# Practical Algorithms for Decomposing Objects into Small Components

**Mikkel Abrahamsen**
Department of Computer Science, University of Copenhagen, Denmark

**Reilly Browne**
Department of Informatics, Dummy College, Country

**Mayank Goswami**
Department of Informatics, Dummy College, Country

**Prahlad Narasimhan Kasthurirangan**
Department of Informatics, Dummy College, Country

**Joseph S.B. Mitchell**
Department of Informatics, Dummy College, Country

**Michael Perk**
Department of Informatics, Dummy College, Country

**Valentin Polishchuk**
Department of Informatics, Dummy College, Country

## Abstract

In 3D printing, fabricators typically subdivide a large object into smaller components each of which fits inside the print bed. In this paper, we study two problems motivated by this workflow. A connected subset of $\mathbb{R}^d$ is defined to be *small piece* if it fits inside an axis-aligned unit box. Our first problem asks for a decomposition of a given simple polygon $P$ into the minimum number of small pieces. We present a greedy 4-approximation algorithm for this problem, improving the current best factor of 13. While the algorithm's correctness relies heavily on non-trivial structural properties of small decompositions that we discover, the algorithm itself is simple to implement.

In practice it may be preferable to print several pieces at once, as long as they altogether fit within the bed. To capture this, we define and study $d$–DiscoSmallPartition: a (possibly disconnected) subset of $\mathbb{R}^d$ is a *small region* if it fits within a unit box; we look to decompose a (possibly non-simple) polyhedron $P \subseteq \mathbb{R}^d$ into the minimum number of small regions. We provide a general $2^{d-1}$-approximation algorithm for the problem and focus on obtaining efficient algorithms for the practically relevant cases of $d = 2$ and $d = 3$. For the 2D case, we describe a 2-approximation algorithm that runs in near-linear time. In 3D, our approximation factor is 4 while the runtime is in the order of $f \cdot \text{OPT}$ where $f$ is the number of faces of $P$ and OPT is the number of boxes in an optimal decomposition of $P$. We provide implementations of our algorithms and make it open source to support further engineering. Furthermore, we compare the performance of our algorithms to an exact primal-dual approach and present experimental results.

**2012 ACM Subject Classification** Theory of computation → Computational geometry

**Keywords and phrases** Polyhedral Decomposition, 3D Printing, Approximation Algorithms

**Supplementary Material** *Software (Source Code)*: `insertlinktorepo`

**Funding** *Mikkel Abrahamsen*: Supported by Independent Research Fund Denmark, grant 1054-00032B, and by the Carlsberg Foundation, grant CF24-1929.
*Reilly Browne*: [funding]
*Mayank Goswami*: [funding]
*Prahlad Narasimhan Kasthurirangan*: [funding]
*Joseph S.B. Mitchell*: [funding]
*Michael Perk*: [funding]




*Valentin Polishchuk*: [funding]

# 1 Introduction

In manufacturing processes, it is typical that we want to produce a large object $P$ but are restricted by fabrication methods that can only produce parts of bounded size. In 3D printing, for example, we can only print parts that fit inside the printing volume, which is typically a box. Once these parts are produced they are assembled appropriately to form $P$. In order to keep the assembly process as simple as possible, we seek to produce as few parts as possible and thus are motivated to partition $P$ into the fewest pieces, each of which is "small" in some sense. In other applications, we seek to cover $P$ with the fewest small pieces, i.e., the pieces may overlap, but must be contained in $P$.

Formal investigation of this class of partitioning/covering problems has been very recent [3] and has focused on the 2-dimensional case. A polygon $Q$ is *small* or is a *small piece* if it fits inside an axis-parallel unit square. A collection $\mathcal{Q}$ of small polygons (which we typically refer to as *small pieces*) is a *small cover* of a polygon $P$ if $P = \bigcup_{Q\in\mathcal{Q}} Q$. Note that each $Q \in \mathcal{Q}$ is a subset of $P$.

A small cover $\mathcal{Q}$ of $P$ is a *small partition* of $P$ if the pieces of $\mathcal{Q}$ are interior-disjoint. The optimization problems SmallPartition and SmallCover seek to minimize the number of pieces in a small partition or a small cover of an input polygon $P$. When the input polygon $P$ is simple, Abrahamsen and Rasmussen give a 13-approximation algorithm [3, Table 1] for SmallPartition with running time $\mathcal{O}(n^2 + \text{OPT} \log n)$, where $n$ is the number of vertices of $P$ and OPT is the number of pieces in the optimal partition of $P$. Abrahamsen and Stade [**?**] show that both SmallCover and SmallPartition are NP-Hard even in simple orthogonal polygons. In private communication (2025), we have learned of the following equivalence, which is proven in a paper submitted to SoCG 2026 (authors redacted):

▶ **Lemma 1.1.** *[1] Let $\mathcal{Q}$ and $\mathcal{Q}'$ denote an optimal small cover and partition of a simple polygon $P$ respectively. Then, $|\mathcal{Q}| = |\mathcal{Q}'|$.*

Lemma 1.1 immediately implies that the 13-approximation for SmallPartition [3] is a 13-approximation for SmallCover. In this paper, we give a 4-approximation algorithm for SmallCover (and therefore, also for SmallPartition).

▶ **Theorem 1.2.** *Consider a simple polygon $P \subseteq \mathbb{R}^2$. There is an algorithm running in $\mathcal{O}(n \cdot \text{OPT} + \text{OPT}^2)$ time that returns a small cover of $P$ with at most $4 \cdot \text{OPT}$ pieces.*

We prove this result in Section 2. Our algorithm first covers the boundary of $P$ (hereon $\partial P$) using a similar greedy strategy as [3]. We then employ a simple line-sweep algorithm covering the left-most uncovered point in each iteration with "maximal" small pieces. While its correctness involves non-trivial structural results of our greedy boundary cover (see Lemma 2.2), the algorithm itself is straightforward to implement; we provide an implementation and run experiments in Section 4.

In 3D printing, it is typically much more economical to print several pieces at once as long as they altogether fit inside the printing volume [2, 19, 22, 18]. We therefore model fabricable substructures as disconnected regions that lie inside a unit hypercube and study the decomposition of a given shape by such regions. Consider a region $Q \subseteq \mathbb{R}^d$. We say $Q$ is small (or that it is a *small region*) if it fits inside an axis-parallel unit hypercube. Note that unlike SmallCover, we do not require $Q$ to be connected; we use "small regions" instead of "small pieces" to distinguish these two cases. A collection $\mathcal{Q}$ of small regions is a *disconnected*

*small cover* of a polyhedron $P$ if $P = \bigcup_{Q \in \mathcal{Q}} Q$; a *disconnected small partition* of $P$ is a cover with interior disjoint regions. Our second set of problems of interest is $d$–DiscoSmallCover and $d$–DiscoSmallPartition: given a (possibly non-simple) polyhedron $P$ in $\mathbb{R}^d$, what is the minimum number of pieces in a disconnected small cover (resp. partition) $\mathcal{Q}$ of $P$?

▶ Remark 1.3. We observe that $d$–DiscoSmallCover is precisely the problem of covering a polyhedron $P \subseteq \mathbb{R}^d$ with axis-aligned unit hypercubes, i.e., finding a family $\mathcal{S}$ of such cubes with $P \subseteq \bigcup_{S \in \mathcal{S}} S$. Geometric covering problems of this type have been extensively studied. In particular, covering a *finite* point set with the minimum number of unit balls in $\mathbb{R}^d$ (typically called $d$–BoxCover or $d$–DiscCover) has a history of more than 40 years, beginning with early complexity results [13] and followed by a long line of approximation algorithms [16, 15, 4, 12, 8]. More recently, the problem of covering a *continuous* region with unit balls chosen from a *prescribed candidate set* has also been investigated [5, 9].

▶ Remark 1.4. $d$–DiscoSmallCover and $d$–DiscoSmallPartition are NP-Hard even when $d = 2$ and $P$ is a simple orthogonal polygon. This follows immediately from the proof of hardness of SmallCover and SmallPartition in [**?**]; their construction does not use the fact that small pieces are connected.

Given an optimal disconnected small cover $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ of $P$, note that $\mathcal{Q}' = \{Q_i \setminus \bigcup_{1 \leq j < i} Q_j \mid i = 1, 2 \ldots k\}$ is a small partition of $P$. Mirroring Lemma 1.1, we have:

▶ **Observation 1.5.** *Let $\mathcal{Q}$ and $\mathcal{Q}'$ denote an optimal disconnected small cover and an optimal disconnected small partition of a polyhedron $P \in \mathbb{R}^d$, respectively. Then, $|\mathcal{Q}| = |\mathcal{Q}'|$.*

We provide a general $2^{d-1}$-approximation algorithm for $d$–DiscoSmallPartition by showing that the problem can be reduced to a series of 1D problems by repeated projection onto relevant hyperplanes, losing a multiplicative 2-factor for each dimension; we then show that it suffices to use a simple greedy sweep algorithm to solve each 1D instance optimally. In general, this algorithm runs in $\text{poly}(\text{OPT}, n)^d$ time. This approach can be seen as a generalization of a sweep algorithm proposed by Gonzalez in [15] for $d$–BoxCover. Unlike their algorithm, we are required to cover a continuum, and OPT can be arbitrarily large when compared to the input size. We provide the following two efficient algorithms for the practically relevant cases of $d = 2$ and $d = 3$ (see Sections 3.3 and 3.4 for details).

▶ **Theorem 1.6.** *For an $n$-vertex polygon $P \subseteq \mathbb{R}^2$ there is an $\mathcal{O}((n + \text{OPT}) \log n)$ time algorithm that returns a disconnected small partition of $P$ with at most $2 \cdot \text{OPT}$ pieces.*

▶ **Theorem 1.7.** *For a polyhedron $P \subseteq \mathbb{R}^3$ with $f$ triangular faces there is an $\mathcal{O}(f \cdot \text{OPT} \log f)$ time algorithm that returns a disconnected small cover of $P$ with at most $4 \cdot \text{OPT}$ pieces.*

We present our experimental results in Section 4. Code and data are made available via HotCRP*. We show that the 4-approximation algorithm for SmallCover can be realized easily in practice and requires no sophisticated geometric data structures. For 2–DiscoSmallCover, which requires exact geometric primitives and exact construction of line intersections, we implement a modification of this approach (which results in a worse theoretical runtime but the same approximation guarantee) and show that it works well in practice. Our test instances are taken from the Salzburg Database of Geometric Inputs [11]. We compare resulting solutions with lower bounds obtained from a constraint–programming formulation of a set cover instance in which the elements are witnesses inside $P$ and the sets correspond to small regions.

* After the review phase, the code will be moved to a permanent repository.

## 2 A 4-Approximation for SmallCover

We can assume that the input polygon $P$ is not itself small. For points $p, q \in \partial P$, the (contiguous) boundary of $P$ in the clockwise direction from $p$ to $q$ is denoted $\partial P[p, q]$. Let OPT denote the optimal number of pieces in a small cover or, equivalently (Lemma 1.1) a small partition of $P$. We utilize the following lemma from [3]:

▶ **Lemma 2.1.** *[3, Lemma 4] Let $a_0 = a_k$ be an arbitrary point on $\partial P$ and let $\mathcal{Q}_\partial = \{Q_1, \ldots, Q_k\}$ be a set of small pieces where $Q_i$ covers the interval $I_i = \partial P[a_{i-1}, a_i]$ and for $i = 1, ..., k-1$, the endpoint $a_i$ is chosen so that a piece of bounded size cannot cover a larger interval starting at $a_{i-1}$. Then, $k = |\mathcal{Q}_\partial| \leq 2 \cdot \text{OPT} - 1$.*

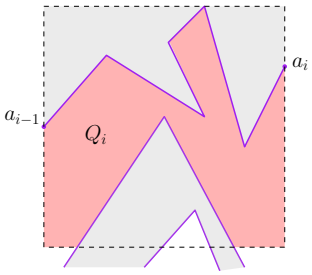


**Figure 1** The piece $Q_i$ is obtained from a unit square containing the interval $I_i$ from $a_{i-1}$ to $a_i$.

Observe that the pieces $\mathcal{Q}_\partial = \{Q_i \mid 1 \leq i \leq k\}$ partition $\partial P$ into $\mathcal{I}_\partial = \{I_i \mid 1 \leq i \leq k\}$. Constructing $I_\partial$ takes $\mathcal{O}(n + \text{OPT})$ time [3, Lemma 5]. By construction, the width or height of $I_i$, where $1 \leq i \leq k-1$, is 1. Since we are looking for a cover of $P$ and not a partition, we modify $\mathcal{Q}_\partial$ a little: first, we no longer enforce that $a_k = a_0$ and extend $a_k$ along $\partial P$ until $I_k$ has width or height 1. Now, consider any $I_i \in \mathcal{I}_\partial$ and let $S$ be any unit square containing $I_i$. Let $Q_i$ denote the connected component of $S \cap P$ that contains $I_i$; we let $\mathcal{Q}_\partial := \{Q_i \mid 1 \leq i \leq k\}$. See Figure 1 for an illustration. Since each $I_i \subseteq Q_i$, the pieces $\mathcal{Q}_\partial$ cover $\partial P$; we therefore refer to these as *boundary pieces*. Given $I_i$, constructing $Q_i$ can clearly be done in $\mathcal{O}(n)$ time; therefore, constructing $\mathcal{Q}_\partial$ takes $\mathcal{O}(n \cdot \text{OPT})$ time. We now show that the bounding box of a set of points in $P$ uncovered by these boundary pieces is a witness to whether they can be covered by the same small piece. For two points $p, q \in P$, we let $SP(p, q)$ denote the euclidean shortest path within $P$ between them. We use $x(p)$ and $y(p)$ to denote the $x$- and $y$-coordinate of a point $p \in \mathbb{R}^2$. We use $BB(\cdot)$ to denote the axis-aligned bounding box.

▶ **Lemma 2.2.** *Consider two points $p, q \in Q \cap P$ that are within a small piece $Q$. If $SP(p, q) \not\subset BB(p, q)$, then at least one of $p$ or $q$ is covered by a boundary piece $Q_i \in \mathcal{Q}_\partial$.*

**Proof.** Consider two points $p$ and $q$ which satisfy our premise: i.e., they can be covered by a small piece $Q$ and $SP(p, q) \not\subset BB(p, q)$. Then, $SP(p, q)$ contains a point that has an $x$- or $y$-coordinate which is either smaller or larger than both $p$ and $q$. We assume that $SP(p, q)$ contains a point with lower $y$-coordinate than $p$ and $q$, the other cases follow symmetrically. Let $v$ be the point with smallest $y$-coordinate in $SP(p, q)$. Clearly, $v$ is a reflex vertex of $P$ and is therefore covered in some interval $I_v$ corresponding to a $Q_v \in \mathcal{Q}_\partial$. Let $\ell$ denote a point with the lowest $y$-coordinate in $I_v$. We will show that $Q_v$ contains either $p$ or $q$ by considering two cases: either $y(\ell) < y(v)$ or $y(\ell) = y(v)$ (note that $y(\ell) \leq y(v)$ since $v \in I_v$).

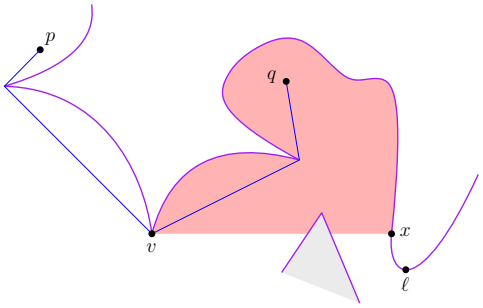


**Figure 2** Illustration of Case 2.2.1. The boundary of the polygon is in purple while $SP(p, q)$ is marked blue. The polygon $P_v$ is the red region. The gray region lies outside the polygon.

▶ Case 2.2.1. Assume that $y(\ell) < y(v)$. We assume that $\partial P[v, l] \subseteq I_v$: i.e., $\ell$ appears after $v$ in the clockwise ordering of $\partial P$. The other case is symmetric. Since $v$ has the lowest $y$-coordinate in $SP(p, q)$, we have $\ell \notin SP(p, q)$. Therefore, the horizontal line through $v$ intersects $\partial P[v, \ell] \setminus SP(p, q)$; let $x$ be the point closest to $v$ in this intersection. Let $P_v$ denote the component of $BB(\partial P[v, x]) \cap P$ containing $\partial P[v, x]$ (see Figure 2 for an illustration). Since $\partial P[v, x] \subseteq I_v$, we have $P_v \subseteq Q_v$. Both $SP(p, v)$ and $SP(v, q)$ lie above $\overrightarrow{vx}$, so either $SP(p, v)$ or $SP(v, q)$ is contained in $BB(\partial P[v, x])$ and therefore in $P_v$. This shows that either $p$ or $q$ is in $Q_v$. ⌟

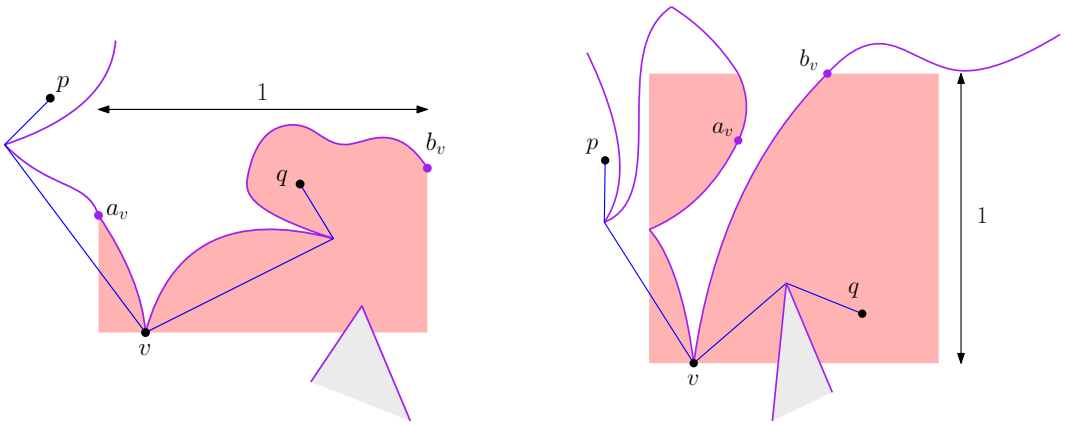


**Figure 3** Illustration of Case 2.2.2. The boundary of the polygon is in purple while $SP(p, q)$ is marked blue. The red region is a subset of $Q_v$. The gray region lies outside the polygon.

▶ Case 2.2.2. Now, assume that $y(\ell) = y(v)$. We further break this up into two cases based on whether the width or height of $I_v = \partial P[a_v, b_v]$ is 1. Let $x_b$, $x_t$, $y_b$, and $y_t$ denote points with minimum and maximum $x$- and $y$-coordinates in $SP(p, q)$ respectively. See Figure 3.
*Subcase (a).* Assume that the width of $I_v$ is 1. Let $x_1 = \min\{x(a_v), x(b_v)\}$ and $x_2 = \max\{x(a_v), x(b_v)\}$. Since $p$ and $q$ are covered by the small piece $Q$, $x_t - x_b \leq x_2 - x_1 = 1$. Therefore, either $SP(p, v)$ lies to the right of the line $x = x_1$ or $SP(v, q)$ lies to the left of $x = x_2$. Moreover, since $SP(p, q)$ lies above the horizontal line through $v$, either $SP(p, v)$ or $SP(v, q)$ is contained in $BB(I_v) \subseteq BB(Q_v)$. This implies that either $p$ or $q$ is in $Q_v$.
*Subcase (b).* Assume that the height of $I_v$ is 1. Since $y(\ell) = y(v)$, we have that either $y(a_v)$ or $y(b_v)$ is $y(v) + 1$. Since $p$ and $q$ is covered by the small piece $Q$ and $v$ is the lowest point in $SP(p, q)$, we have $[y_b, y_t] \subseteq [y(v), y(v) + 1]$. Recall that $Q_v$ was constructed using

a unit square which contained $I_v$, let $S$ be this square: i.e., $Q_v$ is the component of $S \cap P$ containing $I_v$. Let $x_1$ and $x_2$ denote the $x$-coordinate of the leftmost and rightmost vertices of $S$ respectively. As $x_t - x_b \leq 1$, at least one of $SP(p, v)$ or $SP(v, q)$ lies between $x = x_1$ and $x = x_2$; therefore, one of these paths lie in $S$. This shows that either $p$ or $q$ is in $Q_v$. ⌟

This completes the proof of our lemma. ◀

For a point $p \in P$, let $S_p$ (resp., $S'_p$) denote the unit square having $p$ its bottom-left (resp., top-left) vertex, and let $Q_p$ (resp., $Q'_p$) denote the component of $S_p \cap P$ (resp., $S'_p \cap P$) containing $p$. Following this simple consequence of Lemma 2.2, we prove Theorem 1.2.

▶ **Corollary 2.3.** *Consider points $p, q \in P$ that are uncovered by pieces in $\mathcal{Q}_\partial$, where $x(p) \leq x(q)$, and suppose that $p$ and $q$ are contained in the same small piece. Then $q \in Q_p \cup Q'_p$.*

**Proof.** By Lemma 2.2, $SP(p, q) \subseteq BB(p, q)$. Since $x(p) \leq x(q)$, we know $p$ is a left vertex of $BB(p, q)$ and thus that $BB(p, q)$ is a subset of either $S_p$ or $S'_p$, and hence $q \in Q_p \cup Q'_p$. ◀

**Proof of Theorem 1.2.** We first construct $\mathcal{Q}_\partial$, our boundary pieces, in $\mathcal{O}(n \cdot \text{OPT})$ time; note that $|\mathcal{Q}_\partial| \leq 2 \cdot \text{OPT}$ (see the discussion prior to Lemma 2.2). We use an iterative greedy sweep-line algorithm to cover the region of the polygon uncovered by $\mathcal{Q}_\partial$. Let $\mathcal{Q}_\circ$ denote the set of pieces we use; we initialize $Q_\circ$ with $\emptyset$ and add two pieces to it in each iteration. Note that the region of $P$ uncovered by $\mathcal{Q}_\partial$ is a collection of (orthogonal) polygons whose vertices are defined by a union of $\mathcal{O}(\text{OPT})$ squares; the complexity of the uncovered region is therefore $\mathcal{O}(\text{OPT})$ [7].

In the $i^{\text{th}}$ iteration, let $p_i$ be a leftmost vertex of the uncovered region. Consider the small pieces $Q_{p_i}$ and $Q'_{p_i}$ as in Corollary 2.3; we update $\mathcal{Q}_\circ \leftarrow \mathcal{Q}_\circ \cup \{Q_{p_i}, Q'_{p_i}\}$. Repeat this procedure until all points in $P$ are covered. Let $k$ denote the number of iterations of this procedure; note that $|\mathcal{Q}_\circ| = 2k$. Consider $W = \{p_i \mid 1 \leq i \leq k\}$ and two points $p_i, p_j \in W$ where $i < j$. By construction, $p_j \notin Q_{p_i} \cup Q'_{p_i}$ and therefore, by Corollary 2.3, no small piece covers both $p_i$ and $p_j$. Therefore, $W$ is a set of *witness points* (i.e., no two points in $W$ can be covered with the same piece), so $|W| = k \leq \text{OPT}$. This implies that $|\mathcal{Q}_\circ| \leq 2 \cdot \text{OPT}$; overall, our small cover $\mathcal{Q} = \mathcal{Q}_\partial \cup \mathcal{Q}_\circ$ contains at most $4 \cdot \text{OPT}$ pieces. Since each iteration takes $\mathcal{O}(n + \text{OPT})$ time, our runtime follows. See Figures 8 and 9 for examples of the pieces constructed by this algorithm. ◀

## 3 Approximation Algorithms for DiscoSmallPartition

In this section, we describe approximation algorithms for $d$–DiscoSmallCover and $d$–DiscoSmallPartition. Since the optimal number of regions in a disconnected small cover is the same as that in an optimal disconnected small partition (see Observation 1.5), approximations for one problem are immediately valid for the other. Since partitioning is more relevant in practical settings such as 3D printing, we state and prove our results using partitions. We first show that the problem can be reduced to a series of 1D problems, losing a multiplicative 2-factor for each dimension. We then show that it suffices to use a simple greedy sweep algorithm to solve each 1D instance optimally. Naively, each projection can be performed in poly$(n, \text{OPT})$ time in both cases with a series of projections of section of the polygon/polyhedron into 1D. This immediately gives us a $2^{d-1}$-approximation algorithm for dimension $d$ with runtime in poly$(n, \text{OPT})^{d-1}$. A similar approach was used by Gonzalez in [15] to obtain a $2^{d-1}$-approximation for $d$–BoxCover. Our results generalize this idea to

covering continuous domains. Our goal in this section is to design efficient algorithms for the practically relevant cases of $d = 2$ and 3 (i.e., to prove Theorems 1.6 and 1.7 respectively).

▶ Remark 3.1. Consider a polyhedron $P \subseteq \mathbb{R}^d$. The regions obtained by overlaying a unit grid on $P$ is within $2^d$ of OPT – any small region intersects at most $2^d$ cells of this grid. We show here that we can produce disconnected small partition which are $2^{d-1}$ of OPT.

## 3.1 Optimal Cover into Unit Intervals

Consider a collection $\mathcal{I}$ of $n$ intervals on the $x$-axis. We allow $\mathcal{I}$ to contain intervals of two types: $[a, b]$ or $[a, b)$. A collection $\mathcal{I}_u$ of unit intervals *covers* $\mathcal{I}$ if $\bigcup_{I\in\mathcal{I}} I \subseteq \bigcup_{J\in\mathcal{I}_u} J$. We wish to compute the minimum number of unit intervals required to cover $\mathcal{I}$. We first preprocess $\mathcal{I}$ to get an equivalent instance with disjoint intervals by replacing any overlapping pair $I$ and $I'$ in $\mathcal{I}$ with the interval $I \cup I'$. Note that this preprocessing can be done in $\mathcal{O}(n \log n)$ time. We assume hereafter that $\mathcal{I}$ contains disjoint intervals. We show that a simple greedy cover of $\mathcal{I}$ (see `GreedyIntervalCover`) returns an optimal unit interval cover of $\mathcal{I}$.

We first define two subroutines for `GreedyIntervalCover` which can be easily implemented in $\mathcal{O}(\log n)$ time for intervals[†]. In the higher dimensional algorithms we describe later, we show how to implement these subroutines without explicitly constructing $\mathcal{I}$.

**(i)** INTERSECT$(s)$: Return YES if $s$ is lies in some interval of $\mathcal{I}$; return NO otherwise.
**(ii)** NEXT$(s)$: Return $s$ if INTERSECT$(s)$ returns YES; return the minimum $r \in \bigcup_{I\in\mathcal{I}} I$ where $r > s$ otherwise.

The `GreedyIntervalCover` algorithm then proceeds by placing unit intervals from left to right, using NEXT$(\cdot)$ to find the next placement. We give pseudocode in Section A.

▶ **Lemma 3.2** ($\star$). *Given a collection $\mathcal{I}$ of $n$ disjoint intervals on the $x$-axis in sorted order,* `GreedyIntervalCover` *returns an optimal unit interval cover of $\mathcal{I}$ in $\mathcal{O}(\text{OPT} \cdot \log n)$ time.*

▶ Remark 3.3. Note that intervals returned by `GreedyIntervalCover` are interior disjoint.

## 3.2 Dimension Reduction

Consider a given (possibly disconnected) region $P \subseteq \mathbb{R}^d$. For a point $p \in P$, we use $p[i]$ to denote its $i^{\text{th}}$ coordinate. Let $P_t$ be the set of points $p$ in $P$ such that $p[0] \in [t, t + 1)$.

▶ **Lemma 3.4** ($\star$). *Given a region $P \subseteq \mathbb{R}^d$ with an optimal $d$–*DISCOSMALLPARTITION *of size* OPT *and a collection of optimal $d$–*DISCOSMALLPARTITION *solutions $\mathcal{Q}_t$ for each $P_t$ with $t \in \mathbb{Z}$. The following holds:*

$$\left|\bigcup_{t\in\mathbb{Z}} \mathcal{Q}_t\right| \leq 2 \cdot \text{OPT}.$$

▶ **Lemma 3.5** ($\star$). *Consider a region $P \subseteq [0, 1]^{d-1} \times \mathbb{R}$ and let $\mathcal{I}$ denote its projection onto the $d^{th}$-axis. Let $\mathcal{I}_u$ be the optimal unit interval cover of $\mathcal{I}$ that* `GreedyIntervalCover` *returns. Then, $\{[0, 1]^{d-1} \times I \mid I \in \mathcal{I}_u\}$ is an optimal disconnected small partition of $P$.*

[†] Note: INTERSECT$(\cdot)$ does not appear directly in `GreedyIntervalCover`, but is used implicitly by NEXT$(\cdot)$.

## 3.3 A 2-Approximation for 2-DiscoSmallPartition

Consider a polygon $P \subseteq \mathbb{R}^2$. We assume that vertices are stored in order on each component of the boundary of $P$, so that the interior of $P$ is to the right of edges. Translate $P$ until the smallest $x$-coordinate of a point in $P$ is 0. We also assume that exactly one vertex $v$ of $P$ has $x(v) = 0$ for convenience. Let $L \in \mathbb{Z}_+$ be the smallest integer larger than the $x$-coordinate of all points in $P$. We define columns $C_i = [i, i+1] \times \mathbb{R}$ for $0 \leq i \leq L-1$ and, as in Lemma 3.4, let $P_i = C_i \cap P$. From Lemmas 3.2 and 3.5, we know that a greedy cover of $P_i$ is optimal, so performing greedy cover on each $P_i$ of $P$ yields a 2-approximation. However, only Lemma 3.2 is constructive, so implementing an algorithm is still needed. Specifically, we want to develop an algorithm which matches the runtime specified in Theorem 1.6. In Section 4, we implement a much simpler 2-approximation algorithm for 2–DiscoSmallCover which has theoretically worse runtime. However, as we show in that section, it performs very well in practice: it computes solutions for large polygons in fractions of a second.

We process each $P_i$ sequentially from left to right. We begin by sorting the vertices of $P$ in $x$-order; this is stored in a list $V_P$. Each vertex in $V_P$ points to its two incident edges. We are going to march through $V_P$ with a vertical sweep line $s$. We use $\text{POS}(s)$ to denote the current $x$-coordinate of $s$. We maintain two binary search trees associated with $s$: the *head tree* $T_h$ and the *tail tree* $T_t$. We maintain the vertices in $P_i$ in a list $V_i$ sorted by $y$-coordinate.

### Sweeping across $P$

The sweep line $s$ starts with $\text{POS}(s) = 0$. Add all edges incident on the left most vertex of $P$ (i.e., $V_P[0]$) to both $T_h$ and $T_t$ sorted by $y$-coordinate order as in classical line sweep algorithms – see [6] and [10, Chapter 2]. March through the vertices in $V_P$ until we reach the vertex with the largest $x$-coordinate less than 1 (note that this step might involve the sweep line not moving at all). As we march through a vertex $v$, we first insert $v$ to the vertex list $V_0$ to maintain sorted $y$ order. Then, we consider $v$'s two incident edges. If $v$ is the left endpoint of an edge $e$, then we add $e$ to $T_h$ by $y$-coordinate order. This can be achieved with a simple binary search comparing whether $v$ lies above or below each active edge in $T_h$. If $v$ is the right endpoint, then remove $e$ from both $T_h$ and $T_t$. These operations take $\log n$ time (see [6] and [10, Chapter 2]). Note the asymmetry – unlike $T_h$ where we process both edge addition and deletion, we only perform edge deletion from $T_t$ during this march.

The sweep line $s$ is now at the vertex $v$ immediately before $x = 1$ ($v$ is processed too). We update $\text{POS}(s)$ to 1. The obvious discrepancy between the location of $s$ and $\text{POS}(s)$ is intentional – while $x(v) \leq \text{POS}(s)$, there are no vertices between $x = x(v)$ and $x = \text{POS}(s)$. We will use $\text{POS}(s)$ to infer the current column we are processing.

We perform our greedy cover in this first column $C_0$. We will refer to this as *processing* $P_0$ (detailed in the following paragraphs). Once $P_0$ is processed, we update $T_t$ using $V_0$ until it reaches the same vertex as head. We delete vertices from $V_0$ as we update $T_t$. Once $T_t$ reaches $T_h$, we repeat the process of marching forward and processing the remaining columns $C_i$ for $1 \leq i < L$. See Figure 4 for an illustration. We conclude this section with a remark.

▶ Remark 3.6. When processing $P_i$, note that $T_h$ contains the set of edges that intersect $x = i + 1$ while $T_t$ contains the set of edges that intersect both $x = i$ and $x = i + 1$ (we refer to such edges as *spanning edges*).

**Processing a column.** Consider $P_i = C_i \cap P$. We have maintained one list of vertices $V_i$ and two trees ($T_h$ and $T_t$). Using this information, we will implement `GreedyIntervalCover` without explicitly constructing $\mathcal{I}$, the projection of $P_i$ on the $y$-axis, and instead provide imple-

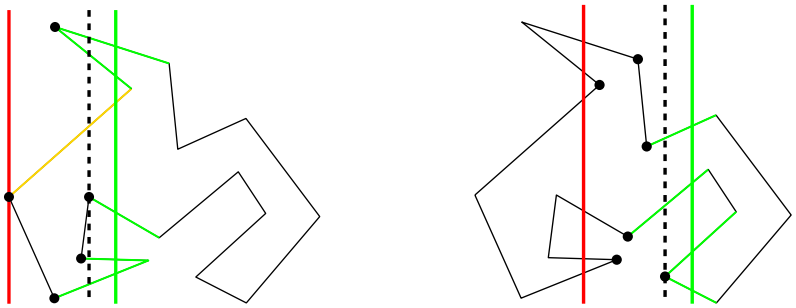

**Figure 4** The edges stored by $T_h$ are given in green, the edges stored by both $T_t$ and $T_h$ are in yellow. The heavy vertical green (resp. red) line is $x = i + 1$ (resp. $x = i$). The dotted line represents the location of the sweep line $s$ while $\text{POS}(s) = i + 1$. Marked vertices are those in $V_i$. *Left*: Status for $P_0$. *Right*: Status for $P_1$.

mentations of $\text{INTERSECT}(\cdot)$, and $\text{NEXT}(\cdot)$.

Before implementing the queries, we first perform some preprocessing. Let $n_i = |V_i|$. We know that $v \in V_i$ lies within this column $C_i$ and is incident on two edges of $P$. For each edge $e$, first compute the intersection $e \cap C_i$ and then project this onto the $y$-axis to form an interval $I_e$. We carefully define whether the right endpoint $b$ of $I_e$ is part of $I_e$: if the line $y = b$ is the right of $e$, we include $b$ in $I_e$; otherwise we do not. Let $\mathcal{I}_i$ represent the collection of all such $I_e$ over all $v \in V_i$. In $\mathcal{O}(n_i \log n_i)$ time, we construct a disjoint representation of the union of all $I_e \in \mathcal{I}_i$, marking the endpoints of each component with the edge $e$ that produced it. See Figure 5, *Far Left*. We first describe $\text{INTERSECT}(\cdot)$ and then $\text{NEXT}(\cdot)$.

▶ **Lemma 3.7** (⋆). *Given $V_i$, $T_h$, $T_t$, and $\mathcal{I}_i$, $\text{INTERSECT}(s)$ can determine if there is a point in $p \in P_i$ with $y(p) = s$ in $\mathcal{O}(\log n)$ time.*

We give the full details in Section A. Let $\ell_s$ denote the horizontal segment between $(i, s)$ and $(i + 1, s)$. To determine if $\ell_s \cap P \neq \emptyset$, we perform a binary search to check if $s$ lies in $\mathcal{I}_i$, a series of binary searches on $T_h$ and $T_t$ to check for an intersection with a edge of $P$, and finally on both $\mathcal{I}_i$ and $T_t$ to check if $\ell_s \subseteq P$. See Figure 5 for an overview of cases.

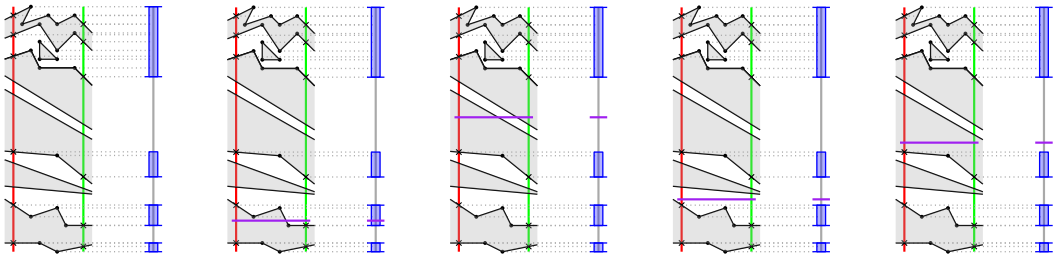

**Figure 5** Illustration of cases in Lemma 3.7. The lines $x = i + 1$ and $x = i$ are in green and red respectively. $P_i$ is shown on the left of each image; gray regions lie outside $P_i$. In the right of each image is $\mathcal{I}_i$ in blue. The query segment $\ell_s$ is given in purple. *Far Left*: The construction of $\mathcal{I}_i$. Note that the third interval is partially open. *Middle Left*: $s$ lies in $\mathcal{I}_i$. *Middle*: $\ell_s$ intersects a spanning edge of $P_i$. *Middle Right*: $\ell_s \subseteq P_i$. *Far Right*: $\ell_s \subseteq \mathbb{R}^2 \setminus P_i$.

▶ **Lemma 3.8.** *Given $V$, $T_h$, $T_t$, and $\mathcal{I}_i$, $\text{NEXT}(s)$ can return $s$ if $\text{INTERSECT}(s)$ returns* YES *or the $y$-coordinate of the lowest point of $P_i$ that lies strictly above $\ell_s$ in $\mathcal{O}(\log n)$ time.*

**Proof.** First, perform INTERSECT($s$); this takes $\mathcal{O}(\log n)$ time by Lemma 3.7. If INTERSECT($s$) returns YES, NEXT($s$) returns $s$. Otherwise, we need to search for the point $r$, the lowest point of $P_i$ which lies above $\ell_s$. Clearly, $r$ is either (i) a $v \in V_i$; or (ii) the intersection of an edge incident on a $v \in V_i$ with $x = i$ or $x = i + 1$; or (iii) an intersection of a spanning edge of $P$ with $x = i$ or $x = i + 1$.

The edge of $\ell_s$ above it in $T_t$ determines the lowest intersection of form (iii). For (i) and (ii), we use $\mathcal{I}_i$: using binary search we find the endpoint in $\mathcal{I}_i$ which is immediately above $s$. Return $r$ as the lower $y$-coordinate among these two. Since $\ell_s$ is not in $P_i$ (we are in the case where INTERSECT($s$) returns NO), there is point in $P_i$ immediately above $\ell_r$. ◀

With these two functions implemented, we use `GreedyIntervalCover` to process $P_i$. Continue onto the next column $C_{i+1}$ and repeat until all of $P$ has been covered.

**Correctness and runtime analysis.** Since we have implemented INTERSECT($\cdot$) and NEXT($\cdot$) correctly (see Lemmas 3.7 and 3.8), `GreedyIntervalCover` returns the optimal set of unit intervals covering the $y$-projection of $P_i$ (see Lemma 3.2) for each $P_i$ with $0 \le i \le L - 1$. Using Lemma 3.5, these unit intervals are converted to an optimal disconnected small partition of each $P_i$; Lemma 3.4 shows that the collection of these small regions is a 2-approximation for $P$. We are just left with showing that the desired runtime given in Theorem 1.6 is met.

▶ **Observation 3.9 ($\star$).** *The total amortized work of marching the sweepline and maintaining $T_h$ and $T_t$ is $\mathcal{O}(n \log n)$.*

▶ **Observation 3.10 ($\star$).** *The total amortized work of maintaining $\mathcal{I}_i$ and $V_i$ is $\mathcal{O}(n \log n)$.*

We can now prove Theorem 1.6. Since NEXT($\cdot$) takes $\mathcal{O}(\log n)$ time for each call, `GreedyIntervalCover` takes $\mathcal{O}(\text{OPT}_i \cdot \log n)$ time in each $P_i$ where $\text{OPT}_i$ denotes the size of an optimal disconnected small partition of $P_i$. As $\sum_{i=0}^{L-1} \text{OPT}_i \le 2 \cdot \text{OPT}$, the total time taken by `GreedyIntervalCover` is $\mathcal{O}(\text{OPT} \cdot \log n)$. From Observations 3.9 and 3.10, we infer that we require $\mathcal{O}(n \cdot \log n)$ time to maintain all required data structures. This gives us an overall runtime of $\mathcal{O}((n + \text{OPT}) \log n)$ as required.

## 3.4 A 4-Approximation for 3-DiscoSmallPart

Consider a polyhedron $P \subseteq \mathbb{R}^3$ with triangulated faces. We assume that each face is oriented so that its normal points towards the interior of $P$. We let $f$ denote the number of faces of $P$. In this section, we prove Theorem 1.7: we describe a 4-approximation algorithm for 3–DiscoSmallPartition that runs in $\mathcal{O}(f \cdot \text{OPT} \log f)$ time. Let $L \in \mathbb{Z}_+$ be the smallest integer larger than $x(p)$ and $y(p)$ over all $p \in P$. We will perform the dimension reduction of Lemma 3.4 twice, once in the $x$ direction to get a set of polyhedral domains $P_i$ for all $0 \le i < L$, and once again on each $P_i$ in the $y$ direction to obtain a set of $P_{i,j}$ for all $0 \le j < L$. Note that this induces a $L \times L$ grid on the $xy$-plane. It is crucial that we only consider this grid implicitly since it can be as large as $\mathcal{O}(\text{OPT}^2)$. Therefore, we will only devote a piece of memory to square $(i, j)$ if there is at least one point of $P$ inside the corresponding column $C_{ij} := [i, i+1] \times [j, j+1] \times \mathbb{R}$. A column $C_{i'j'}$ is a *neighbor* of $C_{ij}$ if its corresponding square $(i', j')$ is in the 8-neighborhood of $(i, j)$. To find the non-empty $P_{i,j}$, we consider compiling a set of faces which touch each column $C_{i,j}$. For $C_{i,j} \cap P = P_{i,j} \neq \emptyset$, we have $C_{i,j} \cap \partial P \neq \emptyset$, so it suffices to check if faces of $P$ intersect $C_{i,j}$. We call a $C_{ij}$ *active* if $P_{ij} \neq \emptyset$.

▶ **Lemma 3.11.** *Computing the set of active columns, along with the collection of faces that intersect each such column can be done in $\mathcal{O}(f \cdot \text{OPT})$ time.*

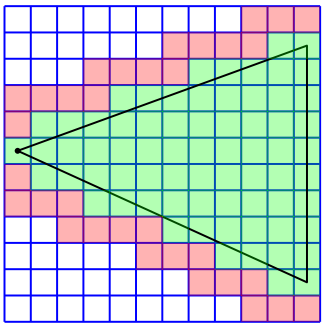

**Figure 6** Green cells are the columns $C_{i,j}$ containing points of the projected face. Red cells are checked by BFS but do not contain points of the projected face, thus causing BFS to terminate.

**Proof.** We iterate through all $f$ faces of $P$, projecting a face $F$ to the $xy$-plane, then checking off all $C_{i,j}$ with non-empty intersection with $F$. Since all faces are triangles, we can simply perform an $\mathcal{O}(1)$ check for intersection with each column of interest, adding $F$ to its ongoing list of intersecting faces or starting a new list if $C_{i,j}$ has not yet been added as active.

We now show how to avoid checking for an intersection against all $C_{i,j}$ (this would incur $\mathcal{O}(\text{OPT}^2)$ time per face). Let $F_{xy}$ denote the projection of $F$ onto the $xy$-plane – note that $F_{xy}$ is a (possibly degenerate) triangle. We start with the column containing the top-left corner of $F_{xy}$ and then check if its neighbors intersect $F_{xy}$. We proceed by performing a breadth-first search (BFS) on the columns, only advancing with a column's neighbors if there is non-empty intersection. This often referred to as a flood fill algorithm. Since $F_{xy}$ is a triangle, the set of all columns intersecting $F$ must be connected, and thus this search suffices to find all columns that intersect $F$. Since each column has at most 8 neighbors, we touch at most 9 times as many columns as those that contain points of $f$; therefore $\mathcal{O}(\text{OPT})$ columns. See Figure 6. ◀

From Lemma 3.11, we can compute the set of active columns, and for every such column, the list of faces that intersect it in $\mathcal{O}(f \cdot \text{OPT})$ time. We cover these columns independently using `GreedyIntervalCover`, and take the union of the column covers as a cover of $P$. This gives us the required approximation guarantee for Theorem 1.7 from Lemmas 3.4 and 3.5. For its required runtime, our goal is to perform $\mathcal{O}(f \cdot \text{OPT} \log f)$ work overall in our invocations of `GreedyIntervalCover`. More specifically, we want to perform $\mathcal{O}(f \log f)$ work in each of our unit columns, which will total $\mathcal{O}(f \cdot \text{OPT} \log f)$ work since we only have $\mathcal{O}(\text{OPT})$ active columns. We show how to achieve this as the last result in this section.

▶ **Lemma 3.12.** *For the intersection of a polyhedron $P$ with a column $C_{i,j}$, computing an equivalent instance of* 1–DiscoSmallPartition *can be done in $\mathcal{O}(f \log f)$ time.*

**Proof.** Consider $P_{i,j} = C_{i,j} \cap P$. For each face $F$ associated with $C_{i,j}$, we compute $I_F$, the projection of each face $F \cap C_{i,j}$ onto the $z$-axis. In $\mathcal{O}(f \log f)$ time, we compute the disjoint representation of the union of these intervals as in Section 3.3. Let $\mathcal{I}'_{i,j}$ denote this collection of intervals with endpoints stored in sorted order. We can use $\mathcal{I}'_{i,j}$ to effectively determine if a given plane $z = s$ intersects $\partial P$ in $P_{i,j}$; we are left with the case where $z = s$ does not intersect $\partial P$ in $P_{i,j}$.

We march through the endpoints in $\mathcal{I}'_{i,j}$. Note that each endpoint is associated with some face of $P$. If the face associated with the top endpoint $t$ of an interval indicates that the points above it are in $P$ (i.e., its normal has a positive $z$ coefficient), we add the section from $t$ to the next bottom endpoint $b$ to $\mathcal{I}'_{i,j}$: i.e., we add $[b, t]$ to $\mathcal{I}'_{i,j}$. After completing this

march through the endpoints of $\mathcal{I}'_{i,j}$, we call the resulting collection of intervals $\mathcal{I}_{i,j}$. $\mathcal{I}_{i,j}$ is the required equivalent instance of 1–DiscoSmallPartition. ◀

▶ Remark 3.13. Constructing the equivalent collection of intervals in Lemma 3.12 was much simpler than our algorithm for 2–DiscoSmallPartition (specifically, designing intersect$(\cdot)$ in Lemma 3.7) since we are allowed $\mathcal{O}(f \log f)$ amortized time in each column (as opposed to only $\mathcal{O}(\log n)$ time in Lemma 3.7). This allows us to charge *spanning faces* (the faces of $P$ which contain no vertices the column) of each column to each such column.

# 4 Experimental Results

In this section, we describe our implementations of the approximation algorithms for SmallCover and 2–DiscoSmallPartition. Our implementation closely follows the descriptions of previous sections, but should be seen as a proof-of-concept that the presented algorithms are indeed implementable without major hidden challenges. No highly optimized geometric data structures were used, and geometric operations were performed using standard libraries. We used Python 3.11 for our implementation, `shapely` [14] for geometric operations and CP-SAT from Google's OR-Tools [20, 17] for solving constraint programming formulations. All experiments were performed on Linux workstations equipped with AMD Ryzen 9 5950X CPUs with 16 cores/32 threads and 128 GB of RAM running Ubuntu 24.04.3. Code and data are made available via HotCRP. Instances were taken from the Salzburg Database of Geometric Inputs [11]; see Figure 8. For each polygon, two sizes (of pieces) were chosen based on the area of the polygon such that a reasonable number of pieces is required to cover the polygon. We define the *relative area* of an instance to be the ratio area/(piece size)$^2$. We used relative areas of 10 and 50, for a total of 378 instances. We first describe an exact primal-dual approach based on constraint programming which provides us lower bounds.

## 4.1 An Exact Primal-Dual Approach

We compare the solutions produced by our algorithms with lower bounds obtained from a CP-SAT formulation of a set cover problem. The universe consists of point witnesses inside the polygon, and the objective is to cover these witnesses with the minimum number of small pieces. Each witness imposes the constraint that at least one of the pieces covering it must be selected. Because the space of possible pieces is infinite, we construct a finite set of candidate pieces that suffices to represent an optimal solution. For 2–DiscoSmallPartition, these candidates can be taken as maximally sized axis-aligned squares (temporarily ignoring that they will later be intersected with the polygon). The additional connectivity constraint in SmallCover makes the selection of candidates substantially more challenging; developing a principled method to generate a sufficient candidate set for SmallCover is an interesting direction for future work. Here, we generate lower bounds for 2–DiscoSmallPartition and therefore, for SmallCover.

Our set cover formulation is as follows: given a polygon $P$, a set of witness points $W \subseteq P$ inside $P$ and a set of candidate pieces $\mathcal{Q}$, find the smallest subset of $\mathcal{Q}$ that covers all points in $W$. We minimize $\sum_{Q \in \mathcal{Q}} x_Q$ with boolean variables $x_Q$ for each candidate piece under constraints $\bigvee_{Q \in \mathcal{Q}: w \in Q} x_Q$ for each witness point $w \in W$. This resembles a standard set cover formulation, which can be solved very efficiently for modest problem sizes using CP-SAT. The correctness of this formulation requires a careful choice of the unit squares $\mathcal{Q}$. A square covering multiple witness points can either be pinned by a single point in one of its corners, or by two points on its boundary (each touching either a vertical or horizontal edge). Thus,

we generate candidate squares by iterating over all points in $W$ and generating squares with that point in each corner, as well as iterating over all pairs of close points and generating squares defined by those points on their boundary. This results in $\mathcal{O}(|W|^2)$ candidate squares, which is manageable for moderate sizes of $W$. Our implementation starts with an initial set of witnesses equally spaced within the polygon and iteratively refines the witness set by adding points in areas not covered by the current best solution until either no uncovered area remains or a maximum number of witnesses is reached. Witnesses are added by sampling points in uncovered areas using a uniform grid with a resolution proportional to the piece size. Once the witness set exceeds the given maximum size, we prune away witnesses that do not define squares in the current solution to reduce the problem size and make room for new witnesses in uncovered areas. Whenever a new solution is found, we try to convert it into a feasible solution by greedily covering the uncovered area with additional pieces until no uncovered area remains, see Figure 10 for an example. We call this method `DualSolver`.

## 4.2 Approximation Algorithms and Heuristic Approaches

The first two algorithms, `Con-GridAPX` and `GridAPX`, place an axis-aligned grid of the desired resolution over the polygon's bounding box (optionally using several fractional offsets). The number of offsets, denoted by $k$, is shown in the plots below. For each grid cell whose intersection with the polygon has non-negligible area, we include the intersection as a piece; we repeat this for all offsets and return the offset that yields the fewest pieces. For SmallCover, a single intersection may yield multiple pieces if the intersection is disconnected. This approach is simple, robust, and easy to parallelize, but it can generate many pieces when the grid is poorly aligned and it requires a geometric intersection computation for every relevant grid cell. Its running time is therefore dominated by the number of generated cells and intersection tests. By Remark 3.1, this is a 4-approximation for 2–DiscoSmallPartition.

`Con-4APX` implements the approach from Section 2 for SmallCover. The algorithm begins by greedily placing the largest possible square pieces along the polygon boundary until no further boundary-anchored pieces fit. It then repeatedly selects the bottom-leftmost point $p$ of the remaining uncovered region and places two squares with $p$ as their bottom-left and top-left corners, respectively. This process continues until the region is fully covered. While conceptually simple, the method requires efficiently finding the largest square anchored to the boundary and the largest squares realizable with a given point as a corner.

`ColumnAPX` partitions the polygon into vertical columns and covers each column optimally by repeatedly placing maximal column-aligned squares from bottom to top, removing each placed square from the column's uncovered region. In the worst case, this yields $\mathcal{O}(n \cdot \mathrm{OPT})$ geometric intersection operations. As shown in Section 3.3, the method is a 2-approximation for 2–DiscoSmallPartition. With a sweep-line strategy it can be implemented in $\mathcal{O}((n + \mathrm{OPT}) \log n)$ time; see Theorem 1.6. Because that implementation requires exact geometric primitives (see, for example, [21]) and line-intersection computations, it is beyond the scope of this work. Nonetheless, even a naive implementation performs well in practice, while remaining easy to understand and requiring only a few lines of code; a desirable property in practical applications.

## 4.3 Results

We implemented all algorithms described above and ran them on the selected set of 378 polygonal instances. For the `DualSolver`, we added 5 new witnesses per iteration and limited the maximum number of witnesses to 750. Our initial witness set consisted of points

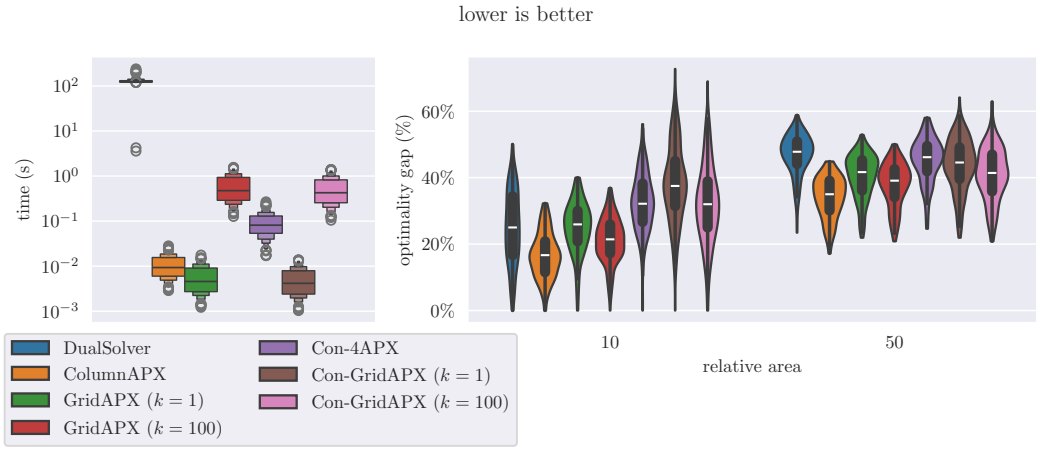


**Figure 7** *Left:* Run times of the different algorithms. Naturally, the exact solver (`DualSolver`) takes significantly longer than the approximation algorithms, which ran into a timeout after 120 s. *Right:* Optimality gaps of the different solvers. Notably, `ColumnAPX` performs best on these instances, often producing solutions within 40% of optimal.

that were placed on the boundary of pieces in `GridAPX` to ensure an even distribution in the beginning. As can be seen in Figure 7 all approximation algorithms run within a few seconds, while `DualSolver` frequently hits the timeout of 120 s. Regarding solution quality, `DualSolver` is suprisingly ineffective at finding primal solutions (i.e. feasible covers). Only a small number of instances are solved to optimality within the time limit, see Figure 8 for an example of an optimal solution. Solutions often have tiny uncovered slivers that are hard to cover with few additional pieces, see Figure 8 for examples. `GridAPX` has been tested with a single grid offset as well as with $k = 100$ offsets and shows only moderate improvements when using multiple offsets. `ColumnAPX` consistently outperforms `GridAPX` and often finds solutions within 40% of optimal. In general, `Con-4APX` performs very well in practice, often producing solutions between 40% and 50% of the optimal solution. Despite the non-linear implementation, `ColumnAPX` is reasonably fast for most instances with runtimes that are comparable to the simple `GridAPX`.

## References


1 Anders Aamand, Mikkel Abrahamsen, Reilly Browne, Mayank Goswami, Prahlad Narasimhan Kasthurirangan, Linda Kleist, Joseph S. B. Mitchell, Valentin Polishchuk, and Jack Stade. Covering and partitioning complex objects with small pieces. In *42nd International Symposium on Computational Geometry (SoCG 2026)*, 2026. To appear.

2 Mikkel Abrahamsen and Dan Halperin. Ten problems in geobotics, 2024. URL: `https://arxiv.org/abs/2408.12657`, `arXiv:2408.12657`.

3 Mikkel Abrahamsen and Nichlas Langhoff Rasmussen. Partitioning a polygon into small pieces. In *SODA 2025*. 2025. `doi:10.48550/ARXIV.2211.01359`.

4 Boris Aronov, Esther Ezra, and Micha Sharir. Small-Size $\eps$-Nets for Axis-Parallel Rectangles and Boxes. *SIAM Journal on Computing*, 39(7):3248–3282, January 2010. Publisher: Society for Industrial and Applied Mathematics. URL: `https://epubs.siam.org/doi/10.1137/090762968`, `doi:10.1137/090762968`.

5 Manjanna Basappa, Rashmisnata Acharyya, and Gautam K. Das. Unit disk cover problem in 2D. *Journal of Discrete Algorithms*, 33:193–201, July 2015. URL: `https://www.sciencedirect.com/science/article/pii/S1570866715000556`, `doi:10.1016/j.jda.2015.05.002`.

6 Bentley and Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979. doi:10.1109/TC.1979.1675432.

7 J.-D. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec. Voronoi Diagrams in Higher Dimensions under Certain Polyhedral Distance Functions. *Discrete & Computational Geometry*, 19(4):485–519, April 1998. doi:10.1007/PL00009366.

8 Timothy M. Chan and Nan Hu. Geometric red–blue set cover for unit squares and related problems. *Computational Geometry*, 48(5):380–385, July 2015. URL: https://www.sciencedirect.com/science/article/pii/S0925772114001308, doi:10.1016/j.comgeo.2014.12.005.

9 Claudio Contardo and Alain Hertz. An exact algorithm for a class of geometric set-cover problems. *Discrete Applied Mathematics*, 300:25–35, September 2021. URL: https://www.sciencedirect.com/science/article/pii/S0166218X21001864, doi:10.1016/j.dam.2021.05.005.

10 Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Heidelberg, 2008. URL: http://link.springer.com/10.1007/978-3-540-77974-2, doi:10.1007/978-3-540-77974-2.

11 Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, and Peter Palfrader. Salzburg database of polygonal data: Polygons and their generators. *Data in Brief*, 31:105984, 2020. doi:10.1016/j.dib.2020.105984.

12 Thomas Erlebach and Erik Jan van Leeuwen. PTAS for Weighted Set Cover on Unit Squares. In Maria Serna, Ronen Shaltiel, Klaus Jansen, and José Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 166–177, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15369-3_13.

13 Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inf. Process. Lett.*, 12(3):133–137, 1981. doi:10.1016/0020-0190(81)90111-3.

14 Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, et al. Shapely, December 2022. doi:10.5281/zenodo.7428463.

15 Teofilo F. Gonzalez. Covering a set of points in multidimensional space. *Information Processing Letters*, 40(4):181–188, November 1991. URL: https://www.sciencedirect.com/science/article/pii/002001909190075S, doi:10.1016/0020-0190(91)90075-S.

16 Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *J. ACM*, 32(1):130–136, January 1985. URL: https://dl.acm.org/doi/10.1145/2455.214106, doi:10.1145/2455.214106.

17 Dominik Krupke, Leon Lan, Michael Perk, et al. The CP-SAT Primer: Using and Understanding Google OR-Tools' CP-SAT Solver. URL: https://github.com/d-krupke/cpsat-primer.

18 Zedi Lu, Kanxin Hu, and Tsan Sheng Ng. Improving Additive Manufacturing production planning: A sub-second pixel-based packing algorithm. *Computers & Industrial Engineering*, 181:109318, July 2023. URL: https://www.sciencedirect.com/science/article/pii/S036083522300342X, doi:10.1016/j.cie.2023.109318.

19 Yosep Oh, Paul Witherell, Yan Lu, and Timothy Sprock. Nesting and scheduling problems for additive manufacturing: A taxonomy and review. *Additive Manufacturing*, 36:101492, December 2020. URL: https://www.sciencedirect.com/science/article/pii/S2214860420308642, doi:10.1016/j.addma.2020.101492.

20 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

21 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.1 edition, 2025. URL: https://doc.cgal.org/6.1/Manual/packages.html.

22 Yizhe Yang, Haochen Li, Kexin Zhang, Xinjian Jia, Gong Wang, and Bingshan Liu. A 3D nesting method based on the convex-concave coding similarity of the voxelized model for additive manufacturing. *Additive Manufacturing*, 64:103429, February 2023. URL: https://www.sciencedirect.com/science/article/pii/S2214860423000428, doi:10.1016/j.addma.2023.103429.


**Figure 8** A selection of solutions produced by the algorithms. The leftmost plot shows the lower bound produced by `DualSolver` with witness points in blue and uncovered areas marked in red.

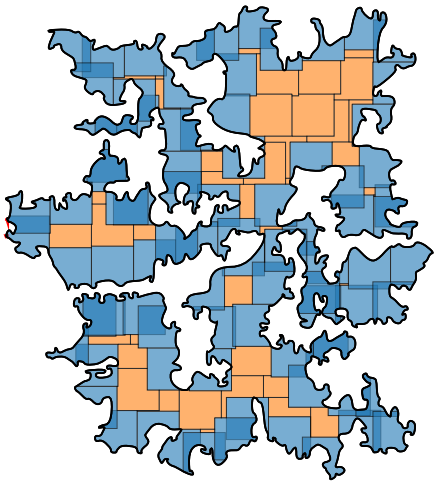

**Figure 9** Example solution of the 4-approximation algorithm for SmallCover. The interior and boundary pieces are shown in different colors.

# A Supplemental Material for Section 3

**Algorithm 1** Greedy Interval Cover

**Input:** $\mathcal{I}$ is collection of disjoint unit intervals given in sorted order
**Output:** An optimal partition of $\mathcal{I}$ into unit intervals
$\mathcal{I}_u \leftarrow \emptyset$
$s \leftarrow$ the first endpoint of $\mathcal{I}$
$t \leftarrow$ the last endpoint of $\mathcal{I}$
**while** $s < t$ **do**
    $s \leftarrow \text{NEXT}(s)$
    $\mathcal{I}_u \leftarrow \mathcal{I}_u \cup \{[s, s+1]\}$
    $s \leftarrow s + 1$
**end while**
**return** $\mathcal{I}_u$

▶ **Lemma 3.2** (⋆). *Given a collection $\mathcal{I}$ of $n$ disjoint intervals on the $x$-axis in sorted order,* `GreedyIntervalCover` *returns an optimal unit interval cover of $\mathcal{I}$ in $\mathcal{O}(\text{OPT} \cdot \log n)$ time.*

**Proof.** Consider $W = \{a \mid [a, b] \in \mathcal{I}_u\}$, the distance between any two points in $W$ is at least one by construction and therefore they cannot be covered by the same unit interval (i.e., $W$ is a witness set). Therefore, $|\mathcal{I}_u| = |W| \leq \text{OPT} \leq |\mathcal{I}_u|$; this proves that $\mathcal{I}_u$ is optimal. Since $\text{NEXT}(\cdot)$ takes $\log n$ time, the runtime of $\mathcal{O}(\text{OPT} \cdot \log n)$ follows. ◀

▶ **Lemma 3.4** (⋆). *Given a region $P \subseteq \mathbb{R}^d$ with an optimal $d$–DiscoSmallPartition of size* OPT *and a collection of optimal $d$–DiscoSmallPartition solutions $\mathcal{Q}_t$ for each $P_t$ with $t \in \mathbb{Z}$. The following holds:*

$$\left| \bigcup_{t \in \mathbb{Z}} \mathcal{Q}_t \right| \leq 2 \cdot \text{OPT}.$$

**Proof.** Consider an optimal disconnected small partition $\mathcal{Q}'$ of $P$ and a small region $Q' \in \mathcal{Q}'$. Since $Q'$ lies within a unit hypercube, $Q'$ intersects at most 2 regions in $\{P_t \mid t \in \mathbb{Z}\}$. If we split each $Q' \in \mathcal{Q}'$ by the hyperplane defined by $p[0] = t$ for all $t \in \mathbb{Z}$, we have a valid set of solutions to each $P_t$ of total size at most $2 \cdot \text{OPT}$. Since each $\mathcal{Q}_t$ is optimal, $\bigcup_{t\in\mathbb{Z}} \mathcal{Q}_t$ must use at most $2 \cdot \text{OPT}$ small regions. ◀

▶ **Lemma 3.5** (⋆). *Consider a region $P \subseteq [0,1]^{d-1} \times \mathbb{R}$ and let $\mathcal{I}$ denote its projection onto the $d^{th}$-axis. Let $\mathcal{I}_u$ be the optimal unit interval cover of $\mathcal{I}$ that* `GreedyIntervalCover` *returns. Then, $\{[0,1]^{d-1} \times I \mid I \in \mathcal{I}_u\}$ is an optimal disconnected small partition of $P$.*

**Proof.** Consider an optimal disconnected small partition $\mathcal{Q}$ of $P$ and let $\mathcal{I}_u$ denote the optimal unit interval cover of $\mathcal{I}$ that Algorithm 1 returns. For each $I \in \mathcal{I}_u$, consider the small region $([0,1]^{d-1} \times I) \cap P$. Using Remark 3.3, we infer that the collection $\mathcal{Q}_\mathcal{I}$ of these regions is a disconnected small partition of $P$; so $|\mathcal{Q}| \leq |\mathcal{Q}_\mathcal{I}| = |\mathcal{I}_u|$. On the other hand, let $I_Q$ denote the projected interval of $Q \in \mathcal{Q}$ onto the $d^{\text{th}}$-axis. Replace $I_Q$ with a unit interval that covers it (since $Q$ is a small region, this is indeed possible). The collection $\mathcal{I}_\mathcal{Q} = \{I_Q \mid Q \in \mathcal{Q}\}$ is now a unit interval cover of $\mathcal{I}$. This shows that $|\mathcal{I}_u| \leq |\mathcal{I}_\mathcal{Q}| = |\mathcal{Q}|$. This proves that $\mathcal{Q}_\mathcal{I}$ is an optimal disconnected small partition of $P$ as required. ◀

▶ **Lemma 3.7** (⋆). *Given $V_i$, $T_h$, $T_t$, and $\mathcal{I}_i$,* INTERSECT$(s)$ *can determine if there is a point in $p \in P_i$ with $y(p) = s$ in $\mathcal{O}(\log n)$ time.*

**Proof.** Let $\ell_s$ denote the line segment $[i, i+1] \times \{s\}$ (i.e., the horizontal segment through $s$ bounded by $C_i$). To determine if $\ell_s$ intersects $P_i$, we dive into three cases; INTERSECT$(s)$goes through these cases in the given order.

▶ Case A.0.1 ($s$ lies in $\mathcal{I}_i$). With the endpoints of each component of $\mathcal{I}_i$ in sorted order, we can determine whether $s$ lies in a semiclosed interval of $\mathcal{I}_i$ or not with binary search in $\mathcal{O}(\log n_i)$ time. If this is true, INTERSECT$(s)$returns YES, since $s$ intersects an edge of $P_i$. See Figure 5, *Middle Left*. ⌟

▶ Case A.0.2 ($\ell_s$ intersects a spanning edge $e$ of $P_i$). If $e$ is not incident on a vertex in $C_i$, $e$ appears in $T_t$. Perform binary search on $T_t$ to determine where the point $(i, s)$ lies. The point either lies between two edges in $T_t$ or above (resp. below) all its edges; let $e$ be highest (resp. lowest) edge in this case. We refer to these as *flanking edges*, plural, for brevity. These flanking edges can be computed in $\mathcal{O}(\log n)$ time. In constant time, we determine if they intersect $\ell_s$[‡]; INTERSECT$(s)$returns YES if either do. See Figure 5, *Middle*. ⌟

Cases A.0.1 and A.0.2 effectively determine if $\ell_s$ intersects an edge of $P$. This leaves only two possibilities: either $\ell_s$ lies completely inside $P$ or completely outside $P$.

▶ Case A.0.3 ($\ell_s$ does not intersect any edge of $P$). Since $\ell_s \subseteq P$ or $\ell_s \subseteq \mathbb{R}^2 \setminus P$, it suffices to check whether $(i+1, s) \in P_i$. Consider its flanking edges in $T_h$ as in Case A.0.2. Let $e$ be a flanking edge of $(i+1, s)$ in $T_h$. Recall that the edges of $P$ are oriented so that their right points towards $P$; INTERSECT$(s)$returns YES if $(i+1, s)$ is to the right of $e$, otherwise it returns NO. Since flanking edges can be found in $\mathcal{O}(\log n)$ time, this case only requires $\mathcal{O}(\log n)$ time. See Figure 5, *Middle Right* and *Far Right*. ⌟

Our cases are exhaustive and require $\mathcal{O}(\log n)$ time each to process; the lemma follows. ◀

▶ **Observation 3.9** (⋆). *The total amortized work of marching the sweepline and maintaining $T_h$ and $T_t$ is $\mathcal{O}(n \log n)$.*

[‡] When this intersection is at $(i, s)$ we return YES if, and only if, $\ell_s$ lies to the right of the edge.

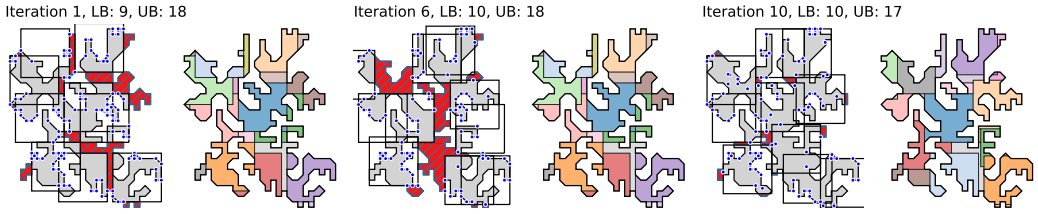


**Figure 10** Iterations of the primal dual approach. The figure shows the upper bound and lower bound maintained by the algorithm after each iteration.

**Proof.** This follows immediately from the complexity of well-known line sweep algorithms – see [6] and [10, Chapter 2]. ◀

▶ **Observation 3.10 (⋆).** *The total amortized work of maintaining $\mathcal{I}_i$ and $V_i$ is $\mathcal{O}(n \log n)$.*

**Proof.** Constructing $V_i$ and $\mathcal{I}_i$ in each $P_i$ takes $\mathcal{O}(n_i \log n_i)$ time where $n_i$ is the number of vertices of $P$ in $P_i$. Since a vertex of $P$ is in exactly one $P_i$, the overall runtime follows. ◀