respective tree. When we want to find q, we can look up the point p in the trees and set q_x to the x returned by the y-axis parallel tree and q_y to the y returned by the x-axis parallel tree (or 1 if the lookup fails). To implement step two, we simply keep a two-dimensional range tree and look up all the points in the tile rectangles T from step one.

We can use the same algorithm for the TilePacking problem if we store the left and bottom border of the tile rectangle in the priority interval trees, rather than the left and bottom border of the rectangle itself. This implies that the tile rectangles never intersect each other and so the total cost of the range tree lookups is $O(n \log n)$.

3 The optimal algorithm

We have seen above that any optimal solution can be given by greedily selecting rectangles in a specific permutation. This immediately suggests an $O(n!\, n \log n)$ algorithm combining a search over all

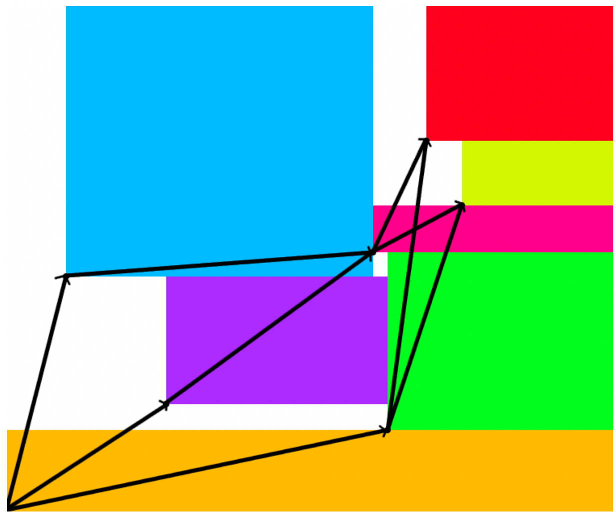

Figure 2: The Dominance DAG

$O(n!)$ permutations with the $O(n \log n)$ TilePacking algorithm above. However, there is a much better algorithm.

3.1 Dynamic Programming

To achieve a $O(2^n\, n \log n)$ algorithm, Jonathan Harder [Har19] proposed to adapt the Held-Karp algorithm that finds an optimal solution to the Traveling Salesman Problem by dynamic programming. Instead of considering all permutations, we find the best permutation for any of the $O(2^n)$ subsets of the points. Indeed, if $\{p_1, \ldots, p_k\}$ is a prefix of the optimal permutation, then it is also the optimal permutation for the instance consisting only of the points $p_1, \ldots, p_k$ (excluding the origin). Thus, we can find the optimal permutation for $\{p_1, \ldots, p_k\}$ by taking the best permutation arising from appending p_i to the best permutation of $\{p_1, \ldots, p_m\} \backslash \{p_i\}$ for some $p_i \in \{p_1, \ldots, p_m\}$.

Unfortunately, this algorithm is quite slow in practice. Next, we discuss some heuristics for speeding it up further.

3.2 The Dominance Directed Acyclic Graph

Recall that q *dominates* p, written $p \leqslant q$ if $p_x \leqslant q_x$ and $p_y \leqslant q_y$. Clearly, $p \leqslant q$ implies $p \preceq q$. A set P is *dominant* if for all $p \in P$ and $p \leqslant q$, we have that $q \in P$.

We want to consider the graph G on the points with an edge (q, p) if $p \leqslant q$ and there is no point s with $p \leqslant s \leqslant q$. For a dominant set, we call its neighbors in this graph the *frontier*. In particular, by the definition of $\preceq'$, for the optimal permutation $[p_1, \ldots, p_n]$, we will have that any $\{p_1, \ldots, p_i\}$ is dominant with p_{i+1} in its frontier. As such, we only have to consider dominant subsets in the dynamic programming algorithm. As Jonathan Harder points out, for instances where G consists of c paths the dynamic programming algorithm runs in time $O(n^c)$.

To construct this graph, we first build the graph $\overleftarrow{G}$ consisting of all points and an edge (p, q) if $p \leqslant q$. Then we consider for every edge (u, w), if there is a point v and edges (u, v) and (v, w). If so, we mark this edge for deletion (but do not delete it yet). Finally, we delete all marked edges and invert the graph. For n nodes and m edges, this procedure runs in time $O(nm)$. In theory, we can achieve a better bound: we can find out if there is such a point v in constant time if we know for any pair of nodes whether they are connected by a length-2-path. This can be computed by squaring the adjacency matrix, yielding an $O(n^{2.373})$ algorithm.

3.3 Implied rectangles

Assume that we have already placed the rectangles for $\{p_1, \ldots, p_k\}$. It may now occur that there is some point p_i that has no other point in its tile. Since we use the TilePacking algorithm to place the rectangles, we can not make the solution worse by simply placing this rectangle directly. We say that this rectangle is *implied* by the partial solution for $\{p_1, \ldots, p_k\}$. In particular, for $k = 0$ this argument implies that the roots of the dominance DAG can be placed greedily before we even start the dynamic programming algorithm: in figure 2 the yellow and red rectangle can be placed immediately (and we would obtain an equally good solution if the yellow one was placed before the red one). We have found this to be an extremely useful optimization in practice, as any placing of c rectangles implied by a partial solution avoids updating 2^{c-1} subsets.

3.4 Dijkstra's Algorithm

Often the dynamic programming approach will continue to update from subsets that have hopelessly wasteful optimal permutations. We can avoid this by using Dijkstra's algorithm on the space of subsets: We think of the subsets as nodes in a graph and any two nodes S_1, S_2 are connected by an edge if there is a point p such that $S_1 \cup \{p\} = S_2$. We define the cost of this edge to be the size of the white space (not covered by a rectangle) in the tile of p when we place it after placing the points of S_1. The dynamic programming approach above

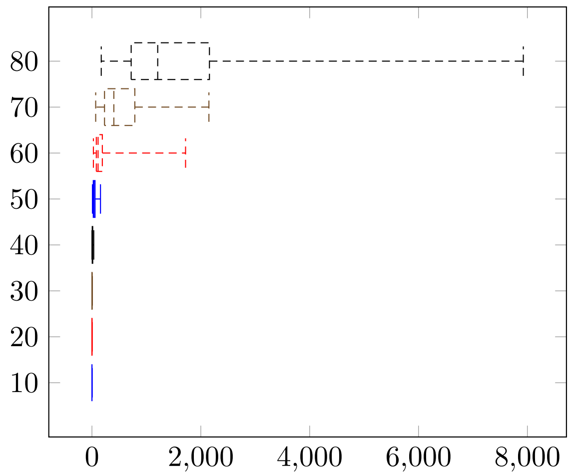

Figure 3: Time (ms) for the Optimal Algorithm on n uniformly distributed points; 100 instances

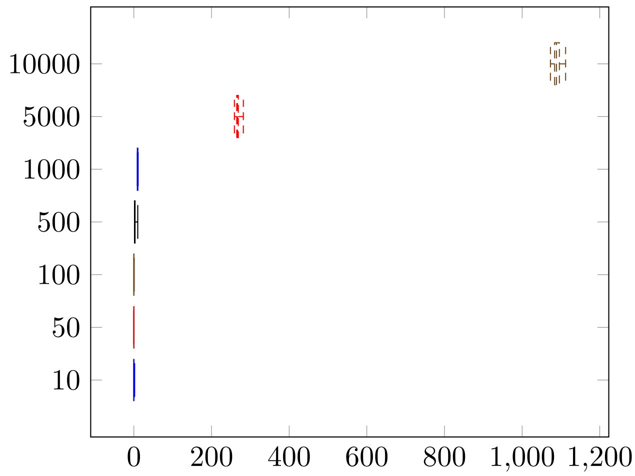

Figure 4: Time (ms) for the Greedy Algorithm on n uniformly distributed points; 100 instances

corresponds to a breadth-first search of this graph; but we can do better by using Dijkstra's algorithm instead. This yields an optimal permutation as maximising the coverage is the clearly same as minimizing the white space.

In other hard combinatorial optimization problems, Dijkstra's algorithm can restrict the search space significantly [HSV15]. In these cases, Dijkstra is usually sped up by changing the edge costs through a feasible potential that acts as a lower bound on future costs. This approach is very similar to the A* algorithm.

However, we observed only modest improvements — and due to the higher overhead, Dijkstra's algorithm can even be slower! We believe that a central reason for this is the lack of a good lower bound on the future edge costs. While Antoniadis et al. [Ant+19] give an algorithm that computes an upper bound on the coverage (and thus a lower bound on the white space), this algorithm is prohibitively slow. Instead we use a simple lower bound that simply measures for each point the white space in its smallest possible tile. This can be computed in time $O(n \log^2 n)$, but can only identify around a tenth of the final white space in random instances.

4 Experimental evaluation

We first evaluate the performance of the optimal algorithm. Here we see the full effect of the optimizations described in that chapter: The dynamic program on its own could only solve instances with 18 points in a second, but with all optimization together, it is often possible to solve even instances with 70 points! However, the optimizations of

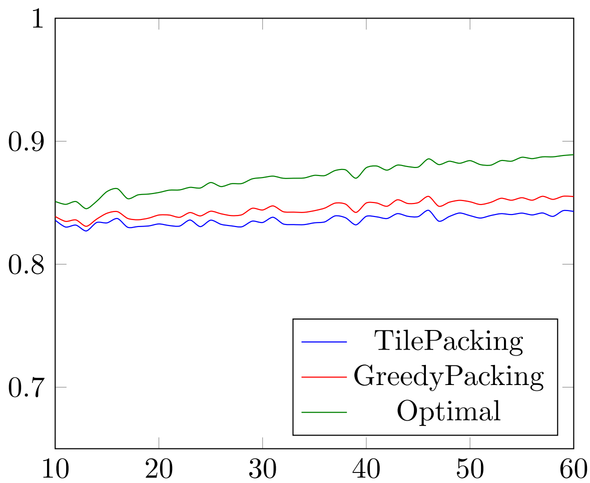

Figure 5: Average coverage of the algorithms on n uniformly random points; 100 instances

course do not change the theoretical runtime of the algorithm and as such we can see outliers that take much more time (see figure 3). Furthermore, the runtime still shows exponential characteristics and so it seems infeasible to solve significantly larger instances.

Naturally, the TilePacking and GreedyPacking algorithms run much faster. We have found that even the quadratic greedy algorithm described at the start of section 2 can solve instances with ten thousand points in about a second (see figure 4). As such, we did not implement the more complicated algorithm described in that section.

Unsurprisingly, the optimal algorithm performs much better on random data than the GreedyPacking and TilePacking algorithms (see figure 5). Even though GreedyPacking is no better in theory than TilePacking, it can still help in practice. The chart also shows greater coverage

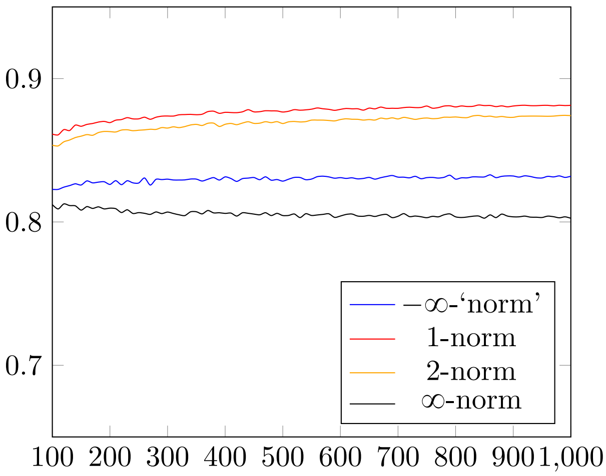

Figure 6: Average coverage of the greedy algorithm on n uniformly random points; 100 instances

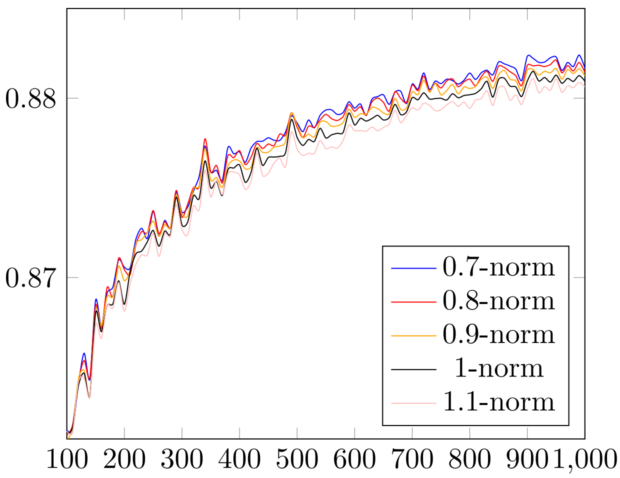

Figure 7: Average coverage of the greedy algorithm on n uniformly random points; 100 instances

for instances with more points (some fluctuations notwithstanding), however, note that this tells us more about the distribution of the points than about the algorithms themselves.

In figure 6 we take a look at the behaviour of different norms on random data for the greedy algorithm. It turns out that the Manhattan norm does best (by a significant factor) on average which justifies the interest in this norm. In figure 7 we present the results for p-norms with $p \leqslant 1$. While smaller p-values seem to be better, the difference is quite small (note the scaled y-axis).

While there is of course no instance where the TilePacking algorithm does better than GreedyPacking or the optimal one, the picture is more muddled for norms. In fact, for each pair of norms in figure 6 we can find an instance where one covers 10% more space than the other. In the `instances` folder, we have selected twelve such instances. As such, we believe that no norm is inherently better than any other when one wants to maximise the covered space.

What if one wants to find an algorithm that always covers half of the space? We can not answer this question with random data – while we have compiled a list of worst random instances in the `instances` folder they are not very bad. For the norms we obtain 'worst' instances on 20 points with a coverage around 63% and around 70% for the optimal algorithm. We believe that it would be interesting to generate the point sets by stochastic processes that introduce more structure than a uniform distribution.

5 User Interface

We have implemented a user interface for our algorithms. In a drawing area one can view instances; add a point by clicking at a free spot; move a point by dragging it or delete a point by a double-click. The menubar at the top has the following entries: A dropdown to select the desired algorithm; buttons to save and load the instances from files and to the left the coverage of the currently displayed instance.

References

[Ant+19] Antonios Antoniadis et al. "On the complexity of anchored rectangle packing". In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[Dam+21] Christoph Damerius et al. "On Greedily Packing Anchored Rectangles". In: *arXiv preprint arXiv:2102.08181* (2021).

[DT15] Adrian Dumitrescu and Csaba D Tóth. "Packing anchored rectangles". In: *Combinatorica* 35.1 (2015), pp. 39–61.

[Har19] Jonathan Gadea Harder. "Anchored Rectangle Cover". 2019.

[HSV15] Stefan Hougardy, Jannik Silvanus, and Jens Vygen. "Dijkstra meets Steiner: a fast exact goal-oriented Steiner tree algorithm". In: (2015).

[Kle06] Rolf Klein. *Algorithmische Geometrie: Grundlagen, Methoden, Anwendungen*. Springer-Verlag, 2006.

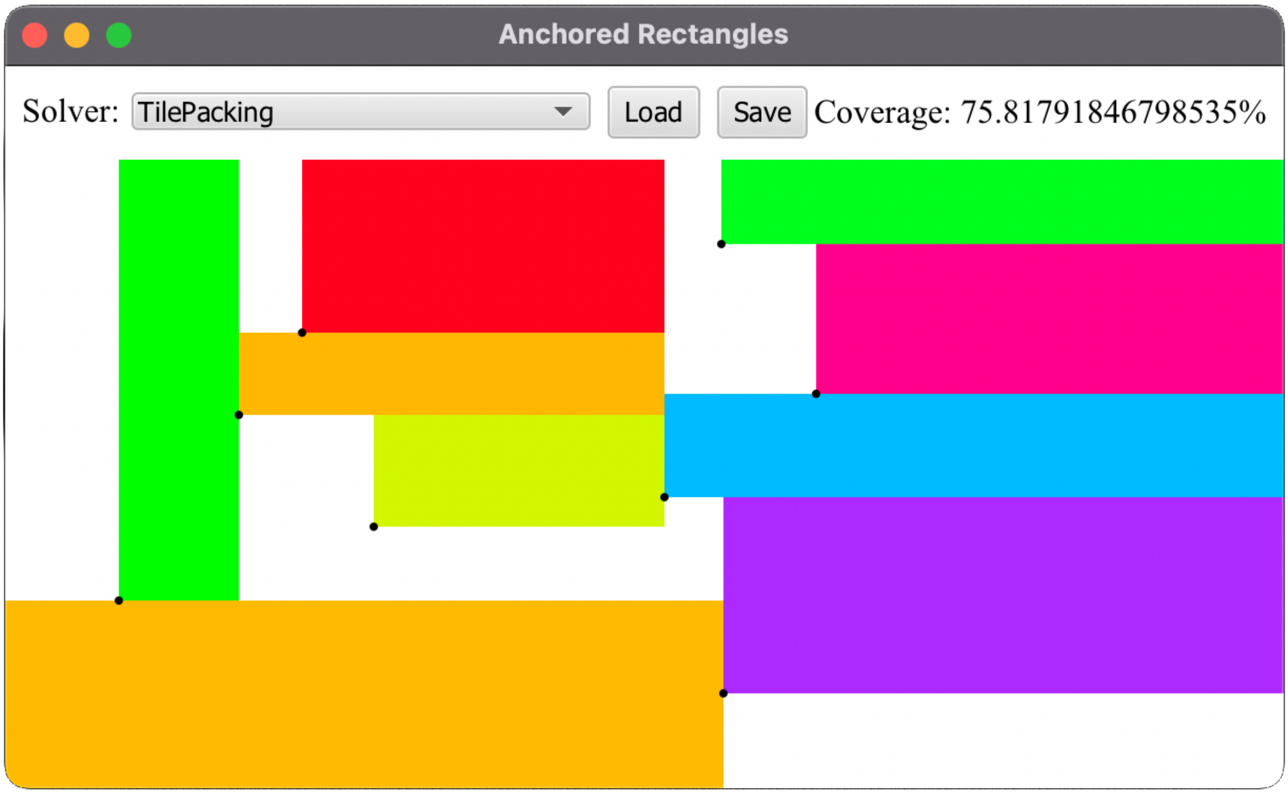

Figure 8: User interface

[Tut69] William Thomas Tutte. "Recent progress in combinatorics (Proceedings of the 3rd Waterloo Conference on Combinatorics)". In: (1969).

[WO93] Peter Widmayer and T Ottmann. *Algorithmen und Datenstrukturen*. Springer, 1993.