Bachelorarbeit

Covering Rectilinear Polygons with Few and Small Axis-Aligned Rectangles

verfasst von

Julian Unterweger

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2023

Studienkennzahl lt. Studienblatt:	UA 033 521
Fachrichtung:	Informatik - Informatik Allgemein
Betreuerin / Betreuer:	Ass.-Prof. Dr. Kathrin Hanauer, B.Sc. M.Sc.

Contents

1 Motivation **4**
1.1 Overview . . . 4

2 Preliminaries **5**
2.1 Problem Statement . . . 5

3 Related Work & Problems **8**
3.1 Rectilinear Picture Compression . . . 8
3.2 Rectilinear Polygon Partition . . . 9
3.3 Weighted Set Cover . . . 9
3.3.1 Weighted Geometric Set Cover . . . 10

4 Algorithms **11**
4.1 Integer Linear Program . . . 11
4.2 Greedy Weighted Set Cover . . . 12
4.2.1 Post-processing . . . 15
4.3 Partition . . . 17
4.3.1 Post-processing . . . 19
4.4 Greedy Strip Cover . . . 22
4.4.1 Post-processing . . . 23

5 Implementation **26**
5.1 Code . . . 26
5.1.1 Instance Conversion . . . 26
5.1.2 Algorithmic Framework . . . 26
5.1.3 Visualization . . . 27
5.2 Usage . . . 27

6 Evaluation **28**
6.1 Methods for Evaluation . . . 28
6.2 Results of Evaluation . . . 30
6.2.1 Result Quality . . . 31
6.2.2 Runtime Results . . . 34

7 Conclusion & Future Work **36**
7.1 Summary . . . 36
7.2 Limitations of the Approaches . . . 37
7.3 Outline of Future Work . . . 37
7.3.1 Hardness of the WRCP . . . 37
7.3.2 Approximability of the WRCP . . . 37
7.3.3 Integer Linear Program for RPC . . . 38

Abstract

Representations of polygons as simple shapes such as sets of rectangles are of interest in many areas. We present a problem that has, to our knowledge, not been studied before in which we seek to represent a polygon with only vertical and horizontal edges which may contain holes as a set of rectangles that are allowed to overlap while minimizing the cost of the rectangles, which is equal to $\alpha + \beta \cdot A(R)$ where $A(R)$ is the area of the rectangle and $\alpha, \beta \in \mathbb{N}_0$ are freely chosen, but the same for all rectangles. In this thesis, we define this special case of the weighted geometric set cover problem, discuss other closely related problems, give several heuristic algorithms as well as an integer linear program formulation for it and evaluate them on a large number of polygons, comparing both the cost of the solutions returned by the algorithms as well as their runtime performance.

1 Motivation

Representing a polygon using simpler shapes is a relevant problem in many fields, including integrated circuit design [12], image compression [18] and construction [24].

When using rectangles or similar simple shapes, the main goal is usually to minimize the number of shapes used to represent the polygon. However, in some applications, there may be a cost associated with both the number of shapes used as well as their area. If shapes may overlap, this can lead to a solution with more shapes costing less than a solution with fewer shapes in some cases.

One such example are some 2D video games in which sets of individual "tiles" may be more compactly represented as "objects". Each object requires a certain amount of time to initialize itself, after which each of its tiles requires a certain amount of time to be rendered. Depending on how long both of these actions take, it may be faster to have more objects, consisting of fewer total tiles, or fewer objects, consisting of more total tiles. Note that for the purpose of this comparison, if two or more tiles overlap, all of them are still rendered and thus contribute to the total required time.

This thesis aims to find and compare approaches specifically for the problem of representing a polygon that may contain holes and only has axis-aligned edges as a set of axis-aligned rectangles. The rectangles may overlap and can be freely chosen. The cost of this set of rectangles depends on the size of the individual rectangles as well as the size of the set. Our goal is to minimize the total cost while ensuring that the union of the rectangles in our set is identical to the original polygon within a reasonable time frame.

As far as we are aware, this is a problem that has not previously been studied. We will refer to it as the *Weighted Rectangle Cover Problem* (WRCP) throughout the rest of this thesis.

The goal of this work is to design, implement and evaluate algorithms in order to provide high-quality solutions for the WRCP within a reasonable time frame as well as to give an integer linear program formulation and implementation that can be used to compute optimal solutions given enough time.

To accomplish these goals, we adapt algorithms intended for various related problems and modify them via post-processing steps to provide improved solutions for our specific problem. These algorithms are then compared against each other on instances with different sizes and parameters, as well as against an optimal solution, when available, to determine their quality and execution time.

1.1 Overview

In section 2, we explain the notation and terminology used throughout the thesis. We then give an overview of problems that are similar to the WRCP, as well as known algorithms for these problems in section 3. In section 4 we discuss the algorithms we have designed as part of this thesis, their implementation is then briefly touched on in section 5, followed by the evaluation of the algorithms

and interpretation of the results in section 6. In section 7, we give a summary of our main results, the limitations of our approaches, as well as potentially interesting future work in this area.

2 Preliminaries

2.1 Problem Statement

In this section, notation and definitions will be given that will be used throughout the rest of this thesis. Parts of the notation and terminology are based on [13, 23, 22, 21].

In an instance of the WRCP, we are given a non-self-intersecting *polygon* P in the form of a list of edges, specifying the polygon's interior, as well as potentially one or more additional lists of edges, specifying any holes in the polygon's interior. We are also given two parameters, α and β with $\alpha, \beta \in \mathbb{N}_0$, which determine the cost of a rectangle when used in a solution to this problem instance. In an instance of the WRCP, the *cost* of a rectangle R is $c(R) = \alpha + \beta \cdot A(R)$ where $A(R)$ denotes the area of the rectangle.

Our goal is to represent P as a set of axis-aligned rectangles C, such that $\bigcup_{R \in C} R \equiv P$, while minimizing $\theta(C) = \sum_{R \in C} c(R)$. A set of rectangles C that fulfills the condition that the union of its elements is equivalent to P is called a *cover*. Note that the rectangles in a cover are allowed to overlap; If they do not overlap, the cover is also a *partition*. Note that all partitions are also covers, but the opposite is not true.

An *edge* of a polygon is a segment connecting two endpoints, which are called *vertices*. All edges share each of their two endpoints with another edge. In the WRCP, all edges are parallel to either the X or Y axis and all vertices are two dimensional and have integer coordinates. A polygon with such edges is called *rectilinear* or *orthogonal*. Since we are only concerned with rectilinear polygons, we will use the term "polygon" to mean "rectilinear polygon" throughout the rest of this thesis.

By *non-self-intersecting*, we mean that none of the polygon's edges intersect, except at their endpoints and every vertex is an endpoint of exactly two edges.

For a rectangle R, which is also a polygon, we will represent it as a tuple of its top-left and bottom-right vertices, meaning that $R = ((x_l, y_t), (x_r, y_b))$ with $x_l < x_r$, $y_b < y_t$ and $x_l, x_r, y_b, y_t \in \mathbb{Z}$. We will also use $\text{tl}(R) = (x_l, y_t)$ and $\text{br}(R) = (x_r, y_b)$ as well as $\text{max_x}(R) = x_r$, $\text{max_y}(R) = y_t$, $\text{min_x}(R) = x_l$ and $\text{min_y}(R) = y_b$.

Note that a rectilinear polygon always has the same number of horizontal and vertical edges, since the direction of two edges that share an endpoint must be different.

A vertex of a polygon is called *concave* if the interior angle between its two edges is 270° or *convex* otherwise.

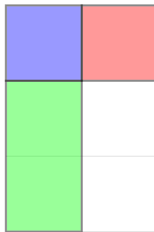


Figure 1: A set of rectangles D with $|D| = 3$.

We will use $E(P)$ and $V(P)$ to refer to the set of edges and vertices of P respectively. Note that when we say $E(P)$, we mean *all* edges of P, no matter if they specify the interior of P or form a hole. The same is true for $V(P)$.

The *bounding box* of a polygon P, which we will refer to using $Z(P)$ is defined as the smallest rectangle R that fully contains P, meaning that $R \cap P \equiv P$ and the area of R is minimal.

A *pixel* $\pi = ((x_l, y_t), (x_r, y_b))$ with $x_r - x_l = 1$ and $y_t - y_b = 1$ of a polygon P is a unit square in the interior of P. We will use $\Pi(P)$ to refer to the set of all pixels that lie inside P.

If we draw one horizontal and one vertical line from each concave vertex of the polygon into its interior until it intersects with an edge from $E(P)$, we will be left with a set of rectangles lying in P's interior. We will denote this set as $B(P)$ and call its elements the *base rectangles* of P.

Lemma 1. *The total number of rectangles in* $B(P)$ *is* $O(v^2)$ *where* v *is the number of vertical edges of* P*.*

Proof. If we extend every vertical and horizontal edge of the polygon in both directions until its length becomes infinite, we are left with a set of rectangular regions bounded by these lines lying inside the bounding box of P. Since a rectilinear polygon has the same number of vertical and horizontal edges, there are v vertical and v horizontal lines bounding $(v-1)(v-1)$ rectangular regions inside the polygon's bounding box, which is $O(v^2)$. Since the polygon consists of some subset of these rectangular regions, it is indeed an upper bound. □

When given a set of rectangles D, we will let $\Gamma(D)$ denote the set of all rectangles whose representation as pixels is equal to a union of subsets of D, meaning a rectangle R belongs to $\Gamma(D)$ if and only if $\exists A \subseteq D$ such that $\Pi(R) \equiv \bigcup_{R' \in A} \Pi(R')$, meaning that $\Gamma(D)$ contains all rectangles which can be created by combining rectangles from D. Figure 1 and Figure 2 show a simple D and $\Gamma(D)$.

Lemma 2. *The total number of rectangles in* $\Gamma(\Pi(P))$ *is* $O(w^2h^2)$*, where* w *and* h *are the width and height of the bounding box of the input polygon respectively.*

Proof. Consider $\Pi(Z(P))$, the set of pixels contained in the polygon's bounding box. Since the bounding box has width w and height h, there are h rows and w

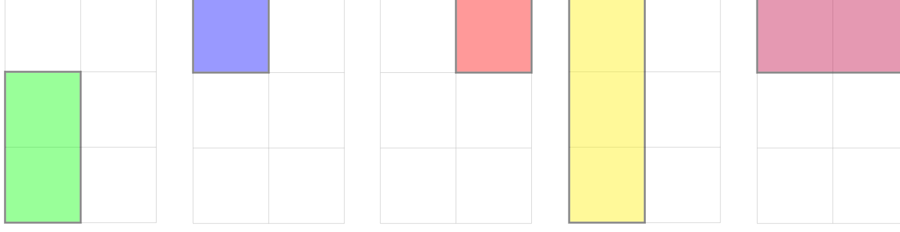


Figure 2: The five rectangles in $\Gamma(D)$ for the D from Figure 1.

columns of pixels in $\Pi(Z(P))$. Let p be a pixel in the ith row and jth column, there are exactly ij pixels in $\Pi(Z(P))$ with row index $\leq i$ and column index $\leq j$. Each of these pixels is the upper left-hand pixel in some unique rectangle belonging to $\Gamma(\Pi(Z(P)))$, with p being the lower right-hand pixel, meaning there are exactly ij rectangles for every pixel in $\Pi(Z(P))$, which gives us a total number of

$$\begin{aligned}
&\sum_{i=1}^{w}\sum_{j=1}^{h} ij \\
=&(\sum_{i=1}^{w} i)(\sum_{j=1}^{h} j) \\
=&\frac{w^2+w}{2}\frac{h^2+h}{2} \\
=&\frac{w^2h^2+w^2h+wh^2+wh}{4} \\
=&O(w^2h^2)
\end{aligned}$$

rectangles when considering all pixels in $\Pi(Z(P))$.

This is an upper bound since the polygon P is made up of a subset of pixels from $\Pi(Z(P))$. □

Lemma 3. *The total number of rectangles in $\Gamma(B(P))$ is $O(v^4)$ where v is the number of vertical edges of the polygon.*

Proof. In the proof of Lemma 1, when we extended the edges of the polygon to infinite length, we saw that the bounded rectangular regions inside the polygon's bounding box formed a grid with v rows and v columns. Combining this with the proof of Lemma 2, we can see that there must be $O(v^2v^2) = O(v^4)$ rectangles in $\Gamma(B(P))$. □

Note that in the worst case, we have $|\Pi(P)| = |B(P)|$ and thus $|\Gamma(\Pi(P))| = |\Gamma(B(P))|$, an example of a polygon for which this is the case is shown in Figure 3.

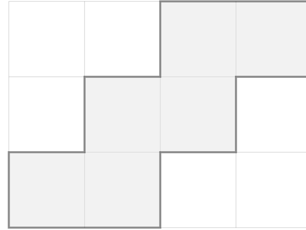


Figure 3: A polygon P for which $\Pi(P) = B(P)$.

3 Related Work & Problems

This section outlines various problems related to the WRCP and discusses different approaches taken to solve them.

3.1 Rectilinear Picture Compression

In the Rectilinear Picture Compression problem (RPC), we are given a binary matrix and must represent it as a minimum set of rectangular submatrices containing only entries which are 1, such that every entry which is 1 is contained in at least one of our submatrices. [18]

This is almost the same problem as the WRCP with $\alpha = 1$ and $\beta = 0$, since the cost function in this case becomes just $c(R) = 1$, which means we are just minimizing the number of used rectangles without considering their areas. In this case, the only difference between the two, as we have defined them, is the format we receive our input in.

This problem has received considerable attention in different fields over the years and was proven to be $\mathcal{NP}$-hard by Masek [20][1].

A simple primal and dual integer linear programming formulation for this problem is given by Heinrich-Litan and Lübbecke [13], some alternative formulations were attempted by Koch and Marenco [18].

The primal formulation given in [13] defines one binary variable per maximal rectangle that can fit inside the polygon and one constraint for each pixel of the polygon. The constraints dictate that every pixel must be covered by at least one rectangle which contains it. This ensures that a valid cover is computed. The objective function which is minimized is the sum of all binary variables, i.e. the number of rectangles used.

The polynomial-time algorithm with the best currently known approximation ratio of $O(\sqrt{\log v})$, where v is the number of vertical or, equivalently, horizontal edges of the polygon, is given by Kumar and Ramesh [3]. This algorithm covers the polygon by looking for *strips*, which are essentially 1-pixel-wide vertical rectangles that cannot be extended further up or down since they would otherwise extend outside the polygon. These rectangles are then extended left

[1] Original source unavailable, but quoted in many sources, including [13]

and right as far as possible. Since two different strips could result in the same extended rectangle, only distinct extended rectangles are kept in the final cover.

Another polynomial-time algorithm shown to give good results in practice is also given by Heinrich-Litan and Lübbecke [13]. This algorithm is based on the greedy set cover algorithm, described in subsection 3.3, but extends it by picking certain rectangles which are guaranteed to be part of some optimal cover before invoking the greedy set cover algorithm to cover the remaining parts of the polygon, pruning any redundant rectangles in a post-processing step. This algorithm is based on an earlier one [11], which has worst-case runtime complexity $O(n^5)$ where n is the the larger of the height and width of the input polygon's bounding box.

Fast approaches for large instances with potentially improved quality are discussed by Koch and Marenco [18]. Their approaches work by finding an initial cover for the polygon by looking for large rectangles, grouping the pixels of the polygon into so-called atomic rectangles using this initial cover and then choosing a subset of the initial cover to cover the obtained atomic rectangles.

3.2 Rectilinear Polygon Partition

Finding a partition (also called decomposition or dissection) of a rectilinear polygon into non-overlapping rectangles is, in contrast to the overlapping case, a problem that is optimally solvable in polynomial time even if the polygon contains holes [21].

The first algorithm, given by Ohtsuki [21], had runtime complexity $O(n^{2.5})$, which was later improved to $O(n^{1.5} \log n)$ [14] with n being the number of vertices of the rectilinear polygon. The algorithm finds an optimal partition by first locating *degenerate diagonals* between concave vertices in the polygon, which are the edges between two concave vertices that can be connected by a straight horizontal or vertical segment that does not leave the polygon's interior and does not intersect any edges of the polygon in more than a single point. The intersections between these vertical and horizontal segments can be represented as a bipartite graph, where a vertical and horizontal node have an edge between them if the segments they represent intersect. Our goal is to pick as many non-intersecting diagonals from the set of intersecting diagonals as possible. This is the same as finding a maximum independent set of the bipartite graph, which can be accomplished in polynomial time. After this, we pick the remaining set of degenerate diagonals which do not intersect any others, as well as an arbitrary vertical or horizontal cut from each remaining concave vertex into the polygon's interior until it hits either an edge of the polygon or one of our picked degenerate diagonals. The rectangles that exist between these picked diagonals and cuts at the end of this process correspond to an optimal partition of the polygon. [21]

3.3 Weighted Set Cover

In the weighted set cover problem, we are given subsets $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ of a universe U, each of which has an associated weight given by a function

$w(S_j)$. The goal is to pick subsets such that their union equals the universe while minimizing the total weight of the chosen subsets. [26]

The unweighted set cover problem is $\mathcal{NP}$-complete [16], making the weighted version $\mathcal{NP}$-hard, since it is equivalent to the unweighted case when all weights are set to the same constant and thus at least as hard as the unweighted problem.

Simple greedy approaches exist both for the unweighted case [19, 15] and for the weighted case [6] which yield an $O(\log n)$ approximation factor, where n is the number of elements in the universe.

If we let L be the set of elements that are already contained in some picked set, the greedy algorithm picks the set $S_j \in \mathcal{S}$ which maximizes $\frac{|S_j - L|}{w(S_j)}$. Then, every element of the picked set is added to L. These steps are repeated until $L = U$. [6]

Interestingly, it is proven by Feige [9] that $O(\log n)$ is the best approximation that can be provided in polynomial time by any algorithm for this problem in the general case unless $\mathcal{P} = \mathcal{NP}$.

The weighted set cover is a generalization of the WRCP. To formulate the WRCP as an instance of the weighted set cover, let the universe U be equal to the set of the input polygon's pixels P and let the subsets $\mathcal{S}$ of U be equal to the set of all possible rectangular subsets of P, R_p. The cost function of the WRCP, $c(R) = \alpha + \beta \cdot |\Pi(R)|$ can be used analogously as a weight function for the weighted set cover problem.

The greedy set cover algorithm is used by Heinrich-Litan and Lübbecke [13] as a subroutine of an algorithm specifically devised for the rectangle cover problem, which was already discussed in subsection 3.1.

3.3.1 Weighted Geometric Set Cover

The weighted geometric set cover is a family of special cases of the general weighted set cover, where the subsets of the universe are geometric objects [25]. The WRCP is part of this family as well.

Even *et al.* give a randomized algorithm for a closely related problem to the weighted geometric set cover with an approximation factor of $O(\log \mathrm{OPT})$ for general geometric objects with "low VC-dimension", where OPT is the weight of an optimal solution [8]. Unfortunately, this result only holds when weights are larger than 1 and, in addition, OPT could be arbitrarily larger than the size of the universe, which would make $O(\log n)$ the superior bound in those cases [1]. Their algorithm is based on solving the relaxed linear program of the problem instance and then rounding it by probabilistically finding an ϵ-net of small size.

Varadarajan gives a similar probabilistic algorithm which uses a "quasi-uniform" sampling of ϵ-nets to round the linear program solution, which can yield improved results when the *union complexity* of the geometric objects is "near-linear" [25]. The union complexity, which is a measure of the complexity of the boundary of a union of geometric objects, of n rectangles is $O(n^2)$ [17], thus Varadarajan's algorithm [25] gives $2^{O(\log^* r)} \log r$ approximation or better

in the WRCP, where r is the number of all possible rectangles which fit inside the polygon.

There are also interesting recent results in the unweighted case by Agarwal and Pan [2] as well as Bus *et al.* [5] based on the works of Brönnimann and Goodrich [4]. These are likewise centered around the computation of ϵ-nets and largely theoretical, though efficient computation of ϵ-nets for disks has been implemented [5].

4 Algorithms

In this section, the different algorithms that were implemented and evaluated for the WRCP as part of this thesis will be discussed.

4.1 Integer Linear Program

Our first algorithm is an integer linear program formulation of the WRCP, which can be used to compute optimal solutions given enough time. Our formulation is based on the primal one used for RPC [13], which we have previously described in subsection 3.1. A crucial difference between their formulation and ours is that we cannot restrict ourselves to only maximal rectangles, as is the case in RPC. This naturally leads to the question of which rectangles we *do* have to consider.

It would suffice to consider all rectangles in $\Gamma(\Pi(P))$, since this is the set of all rectangles that can fit inside P. This is not ideal, since simply scaling up the polygon will result in $\Gamma(\Pi(P))$ growing, because the number of rectangles in $\Gamma(\Pi(P))$ is $O(w^2h^2)$, due to Lemma 2, which scales with the width and height of the polygon's bounding box.

It seems intuitive that we should be able to scale up the polygon without having to increase the number of rectangles that we have to consider. For this purpose, we can use $\Gamma(B(P))$ instead, which has size $O(v^4)$, where $v = \frac{E(P)}{2}$. This is preferable, as it means the complexity of the problem does not grow unless the number of edges of the polygon grows.

Conjecture 1. *For every polygon P with parameters α and β there exists an optimal cover $C \subseteq \Gamma(B(P))$.*

We were unable to prove Conjecture 1 formally, but have compared a formulation of the integer linear program using rectangles from $\Gamma(\Pi(P))$ to our proposed formulation using only rectangles from $\Gamma(B(P))$ on 168 small-size instances and in all cases the two formulations returned covers with equal costs. Furthermore, the formulation using only rectangles from $\Gamma(B(P))$ was never outperformed by one of the heuristic algorithms across all experiments for which it returned a solution within the allotted time frame. This suggests that either our conjecture is true or, alternatively, cases where our formulation does not return an optimal cover are at least rare enough in practice to still make it a useful tool to consider.

The rectangles in $\Gamma(B(P))$ can be enumerated using the procedure shown in Algorithm 1. On a high level, this procedure iterates through all base rectangles and finds all rectangles which are contained in P's interior and have that base rectangle in their top left corner and any other base rectangle in their bottom right corner.

Lemma 4. *Algorithm 1 runs in $O(v^4)$ time, where v is the number of vertical edges of the polygon.*

Proof. We know from the proof of Lemma 1 that the number of base rectangles is $O(v^2)$. The outer for-loop iterates through all base rectangles, so it will iterate at most $O(v^2)$ times. The first nested while-loop iterates from the current top base rectangle rightwards until it hits a hole or the boundary, these are less than v iterations, since our grid has width $v - 1$, the same argument holds in the downward direction for the second nested while loop. □

After we enumerate $\Gamma(B(P))$ using the algorithm above, our integer linear program looks as follows, based on [13]:

$$\theta = \min \sum_{R \in \Gamma(B(P))} x_R \cdot c(R) \tag{1}$$

$$\text{s. t.} \sum_{R \in \Gamma(B(P)): R \ni b} x_R \geq 1 \qquad b \in B(P) \tag{2}$$

$$x_R \in \{0, 1\} \qquad R \in \Gamma(B(P)) \tag{3}$$

The constraints in (2) ensure that every base rectangle is covered by at least one rectangle, while the objective function (1) ensures that the total cost of the picked rectangles is as small as possible.

With this formulation, we need one constraint per base rectangle in $B(P)$ and one binary variable x_R per rectangle $R \in \Gamma(B(P))$. As we saw earlier, the number of base rectangles is $O(v^2)$ and the number of rectangles in $\Gamma(B(P))$ is $O(v^4)$, so we have $O(v^2)$ constraints and $O(v^4)$ binary variables in total in our formulation.

4.2 Greedy Weighted Set Cover

Another algorithm we have implemented and evaluated is the greedy weighted set cover algorithm, which we have already briefly discussed in more general form in subsection 3.3.

Since the greedy set cover algorithm works by selecting the most cost-effective set from the available sets until every element in the universe is covered [6], we have to supply it with a list of rectangles to choose from, just like for the integer linear program. We choose to use $\Gamma(B(P))$ over $\Gamma(\Pi(P))$ here as well due to the same benefits mentioned in the previous section.

Our version of the greedy weighted set cover algorithm for the WRCP is shown in Algorithm 2 and is based on [6].

Algorithm 1 EnumerateBase

Require:
$B = B(P)$

```
1: function EnumerateBase(B)
2:     R ← ∅
3: ▷ For every base rectangle find all rectangles with it in their top left corner
4:     for b ∈ B do
5:         x_l ← min_x(b)                          ▷ X-coordinate of left side of b
6:         y_t ← max_y(b)                           ▷ Y-coordinate of top side of b
7:         t ← b                                  ▷ Current top right base rectangle
8:         l ← −1          ▷ How many times we moved down for the previous t
9:         while true do
10:            x_r ← max_x(t)                      ▷ X-coordinate of right side of t
11:            i ← t                            ▷ Current bottom right base rectangle
12:            l_t ← 0                                            ▷ Temporary l
13:            while true do
14:                y_b ← min_y(i)                ▷ Y-coordinate of bottom side of i
15:                R ← R ∪ {((x_l, y_t), (x_r, y_b))}
16:                if l_t = l then
17:                    break               ▷ Cannot move down one row, try new t
18:                end if
19:                if i has a bottom neighbor then        ▷ Base rectangle below i
20:                    i ← i's bottom neighbor                 ▷ Move down one row
21:                    l_t ← l_t + 1
22:                else                                ▷ Hole or exterior below i
23:                    l ← l_t  ▷ Cannot move down more than l_t times for b now
24:                    break                                       ▷ Try new t
25:                end if
26:            end while
27:            if t has a right neighbor then          ▷ Base rectangle to right of t
28:                t ← t's right neighbor                  ▷ Move right one column
29:            else                               ▷ Hole or exterior to right of t
30:                break                                           ▷ Try new b
31:            end if
32:        end while
33:    end for
34:    return R
35: end function
```

Algorithm 2 GreedyWeightedSetCover

Require:
$B = B(P)$
$G = \Gamma(B(P))$

```
1:  function GreedyWeightedSetCover(B, G)
2:      C ← ∅                                         ▷ Initialize empty cover
3:      for R ∈ G do
4:          a(R) ← |Π(R)|                      ▷ a(R) gives "effective area" of R
5:      end for
6:      while B ≠ ∅ do           ▷ While there exists an uncovered base rectangle
7:          R* ← argmin c(R)/a(R) where R ∈ G   ▷ Get most cost-effective rectangle
8:          C ← C ∪ {R*}                                   ▷ Add it to the cover
9:          G ← G − {R*}                     ▷ Remove it from future consideration
10:         L ← ∅                          ▷ Initialize set of base rectangles in R*
11:         for b ∈ B do
12:             if Π(b) ∩ Π(R*) ≠ ∅ then
13:                 L ← L ∪ {b}                        ▷ R* contains b, add it to L
14:                 B ← B − {b}                           ▷ b is now covered by R*
15:             end if
16:         end for
17:         for R ∈ G do      ▷ For each remaining rectangle under consideration
18:             for l ∈ L do                    ▷ For each base rectangle from R*
19:                 if Π(l) ∩ Π(R) ≠ ∅ then                      ▷ if R contains l
20:                     a(R) ← a(R) − |Π(l)|          ▷ decrease R's effective area
21:                 end if
22:             end for
23:             if a(R) = 0 then                   ▷ If R has no effective area left
24:                 G ← G − {R}                                 ▷ remove it from G
25:             end if
26:         end for
27:     end while
28:     return C
29: end function
```

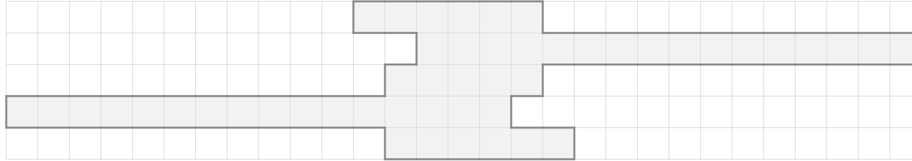


Figure 4: A polygon for which the greedy cover contains a fully redundant rectangle when $\alpha = 13$ and $\beta = 1$.

Lemma 5. *Algorithm 2 runs in $O(|B(P)|^4)$ time.*

Proof. The outer loop runs at most $|B(P)|$ times, the first inner for-loop is dominated by the second one, since $B \subseteq G$ at all times. The second inner for-loop runs at most $|G||L|$ times, where $|G| = O(|B(P)|^2)$ and $|L| = O(|B(P)|)$, so in total we have $O(|B(P)|^4)$ iterations. □

The time complexity of this algorithm could likely be improved to $|B(P)|^3$, but this would involve storing pointers for each base rectangle in $B(P)$ to the rectangles in $\Gamma(B(P))$ that it is contained in, which would drastically increase the memory usage of the algorithm on large instances, which our version of the algorithm avoids.

4.2.1 Post-processing

To improve solutions given by the above greedy weighted set cover algorithm, we use a post-processing step to first remove rectangles that are fully redundant from C and then trim redundant rows and columns of pixels from the remaining rectangles.

By *redundant* rows and columns of pixels of a rectangle $R \in C$, we mean a set $S \subseteq \Pi(R)$ with $|S|$ as large as possible such that $S \subseteq \bigcup_{R' \in C - \{R\}} \Pi(R')$ and $\exists R^* \in \Gamma(B(P))$ such that $\Pi(R^*) \equiv \Pi(R) - S$. Meaning that R can be replaced by R^* without invalidating the cover, since all pixels in S are already covered by some subset of $C - \{R\}$, giving $C^* = (C - \{R\}) \cup \{R^*\}$ where $\Pi(R^*) \subset \Pi(R)$ which gives $\theta(C^*) = \theta(C) - |\Pi(R) - \Pi(R^*)|$, improving the cost of the cover.

By *fully redundant* rectangle, we mean a rectangle $R \in C$ for which all of its rows or, equivalently, columns are redundant, meaning that for the set S of redundant pixels of R we have $S \equiv \Pi(R)$. Such rectangles can be removed from the cover completely without invalidating it, giving $C^* = C - \{R\}$ improving the cost of the cover again.

It may seem somewhat counter-intuitive that the greedy algorithm could return a cover containing a completely redundant rectangle and this indeed does seem to be rare in practice, occurring less than ten times across our thousands of performed experiments.

An example input for which a fully redundant rectangle is picked by the greedy algorithm is shown in Figure 4 and Figure 5.

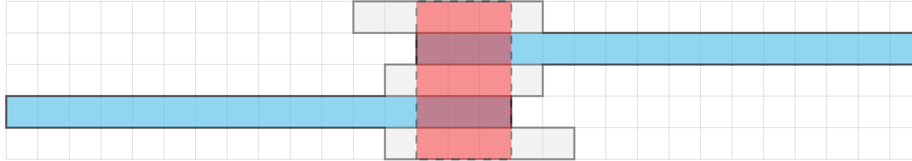


Figure 5: The greedy cover for the polygon and costs from Figure 4 after the first three iterations. The red rectangle will eventually become fully redundant, as the top, middle and bottom rows will be covered by fully extended rectangles in the last three iterations.

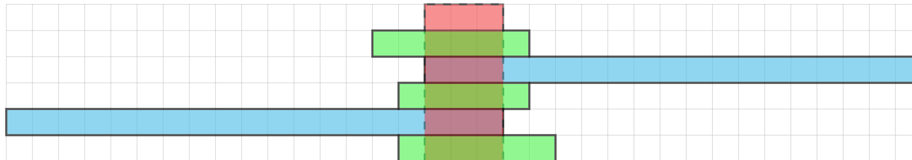


Figure 6: A polygon with $\alpha > 12$ and $\beta = 1$ for which trimming the rectangles in the greedy set cover in the opposite order they were added to the cover gives a greater cost reduction than trimming them in the same order they were added in. If we trim the blue rectangles first, which were picked before the red rectangle, we will reduce the cost by 9, but trimming the red rectangle would have reduced the cost by 15.

Lemma 6. *Removal of fully redundant rectangles from C can be accomplished in $O(|C||B(P)|)$ time.*

Proof. We can iterate through all $R \in C$ and take note of the base rectangles that it covers, giving us a function $T(A)$ telling us how many rectangles in C cover the base rectangle A. This can be done in $O(|C||B(P)|)$ time. Afterward, we can iterate all $R \in C$ a second time and check if $T(A) > 1$ for every base rectangle A which is contained in R. If that is the case, R does not cover any base rectangles uniquely and we can remove it completely, reducing $T(A)$ by 1 for every base rectangle A which it contained. Since we have two loops each running in $O(|C||B(P)|)$ time which are not nested we take $O(|C||B(P)|)$ time overall. □

For the trimming of redundant rows and columns, it is notable that the order in which we trim the rectangles in C can make a difference in the final output. Figure 6 and Figure 7 show two such situations. As is shown by these two examples, the order we should trim rectangles in the cover in to maximally reduce their overall cost depends on the input and it is not immediately obvious how or if an ideal order can be computed or chosen in an obvious way.

Lemma 7. *Removal of redundant rows and columns from rectangles in C can be accomplished in $O(|C||B(P)|)$ time.*

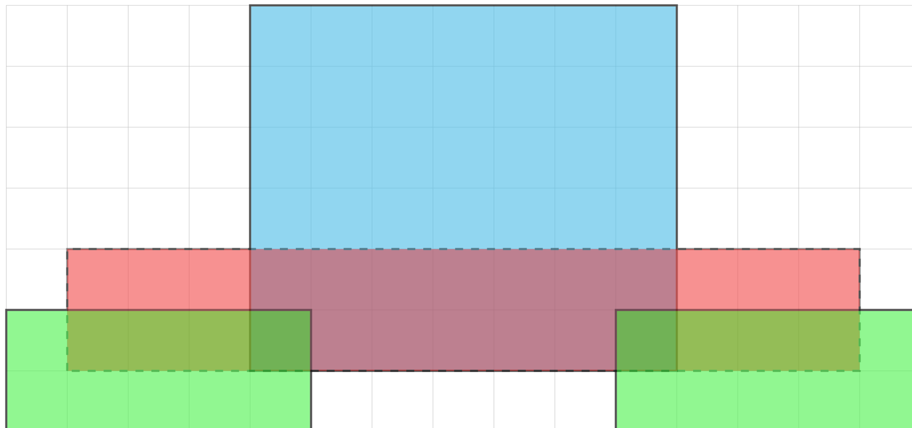


Figure 7: A polygon with $\alpha > 58$ and $\beta = 1$ for which trimming the rectangles in the greedy set cover in the order they were added to the cover gives a greater cost reduction than trimming them in the opposite order they were added in. If we trim the red rectangle first, which was picked after the blue rectangle, we will reduce the cost by 13, but trimming the blue rectangle would have reduced the cost by 14.

Proof. Construct $T(A)$ as in the proof for Lemma 6. Afterward, iterate all $R \in C$ a second time and construct the bounding box z of the base rectangles $\{A_1, A_2, \ldots, A_n\}$ of P which are contained in R and have $T(A_x) = 1$, meaning z is the bounding box of the base rectangles which R covers that are not covered by any other rectangle in C. If z is not equal to R, z must be smaller than R and we can replace R with z in our cover while still keeping every base rectangle in R covered at least once. Afterward, we have to decrement $T(A)$ by 1 for each base rectangle which was covered by R but is no longer covered by z. The time taken by this process is the same as in the proof for Lemma 6, which is $O(|C||B(P)|)$. □

4.3 Partition

We have already described Ohtsuki's polynomial-time algorithm [21], which optimally partitions a polygon into rectangles, in subsection 3.2.

In our experiments, we use both an unaltered version of Ohtsuki's algorithm as well as a version with an additional post-processing step after the initial partition is computed, which potentially turns the partition into a cover, reducing the number of rectangles.

It is worth noting that according to Franzblau and Kleitman [10], the size of a partition of P into rectangles can at worst be "twice the size of the minimum cover plus the number of holes". As we will see in our experiments, this seems to occur rarely in practice and the partition even appears to often give

superior results on small and medium size instances when compared to heuristic algorithms specifically designed to compute a cover.

There also exist certain types of polygons for which an optimal partition is also an optimal cover for the WRCP, no matter which α and β are used.

Lemma 8. *For a polygon P with arbitrary costs α and β, if there exists an optimal partition D of P and an optimal cover C of P with $\alpha = 1$ and $\beta = 0$ such that $|D| = |C|$ then D is also an optimal cover for any $\alpha, \beta \in \mathbb{N}_0$ for P.*

Proof. Since we computed C with $\alpha = 1$ and $\beta = 0$, C covers P using as few rectangles as possible, since it is optimal by definition. D contains the minimum number of rectangles needed to cover P since $|D| = |C|$ and since D is a partition, we know that $\sum_{R \in D} |\Pi(R)| = |\Pi(P)|$, meaning that the sum of the area of the rectangles in D is equal to the area of the polygon itself. For the purpose of contradiction, assume that there exists some C' for some arbitrary α and β such that $\theta(C') < \theta(D)$, meaning C' costs less than D for these parameters. We have $\theta(C') = \sum_{R \in C'} \alpha + \beta|\Pi(R)|$ and $\theta(D) = \sum_{R \in D} \alpha + \beta|\Pi(R)|$, but since D is a partition, we know that the area of all rectangles in it combined is the area of the polygon itself since a partition does not overlap, thus we have $\theta(D) = \alpha|D| + \beta|\Pi(P)|$.

$$\sum_{R \in C'} \alpha + \beta|\Pi(R)| < \alpha|D| + \beta|\Pi(P)|$$
$$\alpha|C'| + \sum_{R \in C'} \beta|\Pi(R)| < \alpha|D| + \beta|\Pi(P)|$$

We know that we have $|D| = |C|$, so $\alpha|C'| \geq \alpha|D|$ or C' would cost less than C when $\alpha = 1$ and $\beta = 0$, a contradiction since C is optimal by definition. That leaves us with

$$\sum_{R \in C'} \beta|\Pi(R)| < \beta|\Pi(P)|$$
$$\sum_{R \in C'} |\Pi(R)| < |\Pi(P)|$$

which means the sum of areas of rectangles in C' must be smaller than the area of P, but this would mean that there is some part of P which is not covered by C', making it an invalid cover.

Since our assumption that C' costs less than D is contradictory for arbitrary α and β, it follows that D must be optimal for all α and β. □

If we were able to tell easily whether a given polygon is of this type, we would be able to provide provably optimal covers for such polygons in polynomial time, though it appears unclear what the exact conditions under which this is the case are, as well as whether it is possible to efficiently verify them.

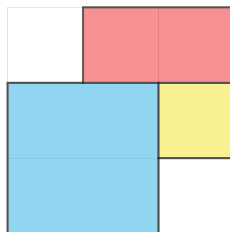


Figure 8: A partition for which there are two rectangles (the red and the yellow one) that are not vertically or horizontally aligned which could be joined into a larger rectangle.

4.3.1 Post-processing

We use two different post-processing methods after we compute the partition. They both potentially give improved solutions, the first is faster than the second, but may miss some possible improvements which the second method considers.

Aligned joins In the first method, we post-process the partition returned for a polygon P by finding vertically and horizontally aligned rectangles and joining pairs of them into one larger rectangle if doing so reduces the overall cost. Note that there may be other ways to join rectangles, such as the situation shown in Figure 8. We do not consider these situations in this method of post-processing in the interest of improved runtime.

By *joining* two rectangles $R_1, R_2 \in C$, we mean replacing C by $C' = (C - \{R_1, R_2\}) \cup \{J(R_1, R_2)\}$ with $J(R_1, R_2) = ((x_l, y_t), (x_r, y_b))$ such that $x_l = \min(\text{min_x}(R_1), \text{min_x}(R_2))$, $y_t = \max(\text{max_y}(R_1), \text{max_y}(R_2))$, $x_r = \max(\text{max_x}(R_1), \text{max_x}(R_2))$ and $y_b = \min(\text{min_y}(R_1), \text{min_y}(R_2))$. In other words, $J(R_1, R_2)$ is the smallest rectangle which contains both R_1 and R_2.

By *horizontally aligned*, we mean a subset of rectangles $S \subseteq C$ such that $\text{min_y}(R_1) = \text{min_y}(R_2) \wedge \text{max_y}(R_1) = \text{max_y}(R_2) \forall (R_1, R_2) \in S \times S$, meaning all bottom edges and all top edges of rectangles in S have the same minimum and maximum Y-coordinate respectively.

vertically aligned is defined the same way, but with $\text{min_x}(R_1) = \text{min_x}(R_2)$ and $\text{max_x}(R_1) = \text{max_x}(R_2)$ and all left and all right edges of rectangles in S having the same minimum and maximum X-coordinate respectively.

Lemma 9. *The vertically or horizontally aligned rectangles in a cover C can be determined in $O(|C|)$ time.*

Proof. We can iterate through all $R \in C$ and add them to a list which is mapped to by $(\text{min_x}(R), \text{max_x}(R))$ or $(\text{min_y}(R), \text{max_y}(R))$. When we have iterated through all rectangles, we will be left with lists containing the vertically or horizontally aligned rectangles in C respectively. □

After we calculate the aligned rectangles, we sort each list of aligned rectangles by the rectangle's minimum Y or X-coordinate, depending on the axis they are aligned on. This is done to ensure that when we iterate through these lists and try to join two rectangles, we know that the next rectangle in the list is the one that is closest to the current one. Note that sorting like this is possible because we are working on a partition, which means none of the aligned rectangles overlap.

Lemma 10. *All lists of vertically or horizontally aligned rectangles can be sorted in $O(|C| \log |C|)$ time.*

Proof. We know that the sum of the length of all lists is $|C|$ since each rectangle appears in exactly one list, as a rectangle cannot have multiple minimum and maximum coordinates on the same axis. In the worst case, all rectangles belong to the same list, in which case we can sort them in $O(|C| \log |C|)$ time using a standard sorting algorithm [7]. □

After the lists are sorted, we proceed to attempt to join pairs of rectangles in them. We do this by trying to join each rectangle R_k in the list with the next rectangle R_{k+1}. If $c(J(R_k, R_{k+1})) < c(R_k) + c(R_{k+1})$, we know that replacing the two rectangles with their joint rectangle would improve the overall cost of the cover. But before we do so, we have to check whether $J(R_k, R_{k+1})$ is contained in $\Gamma(\Pi(P))$, since otherwise, it is not a rectangle which is fully contained in P and would thus invalidate the cover. If the joint rectangle is cheaper than the two original ones and does not invalidate the cover we replace C with $C' = (C - \{R_k, R_{k+1}\}) \cup \{J(R_k, R_{k+1})\}$ and continue by trying to join $J(R_k, R_{k+1})$ and R_{k+2}. Otherwise, we do not replace any rectangles and continue by trying to join R_{k+1} and R_{k+2}.

Note that the fastest way we have found in practice to check whether a rectangle R is fully contained within P is to check whether R contains any part of an edge $e \in E(P)$ strictly on its interior, meaning that the endpoints of the edge do not lie on a common line with any of the rectangle's vertices and the edge intersects with the rectangle. Note that we can get away with only checking half of $E(P)$ because we only have to check edges with the opposite orientation of the axis we are joining our rectangles across.

Lemma 11. *Following the procedure described above on all lists of vertically or horizontally aligned rectangles takes $O(|C||E(P)|)$ time.*

Proof. We iterate every list of aligned rectangles exactly once and we know that the sum of the lengths of these lists is $|C|$, so we have exactly $|C|$ iterations, in each of which we potentially have to iterate $\frac{|E(P)|}{2}$ times to determine whether the joint rectangle would invalidate the cover. □

Lemma 12. *Following the procedure described above for a list of aligned rectangles A sorted as described earlier produces a set of joint rectangles A^* where $\theta(A^*)+\epsilon = \theta(A)$ with $\epsilon \geq 0$ as large as possible while maintaining $\bigcup_{R \in A} \Pi(R) \subseteq \bigcup_{R^* \in A^*} \Pi(R^*)$, meaning A^* gives an optimal reduction in cost while still covering at least the same area as A.*

Proof. Firstly, note that having overlapping joint rectangles is never optimal since we can always join the overlapping joint rectangles, which gets rid of all overlap while still covering the same area, so considering them in sorted order is logical since this will never produce overlap.

Secondly, note that when we choose to join or not to join any two rectangles of A, this does not affect whether we should join or not join the next pair of rectangles. Consider R_1 and R_2 of $A = \{R_1, R_2, \dots, R_n\}$, if we join them, we will then consider joining $J(R_1, R_2)$ and R_3. The area we would add to the cover by joining $J(R_1, R_2)$ and R_3 is the same we would be adding by joining R_2 and R_3. Since every choice we make is locally optimal and there is never a need to reconsider earlier decisions, this procedure is optimal. □

Note that in Lemma 12 the rectangles in different lists never overlap, since we are working with a partition, so it makes no difference in which order we process the lists of aligned rectangles for the same axis.

One might wonder why we have chosen to not calculate and join horizontally and vertically aligned rectangles at the same time. This is because after we join rectangles that are vertically aligned, we would have to recalculate the horizontal alignments, since they may have changed during the process, and vice-versa. For this reason, we have chosen to first calculate horizontally aligned rectangles, then attempt to join them, then calculate vertically aligned rectangles and join them as well. This is an arbitrary choice, whether vertically or horizontally joining first is optimal differs from polygon to polygon and can be changed just by rotating it. A heuristic could potentially be used to attempt to guess the optimal order or, alternatively, we could use both orders and take the better of the two afterward. Both of these approaches would incur a potentially non-negligible amount of overhead.

Arbitrary joins In this method of post-processing for the partition, we consider joins such as the one shown in Figure 8 as well, which are ignored by the previous method. As we will see, this does negatively affect runtime complexity but may offer improved solutions when compared to the previous method.

In this method, we also consider joining pairs of rectangles $R_1, R_2 \in C$ but unlike in the previous method, R_1 and R_2 need not be aligned. Instead, all that is required is that $J(R_1, R_2) \in \Gamma(\Pi(P))$, meaning that the joint rectangle of R_1 and R_2 does not invalidate the cover.

The method works similarly to the previous one. We iterate all $R_k \in C$ and consider joining R with an $R' \in \{R_{k+1}, R_{k+2}, \dots, R_n\}$ with R' in a fixed, arbitrary order. We do not have to consider the rectangles in $\{R_1, R_2, \dots, R_{k-1}\}$ since joining these with R_k would have already been considered in previous iterations. Ideally, we would like to reduce the cost of the cover by as much as possible, so we choose the R' which maximizes $r = c(R) + c(R') - J(R, R')$ with $r > 0$. If we find such an R', we then remove R and R' from C, insert $J(R, R')$ at the kth position in C and continue trying to join from the kth position. Otherwise, we continue at the $(k + 1)$th position.

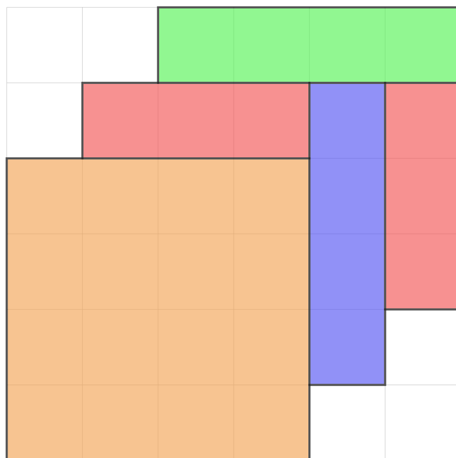


Figure 9: A partition where choosing to join the two red rectangles lets us eliminate one rectangle, but joining one red rectangle with the green one and the other with the blue one could have reduced the number of rectangles by two.

Lemma 13. *The procedure described above takes $O(|C|^2|E(P)|)$ time.*

Proof. Whenever we join two rectangles, we do not move to the $(k+1)$th rectangle, but we do remove R' from $\{R_{k+1}, R_{k+2}, \ldots, R_n\}$, so we still effectively eliminate one iteration. Since we iterate at most $|C|$ rectangles and for each of them have to check the joint rectangle of it and at most $|C|$ other rectangles, we have $O(|C|^2)$ iterations. In each of them, we may have to check if the joint rectangle is in $\Gamma(\Pi(P))$ which, as previously established, we do in $O(E(P))$ time. □

Note that the order in which we join rectangles can make a difference in the cost of the post-processed cover, as can be seen in Figure 9. It is again not clear if an optimal order could be calculated in advance or how costly this would be.

4.4 Greedy Strip Cover

We have already briefly discussed this approach by Kumar and Ramesh [3] in subsection 3.1. We use their algorithm, which offers the best currently known approximation ratio for RPC, for the WRCP, but with some additional post-processing steps to improve the output cover.

In our version of their algorithm, which we will refer to as the *greedy strip cover algorithm* for simplicity, we iterate all horizontal edges of the polygon that have either the polygon's exterior or a hole above them, and then, in contrast to the original version, iterate each integer point lying on this edge and extend a 1-

pixel-wide rectangle, whose top edge lies on the polygon's edge as far downwards as possible and then extend it as far left and right as possible.

Note that this means that the runtime complexity of our version is partially dependent on the width and height of the bounding box of the polygon.

Lemma 14. *The greedy strip cover algorithm runs in* $O(|E(P)|^2 w \max(w, h))$ *time, where* w *and* h *are the width and height of the bounding box of the input polygon respectively.*

Proof. We iterate every horizontal edge with a hole or the exterior above it, which takes $O(|E(P)|)$ time. For each of these iterations, we iterate at worst every integer coordinate lying on the edge, which gives $O(w)$ iterations, then, we have to iterate at worst every integer coordinate downwards, while checking if we can keep going downwards, which gives $O(|E(P)|h)$, since we have to check if the bottom edge of the rectangle is intersecting with an edge of the polygon in a way that prevents it from expanding further. At the end, we extend to the left and right, which again are at worst w iterations, for each of which we have to check if we can expand further on the sides of the rectangle again, which gives $O(|E(P)|w)$. The loops which extend the rectangle downwards and to the sides are sequential rather than nested, so the extending part is $O(\max(w, h)|E(P)|)$, which together with $O(|E(P)|)$ and $O(w)$ from earlier gives $O(|E(P)|^2 w \max(w, h))$. □

4.4.1 Post-processing

Removing redundancies As for the greedy weighted set cover, we also need to remove fully redundant rectangles from the greedy strip cover. Since we avoid enumerating the base rectangles in the greedy strip cover since they are not needed for the algorithm to work, we instead determine whether a rectangle is redundant by checking if every pixel that it contains is already contained in some other rectangle of the cover C. This is the same scenario as we saw in Lemma 6, except we are using $\Pi(P)$ instead of $B(P)$, so the runtime complexity of this process is $O(|C||\Pi(P)|)$.

Likewise, we trim redundant rows and columns the same way as well, which also takes $O(|C||\Pi(P)|)$ time, as we saw in Lemma 7.

Splitting A solution generated by this algorithm in practice often includes a lot of non-trimmable overlap, which is perfectly fine for minimizing the number of rectangles in the cover, but not ideal when the area of the rectangles is also a factor, as in the WRCP. We address this by attempting to split the rectangles in the cover into smaller rectangles when doing so improves the cost of the cover. By *splitting* a rectangle R, we mean removing R from C, which may introduce gaps in the cover, which we fill by adding new rectangles to the cover. By *gap* we mean a polygon that is made up of a maximal set of pixels $G \subseteq \Pi(P)$ where every pixel in the set shares at least one of its edges with another pixel in the set and none of the pixels are contained in any rectangle of C. Note that since the gaps contain *maximal* sets of pixels, we know that there are no gaps that

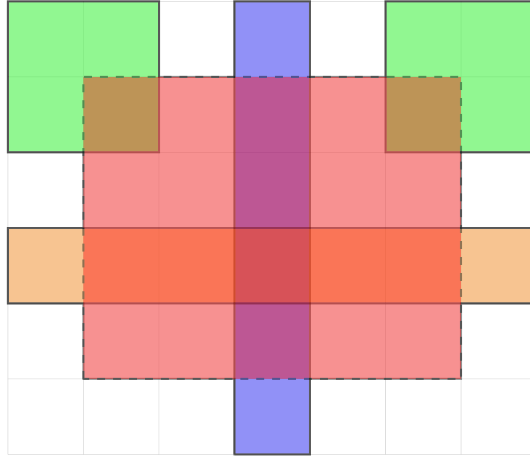


Figure 10: A cover where removing the red rectangle produces four gaps in the cover, two of which are not rectangular.

contain pixels that share an edge with a pixel from any other gap. These gaps are not necessarily just rectangular but can be more complex polygons as well as shown in Figure 10.

Lemma 15. *For a set of gaps $\mathcal{G} = \{G_1, G_2, \ldots, G_m\}$ created by removing a rectangle R from a cover C, the total number of vertices of the gaps $\sum_{G \in \mathcal{G}} |V(G)|$ is $O(|C|^2)$.*

Proof. The gaps in the cover are created by subtracting the union U of the rectangles in $C - \{R\}$ from R. According to [17], the number of vertices in a union of n rectangles is $O(n^2)$ at worst. We thus have $O((|C| - 1)^2) = O(|C|^2)$ vertices in U.

Let $I, X \subseteq V(U)$ denote the set of vertices of U lying inside the interior or on the boundary of R and the set of vertices of U lying outside R respectively. Due to how I and X are defined, we have $I \cup X = V(U)$ and $I \cap X = \emptyset$.

When we subtract U from R, we get our set of gaps, $\mathcal{G}$. Let $\mathcal{V}$ denote the set of all vertices belonging to gaps in $\mathcal{G}$. For any arbitrary vertex $v \in \mathcal{V}$ we may have $v \in I$, $v \in V(R)$ or neither. If we have neither, v must lie on the boundary of R and have been the result of an edge $u \in E(U)$ intersecting an edge $r \in E(R)$. Since $v \notin I$, but u intersects r, we know that one endpoint of u is in I and the other in X.

As we can find one vertex in either $V(R)$, I or X for every vertex in $\mathcal{V}$, we have at most $|V(R)| + |I| + |X|$ vertices in $\mathcal{V}$. Since $|V(R)| = 4$ and $I \cup X = V(U) = O(|C|^2)$, we have $O(|C|^2)$ vertices in $\mathcal{V}$ overall. □

Bounding box gap cover We now consider two ways to cover these gaps. The first is attempting to cover each gap G with its bounding box, meaning the

smallest rectangle $R \in \Gamma(\Pi(P))$ such that $G \subseteq \Pi(R)$. If splitting the rectangle into bounding boxes of its gaps reduces the cost, we perform the split, otherwise, we consider splitting the next rectangle in the cover.

Note that this approach is somewhat limited. If removing the rectangle from the cover only gives us one gap, the bounding box of the gap will be the original rectangle itself, since we remove fully redundant rectangles and trim redundant rows and columns before attempting to split the cover this way.

Lemma 16. *Bounding box gap cover runs on a cover C in $O(|C|^3)$ time.*

Proof. As we saw in the proof of Lemma 15, if we exclude a rectangle R from the cover C, the total number of vertices among the resulting gaps is $O(|C|^2)$. The polygon using the fewest vertices is a rectangle. If every gap is a rectangle, we have $O(|C|^2)$ gaps. As we need to iterate every rectangle in the cover and every gap that is left when removing it from the cover, this gives $O(|C|^3)$ iterations overall. □

Partition gap cover The second way we consider to cover gaps is to view each gap as a polygon and to create a rectangle partition of it using the partition algorithm we discussed in subsection 4.3 without post-processing the resulting partition. If splitting the original rectangle into a partition of its gaps reduces the cost of the cover, we perform the split, otherwise, we consider splitting the next rectangle the same way.

This approach often gives better results than the bounding box cover, but it also has worse runtime complexity.

Lemma 17. *Partition gap cover runs on a cover C in $O(|C|^6)$ time using Ohtsuki's [21] partition algorithm.*

Proof. Again, the total number of vertices among the gaps when excluding a single rectangle from the cover is $O(|C|^2)$ due to Lemma 15. Since Ohtsuki's algorithm runs in $O(n^{2.5})$ time where n is the number of vertices in the polygon, the worst case occurs when all $O(|C|^2)$ vertices belong to a single gap. In this case, we take $O((|C|^2)^{2.5}) = O(|C|^5)$ time to cover the gap. Since we do this for every rectangle in C, we take $O(|C|^6)$ time overall. □

Using an alternative [14] to Ohtsuki's algorithm, the runtime could be improved to $O(|C|((|C|^2)^{1.5} \log |C|^2)) = O(|C|^4 \log |C|^2)$.

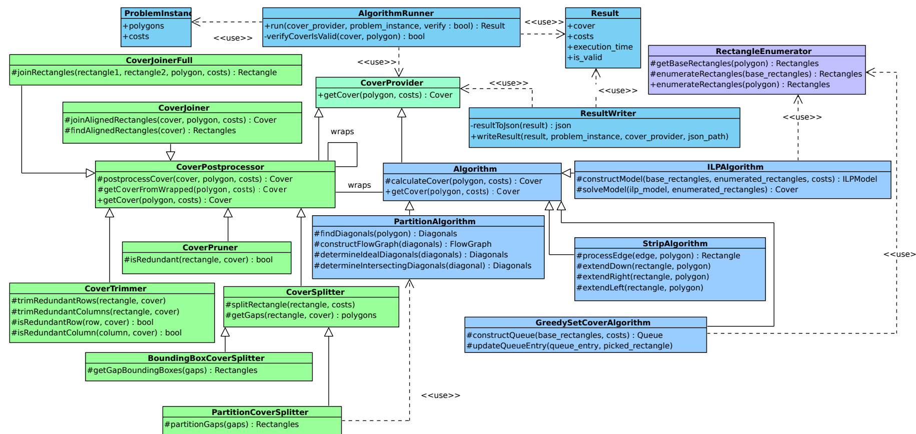


Figure 11: Class diagram for main C++ part of the program.

5 Implementation

5.1 Code

5.1.1 Instance Conversion

Since we are working with polygons, we decided to use the *Well-known text representation of geometry* [2] (WKT) file format to store the input polygons for our experiments, which is well suited for this task. As all instances from the repository we used[3], which is provided by Koch and Marenco in [18], were not in this format, we wrote a Python 3.11[4] script to convert both black-and-white images as well as a custom format used by Heinrich-Litan and Lübbecke [13], from whom we received a few additional instances, to the WKT format. For this purpose, we used the *Pillow*[5], *Shapely*[6], *rasterio*[7] and *numpy*[8] libraries.

5.1.2 Algorithmic Framework

All evaluated algorithms including their post-processing steps were implemented in C++17. The *Computational Geometry Algorithms Library* [9] (CGAL) was used to construct polygons from WKT files, to verify covers returned by the algorithms and as part of some of the algorithms.

Each algorithm uses a common interface with a `getCoverFor()` function which takes an input polygon and a tuple of the costs, (α, β), as input pa-

[2] https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry
[3] https://drive.google.com/drive/folders/1EPj1w_P8Bgg_86dCzOWJVu3JnFsEbrPO
[4] https://www.python.org/
[5] https://python-pillow.org/
[6] https://pypi.org/project/Shapely/
[7] https://pypi.org/project/rasterio/
[8] https://numpy.org/
[9] https://www.cgal.org/

rameters and returns a vector of `Rectangle` instances, which is the resulting cover.

For our integer linear programming implementations, we used *Gurobi*[10] with an academic license.

The post-processors use the same interface as the algorithms and can be wrapped around previous algorithms/post-processors to refine results.

Each algorithm is run on a given problem instance by an `AlgorithmRunner` class, which verifies solutions returned by the algorithm and also records various data, such as the time taken by the algorithm, the cost of the cover, the number of rectangles in the cover, etc.

The main part of the program offers a command-line interface that can be used to run any algorithm with any post-processing steps on a provided input WKT file. The results are output in a *JSON*[11] file at a user-specified location.

Additionally, we implemented a separate program in C++ that can be used to run multiple algorithms and post-processors on multiple instances with multiple different costs to make it easier to run many experiments in sequence. This program also makes it possible to specify a timeout in minutes after which any given experiment is terminated if the corresponding algorithm did not return a solution within this time frame.

Basic functionality of the algorithms as well as some of the other classes was tested using the *GoogleTest*[12] framework.

Boost[13] was used in many places throughout the code as it is already a dependency of CGAL and offers useful functionality for many areas.

All three C++ parts of the project can be built using *CMake*[14].

5.1.3 Visualization

To get a sense of how the algorithms cover a given polygon, it is useful to be able to visualize their results. For this purpose, we implemented a visualization script in Python 3.11 using many of the same libraries which were used for the instance conversion script.

This script can be used to visualize the data from the JSON result files output by the main C++ program. A sample of such a visualization for the cover of a small polygon is shown in Figure 12, though covers with more and larger rectangles can also be visualized the same way without problems.

5.2 Usage

The main program, `covering_run`, has a few command-line parameters that can be passed to it, which are explained when using `covering_run --help`. An example invocation may look like this

[10] https://www.gurobi.com/
[11] https://www.json.org/json-en.html
[12] https://github.com/google/googletest
[13] https://www.boost.org/
[14] https://cmake.org/

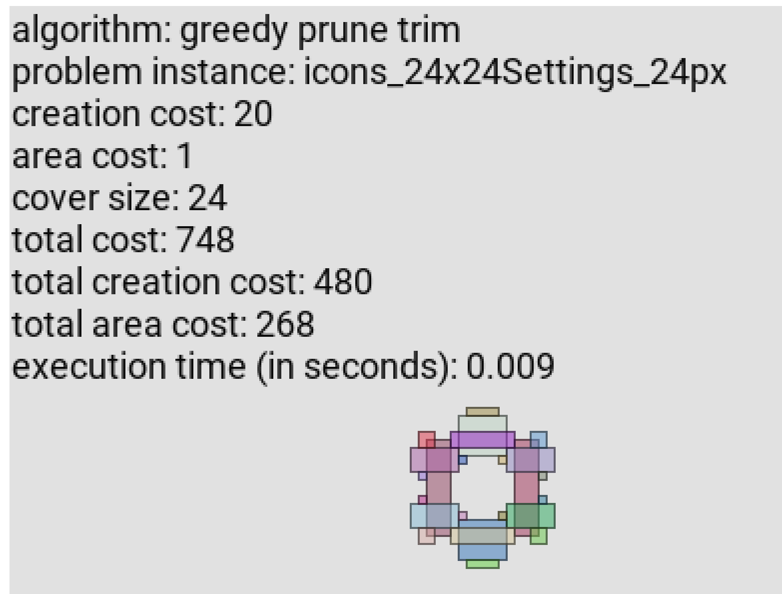


Figure 12: Example output of the result visualizer.

```
./covering_run --input instances/caltech/image_0382.wkt
--costs 100 1 --algorithm strip --postprocessors prune trim
--output result.json
```

For `covering_batch`, which allows running multiple algorithms on multiple instances, `covering_batch --help` also gives an overview of the interface. An example invocation may look like this

```
./covering_batch --instances instances/caltech 100 500 %%
instances/icons 10 20 --algorithms ilp partition
--postprocessors none %% join-full
--results ./results
```

6 Evaluation

6.1 Methods for Evaluation

Experiments were run on all instances from the repository[15] provided by Koch and Marenco [18] as well as on four additional instances we obtained from Heinrich-Litan and Lübbecke [13].

Each experiment was run three times with different cost parameters, with larger α values favoring covers with fewer rectangles and more overlap and

[15] https://drive.google.com/drive/folders/1EPj1w_P8Bgg_86dCzOWJVu3JnFsEbrPO

smaller α values favoring more rectangles and less overlap. β was kept at 1 for all experiments.

For the instance sets containing larger polygons, `caltech`, `nasa`, `ccitt`, `aerials`, `dats`[16] and `textures`, we ran each experiment with $\alpha = 100$, $\alpha = 500$ and $\alpha = 1000$. For the `icons` instance set, which contains small polygons, we instead used $\alpha = 5$, $\alpha = 10$ and $\alpha = 20$.

For the `nasa`, `ccitt`, `aerials`, `dats` and `textures` instance sets, a timeout of one hour was used, while a timeout of 20 minutes was chosen for the `caltech` instance set, as it contains about 800 instances and is generally less complex than the other instance sets containing large polygons. No timeout was used for the `icons` instance set since every algorithm finished almost instantly due to the small size of the instances.

The following algorithms were evaluated:

- `partition`: Partition into rectangles without post-processing
- `partition join`: Partition into rectangles with aligned joins as post-processing
- `partition join-full`: Partition into rectangles with arbitrary joins as post-processing
- `strip prune trim`: Greedy strip algorithm with removal of redundant rectangles and trimming of redundant rows/columns as post-processing
- `strip prune trim bbox-split`: `strip prune trim` followed by attempting to split rectangles in the cover and covering the gaps using bounding boxes
- `strip prune trim partition-split`: `strip prune trim` followed by attempting to split rectangles in the cover and partitioning the gaps into rectangles
- `greedy`: Greedy set cover algorithm without post-processing
- `greedy prune trim`: Greedy set cover algorithm with removal of redundant rectangles and trimming of redundant rows/columns as post-processing
- `ilp`: Integer linear program

Table 1 gives an overview of the characteristics of each instance set we used. The "set size" shows how many instances, meaning WKT files, are part of the corresponding instance set. Since an instance in an instance set may contain several polygons, the third column shows the average number of polygons in each instance of the instance set. The average maximum vertices, average maximum holes and average maximum area columns show the average of the largest

[16] `day.dat`, `marbles.dat`, `mickey.dat` and `night.dat` instances from [13]

corresponding observed feature from each instance in the instance set. Meaning that an average maximum vertex value of 12 for an instance set indicates that the polygon with the most vertices in an instance of the instance set has 12 vertices on average. We have chosen to highlight these features, as it is usually the largest and most complex polygon in an instance that takes up the bulk of execution time.

set	**set size**	**avg. polygons**	**avg. max. vertices**	**avg. max. holes**	**avg. max. area**
`aerials`	38	12587.03	38565.95	3811.42	215934.61
`caltech`	801	97.16	1331.09	81.71	24626.81
`ccitt`	8	1540.38	6608.25	43.88	229907.75
`dats`	4	5208.00	45629.25	7442.25	155889.25
`icons`	168	2.92	24.30	0.60	50.03
`nasa`	13	2899.54	40708.62	2791.69	809030.23
`textures`	23	659.87	41156.39	3755.91	289153.61

Table 1: Characteristics of each set of instances the algorithms were evaluated on.

6.2 Results of Evaluation

Our experiments were performed on a system with Ubuntu 22.04.1 LTS, AMD Opteron 6174 CPU and 32 GB of RAM.

To visualize the results of our evaluation we have chosen to use performance plots. These were produced as follows:

- For every experiment with the parameters we are interested in, the best runtime/solution m^* of any of the algorithms we are interested in is recorded
- For each of these experiments and each of these algorithms, the ratio $\frac{m^*}{m}$ between the algorithm's runtime/solution m and m^* for this particular experiment is computed
- Each algorithm's ratios are sorted in ascending order and plotted on the Y axis

A value of one indicates that the algorithm performed the best in the corresponding metric while a value close to zero indicates that the algorithm performed much worse than the best one.

The X-axis shows the number of experiments for which the algorithm's ratios are smaller or equal to the one at that X-coordinate. For example, if an algorithm's ratio at X-coordinate 5 is 0.8, this means that there were five relevant experiments for which the algorithm's ratio was less than or equal to 0.8.

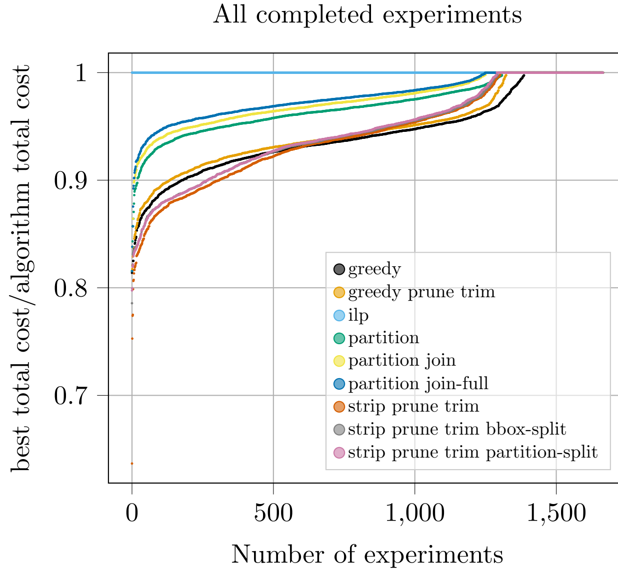


Figure 13: Quality of each algorithm's solution across all experiments for which every algorithm returned a solution.

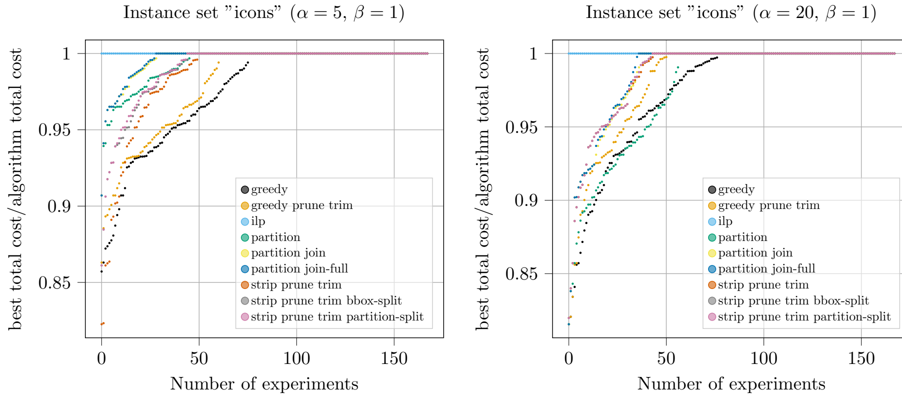
6.2.1 Result Quality

Figure 13 shows the quality of each algorithm's solution for every experiment for which all algorithms returned a solution. As we can see, `ilp` returned the solution with the lowest total cost for every experiment during which it managed to calculate a solution in the allotted time without running out of memory. It is also visible that all versions of the partition approaches have a ratio of 0.9 or better for the vast majority of these experiments, generally performing better than the other non-`ilp` algorithms.

While this gives a quick overview, looking at results for specific instance sets and α-costs is going to prove more insightful.

Figure 14 shows two performance plots for the solutions returned by the algorithms for the `icons` instance set, the one on the left with $\alpha = 5$ and the right with $\alpha = 20$. In general, we can see that for these small instances many of the evaluated algorithms return a solution that costs as much as the one returned by `ilp`. In both plots, we can observe that the greedy strip algorithm and partition algorithms generally outperform the two greedy set cover algorithm versions. Comparing the right and left plots, we can see that with increased α-costs the ratios of the algorithms worsen somewhat and that the difference in the costs of the solutions returned by the algorithms decreases.

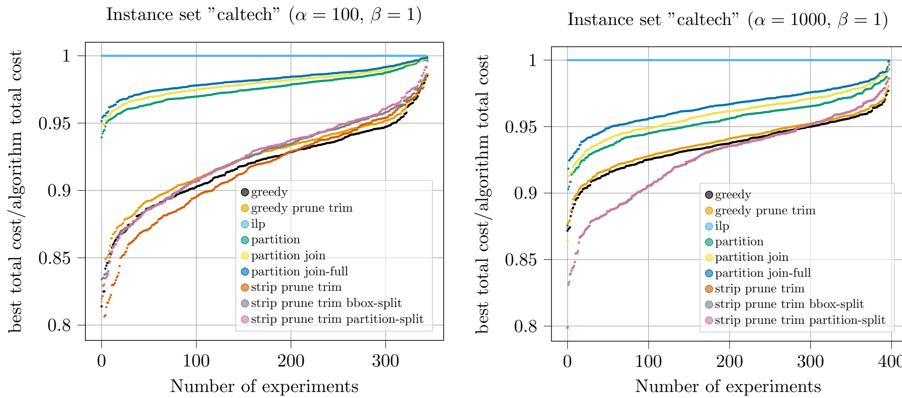
We can observe similar behavior in Figure 15 which shows the same situation for the `caltech` instance set with $\alpha = 100$ and $\alpha = 1000$. While the partition approaches give very good results across the board in the left plot, they again



(a) Quality of each algorithm's solution for the `icons` instance set with $\alpha = 5$.

(b) Quality of each algorithm's solution for the `icons` instance set with $\alpha = 20$.

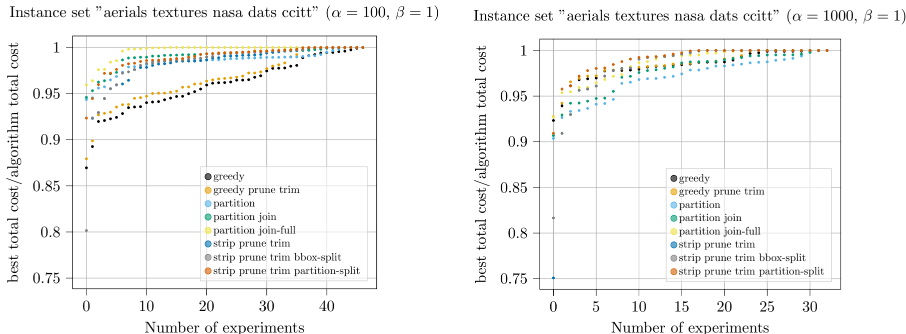
Figure 14: Different performance plots for different costs for the `icons` instance set. All experiments which were performed using the instance set are shown, as none timed out or ran out of memory.



(a) Quality of each algorithm's solution for the `caltech` instance set with $\alpha = 100$.

(b) Quality of each algorithm's solution for the `caltech` instance set with $\alpha = 1000$.

Figure 15: Different performance plots for different costs for the `caltech` instance set. Only experiments for which all evaluated algorithms completed are shown.



(a) Quality of each algorithm's solution for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with $\alpha = 100$.

(b) Quality of each algorithm's solution for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with $\alpha = 1000$.

Figure 16: Different performance plots for different costs for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets. Only experiments for which all evaluated algorithms (except `ilp`) completed are shown.

worsen in the right plot with the increased α-cost, while the greedy strip and set cover algorithms improve somewhat. It is visible that the greedy strip algorithm does especially poorly on this instance set, in particular with the higher α-cost, as it performs worse, or at best similarly, to the greedy set cover approaches. In the right plot, we can also see that the post-processing of the greedy strip algorithm does not appear to improve solutions much, if at all, in this case. This is presumably due to splits being unlikely to decrease the overall cost of the cover since the α-cost is so large. In contrast, the post-processing of the partition appears to pay off in both plots, as there is a clear gap between the plain partition and the post-processed partitions in both of them. It makes sense that the partition approaches would still yield good solutions on these instances since, as Table 1 shows, these instances generally still have few holes and few vertices compared to some of the other instance sets.

Figure 16 shows performance plots for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with $\alpha = 100$ on the left and $\alpha = 1000$ on the right. Note that the instances in these sets proved largely too complex for `ilp` to complete, so for these plots we are only comparing the results of the other algorithms. We can see that the partition algorithms still perform well in both plots, though the `strip prune trim partition-split` algorithm in particular does perform better with the increased α-cost on the right. The two greedy set cover algorithms also perform much better in the right plot, while performing poorly in comparison to the other algorithms for the lower α-cost.

Overall, it appears that the partition approaches followed by joining, especially `partition join-full`, yield very good results for experiments with few holes and/or small α-costs, while the greedy strip cover approaches followed by splitting, especially `strip prune trim partition-split`, may give better

algorithm	mean time in seconds	out of time/memory
`partition`	3.71	9
`partition join`	3.97	9
`partition join-full`	27.75	79
`strip prune trim`	4.17	3
`strip prune trim bbox-split`	7.14	3
`strip prune trim partition-split`	7.59	3
`greedy`	59.84	116
`greedy prune trim`	59.96	116

Table 2: Arithmetic mean runtime of each algorithm (except `ilp`) across all experiments for which all of the listed algorithms completed, as well as the number of times each of them ran out of either time or memory before it could complete.

results for experiments where there are both many holes as well as high α-costs. The greedy set cover approaches overall appeared to usually be outperformed by at least one of the previously mentioned approaches, seemingly performing especially poorly for low α-costs. We can also observe that our post-processing usually improves the results of the algorithms by a non-negligible margin, as there are often large gaps between the plain versions of algorithms and their post-processed variants in the plots.

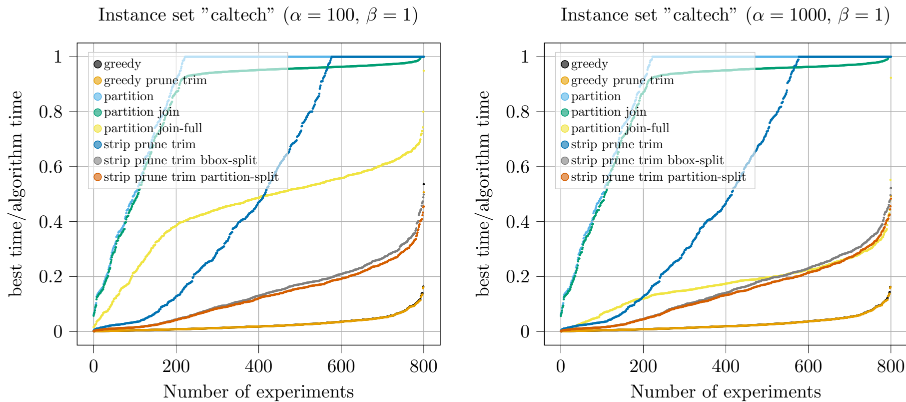
6.2.2 Runtime Results

Table 2 gives an overview of each evaluated algorithm's average runtime across all experiments for which all of them completed, except `ilp` which we have excluded from our runtime evaluation due to its large number of timeouts and very poor speed.

As can be seen, the partition-based approaches are the fastest on average, except the `join-full` version, for which the post-processing turns out to be very costly in practice, while the `join` step showed a comparatively negligible impact on runtime.

The `strip prune trim` version of the greedy strip algorithm on average took a similar amount of time as `partition` and `partition join`. Notably, `bbox-split` and `partition-split` appear to incur a similar runtime cost when used after `strip prune trim`, which is encouraging as `partition-split` generally seems to yield higher quality solutions than `bbox-split`.

Figure 17 shows that the ratios of the algorithms appear to stay similar when α-costs increase, except the `partition join-full` algorithm, which takes longer in comparison in the right plot with the higher α-costs. This is likely because the algorithm checks if joining rectangles would lead to a decrease in the cost of the overall cover before checking if the joint rectangle is valid. With higher α-costs, joins are more likely to be worth it, so the algorithm has to perform more checks for validity in this case, explaining the comparatively



(a) Runtime of each algorithm for the `caltech` instance set with $\alpha = 100$.

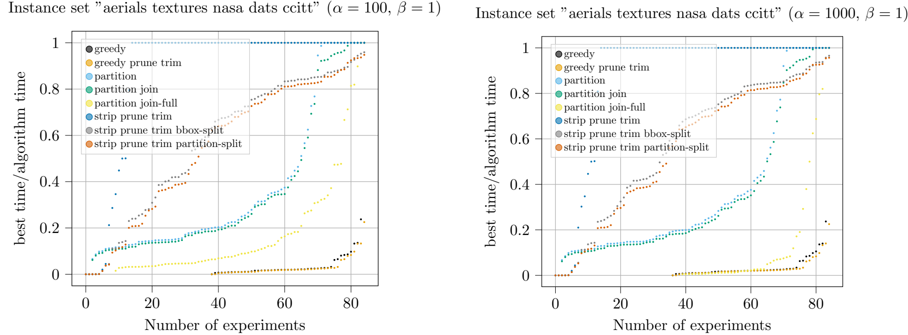
(b) Runtime of each algorithm for the `caltech` instance set with $\alpha = 1000$.

Figure 17: Different performance plots for the runtime of algorithms (except `ilp`) on the `caltech` instance set with different costs. Experiments for which all algorithms (except `ilp`) completed are shown.

worse runtime. Interestingly, the simpler `partition join` does not show a similar deterioration in runtime between the two plots. In combination with the performance plots of the algorithm's quality, this indicates that `join` may be a good alternative to `join-full` when runtime is a concern and α-costs are large.

The performance plots for runtime on the larger instance sets shown in Figure 18 contrast the plots of the `caltech` instance set seen in Figure 17. While in the `caltech` instance set we saw that the `partition` and `partition join` algorithms generally finished faster than the greedy strip algorithms, the reverse was generally true for the larger and more complex `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets. This may be due to the runtime of the partition approaches depending on the number of vertices of polygons, while the runtime of the greedy strip algorithms depends partially on the number of vertices and partially on the dimensions of the polygons. If we look at Table 1, we can see that the larger instance sets generally have a larger increase in their average maximum vertex count than in their average maximum polygon area when compared to the `caltech` instance set, which could explain why the relationship between the runtime of the partition algorithms and greedy strip algorithms appears to change between the two different sets of performance plots.

We can again see that the runtime of the `partition join-full` algorithm deteriorates significantly in the right plot of Figure 18, as it timed out much more frequently for $\alpha = 1000$, while all other algorithms show very similar ratios in both plots.



(a) Runtime of each algorithm for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with $\alpha = 100$.

(b) Runtime of each algorithm for the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with $\alpha = 1000$.

Figure 18: Different performance plots for the runtime of algorithms (except `ilp`) on the `aerials`, `ccitt`, `dats`, `nasa` and `textures` instance sets with different costs. All experiments are shown, an algorithm timing out or running out of memory during an experiment is indicated by the absence of a data point for the algorithm at the given X-coordinate.

Overall, the greedy set cover approaches exhibited very poor runtime performance, being consistently outperformed by all other plotted algorithms. While on average the partition algorithms show the best runtime, with the exception of `partition join-full`, we saw that the greedy strip cover algorithms generally finished faster on larger instances specifically. Importantly, specifically `partition join-full` showed worsening runtime with increasing α-costs, while `partition join` did not appear to suffer from the same behavior, while still providing a similar improvement in the cost of the cover to `join-full`.

7 Conclusion & Future Work

7.1 Summary

In this thesis, we have described the WRCP and shown a formulation of the integer linear program for the problem, as well as several heuristic algorithms, which, as shown in our experiments, provide high-quality solutions in practice.

In particular, the partition algorithm followed by full or aligned joining of the partition appears to give surprisingly good results in practice even on instances with many holes.

As the size of the polygons, the number of holes and α-costs increase, the greedy strip cover algorithm, followed by pruning, trimming and partition-splitting of the cover, as well as the greedy set cover algorithm, followed by pruning and trimming of the cover, appear to sometimes outperform the partition-based approach, though the greedy set cover algorithm in particular proves slow

in our experiments due to its need to enumerate the set of candidate rectangles beforehand.

7.2 Limitations of the Approaches

In the interest of preserving reasonable runtime performance, we perform only a single round of select post-processing methods. Repeating, randomizing or adding further post-processing may potentially yield slightly improved solutions.

As stated, we were unable to formally prove Conjecture 1, but have presented experimental evidence in its favor.

Due to the large number of experiments and often long runtimes, we were only able to run each experiment once. There is thus a chance of some measurement error within the runtime data. In addition, since we have fewer large instances and algorithms were more likely to time out or run out of memory when processing them than for smaller instances, we inherently have less data for large instances, both in terms of runtime and quality of solutions.

7.3 Outline of Future Work

As far as we are aware, the WRCP specifically has not been studied before. There are thus plenty of possible avenues for further research, both into the practical as well as theoretical aspects of the problem.

7.3.1 Hardness of the WRCP

As we have seen in subsection 3.1, RPC, which is $\mathcal{NP}$-hard [20], is very similar to the WRCP when $\alpha = 1$ and $\beta = 0$. Likewise, when $\alpha = 0$ and β is arbitrary, the WRCP is trivial since we can use arbitrarily many rectangles and are just minimizing the covered area, which can even be accomplished by simply filling the polygon with non-overlapping unit squares. As the hardness of the problem depends on the input polygon as well as the costs, it may be interesting to study what exactly makes a particular instance of the problem “hard” and whether we could easily detect such cases in practice.

7.3.2 Approximability of the WRCP

While our heuristics appear to provide good solutions in practice, we do not give approximation factors for them in this thesis, except for the greedy weighted set cover implementation, which is known to give $O(\log n)$ approximation [6], but does not perform as well as others in our experiments, while also showing poor runtime performance. We also do not provide results for the approximability of the WRCP in general, though again, for $\alpha = 1$ and $\beta = 0$, it is worth noting that it is not currently known whether there exists a constant factor approximation for RPC in polynomial time [13].

7.3.3 Integer Linear Program for RPC

Our formulation of the integer linear program, which uses the base rectangles of the polygon, could yield practical improvements in formulations of the integer linear program for RPC, as the number of base rectangles in a polygon can be much smaller than the number of pixels. In such cases, a formulation similar to ours for RPC could drastically reduce the number of constraints of the integer linear program when compared to using pixels.

In addition, Conjecture 1 appears more easily provable for RPC than for the WRCP; As we already saw, in RPC it suffices to consider only maximal rectangles which fit inside of the polygon for the cover [13]. Every rectangle which has the property that it is maximal has to fully contain every base rectangle which it intersects, otherwise, we could extend the rectangle until it fully contains all base rectangles which it intersects, contradicting it being maximal. We know that extending the rectangle this way cannot cause it to become invalid, since there are no holes we could run into, as there are none inside base rectangles due to their definition. It then follows that any relevant rectangle for RPC must be the union of some subset of $B(P)$, meaning a formulation of the integer linear program for RPC using base rectangles for the constraints would give optimal solutions for RPC.

References

[1] Agarwal, P. K., Ezra, E., and Fox, K. *Geometric Optimization Revisited*. Springer-Verlag, Berlin, Heidelberg, 2022, p. 66–84.

[2] Agarwal, P. K., and Pan, J. Near-linear algorithms for geometric hitting sets and set covers. In *Proceedings of the Thirtieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2014), SOCG'14, Association for Computing Machinery, p. 271–279.

[3] Anil Kumar, V. S., and Ramesh, H. Covering rectilinear polygons with axis-parallel rectangles. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1999), STOC '99, Association for Computing Machinery, pp. 445—-454.

[4] Brönnimann, H., and Goodrich, M. T. Almost optimal set covers in finite vc-dimension: (preliminary version). In *Proceedings of the Tenth Annual Symposium on Computational Geometry* (New York, NY, USA, 1994), SCG '94, Association for Computing Machinery, p. 293–302.

[5] Bus, N., Mustafa, N., and Ray, S. Practical and efficient algorithms for the geometric hitting set problem. *Discrete Applied Mathematics 240* (May 2018), 25–32.

[6] Chvatal, V. A greedy heuristic for the set-covering problem. *Math. Oper. Res. 4*, 3 (aug 1979), 233–235.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[8] Even, G., Rawitz, D., and Shahar, S. M. Hitting sets when the vc-dimension is small. *Information Processing Letters 95*, 2 (2005), 358–362.

[9] Feige, U. A threshold of ln n for approximating set cover. *J. ACM 45*, 4 (jul 1998), 634–652.

[10] Franzblau, D., and Kleitman, D. An algorithm for covering polygons with rectangles. *Information and Control 63*, 3 (1984), 164–189.

[11] Hannenhalli, S., Hubell, E., Lipshutz, R., and Pevzner, P. Combinatorial algorithms for design of dna arrays. *Advances in biochemical engineering/biotechnology 77* (02 2002), 1–19.

[12] Hegedüs, A. Algorithms for covering polygons by rectangles. *Computer-Aided Design 14*, 5 (1982), 257–260.

[13] Heinrich-Litan, L., and Lübbecke, M. E. Rectangle covers revisited computationally. *ACM J. Exp. Algorithmics 11* (feb 2007), 2.6–es.

[14] Imai, H., and Asano, T. Efficient algorithms for geometric graph search problems. *SIAM J. Comput. 15*, 2 (may 1986), 478–494.

[15] Johnson, D. S. Approximation algorithms for combinatorial problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1973), STOC '73, Association for Computing Machinery, p. 38–49.

[16] Karp, R. M. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA* (1972), R. E. Miller and J. W. Thatcher, Eds., The IBM Research Symposia Series, Plenum Press, New York, pp. 85–103.

[17] Keller, C., and Smorodinsky, S. On the union complexity of families of axis-parallel rectangles with a low packing number, 2017.

[18] Koch, I., and Marenco, J. A hybrid heuristic for the rectilinear picture compression problem. *4OR* (Jun 2022).

[19] Lovász, L. M. On the ratio of optimal integral and fractional covers. *Discret. Math. 13* (1975), 383–390.

[20] Masek, W. Some np-complete set cover problems. *unpublished manuscript, MIT Laboratory for Computer Science* (1978).

[21] Ohtsuki, T. Minimum dissection of rectilinear regions. In *Proc. 1982 IEEE Symp. on Circuits and Systems* (1982), pp. 1210–1213.

[22] O'Rourke, J. The complexity of computing minimum convex covers for polygons. In *20th Annual Allerton Conference on Communication, Control, and Computing* (1982), pp. 75–84.

[23] O'Rourke, J., Suri, S., and Tóth, C. D. Polygons. In *Handbook of discrete and computational geometry* (2016), pp. 787–810.

[24] Siringoringo, W. S., Connor, A. M., Clements, N., and Alexander, N. Minimum cost polygon overlay with rectangular shape stock panels. *CoRR abs/1606.05927* (2016).

[25] Varadarajan, K. Weighted geometric set cover via quasi-uniform sampling. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing* (New York, NY, USA, 2010), STOC '10, Association for Computing Machinery, p. 641–648.

[26] Vazirani, V. V. *Approximation algorithms*, 2 ed. Springer, Atlanta, USA, 2003.