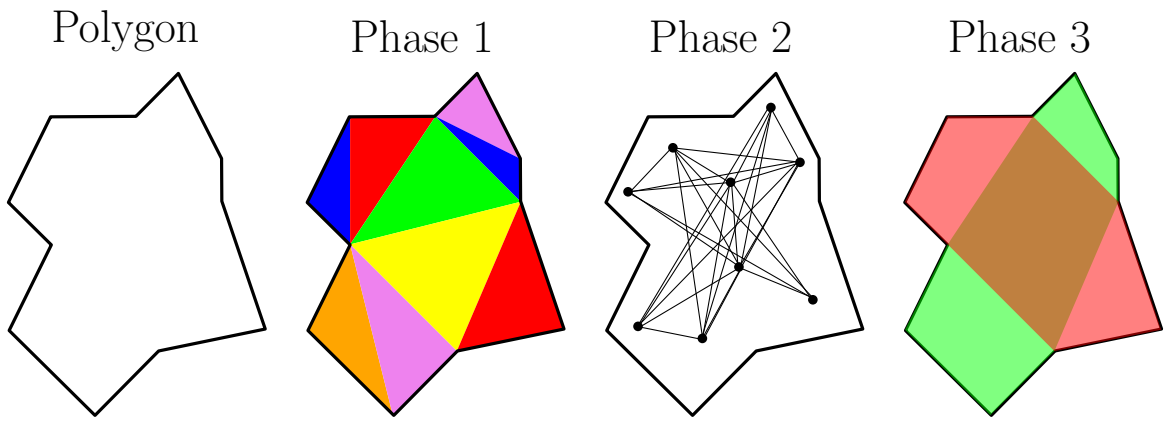
Efficient Construction of Convex Covers via Clique Covers

Bachelor Thesis

Department of Computer Science, University of Copenhagen

William Bille Meyling
Supervised by Mikkel Abrahamsen and André Nusser*

April 2023



Abstract

In the *Minimum Convex Cover* problem we are given a polygon (with holes) and we should find a minimum-cardinality set of convex pieces that cover the polygon. The problem has been shown to be $\exists\mathbb{R}$-complete. In practice we therefore seek small *Convex Covers*. In this thesis we present a heuristic algorithm for doing exactly this. The algorithm was developed as the winning solution to the CG:SHOP 2023 Challenge. The algorithm consists of three phases. In phase one, we compute a triangulation of the polygon. In phase two, we build a visibility graph on the triangulation, where edges exist between pairs of triangles if the triangles fully see each other within the polygon. In phase three, a small clique cover is found in the visibility graph. For most of the cliques, taking the convex hull of the triangles within the clique, corresponds to a valid convex piece in the resulting convex cover. We do thorough benchmarks of the algorithm, and empirically show how to configure the algorithm for different types of polygons. In general, the best results are achieved using constrained Delaunay triangulations with constraints being visibility lines within the polygon that coincide with edges of the polygon. We show how to compute the visibility graphs with an output sensitive exploration of the dual of the triangulation and with very efficient tests to check if two triangles fully see each other. The resulting algorithm is able to compute very small convex covers of polygons with more than 100000 vertices.

*Mikkel Abrahamsen is the official supervisor

Contents

1 Introduction

In the *Minimum Convex Cover* (MCC) problem we are given a polygon with holes, P, and we need to find a minimum-cardinality set of convex polygons, S, such that the union of S equals P. It was recently shown by Abrahamsen [1] that the problem is $\exists\mathbb{R}$-complete. However, there does exist polynomial-time approximation algorithms for MCC. Eidenbenz and Widmayer [4] shows how to compute a $O(\log n)$-factor approximation[1], however since it runs time $O(n^{29} \log n)$ it is impractical. Since even approximating solutions is difficult to do efficiently with good guarantees, we will instead attempt to find small-cardinality convex covers using a heuristic algorithm.

The algorithm we present in this thesis was developed as part of the participation of team `DIKU(AMW)` in the CG:SHOP 2023 Challenge (`https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2023/#problem-description`), which is the fifth Computational Geometry Challenge of its kind. Therefore we needed the algorithm to be very efficient. In the CG:SHOP Challenges participating teams are presented with geometric optimization problems. This year the Challenge was to compute small convex covers of given polygons. For an overview of the Challenge and the results a survey is available[5]. In this thesis, we describe the approach used by team, `DIKU(AMW)`, to find convex covers of the 206 Challenge instances (polygons). After a close race, team `DIKU(AMW)` ended up as the first place winner of the Challenge against 21 other teams. Team `DIKU(AMW)` consisted of Mikkel Abrahamsen, William Bille Meyling and André Nusser. William did all the programming and is the author of this thesis. The implementation is available at github: `https://github.com/willthbill/ExtensionCC`.

We used a three-phase approach. In phase one, we compute a triangulation, $\mathcal{T}$, of P. In phase two we then construct a visibility graph, $G = (\mathcal{T}, E)$, with a unique node representing each triangle of $\mathcal{T}$. It should hold that if there is an edge between two nodes then the corresponding two triangles are fully visible from each other within P. In phase three we compute a clique cover of G. We want the convex hull of the triangles in a clique to be a piece in the resulting convex cover. However, it is not guaranteed that this convex hull is contained in P. In that case we split these cliques into smaller cliques, and get a clique cover corresponding to a valid convex cover. Afterwards, we do a post-processing that removes convex polygons from the solution such that we get a minimal convex cover.

A short paper describing the approaches and results of team `DIKU(AMW)` was already written for the SoCG 2023 proceedings (`https://cs.utdallas.edu/SOCG23/challenge.html`). Some figures and small pieces of text appear in both this thesis and in the short paper. The paper can be found in appendix 7.1. This thesis will work as an extended and detailed report of the approaches developed by team `DIKU(AMW)` during the Challenge. The results, the implementation and the data referred to in this thesis is also (mostly) from the Challenge. Thus, this thesis is not meant to be a finalized research result, but instead a source of inspiration for further research.

In section 3 we present the overall algorithm without diving into all the underlying algorithmic optimizations. These optimizations are instead presented in section 4. Section 3 provides the conceptual ideas and can be read without reading section 4. In section 5 we present the results of the Challenge and some benchmarks performed after the end of the Challenge.

2 Theory and preliminaries

In this section relevant theory, definitions and notation is presented such that later sections will be unambiguous.

2.1 Visibility and containment

Given a set of points, S, we say that it is *inside* (or *contained in*) another set of points, T, if $S \subseteq T$ and *outside* if S and T are disjoint. S is *partially contained* in T if it is neither inside nor outside. Two points p and q see each other within T if line-segment $\overline{pq}$ is contained in T. For two sets of points, S_1 and S_2, we say that — in the context of T — S_2 is *fully visible* from S_1 iff. for every pair of points, $p_1 \in S_1$ and $p_2 \in S_2$, p_1 and p_2 see each other. In this case we also say that S_1 and S_2 *fully see* each other. S_2 is *partially visible* from S_1 iff. for every point, $p_1 \in S_1$, there exist point, $p_2 \in S_2$, s.t. p_1 fully sees p_2. In this case we also say that S_1 *partially sees* S_2.

[1] n being the number of vertices of P

2.2 Polygons

A **simple polygon** is defined by a cyclic polygonal chain of two-dimensional points, called the boundary. A line-segment connects every pair of adjacent points, and is called either and edge or a side. No two sides intersect except possibly at their endpoints. The polygon has a bounded interior and an unbounded exterior. We may perceive a polygon as the set of all the points that are in its interior or on the boundary. The vertices of a polygon are the points on the boundary. In this thesis only simple polygons will used.

A **polygon with holes**, $P = (B, \{H_1, H_2, ..., H_k\})$, is defined by an outer-boundary (which is a simple polygon), B, a set of holes, $\{H_1, H_2, ..., H_k\}$. Each hole is a simple polygon contained in B. The set of points represented by a polygon with holes is given by $P - H_1 - H_2 - ... - H_k$. The vertices of a polygon with holes are the vertices on B and the vertices on each of the holes. In later sections n will refer to the total number of vertices in P.

A **star-shaped polygon**, P, is a simple polygon for which there exists a point, $c \in P$ (the star-center) s.t. for all points $p \in P$ the segment $\overline{cp}$ is contained in P. The set of all star-centers is called the kernel.

A **convex polygon** is a star-shaped polygon, P, with kernel, K s.t. $P = K$. Equivalently all interior angles are at most 180 degrees[2].

A **visibility polygon** is a type of star-shaped polygon. We will define it in the context of a polygon with holes. Given a point q and a polygon with holes $\mathcal{P}$ the visibility polygon of q in $\mathcal{P}$ is the set of points, $p \in \mathcal{P}$, s.t. q sees p. We can show that this subset is a star-shaped polygon with star-center, q.

The **convex hull** of a set of points S is the smallest convex polygon, P, s.t. $S \subseteq P$. We will use the terminology that the convex hull of two sets S_1 and S_2 is the convex hull of $S_1 \cup S_2$.

2.3 Polygon decomposition

A set of points, S, is covered by a set of polygons $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ if $P_1 \cup P_2 \cup ... \cup P_k = S$. We say that $\mathcal{P}$ is a **cover** of S. If additionally all polygons in $\mathcal{P}$ are disjoint we say that $\mathcal{P}$ is a **partition** of S. If all polygons in $\mathcal{P}$ are convex we call it a **convex cover/partition**. $\mathcal{P}$ is a **minimal cover** if we cannot choose a polygon, $P \in \mathcal{P}$, s.t. $\mathcal{P} - \{P\}$ is a cover.

A **triangulation** of a set of points S is a convex partition, $\mathcal{T}$, of S, where each polygon $T \in \mathcal{T}$ is a triangle. A polygon with holes with n vertices can always be triangulated into $O(n)$ triangles.

A **Delaunay triangulation** is a triangulation for which the smallest angle in any triangle is maximized, then the second smallest angle is maximized, and so on.

For a **constrained Delaunay triangulation** we are given a set, L, of line segments and we must find a Delaunay triangluation, s.t. given a side, s, of any triangle and any line segment $l \in L$, either s and l does not cross each other.

2.4 Graph theory

A **clique**, C, in an undirected graph, $G = (V, E)$, is a subset of nodes, $C \subseteq V$ s.t. for any pair, $(a, b) \in C \times C$, the undirected edge $\{a, b\}$ is in E. A vertex **clique cover** (VCC), $\mathcal{C} = \{C_1, C_2, ..., C_k\}$ is a collection of cliques s.t. $C_1 \cup C_2 \cup ... \cup C_k = V$. Note that if we remove any node from a clique the resulting subset of nodes is still a clique. Therefore, any clique cover can be converted into a clique partition (clique cover where cliques are disjoint) of the same cardinality. When a clique cover is mentioned it is therefore assumed that the cliques are pairwise disjoint.

[2] allowing for consecutive colinear vertices

The **dual graph** (or just the dual) of a cover, $\mathcal{P}$, (of a set of points) is the induced graph with faces (pieces) as nodes, and edges between nodes sharing an edge in the cover. We denote it by $D_{\mathcal{P}}$.

3 The algorithm

We wish to find a small convex cover (CC) of a polygon with holes, P. A three-phase approach is used. Firstly, in phase one we compute a partition (which we always choose to be a triangulation), $\mathcal{T}$, of P. In phase two we then construct a visibility graph, $G = (\mathcal{T}, E)$, where if there is an edge between two nodes then the corresponding two triangles fully see each other within P. Lastly, in phase three, we compute a clique cover of G, where the convex hulls of the cliques (meaning the convex hull of the triangles of a clique) represent the convex polygons of the convex cover. Not all such convex hulls are necessarily contained in P, and thus we need to fix those. Afterwards, we perform a generic post-processing of the solution to get a minimal convex cover. We believe the advantages of this approach are the following.

- During phase three all the pieces of the convex cover are constructed at the same time. This means that all the pieces of the convex cover are somewhat related to each other, and we cannot decompose the convex cover into solutions to subproblems. In general, we cannot solve MCC using divide and conquer, and thus, in that sense, it is a good idea that we do not do that either.
- As the pieces are constructed in phase three while constructing the cover we do not need to pregenerate a large set of pieces to choose from.
- Reducing the convex cover problem to a graph problem means that we can make use of all the existing graph algorithms and implementations.
- The approach is very generic. We can choose a type of triangulation, a type of visibility graph and any clique decomposition algorithm.

In this section we present the three phases and the post-processing, but only including the most important underlying algorithmic strategies used to make the algorithm efficient. In section 4 all the optimization strategies are explained.

3.1 Phase one: Partitioning the polygon

The first phase of the algorithm is to find a partition, $\mathcal{T}$, of P, of which we can compute a visibility graph in phase two. All the partitions will be triangulations, since then it is easier to design fast polygon-in-polygon visibility tests. We will therefore use the term "triangulation", but conceptually, what is important is that we have a partition of P, and for most of the presented techniques any partition will work.

We will present different types of triangulations with different properties and varying space complexity. Every type of triangulation we present can be defined by a set of line segments that are used as the set of constraints in a constrained Delaunay triangulation of P. We use a Delaunay triangulation, instead of an arbitrary triangulation, since a Delaunay triangulation produces "fat" triangles, i.e. inner-angles are as large as possible. Intuitively, such "fat" triangles are more likely to fully see each other, which leads to more edges in the visibility graph. A larger visibility graph gives more freedom when picking cliques, and thus intuitively should lead to a smaller convex cover.

The simplest approach is to not have any constraints, and simply compute a `Delaunay triangulation` of P. However, our main choice of approach is, what we call, the `extension triangulation`. A much slower approach, is, what we call, the `visibility-pair triangulation`, which we expect will be able to produce better results than the `extension triangulation`. Lastly, we present approaches (`constrained-extension triangulation`, `extension-sampling triangulation` and `concave-chain triangulation`) that, during the Challenge, were discovered to be necessary, in terms of computation time, for special types of polygons (see section 5.2).

3.1.1 Extension triangulation

Consider two points, p_1 and p_2, and triangles, T_1 and T_2, containing these points. Ideally we would like that if p_1 and p_2 fully see each other, then T_1 and T_2 fully see each other. This is very difficult to achieve.

However, for most triangulations if p_1 and p_2 are close and see each other, then T_1 and T_2 are also close and "most likely" see each other. In a small convex cover the convex polygons may sometimes stretch over large distances, and therefore we would like to also have pairs of distant points that see each other. We achieve this by including, in the set of constraints, important visibility segments within the polygon. We imagine that edges of pieces in a small convex cover often coincide with edges of P. And thus, we want the important visibility segments to lie on top of edges of P. We call these important visibility segments `extensions segments`. We formally define an `extension segment` as the following.

Definition 3.1 (`extension segment or extension`). *An* ***`extension segment`*** *of a polygon with holes, P, is a segment $s = \overline{p_1 p_2} \subseteq P$ of maximal length with the following property: there exists at least one edge, $e = \overline{p_3 p_4}$, on the outer boundary of P or on the outer boundary of a hole in P s.t. either $p_1 = p_3$ or $p_1 = p_4$ and $e \subseteq s$.*

In the `extension triangulation` we use all extension segments of P as the set of constraints. In figure 1 is shown the extension segments of a polygon and the corresponding `extension triangulation`.

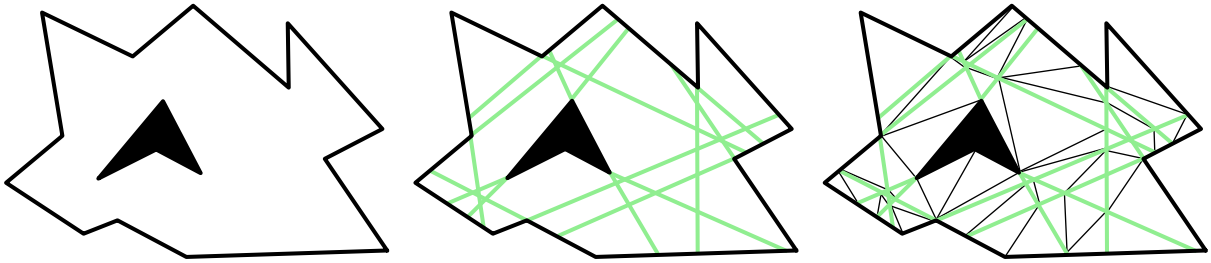


Figure 1: Polygon P (left), extension segments of P (middle), and extension triangulation of P (right).

As we will see in later sections we cannot afford large triangulations. We can easily show that an `extension triangulation` has $O(n^2)$ pieces in the worst case (proof in appendix 7.2). A quadratic number of pieces is very expensive, however it seems likely that this, in practice, is often a weak upper bound. For a description of how compute the extension segments, see section 4.3.

3.1.2 Visibility-pair triangulation

In section 3.1.1 we argued that `extension segments` are probably the best kinds of visibility segments to include in the set of constraints. In the `visibility-pair triangulation` we want to have even more visibility segments than in the `extension triangulation`. We will define a `visibility-pair segment` as the following.

Definition 3.2 (`visibility-pair segment`). *A* ***`visibility-pair segment`*** *of a polygon, P, is a segment $s \subseteq P$ of maximal length with the following property: there exists at least one vertex of P, whose visibility-polygon within P has two vertices, v_1 and v_2, such that the segment $\overline{v_1 v_2} = s$.*

The set of constrains of the `visiblity-pair triangulation` is the set of all `visibility-pair segments`. We note that any `extension segment` is a `visibility-pair segment`. Hence the set of all `extension segments` is a subset of the set of all `visibility-pair segments`, and therefore we expect convex covers based on a `visibility-pair triangulation` to be at least as small as convex covers based on an `extension triangulation`.

Every visibility polygon has $O(n)$[3] vertices. Thus the total number of `visibility-pair segments` will be $O(n^2)$. Every segment may intersect every other segment and thus we will have $O(n^4)$ intersections. As shown in appendix 7.2 the size of a constrained Delaunay triangulation is linear in the number of constraints plus the number of segments of P. Therefore the size of the `visibility-pair triangulation` is $O(n^4)$. A visibility-graph of a triangulation, $\mathcal{T}$, may contain $O(|\mathcal{T}|^2)$ edges. Therefore the number of edges in the visibility-graph of a `visibility-pair` triangulation may be $O(n^8)$. We expect this to be a weak upper bound, but still very expensive.

[3]It is just an upper bound. A visibility polygon might have asymptotically less vertices.

3.1.3 Constrained-extension triangulation

The `constrained-extension triangulation` is simply an `extension triangulation`, but where only `extension segments` below a certain length, L, are included in the set of constraints.

3.1.4 Extension-sampling triangulation

Just like the `constrained-extension triangulation` the `extension-sampling triangulation` is a triangulation, where we only include a subset of all the extension-segments in the set of constraints. In this case we randomly sample `extension segments` inversely proportional to their length. To be more specific we have a parameter C and the average length, $\overline{L}$, of a side of P. Given this, an `extension segment` of length $L \geq C\overline{L}$ is sampled with the following probability.

$$P(\texttt{extension segment} \text{ of length } L \text{ is sampled}) = \frac{C\overline{L}}{L}$$

Any `extension segment` with length $L < C\overline{L}$ is always sampled. In the `extension-sampling triangulation` the set of sampled `extension segments` is the set of constraints for the constrained Delaunay triangulation.

3.1.5 Concave-chain triangulation

Let a concave chain of P be a continuous sequence of sides of P such that the inner angles on this chain are greater than 180 degrees. Suppose that P has a long concave chain of maximal-length on the boundary. Take the sides of P on this concave chain. For any two such sides, s_1 and s_2 and any two interior points, $p_1 \in s_1$ and $p_2 \in s_2$, it holds that p_1 and p_2 must be part of distinct pieces in any convex cover.

Now, take two segments of two different concave chains. We would like these segments to be contained in triangles (of the triangulation) such that these two triangles fully see each other within P. The reason being that the visibility graph will then have an edge between these triangles, and the two segments can share piece in the convex cover.

To do this we will do a local triangulation along each concave chain. More specifically we define a `concave-inner segment` as the following.

Definition 3.3 (`concave-inner segment`)**.** *Consider a concave chain, C, consisting of k vertices of P, $v_1, v_2, ..., v_k$. Then consider 4 of these vertices $v_i, v_{i+1}, v_{i+2}, v_{i+4}$, $i \leq k-3$. Let e_1 be the* `extension segment` *coinciding with segment $\overline{v_i v_{i+1}}$ and let e_2 be the* `extension segment` *coinciding with segment $\overline{v_{i+3} v_{i+2}}$. Let p be the intersection point of e_1 and e_2. Given this, we define a* `concave-inner segment` *as a segment, $s \subseteq P$, such that there exists a concave chain, C, and a value i, where p exists and either $s = \overline{v_{i+1} p}$ or $s = \overline{v_{i+2} p}$.*

For a `concave-chain triangulation` the set of constraints for the constrained Delaunay triangulation is the set of all `concave-inner segments` of P (see figure 2). Effectively the `concave-chain triangulation` puts small triangles on the segments of the concave chains. Additionally, we may overlay a grid of equally-spaced vertical and horizontal segments on top of P and include these segments in the set of constraints. This is done to have more edges in the visibility graph, and thus more flexibility when constructing a clique cover. As we will see in section 5.5.3 overlaying a grid does improve the results.

Under the assumption that the number of vertical and horizontal segments is $O(1)$, we can show that overlaying the grid does not change the space complexity. The grid will itself will have $O(1)$ intersections. Each `concave-inner segment` will intersect $O(1)$ grid segments. Thus, the number of additional intersections created by the grid is $O(n)$, since there are $O(n)$ `concave-inner segments`. However, each `concave-inner segment` already contributes $\Omega(1)$ space.

Since each `concave-inner segment` is a subsegment of an `extension segment`, we note that the union of all `concave-inner segments` is a subset of the union of all `extension segments`.

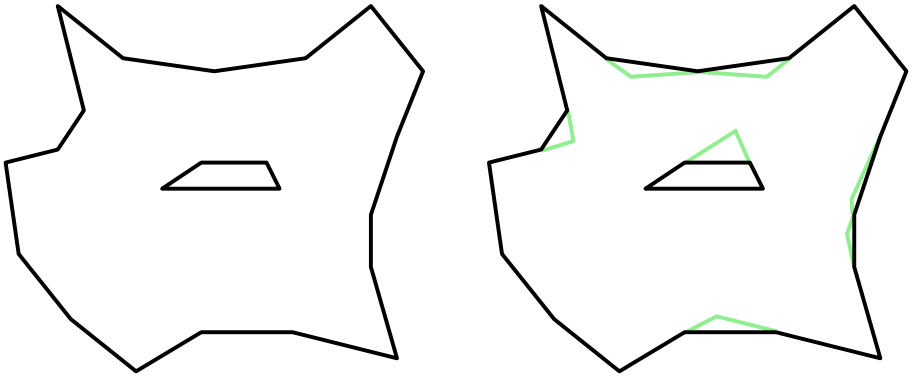


Figure 2: Polygon P (left) and `concave-chain triangulation` with no grid (right).

3.2 Phase two: Constructing a visibility-graph

In phase one we found a triangulation, $\mathcal{T}$, of P, and now we wish to build a visibility graph, $G = (\mathcal{T}, E)$, based on the triangulation. We define a visibility graph like this.

Definition 3.4 (visibility graph)**.** *A visibility graph, of a triangulation $\mathcal{T}$ of P, is an undirected graph, $G = (\mathcal{T}, E)$, where for each edge, $\{T_1, T_2\} \in E$, T_1 and T_2 fully see each other.*

Note that for a graph to be a visibility graph, according to definition 3.4 it is not necessary that it has all possible edges, $\{T_1, T_2\}$, for which T_1 and T_2 fully see each other. We also note that $|E| = O(|\mathcal{T}^2|)$, and thus sometimes we will have to compute only a subset of the all the possible edges. We will have two approaches to constructing a visibility graph.

- In the first approach, for each node $T \in \mathcal{T}$, we first find a set of candidate nodes, such that the set is small, but is guaranteed to contain all neighbours of T. Then, for each candidate node, we check if it is actually a neighbour of T. We will refer to this approach as the **full visibility graph**.
- In the second approach we only add an edge between two nodes, $T_1 \in \mathcal{T}$ and $T_2 \in \mathcal{T}$, that fully see each other, if there exists a path between T_1 and T_2 in the dual graph of $\mathcal{T}$, where either all nodes on the path are fully visible from T_1 or all nodes on the path are fully visible from T_2. We will refer to this approach as the **partial visibility graph**.

Throughout this section we will need efficient polygon-in-polygon containment tests. We will explain where and how they are used and refer to section 4.4 for more details.

- `Naive-test`: Efficiently tests if a polygon, P_1, is fully contained within another polygon with holes, P_2, given that P_1 is not outside P_2. See section 4.4.1 for details.
- `Star-test`: Answers whether a polygon, P_1, is inside, partially contained or outside a star-shaped polygon (in particular a visibility polygon), P_2. We require that a star-center of P_2 is known. The time complexity is the same as the `naive-test`. See section 4.4.2 for details.
- `Fast-test`: Given a polygon with holes, P, the `fast-test` tests if disjoint convex polygons, $P_1 \subset P$ and $P_2 \subset P$, fully see each other within P. The running time does not depend on the size of the convex hull of P_1 and P_2. However, it requires access to visibility polygons of a subset of the vertices of P_1 and P_2. This is inconvenient, but fortunately the test can be divided into two, so-called, `fast-half-tests` (one for P_1 against P_2 and one for P_2 against P_1), which can be performed in any order. In a `fast-half-test` we will not need the visibility polygons of both P_1 and P_2. After the last **fast-half-test** has been performed, if both **fast-half-tests** concluded visibility then the P_1 and P_2 fully see each other. See section 4.4.3 for details.
- **One-sided-visibility-test**: Given a triangulation $\mathcal{T}$, it tests if a triangle $T_2 \in \mathcal{T}$ is fully visible from another triangle $T_1 \in \mathcal{T}$ within a polygon with holes, P. The test can have false negatives, but it is very efficient. See section 4.4.4 for details.

- **One-sided-invisibility-test**: Tests if triangle T_2 is not fully visible from another triangle T_1 within a polygon with holes, P. The test can have false negatives, but it is very efficient. See section 4.4.5 for details.

3.2.1 Constructing the full visibility graph

In this approach we wish for G to have all possible edges. Formally we define the full visibility graph like this.

Definition 3.5 (the full visibility graph)**.** *Given a triangulation, $\mathcal{T}$, we define $G = (\mathcal{T}, E)$, as having the triangles of $\mathcal{T}$ as nodes, and $\{T_1, T_2\} \in E$, iff. T_1 and T_2 fully see each other.*

Constructing the full visibility graph will consist of two steps. The first step is to, for each node, T, **find candidate nodes** — nodes that are "likely" to be neighbours of T in G. Consider triangles $T_1, T_2 \in \mathcal{T}$ that fully see each other. Then consider the convex hull, C, of T_1 and T_2 and the set, $\mathcal{T}' \subseteq \mathcal{T}$, of all triangles that are not outside of C (see figure 3). For any triangle, $T' \in \mathcal{T}'$ it clearly holds that T' is partially visible from both T_1 and T_2.

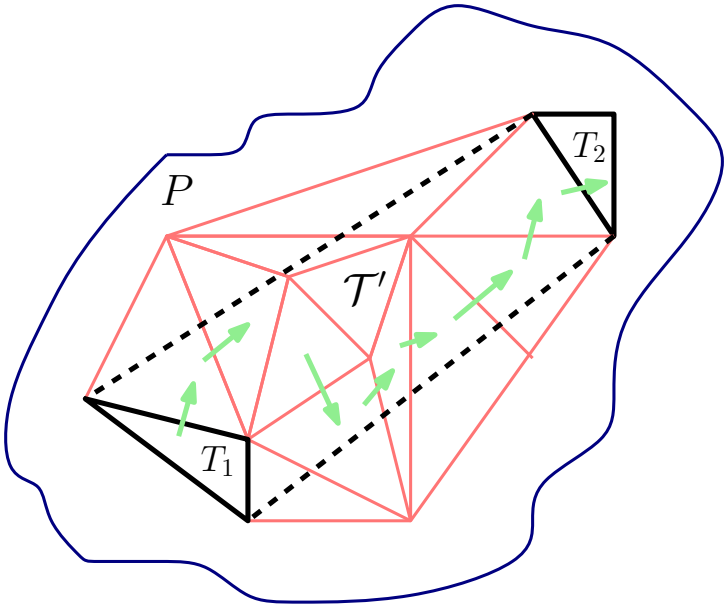


Figure 3: T_1 and T_2 fully see each other. $\mathcal{T}'$ is the set of triangles that overlap with their convex hull. Green arrows indicate a path in D_T.

Now consider the dual, $D_\mathcal{T}$, of $\mathcal{T}$; it holds that the induced subgraph of $\mathcal{T}'$ in $D_\mathcal{T}$ is connected. This implies that there is a path in $D_\mathcal{T}$ between T_1 and T_2, where all triangles on the path are partially visible from T_1 and T_2. Based on this we define the set of candidates nodes like this.

Definition 3.6 (candidate nodes)**.** *For a node, $T \in \mathcal{T}$, let us perform a depth-first-search from T in $D_\mathcal{T}$, but only visit nodes that are partially visible (which could also be fully visible) from T. The visited nodes are the candidate nodes, $\mathcal{C}_G(T)$ of T.*

Based on the above arguments we also state the following lemma.

Lemma 3.1. *For a node, $T \in \mathcal{T}$, it holds for the set of neighbours, $N_G(T)$, that $N_G(T) \subseteq \mathcal{C}_G(T)$.*

At this point we have a valid definition of a set of candidate nodes, i.e. $\forall T \in \mathcal{T} : N_G(T) \subseteq \mathcal{C}_G(T)$. However, it is unclear how we find the candidate nodes, i.e. how we test if a triangle, $T_2 \in \mathcal{C}_G(T_1)$, is partially visible of fully visible from another triangle, T_1. And even more unclear, how we would do so efficiently. We could compute the convex hull of T_1 and T_2 and test if it is not ouside P using the `naive-test`. But, if the triangles are far apart then the convex hull will be large, and the `naive-test` will be slow.

Let us therefore instead relax the condition of candidate nodes to be the following.

Definition 3.7 (relaxed candidate nodes)**.** *For a node, $T \in \mathcal{T}$, let us perform a depth-first-search from T in $D_{\mathcal{T}}$, but only visit nodes that are partially visible from each of the three vertices of T. The visited nodes are the relaxed candidate nodes, $\mathcal{R}_G(T)$ of T.*

If all the points in a triangle, T_1, partially see another triangle, T_2, then the three vertices of T_1 also partially see T_2. Based on this and lemma 3.1, we get the following lemma.

Lemma 3.2. *For a node, $T \in \mathcal{T}$, it holds for the set of neighbours, $N_G(T)$, of T in G that $N_G(T) \subseteq \mathcal{C}_G(T) \subseteq \mathcal{R}_G(T)$.*

To find the set of relaxed candidate nodes for a node, T_1, we compute the visibility polygons of the three vertices of T_1. Then we perform a *depth-first-search* from T_1 in $D_{\mathcal{T}}$ only visiting nodes that fulfill the condition of a relaxed candidate node. To test if a node, T_2, is a relaxed candidate node, we test, for each vertex of T_1, if T_2 is not outside the visibility polygon of the particular vertex. To perform these tests we can use the `star-test`, since a visibility polygon is a star-shaped polygon.

Given that we have all sets of relaxed candidate nodes, the second step is to **find neighbours based on the relaxed candidate nodes**. For a node T and each node, $T' \in \mathcal{R}_G(T)$, we wish to check if T and T' are neighbours in G, i.e. that they fully see each other. We could compute the convex hull of T and T', and test if it is contained in P using the `naive-test`. But, like before, if the triangles are far apart then the convex hull will be large, and the `naive-test` will be slow (see section 4.4.1). Instead, we will use the `fast-test`. The `fast-test` requires access to visibility polygons of one vertex of T and one vertex of T' (see section 4.4.3). We could compute and store all visibility polygons of all vertices of all triangles initially, but this would require a lot of storage. Thus, to avoid storing many visibility polygons at once, we instead use the `fast-half-tests` in the the following way.

1. Process the triangles one-by-one. For a given triangle, T do the following. For each node, $T' \in \mathcal{R}_G(T)$, we perform the **fast-half-test** for T against T', and save the result for later.

2. For each unordered pair, $\{T_1, T_2\}$, for which T_1 and T_2 are relaxed candidate nodes of each other, we now know the result of the **fast-half-test** for T_1 against T_2 and the **fast-half-test** of T_2 against T_1. Thus, we know the result of the **fast test** between T_1 and T_2, i.e. we know if T_1 and T_2 fully see each other within P.

After the second step we have computed all unordered pairs $\{T_1, T_2\}$, for which T_1 and T_2 fully see each other. Therefore, we have computed all the undirected edges of G, and thus constructed the full visibility graph.

3.2.2 Constructing the partial visibility graph

Unfortunately, the full visibility graph can become very large and take a long time to compute (see section 5.3). Therefore, in this section, we will compromise solution quality in return of faster running time and a smaller graph.

We will define the visibility graph, G, in this case, as the following.

Definition 3.8 (the partial visibility graph)**.** *Given a triangulation, $\mathcal{T}$, we define $G = (\mathcal{T}, E)$, as having the triangles of $\mathcal{T}$ as nodes, and such that there is an edge between node, T_1 and T_2 iff. T_1 and T_2 fully see each other, and there exists a visibility path from T_1 to T_2 or/and a visibility path from T_2 to T_1. A visibility path is a path in $D_{\mathcal{T}}$ from a source, U, to a target V, such that all nodes on the path are fully visible from U.*

Definition 3.8 indicates that the existence of a visibility path from T_1 to T_2 does not imply the existence of a visibility path from T_2 to T_1. We show this using the counter-example in figure 4.

For each node, T, we need to finds its neighbours, $N_G(T)$. To do this will run a procedure, `visibility-BFS`, which has some of the characteristics of a *breadth-first-search*, in $D_{\mathcal{T}}$ with T as the source. The procedure will only visit nodes that are fully visible from T. The induced subgraph of the visited nodes will always be connected, and have what we call a "boundary".

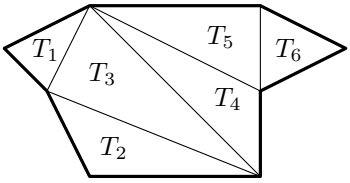


Figure 4: There is a visibility path from T_1 to T_6: $T_1 \to T_3 \to T_4 \to T_5 \to T_6$. However, since T_6 does not fully see T_3, there cannot exist a visibility path from T_6 to T_1.

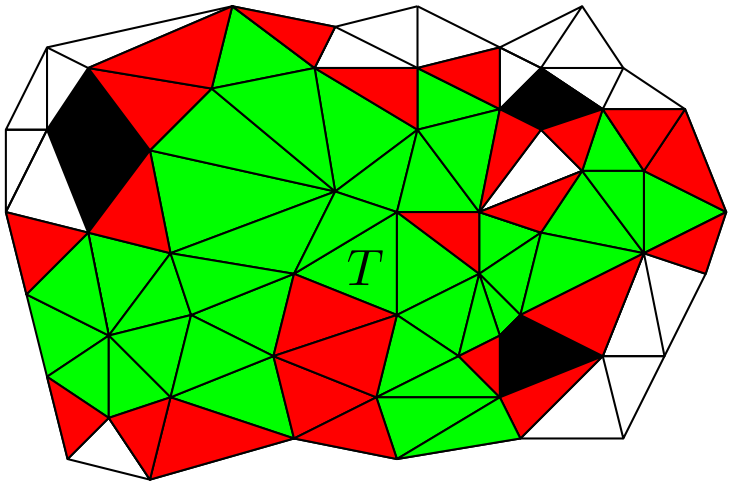


Figure 5: `visibility-BFS` is running, with source T, on a triangulation. Green triangles are visited. Red triangles are on the boundary. White triangles are visited and on the boundary. Black areas are holes.

Definition 3.9 (boundary)**.** *At any point during the* ***visibility-BFS****, the boundary is the set of nodes that are unvisited, but adjacent to at least one visited node.*

Figure 5 shows an example triangulation with visited nodes and the corresponding boundary. For a source node, T, the set of visited nodes is initially $\{T\}$, and thus the boundary is the neighbours of T, $N_{D_\mathcal{T}}(T)$. During the `visibility-BFS` we always want to find nodes that can be visited, i.e. are fully visible from the source. At any point in time, the next node that becomes visited will always be a node on the boundary. Thus, we always want to, as "cheaply" as possible, find a node on the boundary that is fully visible from T, and add it to the set of visited nodes. At some point we will reach a state where either the boundary is empty or there does not exist a node on the boundary which is fully visible from the source. In this case `visibility-BFS` terminates. We suggest that `visibility-BFS` correctly computes the partial visibility graph.

Lemma 3.3. *Suppose $\{T_1, T_2\} \in E$. After running* ***visibility-BFS*** *with T_1 as source and* ***visibility-BFS*** *with T_2 as source, then the following holds: T_1 was visited from T_2 or T_1 was visited from T_2 or both was visited.*

Proof. If there is a visibility path from T_1 to T_2 then T_2 is reachable from T_1 through this path, and T_2 will therefore be visited during the `visibility-BFS`. Since $\{T_1, T_2\} \in E$, then if there is no visibility path from T_1 to T_2 there must be a visibility path from T_2 to T_1. Symmetrically T_1 will be reachable from T_2 in this case. □

Thus, in order to built the partial visibility graph we simply run `visibility-BFS` with each node, $T \in \mathcal{T}$ as source. Then to find the set of edges, E, we add, for each source T and an visited node T', the unordered pair $\{T, T'\}$ to E. Since we run `visibility-BFS` from each node lemma 3.3 implies that each edge of the partial visibility graph is found.

There is one unanswered question left, which is how we can efficiently find a fully visible node on the boundary. An exact answer will be given in section 4.5, but the general idea is that the cheap

`one-sided-visibility-test` can be used very often, since we are only dealing with fully visible nodes. This is the strength of `visibility-BFS` which makes the partial visibility graph efficient to compute in practice (see section 5.3). We can also use the cheap `one-sided-invisibility-test`. However, these cheap tests cannot always be used, and thus we sometimes have to fall back to the `naive-test`.

3.3 Phase three: Finding a valid clique cover of the visibility graph

Now that we have a visibility graph, $G = (\mathcal{T}, E)$, we wish to find a small clique cover of G. Let the convex hull of a clique, C, of G be the convex hull of the union of the triangles of C. We want the convex hulls of the cliques to represent the convex polygons in the convex cover. For a clique it holds that the convex hull of every pair of triangles is contained in P. This is a necessary condition for the convex hull to be contained in P, but it is unfortunately not sufficient. Still, because of this necessary condition, it seems likely that the convex hull of most cliques are contained in P. Still we must be able to handle the case, where the convex hull is not contained in P. We call such a clique an *invalid clique*.

Definition 3.10 (invalid clique, invalid clique cover)**.** *The convex hull, ch_C, of a clique of size k, $C = \{t_1, t_2, ..., t_k\}$, is defined as the convex hull of $t_1 \cup t_2 \cup ... \cup t_k$. If $ch_C \not\subseteq P$ then we call C an invalid clique. An invalid clique cover has at least one invalid clique.*

The overall approach is to first find any small clique cover of G, and then turn this clique cover into a valid clique cover that represents a valid convex cover of P. In section 3.3.1 we will explain, how we find a small clique cover of G, and in section 3.3.2, we explain exactly why invalid cliques occur and how we can fix them without increasing the solution size too much.

3.3.1 Finding a small clique cover

In principle any method for finding a clique cover could be used. We used the iterated greedy heuristic algorithm by Chalupa [3]. From now on let us refer to it as `Chalupa`. `Chalupa` iteratively finds better and better clique covers. It does so by decreasing the size of a maintained clique cover, and at the same time maintaining a lower bound on the minimum-cardinality clique cover using independent sets. If the size of the maintained clique cover reaches the lower-bound then the clique cover is optimal. We do not expect `Chalupa` to have time to converge on very large visibility graphs, but `Chalupa` does have the convenient property that the maintained clique cover is always the smallest clique cover found so far. Thus, we can simply let `Chalupa` run for a fixed amount of time, and use the clique cover that it was able to produce.

Another option is the state-of-the-art VCC solver `ReduVCC`, by Strash and Thompson [11]. `ReduVCC` applies reductions to the input-graph to reduce the size, and then runs `Chalupa` on the reduced graph. A branch-and-reduce (with pruning) approach is used to efficiently perform the reductions. In practice `ReduVCC` is much faster than `Chalupa` as shown in [11].

3.3.2 Fixing invalid cliques

As illustrated in the figure 6 there might be invalid cliques in the clique cover of the visibility graph. In practice invalid cliques are rare, but it is essential to be able to fix them efficiently and without increasing the solution size too much.

To do this, we first observe that the reason a clique is invalid is never that its convex hull intersects the unbounded side of P. Thus, any invalid clique contains one or more holes. The reason that we can have invalid cliques is that the nodes (triangles) of the clique might induce a subgraph of $D_{\mathcal{T}}$ with multiple components that surrounds a hole of P (see figure 6). Additionally, it is worth mentioning that there exists minimum convex covers where a clique will have to contain multiple of such induced components (see figure 7).

We use the following strategy to fix invalid cliques. Given an invalid clique we pick any hole, h, and find any half-plane defined by an infinite line, l, that intersects h. Then we split the triangles of the clique into two subsets based on containment in the half-plane. If a triangle intersects l it does not matter, which subset we put it into. Thus, we can choose the following containment rule: we put a triangle in the first subset if it is contained in the half-plane, otherwise we put it in the second subset. This leads

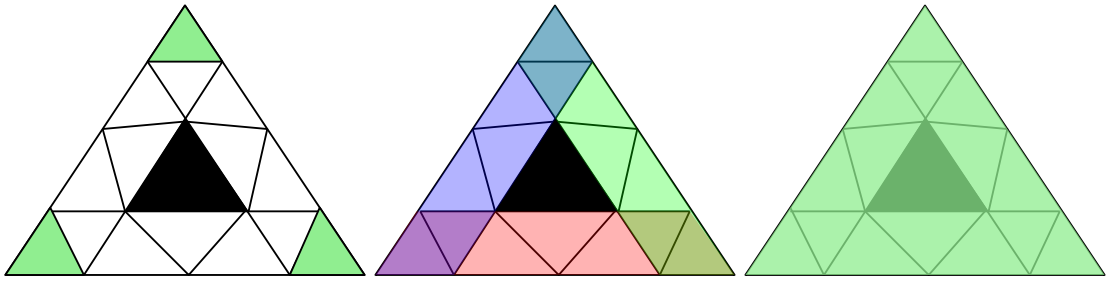


Figure 6: For the set of green triangles (left), every pairwise convex hull is contained in P (middle), but the convex hull of all the green triangles is not (right).

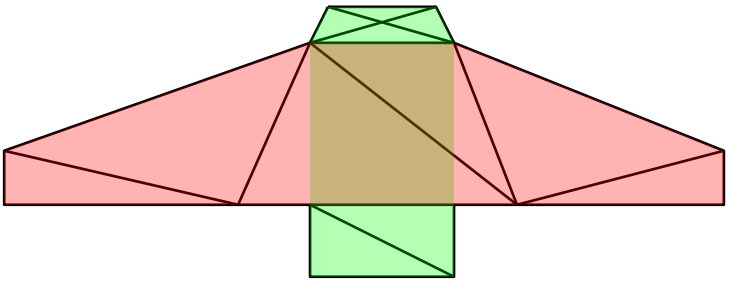


Figure 7: Example of a polygon with an extension triangulation, where it is necessary in the optimal solution for the green clique to not be connected in the dual-graph of the triangulation.

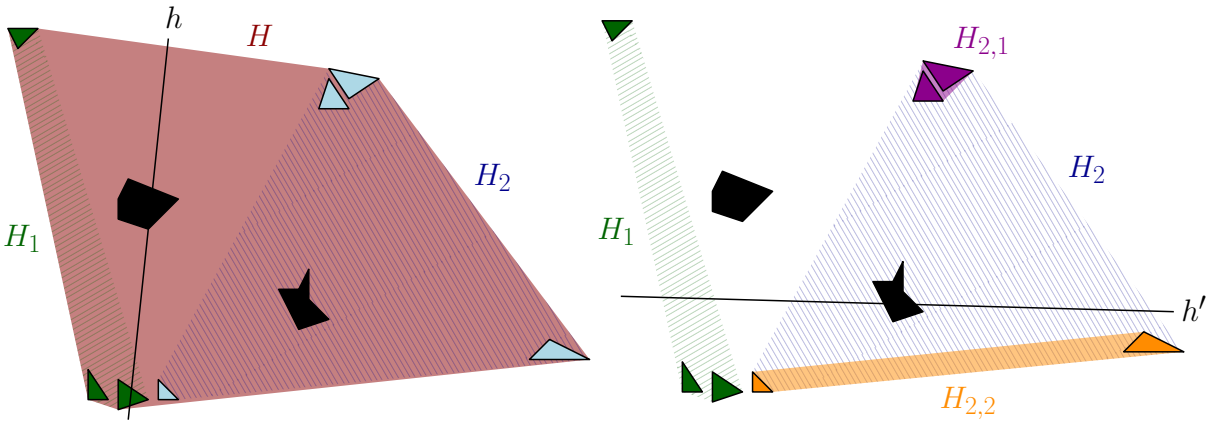


Figure 8: Fixing an invalid clique: All visible triangles form the initial piece H that we then split into pieces H_1 and H_2 using the half-plane h (left). As H_2 still contains a hole, we again split it into pieces $H_{2,1}$ and $H_{2,2}$ using half-plane h' (right). The result is the valid pieces H_1, $H_{2,1}$, and $H_{2,2}$.

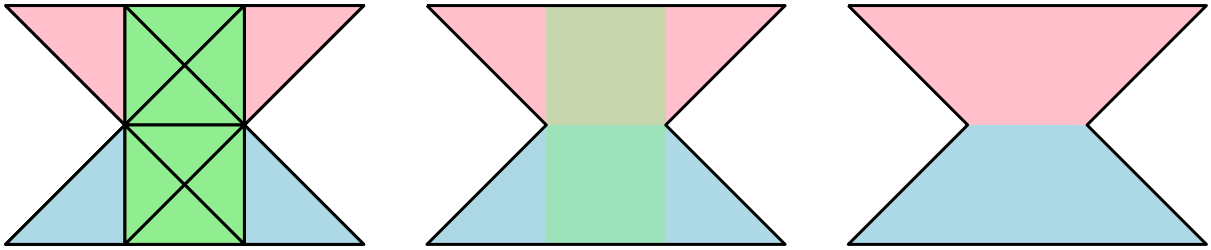


Figure 9: Extension partition with suboptimal clique cover (left), the corresponding convex cover (middle), and a convex cover without the unnecessary green polygon (right).

to the following situation: The convex hull of the triangles of any of the two subsets will not intersect h. To fix cliques containing more than one hole, we recursively fix the two subsets. See figure 8.

An important observation is that if an invalid clique has more than one hole, then after splitting it into the two subsets, there cannot exist a hole s.t. the convex hulls of the two subsets both overlap with it. This guarantees that an invalid clique, whose convex hull overlaps with n_h holes, can be converted into $n_h + 1$ (or less) valid cliques, since each hole adds (at most) one new clique. But, we can do even better. We can always pick the line defining the half-plane s.t. it intersects two holes, h_1 and h_2, (if there are at least two holes). The convex hull of the two produced subsets of triangles cannot contain h_1 or h_2. Thus for each pair of holes one clique is converted into two cliques. If n_h is odd then we will, in the worst case, have to assign a single half-plane to the last hole. Thus we finally conclude that the invalid clique can be converted into $\lceil \frac{n_h}{2} \rceil + 1$ (or less) valid cliques.

3.4 Post-processing

Although phases one, two and three, hopefully should produce a small convex cover, we can show that the produced convex cover is not necessarily a minimal convex cover (see figure 9).

To fix this we will perform generic post-processing that converts any convex cover into a minimal one. Essentially we are dealing with the classic *set-cover-problem* with the sets being the convex polygons. In order to make it practical to work with we need to discretize the convex polygons. We do this by computing the overlay of the pieces of the convex cover. Then we compute a constrained triangulation of the induced subdivision. In practice all of this is done by simply computing a constrained triangulation with constraints being the sides of all pieces. Afterwards, any piece will be represented by a subset of this triangulation.

With this setup we can use any existing set cover solver. However, what we did during the Challenge was to use a heuristic algorithm that selects large pieces until a minimal convex cover is reached. In appendix 7.3 this approach is described in detail.

4 Algorithmic optimizations

In section 3 the conceptual ideas behind the algorithm were described. We left out details related to, how we concretely make an efficient algorithm. Therefore, in this section, we will suggest, how essential aspects of an efficient algorithm might look like. In general the geometric procedures and primitives, such as polygon-in-polygon containment tests, are very expensive compared to everything else. Therefore we did not spend time, during the Challenge, optimizing away polylogarithmic factors arising from the use of binary-search-trees and so on.

4.1 Computing a constrained Delaunay triangulation

This is our take on, how to compute a constrained Delaunay triangulation from a set of potentially intersecting segments. We may convert P and the set of constraints as a planar straight-line graph. Let S be the set of segments which are either an edge of P or a constraint. Then we can find all intersections of segments in $O(|S| \log(|S|) + k \log(|S|))$ time [2][4]. The intersections and the sub-segments connecting

[4] chapter 2

intersections induces a subdivision, which is a planer straight-line graph, and k is the complexity (number of vertices + number of edges + number of faces) of this subdivision. From this planer straight-line graph we may use existing algorithms to compute the a constrained Delaunay triangulation in $O(k \log(k))$ time[10].

4.2 Computing visibility polygons

For computing visibility polygons we first compute a Delaunay triangulation of P. Then when querying the visibility polygons of a point, p, we first locate (for example using range-trees) a triangle, T, for which p is not outside. In the dual of the triangulation we then explore, with T as the source, the triangles that are either partially or fully visible from T. Thus, the algorithm is output sensitive, since the running time depends on the size of the visibility polygon. This method is used in the CGAL visibility package [7].

4.3 Computing extension segments

At some point in phase two we need to compute at least as many visibility polygons as there are vertices in P. Thus, we might as well just use visibility polygons to compute the extension segments. For each vertex, v, of P we compute the visibility polygon. v has at most two incident edges, on which an extension segment lies. Thus, we shoot a ray out from v along the incident edges and find the intersection point between the ray and the boundary of the visibility polygon. We can simply iterate over the vertices of the visibility polygon to find the intersection, since we only do this a constant number of times. Ignoring the complexity of computing the visibility polygons, the complexity will be linear in the sum of the number of vertices of all the visibility polygons.

4.4 Polygon-in-polygon containment predicates

Polygon-in-polygon containment tests, as talked about in section 3.2, are very important to the efficiency of the algorithm. In this section we describe these tests.

4.4.1 The naive-test

Given a polygon with holes, P_2, we wish to be able to query the answer to the following: Given polygon, P_1, test if it is inside P_2. It is a precondition that the area of $P_1 \cap P_2$ is positive. We call the query the the `naive-test`, and in order to use it we need to perform some associated preprocessing. If this precondition is not fulfilled then the `naive-test` will fail iff. P_1 is not inside P_2 and the area of $P_1 \cap P_2$ is positive. Otherwise it succeeds. Thus, the `naive-test` serves a dual purpose depending on whether the precondition is fulfilled.

The `naive-test` is "naive", because it does not take advantage of any properties of P_1 or P_2 to speed up the test. Still, if P_1 is much smaller than P_2 then this test is typically fast.
The `naive-test` is based on the following simple idea. P_1 being partially contained in P_2 is equivalent the existence of any point on any side of P_2, that lies strictly inside P_1. The side, on which such a point lies, must have the interior of P_1 to either its left or right. Similarly, the exterior of P_1 must be either to its left or right. Therefore any point on this side must touch the interior and the exterior of P_1, and thus this equivalence holds. If no such point exists, we can conclude that P_1 is fully contained in P_2, since we have the precondition that $P_1 \cap P_2 \neq \emptyset$. To use this conceptual idea, we will build the following two datastructures on top of P_2.

- A two-dimensional range-tree on the vertices of P_2. Given a rectangular window, W, the range-tree allows us to query the vertices, V, of P_2 that are not outside of W in $O(\log(n)^2 + |V|)$[5] time [9].
- A two-dimensional segment-tree on the sides of P_2. Given a rectangular window, W, the segment-tree allows us to query the sides, S, of P_2 that are not outside of W in $O(\log(n)^2 + |S|)$ time[9].

After this preprocessing a `naive-test` can be performed like this.

[5] this can theoretically be done faster[2]

```
let RT be a range-tree on the vertices of P_2
let ST be a segment-tree on the sides of P_2
Naive-Test(P_1) # returns true if P_1 is inside P_2
    let W be the minimum bounding box of P_1
    let V be the vertices of P_2 inside W. Find V using RT
    for v in V
        if v is inside P_1 then return false
    let S be the sides of P_2 inside W. Find S using ST
    for s in S
        if s crosses a side of P_1 then return false
    return true
```

There are two reasons we decide to query the vertices of P_2.

- *(simplicity)* If no vertex of P_2 is inside P_1 then a side of P_2 cannot be contained P_1, and therefore we just have to check if a given side crosses a side of P_1.
- *(speed)* A range-tree is simple and efficient and produces smaller outputs, since, if a vertex, v, is inside W then the sides incident to v are not outside W, but the converse does not hold. In many cases it is not necessary to query the segment-tree after querying the range-tree.

Assuming that P_1 has a constant number of vertices the time complexity of querying is $O(\log(n)^2 + |V| + |S|)$. The preprocessing time is $O(\log(n)^2 n)$ (building range- and segment-tree) and the space complexity is $O(\log(n)^2 n)$[9].

4.4.2 Star-test

Given a star-shaped polygon, P_2, with a known star-center, we wish to query the answer to the following: Given polygon, P_1, with positive area, test if it is outside, partially contained or inside P_2. In this section partially contained means that the area of $P_1 \cap P_2$ is positive and $P_1 \not\subset P_2$, and thus if P_1 is outside P_2 then the boundaries are allowed to touch. We call this query the `star-test`, and in order to use it we need to perform some associated preprocessing.

We will do this by first performing the preprocessing of the `naive-test` on P_2. Then we use the `naive-test` on P_1. If the `naive-test` returns false then P_1 is partially contained in P_2 i.e. at least one point of any side of P_2 lies strictly inside P_1. Thus, if the `naive-test` returns `true`, we simply need to determine if P_1 is inside or outside P_2.

Given that P_1 is not partially contained in P_2, then P_1 being inside P_2 is equivalent to $P_1 \subset P_2$. Thus, to determine if P_1 is inside or outside P_2 we can pick any point in the interior of P_1 (which is guaranteed to not touch the boundary of P_2) and test if it is inside P_2. We can test if a point is inside a star-shaped polygon in $O(\log(n))$ time using binary search, given that we have a star-center [2][6].

Assuming that P_1 has a constant number of vertices, then the complexity of the star-test is $O(\log(n) + |V| + |S|)$ (this is the complexity of the `naive-test`).

4.4.3 Fast-test

Given a polygon with holes P, we wish to query the answer to the following: Given two disjoint[7] convex polygons, $P_1 \subset P$ and $P_2 \subset P$, test if P_1 and P_2 fully see each other within P. We call this query the `fast-test`, and in order to use it we need to perform some associated preprocessing.

An easy solution to this problem is to compute the convex hull of $P_1 \cup P_2$ and, using the `naive-test`, test if it is inside P. Since P_1 and P_2 by assumption is already inside P, then the convex hull will not be outside P. The problem with this is that the convex hull might span a very large area and result in the check taking $\Omega(n)$ time. Instead lets make the following observation.

Lemma 4.1. *Let L_1 and L_2 be the two outer common tangents of P_1 and P_2, and let p_1 be a vertex of P_1 supporting L_1 and p_2 be a vertex of P_2 supporting L_2. Then the convex hull of $P_1 \cup P_2$ is inside P iff. p_1 fully sees P_2 and p_2 fully sees P_1 within P.*

[6] exercise 6.7

[7] disjointness is assumed for simplicity, it is possibly not necessary

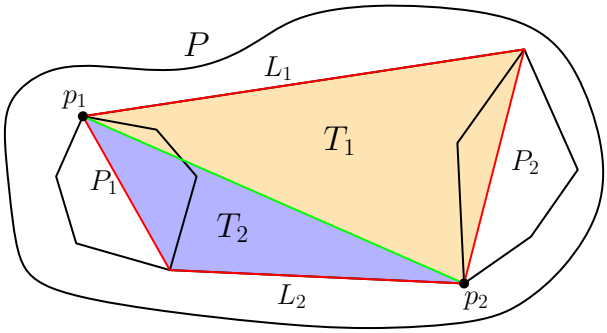


Figure 10: Visualization of common outer tangents, L_1 and L_2 of P_1 and P_2, and the corresponding supporting points p_1 and p_2.

Proof. Consider the convex quadrilateral, Q, defined by the 4 (possibly coincident) supporting points of L_1 and L_2 (the red shape in figure 10). Now consider segment $\overline{p_1p_2}$ (the green segment in figure 10). $\overline{p_1p_2}$ partitions Q into two triangles. Let triangle, T_1, be the triangle sharing two vertices with P_2, and T_2 be the triangle sharing two vertices with P_2. We first show the right-to-left implication. If p_1 fully sees P_2, then $T_1 \subset P$, and if p_2 fully sees P_1, then $T_2 \subset P$. Thus, if p_1 fully sees P_2 and p_2 fully sees P_1 then $Q \subset P$. If $Q \subset P$ and P_1 and P_2 by assumption is inside P, then $P_1 \cup Q \cup P_2 \subset P$. $P_1 \cup Q \cup P_2$ is the convex hull of $P_1 \cup P_2$, and thus P_1 and P_2 full see each other. Then we show the left-to-right implication. By definition if P_1 and P_2 fully see each other then every pair of points $p_1 \in P_1, p_2 \in P_2$ see each other. □

Based on lemma 4.1 we can test if P_1 and P_2 fully see each other by simply finding the supporting vertices, p_1 and p_2, and check if p_1 fully sees P_2 and p_2 fully sees P_1. This can be done by computing the visibility polygons, V_{p_1} and V_{p_2}, of p_1 and p_2 correspondingly. Then, we test if P_2 is contained in V_{p_1} and P_1 is contained in V_{p_2}. For this we can use the `star-test`, since V_{p_1} and V_{p_2} are visibility polygons, which are star-shaped. Note that in the `star-test` the meaning of "outside" and "partially contained" was slightly modified, however the meaning of "inside" is preserved. Thus, using the `star-test` in this case, does in fact give us precisely what we want.

The method seems smart, however it requires expensive calculation of V_{p_1} and V_{p_2}, and expensive `star-test` preprocessing of V_{p_1} and V_{p_2}. In order to justify this, suppose we are given k polygons, $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$, and we need to perform a lot of `fast-test` queries on pairs of polygons, $(P_i, P_j) \in \mathcal{P} \times \mathcal{P}$. We could compute all visibility polygons of all vertices of all polygons, and then each query can be performed quickly. However, the space usage after such prepossessing will be very high. Thus, suppose additionally that we initially get all the queries, and we have to report the answers to all queries at the end. This can be done while only storing a single visibility polygon at any point in time, and only constructing each visibility polygon once. Notice that the order of the two tests, $P_2 \subseteq V_{p_1}$ and $P_1 \subseteq V_{p_2}$, does not matter. Lets define a `fast-half-test` as one of $P_2 \subseteq V_{p_1}$ and $P_1 \subseteq V_{p_2}$. For each polygon P_i and a vertex v of P_i we will find all queries of the form (P_i, P_j) or (P_j, P_i), where the supporting vertex of P_i is v. Then we perform all `fast-half-tests` of the form $P_j \subseteq V_v$. We store the results of each `fast-half-test` and, in the end, for a query, (P_i, P_j), we will know the result of both `fast-half-tests`, and thus the result of the `fast-test`.

The `fast-test` is especially useful if the number of queries is much larger than the number of distinct polygons, since then every visibility polygon will be used many times. However, we can hardly provide any theoretical guarantees on the total running time. The `fast-test` is important, because it never performs a `naive-test` on the convex hull of P_1 and P_2.

4.4.4 One-sided-visibility-test

Given a triangulation, $\mathcal{T}$, of a polygon with holes, P, and two triangles $T_1, T_2 \in \mathcal{T}$, we wish to perform the following test: Is T_2 fully visible from T_1 within P. For the `one-sided-visibility-test` to be useful we also need to have a subset of triangles, $\mathcal{T}' \subset \mathcal{T}$, for which we know that every triangle, $T' \in \mathcal{T}'$, is fully visible from T_1.

Given this setup, the `one-sided-visibility-test` works as follows. We compute the convex hull, CH,

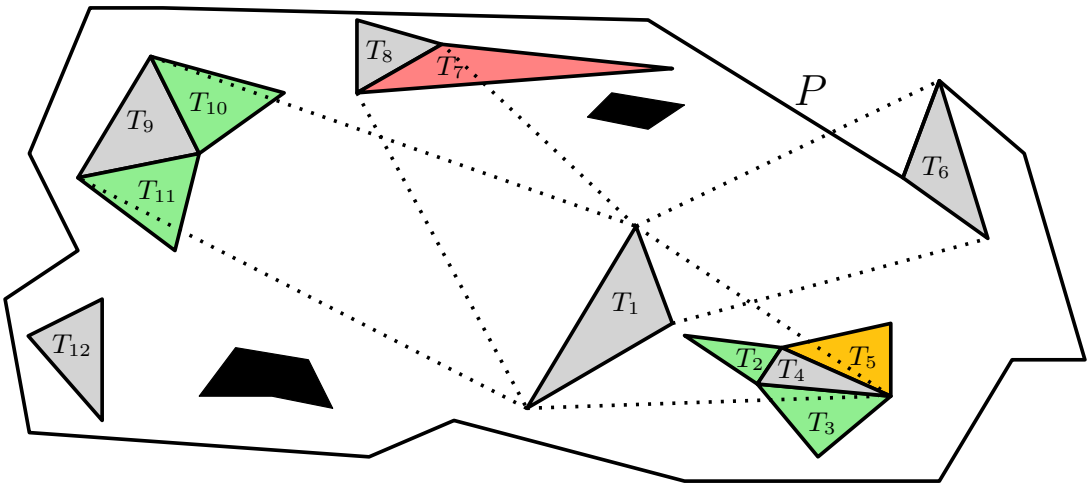


Figure 11: Green triangles are known to be fully visible from T_1, red ones are known to not be fully visible from T_1 and for orange ones it is unknown. We conclude that T_9 is fully visible, as T_{10} and T_{11} are. We cannot conclude that T_{12} is not fully visible, since the one side not on the boundary of the convex hull does not coincide with a side of P. We cannot conclude that T_8 is fully visible since T_7 is not. We cannot conclude right now that T_4 is fully visible as T_5 is unknown. We conclude that T_6 is not fully visible since a side not on the boundary of the convex hull coincides with a side of P.

of T_1 and T_2. We find all the sides, S, of T_2 that are not on the boundary of CH. For each side, $s \in S$, we find the neighbour, T_s, of T_2 in $D_{\mathcal{T}}$ who shares side s with T_2. Note that it might be the case that T_s does not exist if s coincides with a side of P. We can conclude that T_2 is fully visible from T_1 if for every $s \in S$ it holds that T_s exists and that $T_s \in \mathcal{T}'$. Otherwise we cannot conclude anything. See figure 11 for examples.

4.4.5 One-sided-invisibility-test

Given a polygon with holes, P, and two triangles T_1 and T_2, we wish to perform the following test: Is T_2 not fully visible from T_1 within P. Like for the `one-sided-visibility-test` we compute the set S (see section 4.4.4). If a side $s \in S$ coincides with a side of P then we can conclude that T_2 is not fully visible from T_1. Otherwise we cannot conclude anything. See figure 11 for examples.
If T_1 and T_2 are part of an extension triangulation, $\mathcal{T}$, then we expect the `one-sided-visibility-test` to be much more effective than the `one-sided-invisibility-test`, since many triangles will be strictly in the interior of P.

4.5 Efficient `visibility-BFS`

In section 3.2.2 we described the overall idea of how `visibility-BFS` works. Therefore section 3.2.2 is also a prerequisite for this section. `Visibility-BFS` may be made efficient in a lot of different ways. In this section we propose one such method.

What we need to speed-up is how to find a fully visible node on the boundary. We can either test if a node is not fully visible from the source and thus ignore it in the future, or we can test if a node is fully visible from the source and thus mark it as visited. We will use two types of tests. We may use the expensive `naive test`, which decides if a node is fully visible or note, or we may use the following **cheap tests**.

- `one-sided-visibility-test`: If it tells us that a node fully visible, then it is. Otherwise we cannot conclude anything.
- `one-sided-invisibility-test`: If it tells us that a node is not fully visible, then it is. Otherwise we cannot conclude anything.

- Cached results: Given that we want to test if T' if fully visible from source, T. Suppose we already from a previous run of `visibility-BFS` with T' as source know that T is fully visible T', then we also know that T' is fully visible from T. The same approach can be used to test if T' is not fully visible from T.

The **cheap tests** cannot always be used, and thus we have to fall back to the `naive test`. However, the strength of `visibility-BFS` is that it only visits fully visible nodes and thus the `one-sided-visibility-test` can be used often.

Our optimization of `visibility-BFS` will consist of two steps that are repeated until the boundary consists of only nodes that are not fully visible. In the first step we perform a *breadth-first-search* from the source and try to visit as many fully visible nodes as possible using only cheap tests. We also try to mark nodes as not fully visible using cheap tests, and we keep track of the boundary. When we cannot use the cheap tests anymore we go to the second step. In the second step we iterate the boundary and use the `naive test` until we find a node that is fully visible, which we then add to the BFS-queue and perform the first step again. If we do not find a fully visible node on the boundary during the second step then `visibility-BFS` terminates. The following is the pseudocode for `visibility-BFS`.

```
1  Visibility-BFS(T)
2      q = {T}
3      boundary = {}
4      while q is not empty
5          while q is not empty
6              node = q.front
7              mark node as visited
8              if node is on the boundary
9                  remove node from boundary
10             for neighbour, nb, of node in dual
11                 if nb is visited or nb is in q or nb is not fully visible
12                     continue
13                 if a cheap-test concludes nb being fully visible
14                     add nb to q
15                 else if a cheap-test concludes nb not being fully visible
16                     mark nb as not fully visible
17                 else # inconclusive
18                     add nb to boundary if it not already on the boundary
19         for node on the boundary
20             if naive test concludes node being fully visible
21                 add node to q
22                 break
23             else
24                 mark node as not fully visible
25                 erase node from boundary
26     return nodes that are marked as visited
```

Let us analyze the time complexity of `visibility-BFS` as a function of the total number of nodes, K, considered on lines 11-18. We will assume both the cheap tests and the `naive test` takes constant time, and we will ignore any polylogarithmic factors resulting from managing the boundary, queue, marking nodes and so on. Notice that a node is marked as visited at most once. Since, the nodes are triangles their degree in $D_{\mathcal{T}}$ is at most three. Thus, each node will be considered on lines 11-18 a constant number of times and therefore we can conclude that the (total) complexity of lines $5-18$ is $\tilde{\Theta}(K)$. Next up, each node will be added to the boundary at most once. If a node is on the boundary it will not be added, and after having been on the boundary it will be marked as either visited or not fully visible. Thus the (total) complexity of lines 19-25 $\tilde{\Theta}(K)$. In total the complexity is $\tilde{\Theta}(K)$, which is the best we can hope for.

4.6 Fixing cliques

In phase three we need to validate that the clique cover is valid. For each convex piece we simply use the `naive-test` (see section 4.4.1) to test if the piece is inside P. If not the clique is invalid, and we need to fix it. When fixing an invalid clique we need to find hole inside the clique. We can do this by constructing a range tree on the vertices of all the holes of P. Then we query the vertices that are not outside the smallest bounding box of the piece. Since the holes in an invalid clique are fully contained in the piece, we know that each hole in the piece has all its vertices among the the queried vertices. Given that we have a mapping from vertices to holes, we can find out, for each queried vertex, which hole is belongs to. That way we get the set of holes inside the piece.

5 Results

In this section we analyse results from the Challenge and additional benchmarks performed after the end of the Challenge. Section 5.1 describes the data, the implementation and the environment used for evaluations. In that section we give names to certain configurations of the algorithm, and we refer to these configurations throughout this section. Section 5.2 summarizes the results from the Challenge. Section 5.3 compares the partial visibility graph and the full visibility graph. Section 5.4 compares extension- and Delaunay-based configurations, and section 5.5 analyses other (special) configurations. Lastly, section 5.6 looks at the effects of phase three and section 5.7 looks at the effects of the post-processing.
As we see in figure 16, figure 17 and figure 19 the running time of the construction of visibility graphs completely dominates the running time of extension based configurations. This means that we did not spend time optimizing the implementations of the other parts of the algorithm during the Challenge, and therefore we will only focus our attention on the running time of the construction of the visibility graph in this section.

5.1 Experimental setup

In this section we describe the data, the implementation and the server used for evaluations.

5.1.1 Data

The testing data is based on the problem-instances used for evaluation during the Challenge. Several different kinds of polygons were present among the Challenge instances, which we summarize here. Examples of all the instance-types can be seen in figure 12.

- `cheese instances`: The cheese instances have an outer-boundary with few vertices, but with many small holes inside.
- `fpg instances`: The fpg instances have very sharp corners with few holes inside.
- `maze instances`: The maze instances have an outer-boundary with a grid of holes inside. Some of the holes are not quadratic and some are left out.
- `iso instances`: The iso instances are orthogonal[8] polygons that have a complicated outer-boundary with few holes inside.
- `mc instances`: The mc instances have a complicated outer-boundary with few holes inside. In the Challenge these were the instances prefixed with `srpg_mc`.
- `octa instances`: The octa instances are octagonal[9] polygons that have a complicated outer-boundary with few holes inside.
- `smo instances`: The smo instances have a smooth[10] outer-boundary boundary with few holes inside.

[8] All edges are either horizontal or vertical.
[9] The angle of any edge is a multiple of 45 degrees.
[10] Very long concave chains

- `smr instances`: The smr instances have a very smooth[10] outer-boundary boundary with few holes inside.

Among the Challenge instances were also the instance type, `socg`, which were small well-known examples. We exclude this type of instance in the coming sections.
In the benchmark test-sets we include a selection of instances of all types. These instances are not necessarily worst case instances, but are chosen to be representative of all the instances that are small enough to be evaluated in the given context.

5.1.2 Implementation

The implementations were made during the Challenge with very limited time constraints. Therefore, there are a few small differences between the Challenge implementation and the theory described in section 3 and section 4. These are the following.

- Instead of turning an invalid clique into (at most) $\lceil n_h/2 \rceil + 1$ valid cliques we turn it into (at most) $n_h + 1$ valid cliques. See section 3.3.2.
- In a `fast-half-test` we check all three corners of a given triangle instead of just a single supporting vertex (see section 4.4.3). This is a stronger test and thus it might, more often, make the other (opposite) `fast-half-test` unnecessary. However, this is still suboptimal.
- Since triangles of a triangulation share vertices, we cache (some) visibility polygons, when computing the full visibility graph. To get the most out of this cache, and the hardware cache, we run a DFS on the dual of the triangulation, and use the pre-order of the DFS-tree as the order to process the nodes when constructing the full visibility graph.

In phase one we choose a type of triangulation, in phase two we decide which variant of the visibility graph we want and in phase three we pick an algorithm to compute a clique cover. A lot of combinations of choices exist, but not all combinations leads to a useful algorithm. We will call such a combination a **configuration** of the Algorithm. The configurations that produced the best results during the Challenge were the following.

- `Delaunay-full`: The full visibility graph build on a `Delaunay-triangulation`.
- `Extension-full`: The full visibility graph build on an `extension-triangulation`.
- `Extension-partial`: The partial visibility graph build on an `extension-triangulation`.
- `Constrained-partial`: The partial visibility graph build on a `constrained-triangulation`.
- `Sampling-partial`: The partial visibility graph build on a `sampling-triangulation`.
- `Concave-full`: The full visibility graph build on a `concave-triangulation`.
- `Vispair-full`: The partial visibility graph build on a `vispair-triangulation`, but with only 10% of the visibility-pair segments included (randomly sampled).

Additionally, we implemented `Delaunay-partial`, which is the partial visibility graph build on a `Delaunay-triangulatio`
We also implemented many other configurations, some of them with types of triangulations not described in section 3.1. Some of these taught us, in which direction to take our approaches. The implementation only has open-source dependencies and is available at github: `https://github.com/willthbill/ExtensionCC` (commit hash: `cd780bb3a3e727194203d518f2da7430feedfcb6`).

5.1.3 Environment

All evaluations were performed on a server with two Intel Xeon E5-2690 v4166 processors, each with 14 cores (and 28 threads), and 504GB ram. Within the last 24 hours of the Challenge a multi-threaded implementation of the computation of full visibility graph was implemented using OpenMP (15.0.7-1), but all running reported running times in this section will be single-threaded. We used Boost (1.80.0) for various other purposes. The Challenge code was developed in C++ and compiled using GCC (11.3.0)

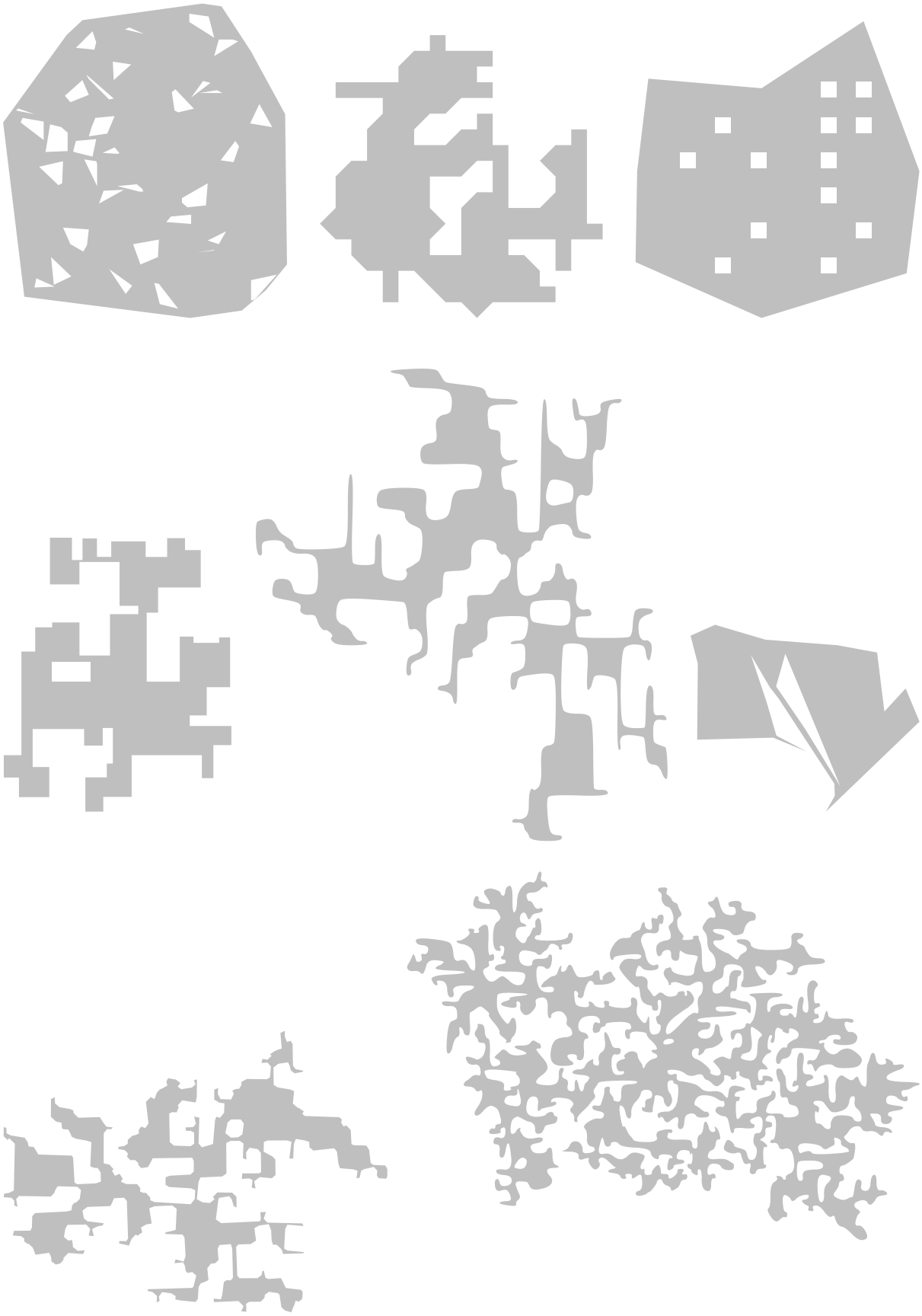


Figure 12: Examples of instance types `cheese`, `octa`, `maze`, `iso`, `smr`, `fpg`, `mc` and `smo` from left to right and top to bottom.

with `-O3` optimization turned on. CGAL (5.5.1-1) was used for geometric primitives and we used a Kernel that represented numbers as fractions with unlimited precision for exact predicates and constructions. We also also used the CGAL packages for triangulations[12], visibility polygons[7], segment- and range-trees[9], convex hulls[8] and boolean-set-operations[6].

5.2 Results on the Challenge instances

In this section we summarize the results from the Challenge. More than 1500 convex covers were computed for the 206 Challenge instances. The results on all of these instances is available in appendix 7.4. Evaluation time took anywhere from a less a few seconds to a few days. In figure 13 we show the solutions to the instances from figure 12. When a configuration is too slow (and was thus not evaluated on a given instance) it means that it took approximately more than 48 hours to evaluate[11]. We should also mention that most of the produced results include the post-processing step (see section 3.4), but not all. In figure 14 we show, for each instance, which configuration gave the best result.

- `cheese instances`: For small instances the difference between `extension-full` or `extension-partial` was insignificant and the solution quality was mainly controlled by the time allowed for phase three. For medium instances `extension-partial` was used, since `extension-full` became too slow. For large instances `constrained-partial` (see section 5.5.1) was used, since `extension-partial` became too slow.
- `fpg instances`: All instances, except the two largest, were solved using `extension-full`. `extension-partial` solved the two largest instances as `extension-full` became too slow.
- `maze instances`: A few maze instances were improved by `vispair-full` over `extension-full`. Small and medium instances were solved using `extension-full`, but `extension-full` became too slow on the large instances, and thus `extension-partial` was used instead.
- `iso instances`: For most instances (small and medium especially) `extension-full` was used. For 5 medium and large instances `extension-partial` was used as `extension-full` became too slow. For three very large instances `sampling-partial` was used, since `extension-partial` became too slow.
- `mc instances`: For three small instances `extension-full` was used. For most medium instances and one large instance `extension-partial` was used, since `extension-full` became too slow. For three large instance `sampling-partial` was used, since `extension-full` became too slow. For three of the largest instances we did not have time to sample enough extensions, and thus `delaunay-full` was used.
- `octa instances:` For all small, and some medium instances `extension-full` was used. For the rest of the instances, except one, `extension-partial` was used, since `extension-full` became too slow. For that last large instance `sampling-partial` was used.
- `smo-` and `smr instances:` Any extension based configuration was too slow, but `concave-full` easily fast enough and provided much better results that `delaunay-full`. A single `smr instance` used `delaunay-full`, since we only started evaluating `concave-full` on it a few hours before the deadline of the Challenge.

To summarize, the vast majority of "best" results were found using either `extension-full` (smaller instances) or `extension-partial` (larger instances). It seems that `vispair-full` does, as expected (see section 3.1.2, give better results than extension based configurations, however it is too slow for almost every instance. Figure 15 shows an example where `vispair-full` performed better than `extension-full`. `constrained-partial` and `sampling-partial`, which are based on extensions, worked well (were fast) on special types of instances. For `smo-` and `smr instances` the `concave-full` configuration provided the best results, since any extension based configuration was too slow. A solution using `delaunay-full` was computed for every single instance, but only 4 instances did not improve using another configuration (because every other configuration was too slow).

[11] this was based on our judgement during the Challenge.

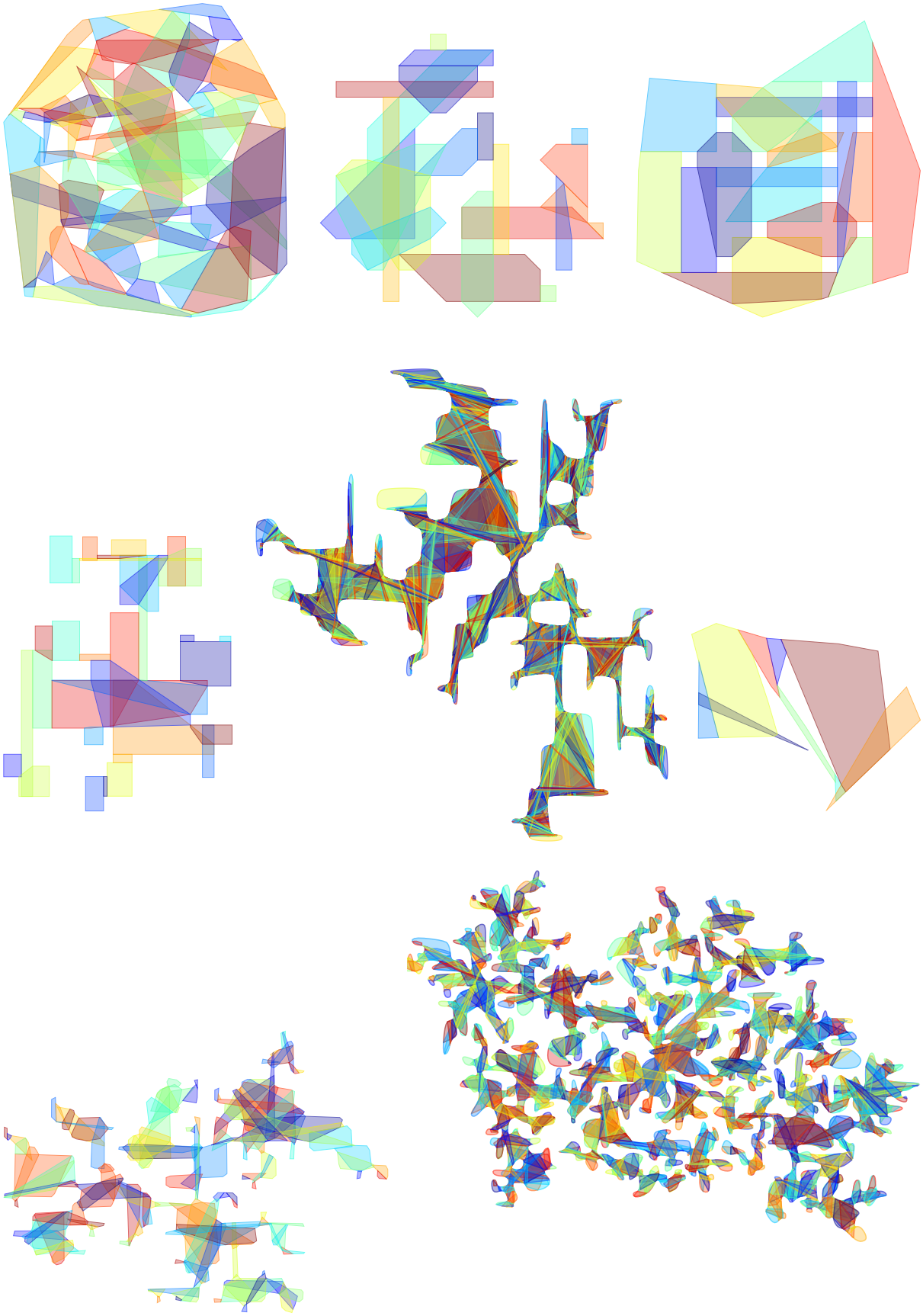


Figure 13: Solutions to the examples from figure 12. Solutions sizes are 65 for `cheese`, 22 for `octa`, 18 for `maze`, 33 for `iso`, 1312 for `smr`, 9 for `fpg`, 152 for `mc` and 1179 for `smo` from left to right and top to bottom.

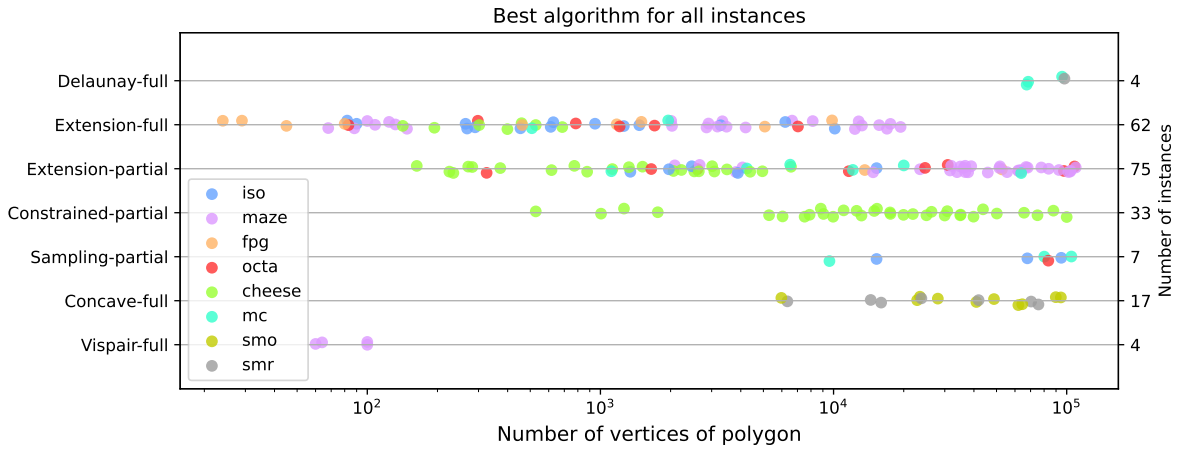


Figure 14: Each instance has a point, which is located next to the algorithm which produced the best solution for that instance. Note that for some of the small instances many configurations got the same "best" result, we chose `extension-full` to represent these instances.

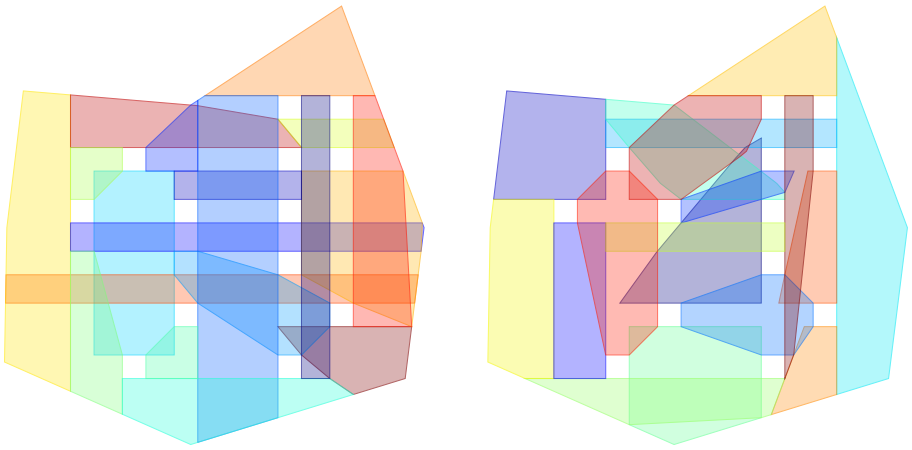


Figure 15: For the `maze instance` from figure 12. To the left a solution using `extension-full` with 19 pieces and to the left a solution using `vispair-full` with 18 pieces.

5.3 Comparing the partial visibility graph and the full visibility graph

In phase two of the algorithm we construct a visibility graph. This is a critical part of the algorithm in terms of both use of resources and solution quality. The largest observed graph had 360592 vertices and 2133838660 edges, and anything larger than this is very impractical. The vision for the partial visibility graph was that it should make a sensible compromise between decreasing resource spending and increasing solution size. Consider figure 16 comparing `delaunay-partial` and `delaunay-full`, and also consider figure 17 comparing `extension-partial` and `extension-full`. In both figures we observe that the use of the partial visibility graph decreases computation time significantly, which is mostly due to the use of the two fast visibility tests, `one-sided-visibility-test` (see section 4.4.4) and `one-sided-invisibility-test` (see section 4.4.5).

For figure 16 the memory usage is reduced as the size of the visibility graph is reduced. In contrast the decrease in memory usage is not as significant in figure 17, which corresponds well with the fact that there is a only a minuscule difference in the size of the graph. Still, however, in both figures the difference in graph size does not seem to justify the difference in memory usage. We postulate that this is because we cache visibility polygons during the computation of the full visibility graph.

Lastly, we see that in figure 16 there is a noticeable increase in solution size when using the partial visibility graph, whereas in figure 17 the difference is minuscule. For the middle instances in figure 14 it is unclear if `extension-partial` or `extension-full` is better in general, but this minuscule difference indicates that the winner is determined by the amount of time given to phase three. Either way, this tells us that extension-based configurations are more resilient in preserving solution quality, while reducing running time, when using the partial visibility graph. It seems logical that this is due to the fact that extension segments represent important visibility lines within the polygon. It also seems likely that the use of of more extension segments correlates with a triangulation that is more resilient to the use of a partial visibility graph instead of a full visibility graph. Still we must admit that both figures indicate that the increase in solution size is, in any case, a good compromise considering the significant decrease in running time.

5.4 Comparing `Extension-triangulation` and `delaunay-triangulation`

In this section we will use the `extension-partial` configuration to learn about why extension segments are a good choice. The use of `extension-partial` instead of `extension-full` in this analysis will also allow us to benchmark some larger instances[12]. In figure 18 is shown an example of a delaunay-based convex cover and an extension-based convex cover.

From figure 16 and 17 it is not easy to compare Delaunay-based configurations and extension-based configurations. Consider instead figure 19 comparing `delaunay-partial` and `extension-partial`. We see that solutions produced with `extension-partial` are consistently better than solutions produced with `delaunay-partial`. As mentioned in previous sections, we believe this is caused by how extension segments incorporate important visibility lines of the polygons into the triangulation. Still, however, the quality of the solution comes at the cost of a much larger visibility graph that leads to much higher memory usage. Also, the running time increases and for `extension-partial` the running time is completely dominated by the construction of the visibility graph. In contrast, for `delaunay-partial` constructing the visibility graph takes up 32.8% of the running time on average. Still the top priority of the algorithm (and especially in the Challenge) is to find small convex covers.

Based on this section, section 5.3 and section 5.2 we conclude the following. For small instances a full visibility graph is preferred, since the running time will be reasonable, and the full visibility graph produces slightly better solutions than the partial visibility graph. However, the difference in solution quality between `extension-partial` and `extension-full` is insignificant compared to the difference in running time. Moreover, `vispair-full` is extremely slow, `delaunay-partial` is much worse than `extension-partial` and the union of the constraints (segments) in either of the triangulations used in `constrained-partial`, `sampling-partial` and `concave-full` is a subset of the set of all `extension segments`. Based on this we (empirically) conclude that, in general, if we do not know the properties of the input polygon, then `extension-partial` is the best configuration.

[12] we could also have compared with `delaunay-full`, since that is the best Delaunay-variant, but it also seems sensible to only change the type of triangulation.

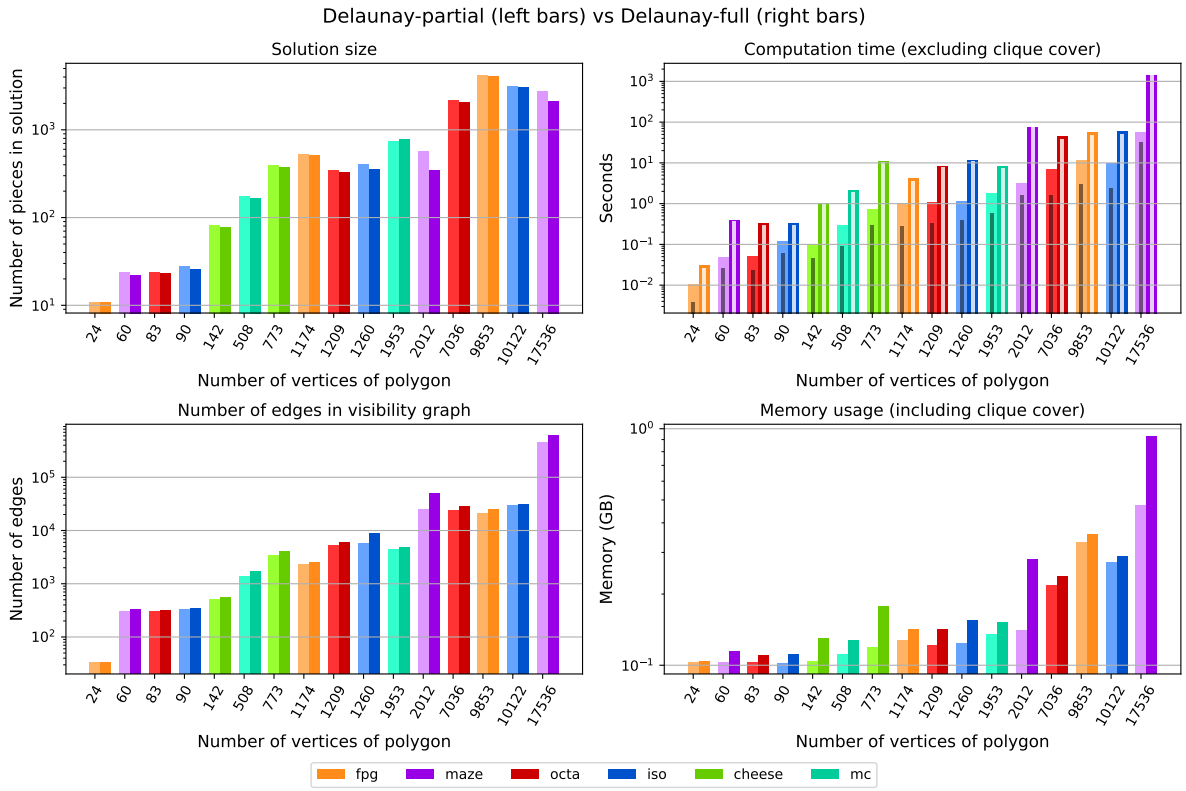


Figure 16: Experiments comparing `delaunay-partial` (left bars) to `delaunay-full` (right bars). We compare the solution size (top left), running time (top right; thin bars showing the running time of the visibility graph construction), number of edges in the visibility graph (bottom left) and memory usage (bottom right).

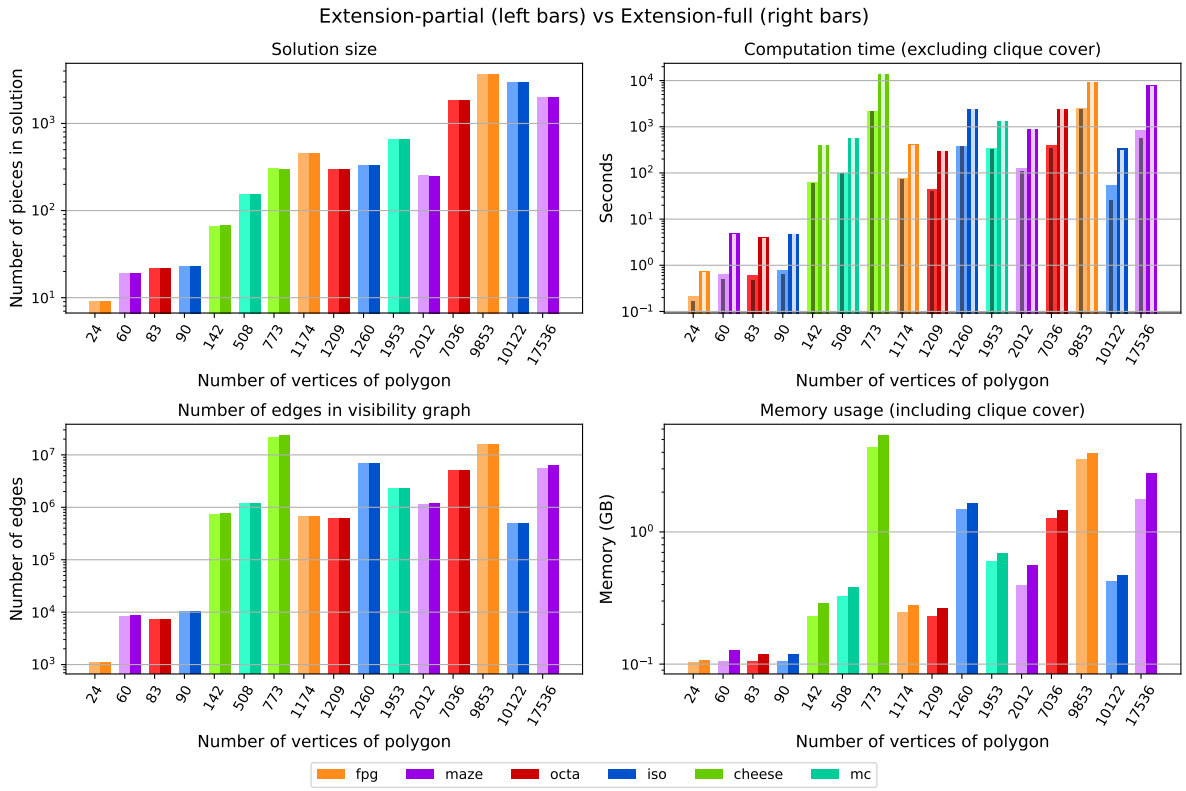


Figure 17: Experiments comparing `extension-partial` (left bars) to `extension-full` (right bars). We compare the solution size (top left), running time (top right; thin bars showing the running time of the visibility graph construction), number of edges in the visibility graph (bottom left) and memory usage (bottom right).

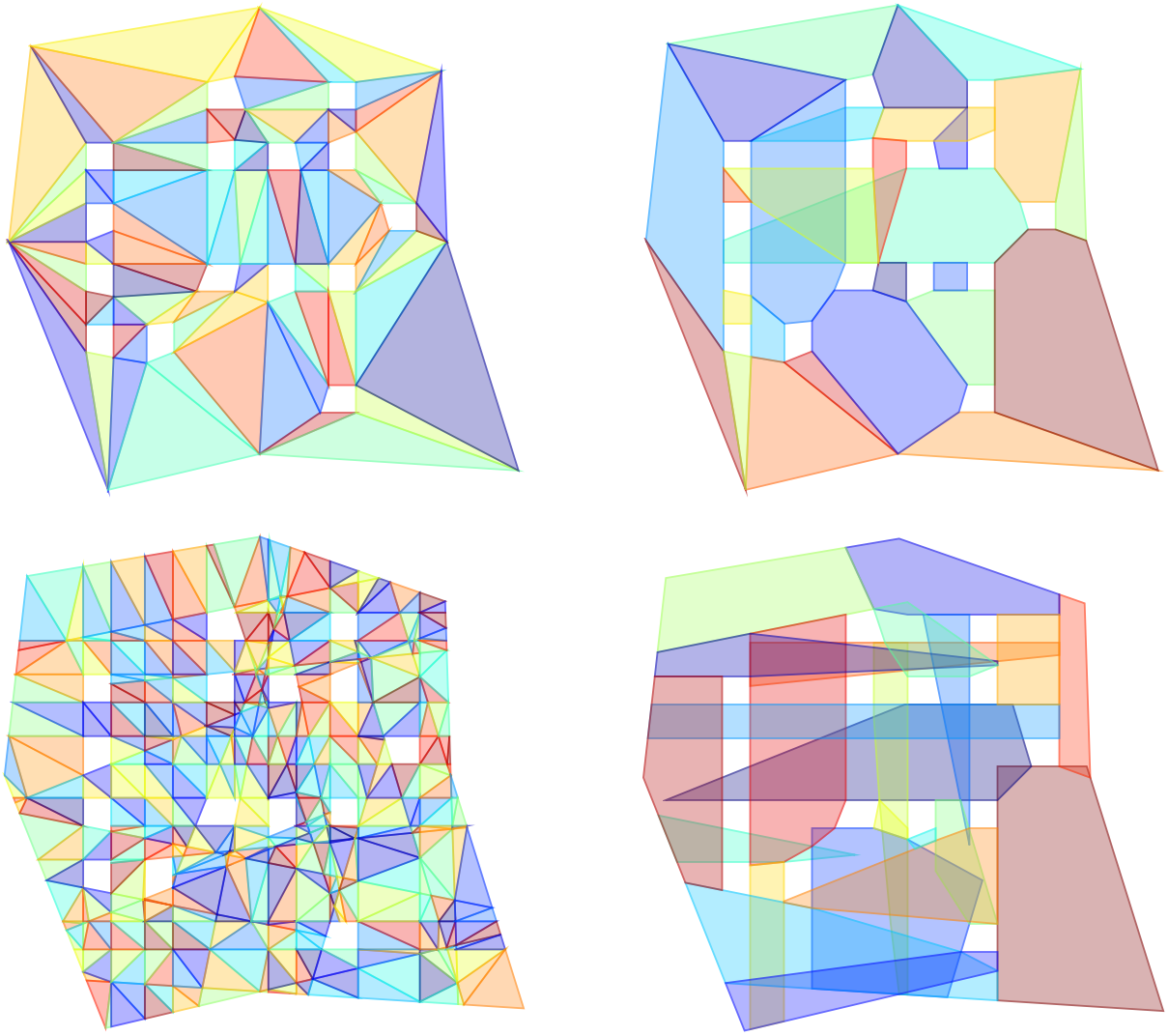


Figure 18: The Delaunay triangulation (top left) and the resulting cover of size 27 (top right). The extension partition (bottom left) and the resulting cover of size 23 (bottom right).

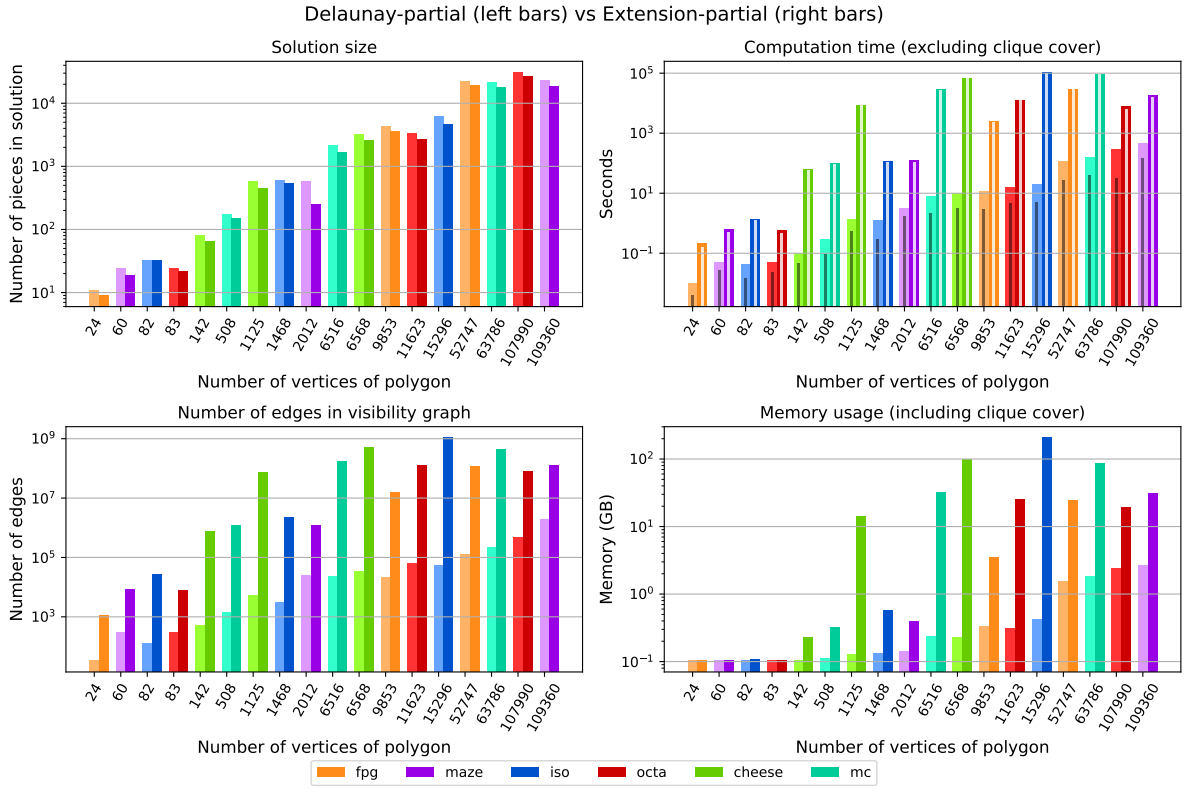


Figure 19: Experiments comparing `delaunay-partial` (left bars) to `extension-partial` (right bars). We compare the solution size (top left), running time (top right; thin bars showing the running time of the visibility graph construction), number of edges in the visibility graph (bottom left) and memory usage (bottom right).

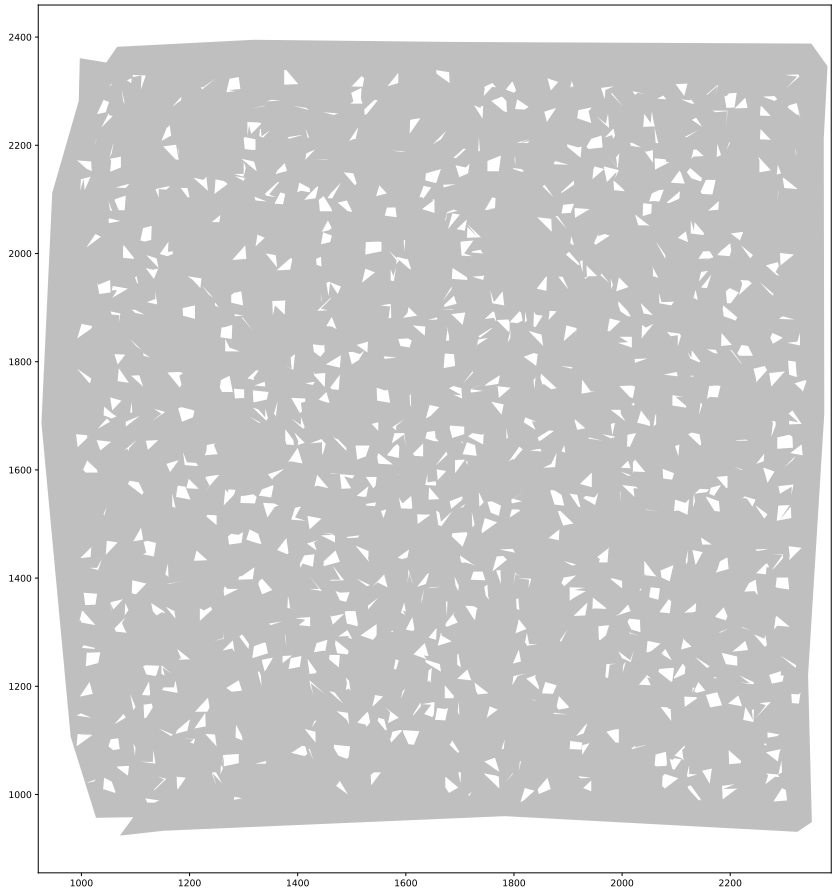


Figure 20: A cheese instance. Holes are (close to) evenly distributed inside the boundary.

5.5 Special types of triangulations

Ideally we would have liked to run either `extension-full` or `extension-partial` on all instances. However, for some instances this was infeasible in terms of both running time and space consumption. For these instance we instead designed the specialized configurations, `constrained-partial`, `sampling-partial` and `concave-full`.

5.5.1 Constrained-partial

The `constrained-partial` configuration uses the `constrained-extension triangulation` (see section 3.1.3). This configuration was used for large `cheese instances`, since it turned out to be much faster than `extension-partial` and still produced good solutions. Consider the large cheese instance shown in figure 20. It is clear that holes are distributed evenly inside the outer convex boundary. It seems likely that good solutions to a cheese instance does not use very long pieces. Thus, we assume pieces to be local (i.e. made up of triangles close to each other), and thus we simply throw away all extension segments of length greater than some constant. In this case we chose the constant to be 160. We could choose the same constant for all `cheese instances`, since the distances between points that see each other is (approximately) the same. With this constant we had time to run the `constrained-partial` configuration on all cheese instances. `extension-partial` is most likely slow on cheese instances, since the holes are small and therefore extension segments can stretch over large distances without intersecting holes. This results in many intersections.

5.5.2 Sampling-partial

Suppose we have a polygon with the following properties.

- Distances between points that see each other varies a lot.
- The polygon has large open spaces.

See the `mc` instance in figure 12 for an example of such a polygon. In the large open spaces a lot of intersections between extension segments will occur, and therefore we need to restrict the number of extension segments used. We cannot use `constrained-partial`, since the distances between points that see each other vary. Instead we will use `sampling-partial`, which uses the `extension-sampling triangulation` (see section 3.1.4) that throws away segments of different lengths. In general, we expect convex covers to be local in the sense, that pieces of the cover do not stretch across large distances. That is why `sampling-partial` works well, since it samples more short segments than long segments. We can use `sampling-partial` for most types of instances and simply adjust the number of sampled extension segments. `sampling-partial` came in useful for `mc instances`, `iso instances` and `octa` instances with large open spaces. We sampled as many extension segments as possible, given the time constraints of the Challenge. However, for three of the `mc instances` with very large open spaces, we were not able to sample enough extension segments to beat `delaunay-full`, whose strength is the use of the full visibility graph.

5.5.3 Concave-full

The `concave-full` configuration was used for `smo-` and `smr instances` since all extension-based configurations were too slow to for theses instances. Aside from the local triangulation along concave chains the `concave-full` configuration also places a grid on top of the polygon. As mentioned in section 3.1.5, assuming the number of gridlines is a constant, then the cardinality of the `concave-triangulation` is $O(n)$. However, even though overlaying a grid lead to slightly better results, it did so at the cost of noticeably higher running time. This is most likely caused by the quadratic number of edges that occur within a grid, and most of these edges will be unnecessary, since the grid is oblivious to the composition of the polygon. The local triangulation part of the `concave-full` configuration is much more important than the grid. For example on the smallest `smr instance`, as shown in figure 12, `delaunay-full` gave 1905 pieces, `concave-full` with a 200x200 grid gave 1312 pieces and `concave-full` with no grid gave 1430 pieces. Whether or not a grid is overlaid, `concave-full` is better than `delaunay-full`, and we have the following idea, as to why this is. Segments of a concave chain must be part of different pieces in the convex cover. Therefore, the idea behind the `concave-full` configuration is to have edges between triangles (representing segments) of different concave chains. Since the triangles along a concave chain are small it is likely that such edges exist. Figure 21 shows how the local triangulation allows for larger pieces connecting more than two segments. In contrast, `Delaunay-full` seems to create a partition where only a single piece connects more than two segments.

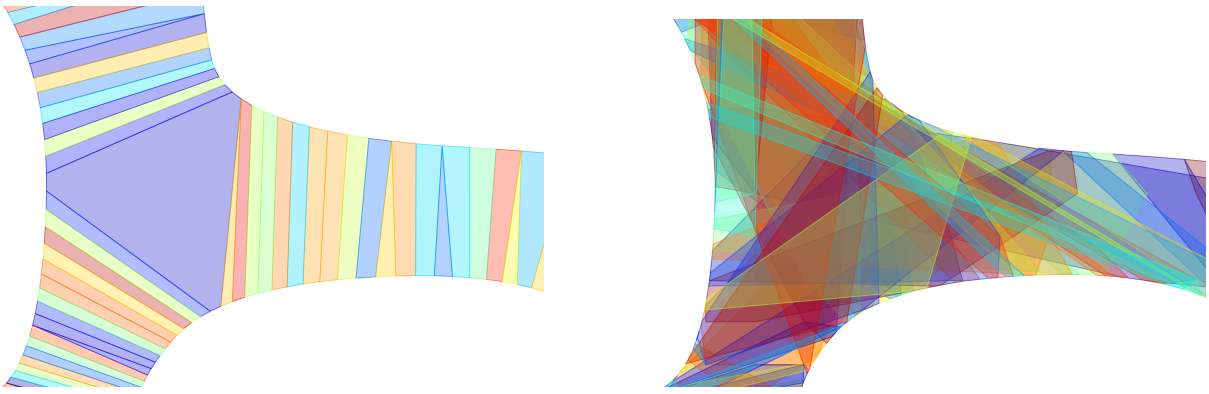


Figure 21: Part of a polygon with multiple long concave chains. While `delaunay-full` almost exclusively creates pieces adjacent to only two concave chains (left), `concave-full` creates pieces that contain parts of all three concave chains (right).

5.6 Finding a small valid clique cover

In this section we describe the effects of phase three (see section 3.3) - finding a small valid clique cover.

5.6.1 Computing a clique cover

We used `chalupa` to compute small clique covers (see section 3.3.1). `Chalupa` iteratively finds smaller and smaller clique covers. For most instances `chalupa` reaches a point where it takes a long time to find improvements. `chalupa` did not converge, i.e. find an optimal clique cover, on any instances except the smallest ones. We gave `chalupa` a time-limit of anywhere between 5 minutes to 25 hours depending on the size of the instance. This time-limit was divided evenly into 5 independent runs of `chalupa`, since sometimes `chalupa` converged quickly with a bad solution. Only a few times did we not get a small clique cover with `chalupa` among the 5 runs, but to fix this we simply reran the algorithm and got a good result. We chose the time-limit such that, we expected any further improvements to take hours. After the Challenge we have observed instances that did manage to improve further when given an even higher time-limit.

5.6.2 Invalid cliques

Invalid cliques occur when the convex hull of a clique contains a hole. Based on our benchmarks we observed invalid cliques in `cheese`, `maze`, `iso`, `mc` and `octa` instances. However, only `cheese` and `maze` instances are interesting to consider further as they are the only instance-types containing many holes. It was only for `cheese` instances that we saw the cover size increase by more than 6 cliques. For large `cheese` instances it seems, based on benchmarks, that the cover size increases by at most 6%. For the largest `cheese` instance the clique cover increased from having 36583 cliques to 37190. In contrast there were almost no invalid cliques in `maze` instances even though they contain many holes. This is most likely due to the sides of the holes being aligned horizontally and vertically, and therefore good solutions to `maze` instances uses very long horizontal and vertical pieces that cannot wrap around holes (see figure 22). By far, the most invalid cliques contain only a single hole. Therefore, it does not make much of a

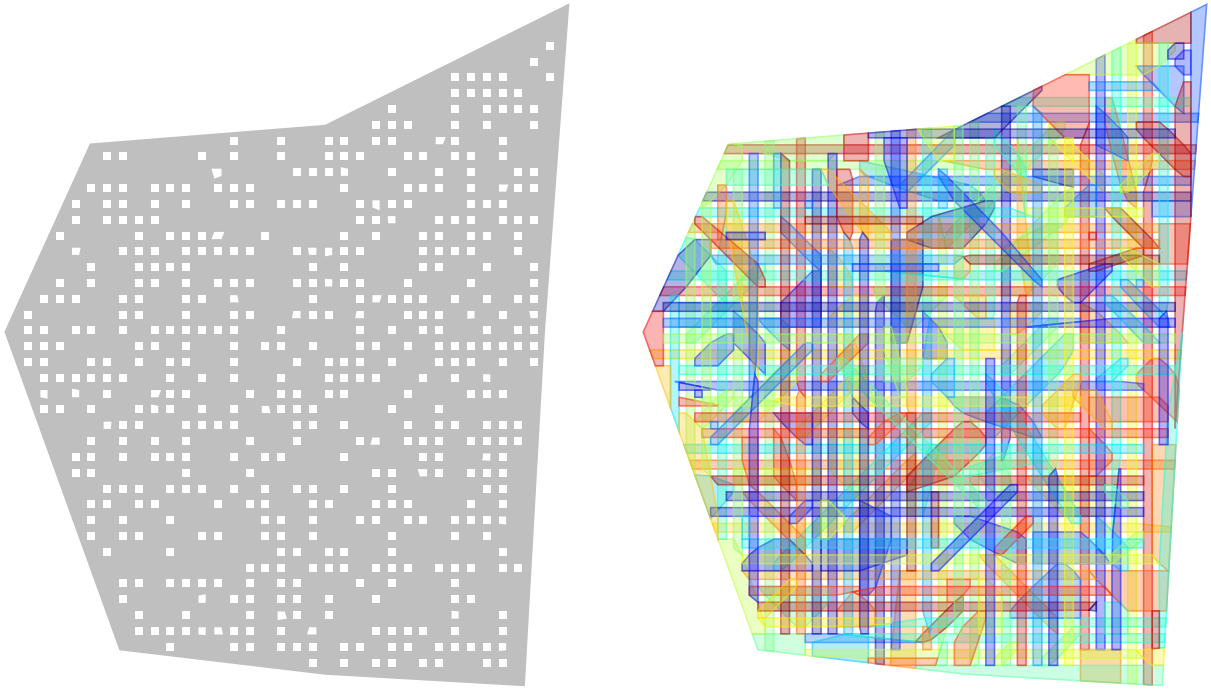


Figure 22: The best solution found to a `maze` instance. A lot of long horizontal and vertical pieces are used.

difference whether we choose half-planes intersecting two holes or one hole (see section 3.3.2).

5.7 Effects of post-processing

Our algorithm is not guaranteed to produce a minimal convex cover after phase three. Thus as part of the post-processing we make it minimal (see section 3.4). The largest observed improvements were 10 pieces for `maze` instances, two pieces for `octa` instances, three pieces for `mc` instances, 83 pieces for `cheese` instances, two pieces for `iso` instances, 7 pieces for `smo` instances and 13 pieces for `smr` instances. In general all large cheese instances were improved by 10 to 83 pieces.

6 Conclusion

We have proposed a heuristic algorithm for computing small convex covers efficiently. The high quality output of the algorithm is especially due to the use of extension segments — segments within the polygon coinciding with edges of the polygon — which are used as constraints in a constrained Delaunay triangulation. From the triangulation a visibility graph is efficiently constructed using output-sensitive explorations of the dual of the triangulation, and using efficient polygon-in-polygon containment tests. This allows for the construction of visibility graphs with more than two billion edges. Using the state of the art `ReduVCC` implementation of the `Chalupa` clique cover algorithm, we are able to find very small clique covers of the visibility graph. The convex hull of the triangles in the cliques represent pieces in the convex cover, and only few of the cliques do not correspond to valid pieces and therefore need to be fixed.

The algorithm is generic and we describe various configurations of the algorithm that perform well in practice. Most successful is the use of extension segments and slightly reduced visibility graphs, which is the best compromise between quality and speed. Variations of this configuration allow for high quality solutions to many different types of polygons with over 100000 vertices.

6.1 Future work

The presented algorithm shows promise, but it is still slow in practice for large polygons. Exploring how to reduce the size of the visibility graph is critical for reducing the running time and memory usage. We may do this by better recognizing which extension segments are more important than others.

It would be interesting to see a highly optimized implementation of the algorithm. Especially, an implementation using the ReduVCC clique cover solver (with reductions).

It would be interesting to explore how a general version of the algorithm that requires no parameters could look like. Potentially, such an algorithm could work like this. First we locally triangulate concave chains, then we sample extension segments, and lastly we sample other visibility segments of the polygon. We suggest adding segments in that order and stopping once the visibility graph is expected to become too large. It would be interesting to see if the solution quality increases linearly with the number of used segments or not.

Lastly it could be interesting to explore, what would happen if we purposely ignore potential edges of the visibility graph, and at the same time look for communities[13] instead of cliques in the visibility graph.

Acknowledgements

I would like to thank my teammates, André Nusser and Mikkel Abrahamsen, for great teamwork during the Challenge. I also thank the CG:SHOP 2023 organizers and the other participants of the Challenge — especially Guilherme Dias da Fonseca of team `Shadoks` (second place in the Challenge) for making us fight for the first place until the very last minute. Thanks to Martin Aumüller and Rasmus Pagh for assistance with the server used for evaluations during and after the Challenge. Thanks to Darren Strash for helping with the ReduVCC implementation. The ReduVCC implementation was invaluable to the success of our algorithm. Lastly, I would like to thank my good friend, Ásþór Björnsson, for listening to me explain the project and for the many valuable discussions.

[13] Subsets of nodes that are densely connected.

References

[1] Mikkel Abrahamsen. "Covering Polygons is Even Harder". In: *Symposium on Foundations of Computer Science (FOCS)*. 2021, pp. 375–386. DOI: 10.1109/FOCS52979.2021.00045. URL: https://doi.org/10.1109/FOCS52979.2021.00045.

[2] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736.

[3] David Chalupa. "Construction of Near-Optimal Vertex Clique Covering for Real-World Networks". In: *Comput. Informatics* 34.6 (2015), pp. 1397–1417. URL: http://www.cai.sk/ojs/index.php/cai/article/view/1276.

[4] Stephan J. Eidenbenz and Peter Widmayer. "An Approximation Algorithm for Minimum Convex Cover with Logarithmic Performance Guarantee". In: *SIAM Journal on Computing* 32.3 (2003), pp. 654–670. DOI: 10.1137/S0097539702405139. eprint: https://doi.org/10.1137/S0097539702405139. URL: https://doi.org/10.1137/S0097539702405139.

[5] Sándor P. Fekete et al. *Minimum Coverage by Convex Polygons: The CG:SHOP Challenge 2023*. 2023. arXiv: 2303.07007 [cs.CG]. URL: https://arxiv.org/abs/2303.07007.

[6] Efi Fogel et al. "2D Regularized Boolean Set-Operations". In: *CGAL User and Reference Manual*. 5.5.2. CGAL Editorial Board, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgBooleanSetOperations2.

[7] Michael Hemmer et al. "2D Visibility Computation". In: *CGAL User and Reference Manual*. 5.5.2. CGAL Editorial Board, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgVisibility2.

[8] Susan Hert and Stefan Schirra. "2D Convex Hulls and Extreme Points". In: *CGAL User and Reference Manual*. 5.5.2. CGAL Editorial Board, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgConvexHull2.

[9] Gabriele Neyer. "dD Range and Segment Trees". In: *CGAL User and Reference Manual*. 5.5.2. CGAL Editorial Board, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgSearchStructures.

[10] L. Paul Chew. "Constrained delaunay triangulations". In: *Algorithmica* 4.1 (June 1989), pp. 97–108. ISSN: 1432-0541. DOI: 10.1007/BF01553881. URL: https://doi.org/10.1007/BF01553881.

[11] Darren Strash and Louise Thompson. "Effective Data Reduction for the Vertex Clique Cover Problem". In: *Symposium on Algorithm Engineering and Experiments (ALENEX)*. 2022, pp. 41–53. DOI: 10.1137/1.9781611977042.4. URL: https://doi.org/10.1137/1.9781611977042.4.

[12] Mariette Yvinec. "2D Triangulations". In: *CGAL User and Reference Manual*. 5.5.2. CGAL Editorial Board, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgTriangulation2.

7 Appendix

7.1 Team DIKU(AMW) CG:SHOP23 article

Constructing Concise Convex Covers via Clique Covers

Mikkel Abrahamsen
University of Copenhagen, Denmark

William Bille Meyling
University of Copenhagen, Denmark

André Nusser
University of Copenhagen, Denmark

Abstract

This work describes the winning implementation of the CG:SHOP 2023 Challenge. The topic of the Challenge was the convex cover problem: given a polygon P (with holes), find a minimum-cardinality set of convex polygons whose union equals P. We use a three-step approach: (1) Create a suitable partition of P. (2) Compute a visibility graph of the pieces of the partition. (3) Solve a vertex clique cover problem on the visibility graph, from which we then derive the convex cover. This way we capture the geometric difficulty in the first step and the combinatorial difficulty in the third step.

2012 ACM Subject Classification Theory of computation → Computational geometry; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Convex cover, Polygons with holes, Algorithm engineering, Vertex clique cover

Category CG Challenge

Supplementary Material `https://github.com/willthbill/ExtensionCC`

Funding *Mikkel Abrahamsen*: Supported by Starting Grant 1054-00032B from the Independent Research Fund Denmark under the Sapere Aude research career programme. Part of BARC, supported by the VILLUM Foundation grant 16582.
William Bille Meyling: Supported by Starting Grant 1054-00032B from the Independent Research Fund Denmark under the Sapere Aude research career programme.
André Nusser: Part of BARC, supported by the VILLUM Foundation grant 16582.

Acknowledgements We want to thank the CG:SHOP 2023 organizers and the other participants (especially Guilherme Dias da Fonseca) for creating such a fun challenge and for helpful comments on our write-up. We also want to thank Martin Aumüller and Rasmus Pagh for access and help with using their server. Finally, we want to thank Darren Strash for quick and last-minute support using the ReduVCC implementation.

1 Introduction

Covering a polygon with the minimum number of convex pieces is a fundamental problem in computational geometry and the problem chosen for the CG:SHOP 2023 Challenge. In this problem we are given a polygon P (potentially with holes) and we have to find a smallest possible set of convex polygons whose union equals P. This problem is NP-hard [3] and was later shown to be even $\exists\mathbb{R}$-complete [1]. Note that in the Challenge, all coordinates of the solutions had to be rational, and then the decision problem is not even known to be decidable. Thus, it is not expected that there exists any fast algorithm that always finds the optimal solution. Likewise, we are aware of no previously described algorithm which is fast in practice. In this 5th CG:SHOP Challenge, there were a total of 22 teams who signed up, out of which 18 submitted solutions. Our team — named *DIKU (AMW)* — obtained

the highest total score, despite finding fewer smallest solutions than the runner-up [4], as we achieved significantly smaller solutions on many of the largest Challenge instances. See [5] for a survey about the Challenge.

Our algorithm consists of three steps. In the first step (see Section 2.1), the aim is to capture the geometry of the problem. We do this by partitioning the input polygon P into triangles. Note that the corners of these triangles do not have to be corners of P but can be Steiner points. In the second step (see Section 2.2), we move from the geometric structure to a combinatorial structure. We do this by computing a visibility graph G of the partition, with each triangle corresponding to a vertex and an edge is inserted for two vertices only if the convex hull of the corresponding triangles lies within P (i.e., the convex hull would be a valid piece of the convex cover). Finally, in the third step (see Section 2.3), we solve a combinatorial problem: we find a vertex clique cover (VCC) of G with small cardinality. Recall that a *vertex clique cover* is a set of cliques in G, for which each vertex in G appears in one of the cliques. When possible, we use the convex hull H of the triangles of a clique C as a piece of our convex cover. However, H may intersect holes of P, which makes H an invalid piece. This happens rarely for the Challenge instances, but in that case we split C into smaller cliques. For us, the main insights and highlights of our approach are:

1. Assembling the pieces and the cover at the same time (instead of first deciding on the pieces and then assembling the cover) allows for great flexibility and adaptivity.
2. Reduction to a fundamental graph problem allows for usage of a powerful set of already existing tools.
3. As we can arbitrarily choose a partition in the first step of the algorithm, our approach is very adaptive with respect to input structure and instance size (simpler partitions can be chosen for larger instances).

2 Algorithm

In this section we describe our algorithmic approach to solve the convex cover problem.

2.1 Partition

First, we partition the polygon. While our approach in principle works with any kind of partition, for simplicity we only used partitions consisting of triangles. Recall that the goal of the partition is to obtain triangles from which we can later assemble good pieces for a convex cover and that the corners of these triangles are not restricted to lie on the corners of P. In fact, to obtain good solutions one often needs Steiner points.

The simplest partition that we use is a Delaunay triangulation. We prefer a Delaunay triangulation over an arbitrary triangulation because it leads to fat triangles, which we intuitively assume to create better pieces for the convex cover. The main issues of using a Delaunay triangulation of P as partition are that its vertices are restricted to the corners of P and that the pieces can be too coarse to merge into convex pieces. See Figure 6 for an example for which this leads to a suboptimal cover. Thus, the question is: which Steiner points should we introduce to obtain better solutions?

Consider a directed edge e of P and suppose that the interior of P is to the left of e. We define the *extension* of e to be the maximal directed segment s such that $e \subseteq s \subseteq P$; see Figure 1 (middle). Note that a piece Q of a convex cover can contain an interior point of e only if Q does not contain a point to the right of s. Thus, intuitively it make sense to include pieces of the cover that are bounded by s. This intuition is captured by the *extension*

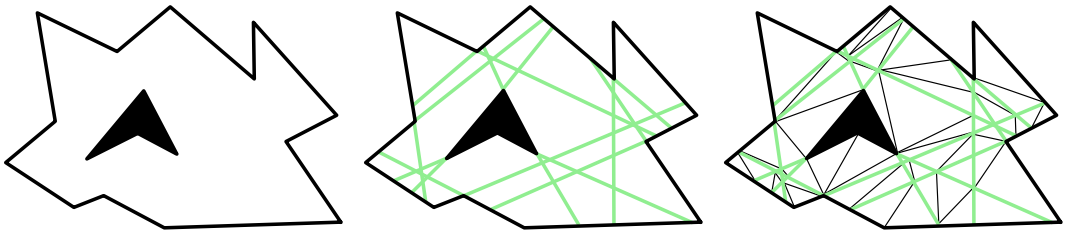


Figure 1 Left: Polygon P (left), extensions of P (middle), and extension partition of P (right).

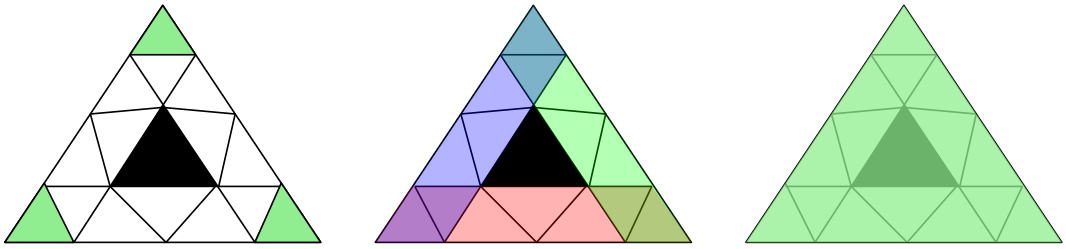


Figure 2 For the set of green triangles (left), every pairwise convex hull is contained in P (middle), but the convex hull of all the green triangles is not (right).

partition, which is the constrained Delaunay triangulation of the extensions of all edges of P; see Figure 1 (right).

2.2 Visibility Graph

In order to create a convex cover, we first want to understand which triangles we can potentially combine to form pieces for the cover. Given a partition $\mathcal{P}$ of the polygon P and two triangles $p, q \in \mathcal{P}$, we say that q is *fully visible* from p if every point in p sees all of q and *partially visible* if every point in p sees some point in q. We define the visibility graph $G = (\mathcal{P}, E)$, which contains an edge pq if the convex hull of $p \cup q$ is contained in P. We can compute G naively by checking for each pair $p, q \in \mathcal{P}$ whether its convex hull is contained in P. However, the running time $\Omega(|\mathcal{P}|^2)$ renders this impractical. A simple observation comes in handy here: For any triangles $q \in \mathcal{P}$ fully visible from $p \in \mathcal{P}$, there exists a path from p to q in the dual graph[1] of $\mathcal{P}$ using only vertices that correspond to triangles that are partially visible from p. Thus, instead of checking all pairwise visibilities, we can simply perform a BFS on the dual graph, only using partially visible triangles and stop exploring on triangles that are not partially visible. While this significantly reduces the running time in practice, it can still be too expensive. For further speedup, we resort to building a subgraph of G by only exploring fully visible triangles in the BFS. To speed up the visibility graph construction, we engineered fast visibility checks that we do not further describe here.

2.3 Compute Cover

We employ the following three steps to compute a convex cover using the visibility graph.

Compute Vertex Clique Cover: We first compute a vertex clique cover (VCC) on the visibility graph. The problem of finding a minimum VCC is one of Karp's classical NP-hard

[1] The dual graph of a partition is defined as follows: the vertex set consists of the triangles of the partition and there is an edge between two vertices iff the two corresponding triangles touch.

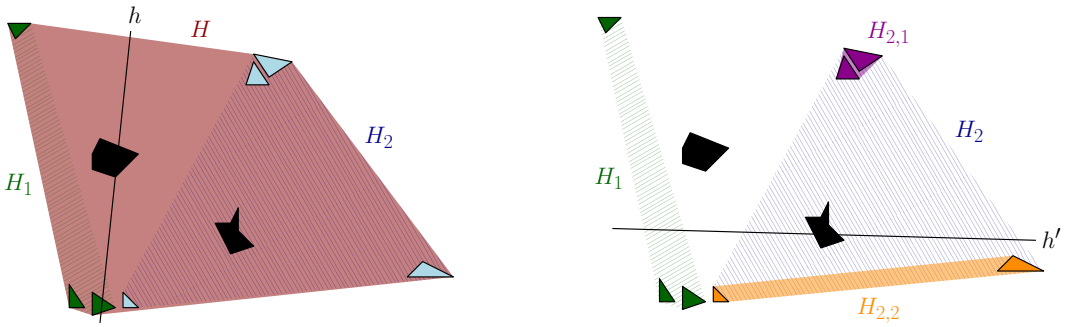


Figure 3 Fixing an invalid clique: All visible triangles form the initial piece H that we then split into pieces H_1 and H_2 using the half-plane h (left). As H_2 still contains a hole, we again split it into pieces $H_{2,1}$ and $H_{2,2}$ using half-plane h' (right). The result is the valid pieces H_1, $H_{2,1}$, and $H_{2,2}$.

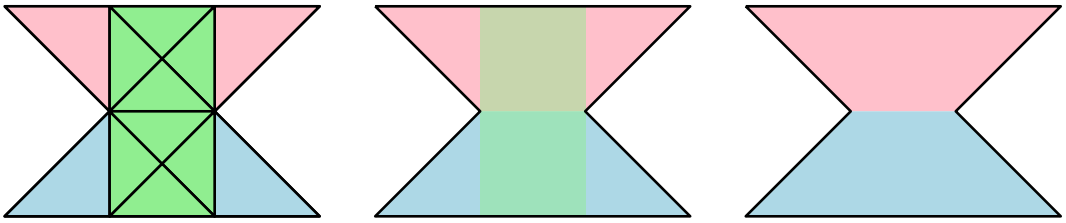


Figure 4 Extension partition with suboptimal clique cover (left), the corresponding convex cover (middle), and a convex cover without the unnecessary green polygon (right).

problems, and there exists no $n^{1-\varepsilon}$-approximation algorithm for any $\varepsilon > 0$ unless P = NP. However, there exist implementations that compute small solutions on practical instances. Namely, Chalupa [2] presented a randomized clique-growing approach that was subsequently used as a subroutine by Strash and Thompson [8] in their state-of-the-art solver `ReduVCC` that uses sophisticated reduction rules with a branch-and-reduce approach.

Fix Cover: Recall that a clique C corresponds to a set of triangles that are pairwise fully visible. We would like to use the convex hull H of the triangles as a piece in our convex cover, but H may not be contained in P; see Figure 2 for a simple example. While this rarely happens on the Challenge instances (see Section 3.3), we nonetheless have to post-process such pieces to obtain a feasible convex cover.

First, note that the only way a piece H can be invalid is by containing a hole of P; in particular, it is not possible that H intersects the unbounded region of the complement of P. We fix an invalid piece H as follows: Pick any hole h that invalidates H and consider an arbitrary half-plane whose boundary intersects h. Now partition the triangles of C according to whether they intersect the half-plane or not. This creates two new pieces, which both do not intersect the hole h and which partition the remaining holes in H. We apply this procedure recursively to the new pieces (always reducing the number of holes intersected by these pieces by at least one) until all newly created pieces are valid; see Figure 3. We omit the proof of correctness of this procedure due to space constraints.

Make Cover Minimal: At this point, we may end up with a non-minimal cover, i.e., there may exist redundant pieces; see Figure 4. To make the solution minimal, we iterate over the pieces and remove them from the cover if their removal does not invalidate it.

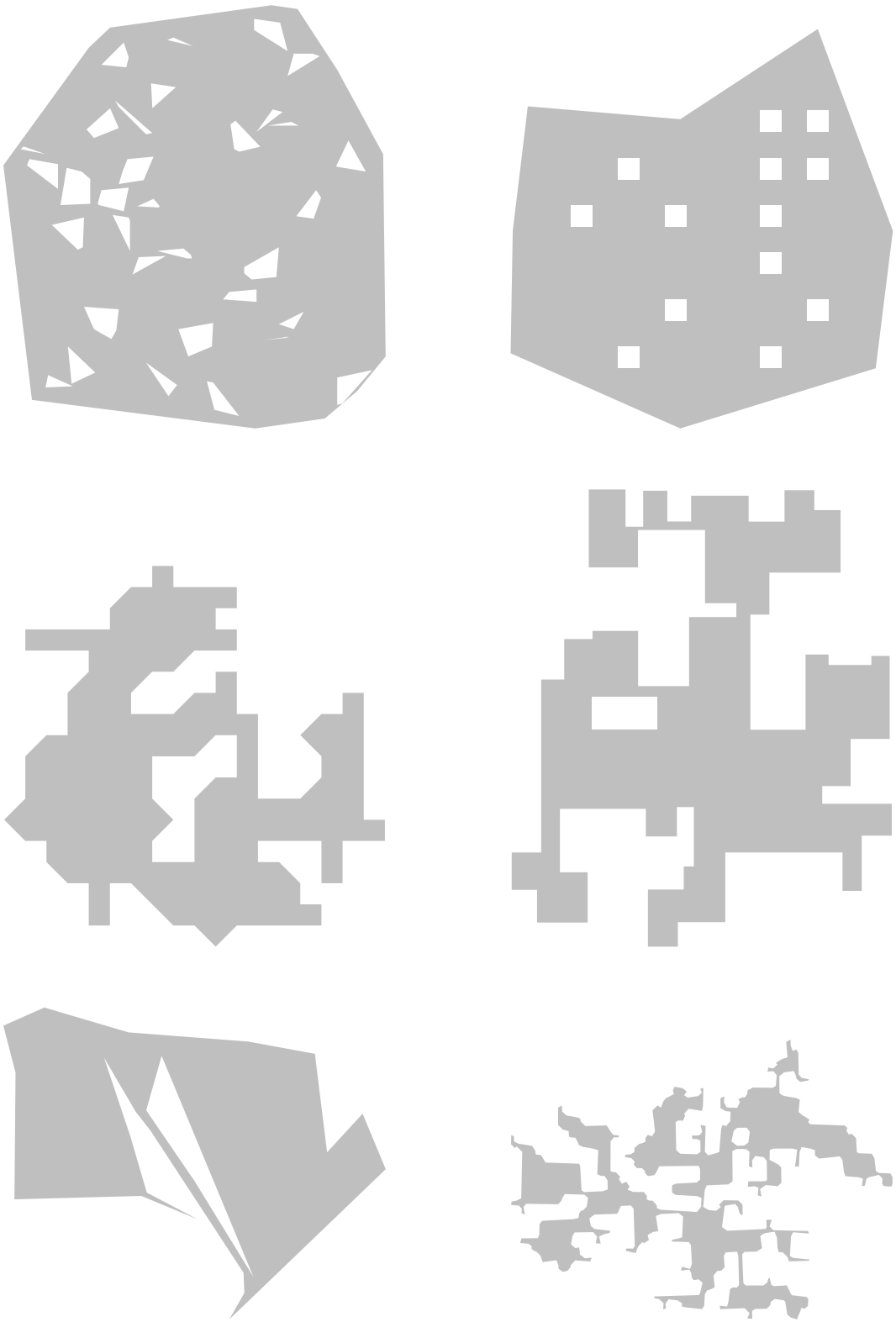


Figure 5 The smallest instances of some of the Challenge instance types. From left to right and top to bottom, these are: `cheese`, `maze`, `octa`, `iso`, `fpg`, `srpg_mc`.

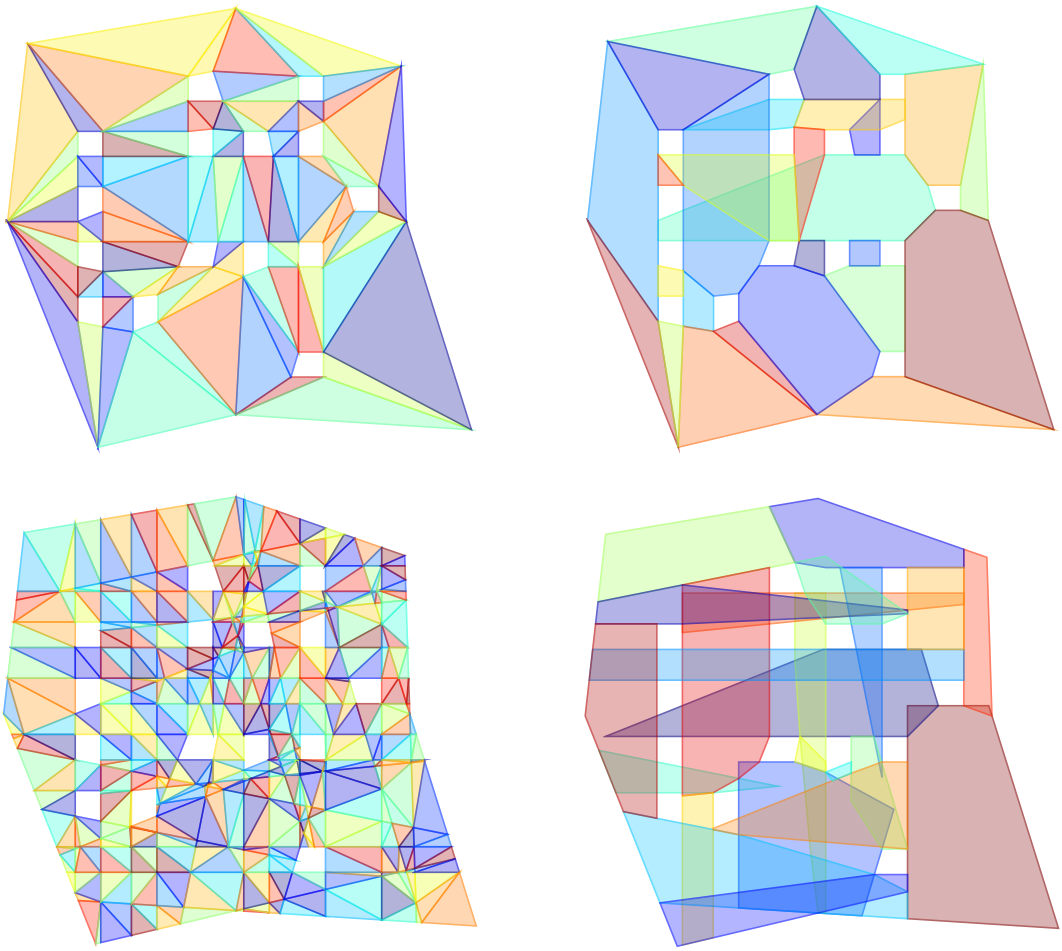


Figure 6 The Delaunay triangulation (top left) and the resulting cover of size 27 (top right). The extension partition (bottom left) and the resulting cover of size 23 (bottom right).

3 Evaluation

3.1 Implementation and Data

The competition code is written in C++ and compiled using GCC 11.3 with `-O3` optimization turned on. We use CGAL [9] for all geometric primitives with a Kernel that uses a number type that saves numbers as fractions and performs exact computations. For the partitioning and to construct the visibility graph, we use the triangulation, visibility, and convex hull packages of CGAL [6, 7, 10]. To compute the vertex cover, we use ReduVCC [8]. We show different types of instances of the problem set in Figure 5.

3.2 Examples

An important part of our approach is the choice of the partition. In particular, while the Delaunay triangulation is fast to compute, the extension partition creates partitions with significantly more pieces and thus slows down our approach. To justify this kind of partition, it must lead to significantly better solutions. Figure 6 shows a cover of the same instance using the Delaunay partition and the extension partition — one can clearly see that the extension

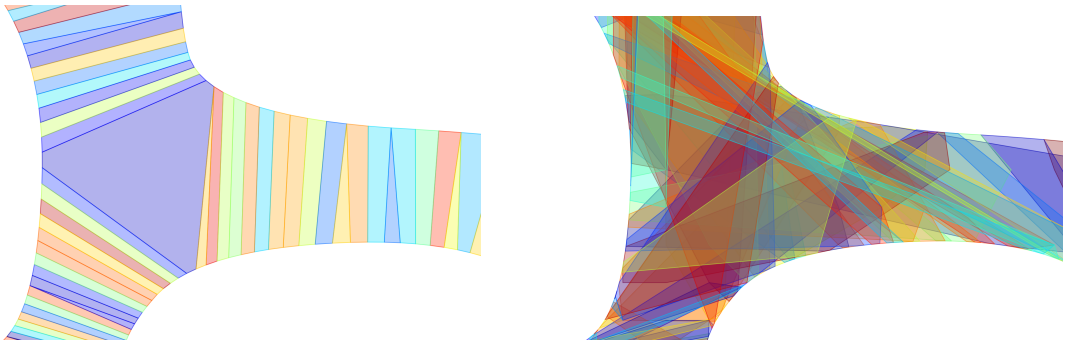


Figure 7 Part of a polygon with multiple long concave chains. While the cover using a Delaunay triangulation almost exclusively creates pieces adjacent to only two concave chains (left), local triangulation allows for creation of pieces that contain parts of all three concave chains (right).

partition better adapts to the geometry of the input polygon. Unfortunately, extension partitions can lead to a blow-up of the partition size that makes the approach practically infeasible for some instances. This blow-up happens when many extensions intersect. We circumvent this issue by computing restricted extension partitions in two different ways.

1. We want to preserve extensions locally. Thus, we only choose a subset of the extensions favoring short extensions. We either only insert extensions below a certain length, or we randomly sample extensions inversely proportional to their length.
2. Some Challenge instances have long concave chains on the boundary of P. Note that the midpoint of each edge of such a chain has to be part of a distinct piece in the convex cover. To allow for creation of pieces that combine segments from multiple concave chains, we locally triangulate long concave chains instead of creating extensions; see Figure 7.

3.3 Experiments

For this section, we selected a subset of the instances for more thorough experiments and subsequently only refer to these. See the sizes and types in the plots of Figure 8. Recall that we compute an intermediate, potentially infeasible solution via a vertex clique cover that is subsequently fixed. We argue above that we expect that only few cliques have to be fixed on practical instances. Indeed, on all except the `cheese` instances, the solution size increased by at most 6 pieces when fixing cliques, while most small instances did not have any invalid clique. However, the largest increase in solution size was for the largest `cheese` instances with an increase of 110 pieces. Our algorithm may create redundant cliques that are removed in a post-processing step, so it is interesting to consider how much this post-processing reduces the size of the solution. This decrease in pieces is very much dependent on the instance: While for `octa` the maximal decrease was 2 pieces, it was 83 pieces for `cheese` instances and all larger `cheese` instances saw significant improvements.

For the competition and our experiments we used a server with two Intel Xeon E5-2690 v4 CPUs with 14 cores (28 threads) each, and a total of 504GB RAM. All reported running times are single-threaded. The bottleneck of our approach is the visibility graph computation discussed in Section 2.2. To better understand the trade-off between running time, memory usage, and solution quality with respect to the choice of partition, we conduct experiments comparing Delaunay triangulation and extension partition; see Figure 8. The extension partition introduces a large overhead in running time and memory consumption compared to the Delaunay triangulation, but it reduces the solution size by a significant fraction. While

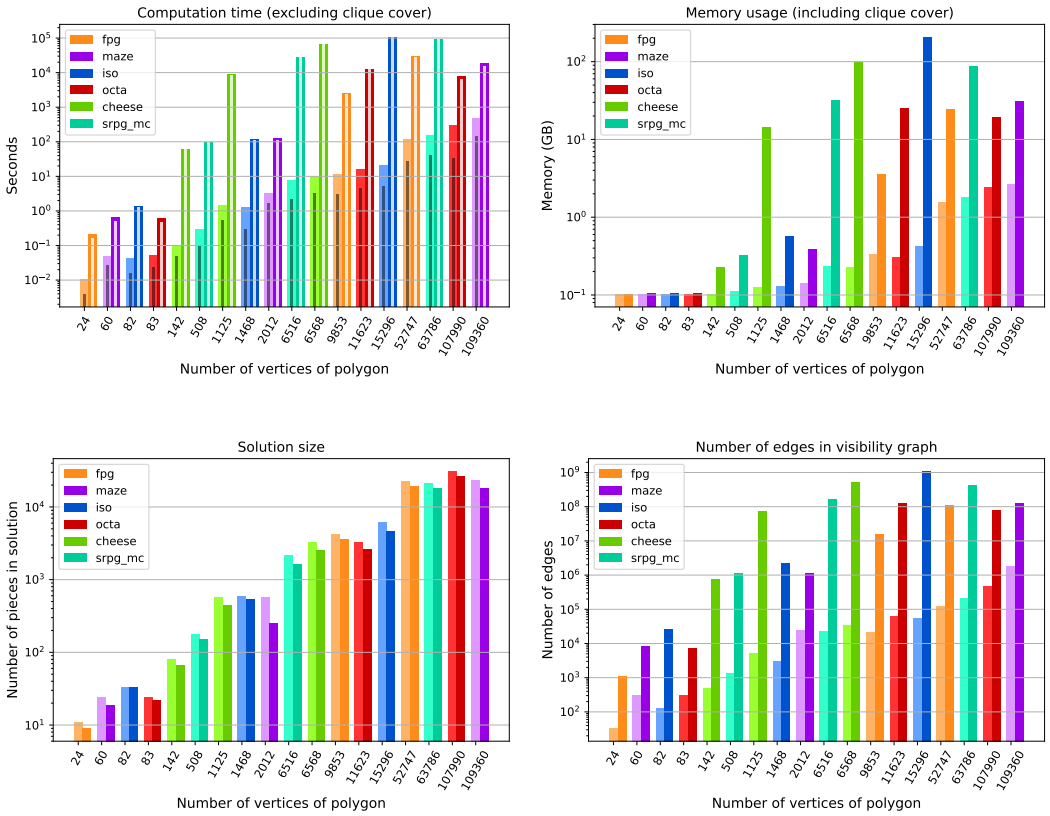


Figure 8 Experiments with the Delaunay triangulation (left bars) and the extension partition (right bars) as underlying partitions for a selected set of instances. We measure the running time (top left; thin bars showing the running time of the visibility graph computation), memory usage (top right), solution size (bottom left), and number of edges in the visibility graph (bottom right).

for the extension partition the visibility graph computation clearly dominates the running time, for the Delaunay triangulation it only makes up 32.8% of the running time on average.

References

1. Mikkel Abrahamsen. Covering polygons is even harder. In *Symposium on Foundations of Computer Science (FOCS)*, pages 375–386, 2021. doi:10.1109/FOCS52979.2021.00045.
2. David Chalupa. Construction of near-optimal vertex clique covering for real-world networks. *Comput. Informatics*, 34(6):1397–1417, 2015. URL: http://www.cai.sk/ojs/index.php/cai/article/view/1276.
3. Joseph C. Culberson and Robert A. Reckhow. Covering polygons is hard. *J. Algorithms*, 17(1):2–44, 1994. doi:10.1006/jagm.1994.1025.
4. Guilherme D. da Fonseca. Shadoks approach to convex covering. In *Symposium on Computational Geometry (SoCG)*, volume 258, page ?, 2023. URL: https://pageperso.lis-lab.fr/guilherme.fonseca/CGSHOP23conf.pdf.
5. Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. Minimum coverage by convex polygons: The CG:SHOP Challenge 2023, 2023. URL: https://arxiv.org/abs/2303.07007, arXiv:2303.07007.
6. Michael Hemmer, Kan Huang, Francisc Bungiu, and Ning Xu. 2D visibility computation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5.2 edition, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgVisibility2.
7. Susan Hert and Stefan Schirra. 2D convex hulls and extreme points. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5.2 edition, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgConvexHull2.
8. Darren Strash and Louise Thompson. Effective data reduction for the vertex clique cover problem. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 41–53, 2022. doi:10.1137/1.9781611977042.4.
9. The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5.2 edition, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html.
10. Mariette Yvinec. 2D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5.2 edition, 2023. URL: https://doc.cgal.org/5.5.2/Manual/packages.html#PkgTriangulation2.

7.2 Proof of $O(n^2)$ triangles in an extension triangulation

Since an extension segment can be defined by a pair of adjacent points on the outer boundary of P or on the boundary of a hole of P, the number of extension segments is $O(n)$. Still there might be $O(n^2)$ intersections between extension segments. The induced subdivision of P and the intersection points represents a plane graph, and for a vertex the number of incident faces equals the degree. Thus, the sum of number of incident faces of all points equals the sum of degrees of all points, which is two times the number of edges, which is linear in the number of vertices, since it is a planar graph. Viewing each face as a simple polygon, this implies that the sum of the number of vertices on the boundaries of all faces is linear in the number of vertices (or intersections). Since a Delaunay triangulation has $\Theta(k)$ triangles of a polygon with k vertices, the sum of sizes of Delaunay triangulations of all faces is $O(n^2)$. We therefore conclude that an **extension triangulation** has $O(n^2)$ triangles in the worst case.

7.3 Algorithm for making convex cover minimal

The convex cover corresponding to a clique partition without **invalid cliques**, does not guarantee a minimal convex cover, i.e. there might exist a convex polygon that is completely covered by other convex polygons. Therefore, we would like to turn a convex cover, produced by phases 1-3, into a small minimal convex cover. We discretize the convex polygons by finding a constrained Delaunay triangulation of the convex cover (the edges of the convex polygons are used as constraints). Thus any convex polygon will represent a subset of triangles. Then we repeatedly do the following (at most 5 times, or until convergence).

1. Let a convex polygon, X, be needed iff. there exist a triangle s.t. X is the only polygon containing the triangle. Add all needed convex polygons to the output convex cover.
2. For the remaining convex polygons. Sort in non-increasing order: first by the number of points on the boundary (representing its involvement with other convex polygons), and then by area (potential to cover other polygons).
3. Process the convex polygons in the order defined above. If all triangles in a convex polygon are already covered by polygons from the output convex cover, then discard it. Otherwise, add the convex polygon to the output convex cover.

7.4 All results from the Challenge

CG:SHOP 2023

Download | Submit Solution

Organized by: Sándor Fekete (TU Braunschweig),
Phillip Keldenich (TU Braunschweig),
Dominik Krupke (TU Braunschweig),
Stefan Schirra (University of Magdeburg)

22 teams participating
Oct. 28, 2022, midnight (UTC) - Jan. 27, 2023, 11:59 p.m. (AoE)

view all competition news

The official ranking for the 2023 challenge is now public! The two top teams have been invited to present their … 02/06/2023 12:54 p.m.

read more ...

We finished verification. There have been some smaller changes in the middle. The leading teams should receive a mail soon. … 02/01/2023 3:55 p.m.

read more ...

We are still verifying... this bug prevents 5 submissions of two teams from being verified, one of the teams being … 01/28/2023 2:57 p.m.

read more ...

The competition has ended. To view your score and the score of the best teams, please refer to the ranking tab.

Announcement | Rules | Instance Format | Ranking | Your Teams | Standings

	Team	*Best Score*	DIKU (AMW)
	Objective	nr_polys	nr_polys
	Overall Rank		1
	Junior Rank		
ccheese11045		4163	4306
ccheese1125		430	449
ccheese13171		4976	5166
ccheese1325		503	534
ccheese142		65	65
ccheese15357		5815	6056
ccheese17543		6605	6847
ccheese1762		662	710
ccheese194		81	81
ccheese21892		8347	8535
ccheese2225		841	881
ccheese234		100	100

ccheese26314	10100	10301
ccheese2643	993	1029
ccheese282	115	117
ccheese30746	12010	12010
ccheese3082	1182	1250
ccheese35084	13622	13622
ccheese372	148	148
ccheese43760	17078	17078
ccheese4390	1634	1720
ccheese459	175	178
ccheese528	201	209
ccheese5289	1968	2091
ccheese65632	25610	25610
ccheese6568	2464	2584
ccheese686	265	269
ccheese7893	2993	3147
ccheese877	329	348
ccheese87712	34264	34264
ccheese8817	3316	3480
cheese100003	37131	37131
cheese1006	365	387
cheese12554	4572	4694
cheese1262	457	483
cheese14941	5353	5557
cheese1516	542	570
cheese163	68	69
cheese17474	6324	6560
cheese19947	7296	7431
cheese2058	742	762

cheese225	85	85
cheese25028	9177	9356
cheese2527	916	953
cheese271	103	103
cheese29976	11180	11180
cheese3015	1064	1106
cheese302	108	113
cheese3489	1241	1307
cheese34962	13092	13092
cheese39834	14823	14823
cheese399	149	153
cheese4951	1736	1838
cheese50158	18691	18691
cheese528	190	193
cheese6044	2142	2281
cheese617	229	239
cheese74951	27892	27892
cheese7514	2674	2817
cheese773	290	298
cheese9051	3210	3409
cheese9957	3548	3763
fpg-poly_0000000020_h1	9	9
fpg-poly_0000000020_h2	12	12
fpg-poly_0000000040_h1	18	18
fpg-poly_0000000070_h2	31	31
fpg-poly_0000000400_h2	172	173
fpg-poly_0000000800_h7	454	454
fpg-poly_0000001300_h7	563	564
fpg-poly_0000004900_h2	1818	1822

fpg-poly_0000008100_h6	3626	3626
fpg-poly_0000009600_h8	4901	4901
fpg-poly_0000050000_h9	19083	19083
maze_10174_500_05_001	833	833
maze_104559_1200_025_001	5687	5687
maze_109_50_025_00	17	17
maze_115769_1200_025_01	17121	17121
maze_124_50_01_001	17	17
maze_124_50_01_005	26	26
maze_124_50_025_01	29	29
maze_126979_1200_01_00	2738	2738
maze_129304_1200_01_005	11320	11320
maze_129819_1200_001_005	9524	9524
maze_134_50_01_01	25	25
maze_136699_1200_01_01	18325	18325
maze_15429_500_025_01	2338	2363
maze_154_50_001_001	18	18
maze_16019_500_025_005	1671	1686
maze_164_50_001_01	30	30
maze_16549_500_01_01	2358	2376
maze_18474_500_01_001	755	763
maze_184_50_001_005	21	21
maze_19609_500_001_001	548	548
maze_21184_500_001_005	1656	1670
maze_21919_500_01_005	1964	1981
maze_24234_500_001_00	282	282
maze_2514_250_05_001	247	247
maze_2534_250_05_01	433	453
maze_2604_250_05_00	233	233

maze_29364_750_05_005	3631	3631
maze_3329_250_025_001	259	262
maze_3564_250_025_005	420	428
maze_3634_250_025_01	599	607
maze_39479_750_01_005	3524	3557
maze_3969_250_01_01	589	602
maze_39929_750_025_001	2275	2275
maze_4169_250_025_00	240	240
maze_42329_1000_05_01	6942	6942
maze_4344_250_001_01	593	605
maze_43729_750_001_00	324	324
maze_45284_1000_05_00	2809	2809
maze_45859_750_001_01	5733	5793
maze_45884_750_01_00	1117	1117
maze_47214_1000_05_005	5739	5739
maze_4759_250_01_001	237	239
maze_48674_750_001_001	1216	1216
maze_5084_250_001_005	408	416
maze_5244_250_001_001	210	210
maze_57334_1200_05_01	9372	9372
maze_64279_1000_025_001	3491	3491
maze_64679_1000_025_005	6663	6663
maze_69374_1000_025_00	2770	2770
maze_74_50_05_00	18	18
maze_77384_1000_01_00	1736	1736
maze_77579_1000_001_01	9573	9698
maze_79369_1000_01_005	7028	7033
maze_79619_1000_001_005	5814	5841
maze_79_50_05_005	20	20

maze_8299_500_05_005	1077	1079
maze_83604_1000_01_001	3146	3146
maze_84994_1000_001_001	1878	1878
maze_84_50_05_01	23	23
maze_97289_1200_01_001	3568	3568
socg60	10	10
socg92	15	15
socg_fixed60	9	9
socg_fixed92	14	14
srpg_iso_aligned_mc0000088	23	23
srpg_iso_aligned_mc0000286	84	84
srpg_iso_aligned_mc0000451	129	129
srpg_iso_aligned_mc0000609	147	147
srpg_iso_aligned_mc0000624	178	178
srpg_iso_aligned_mc0001258	333	333
srpg_iso_aligned_mc0001336	398	399
srpg_iso_aligned_mc0003255	951	954
srpg_iso_aligned_mc0003883	960	967
srpg_iso_aligned_mc0006138	1799	1806
srpg_iso_aligned_mc0010085	2947	2948
srpg_iso_aligned_mc0015285	3809	3951
srpg_iso_aligned_mc0067791	18067	18092
srpg_iso_aligned_mc0094745	24921	25633
srpg_iso_mc0000080	33	33
srpg_iso_mc0000261	97	97
srpg_iso_mc0000266	100	100
srpg_iso_mc0000946	301	302
srpg_iso_mc0001463	543	545
srpg_iso_mc0001964	594	604

srpg_iso_mc0002462	934	934
srpg_iso_mc0015281	4706	4717
srpg_mc0000506	152	152
srpg_mc0001112	316	316
srpg_mc0001937	655	658
srpg_mc0004240	1216	1216
srpg_mc0006513	1653	1653
srpg_mc0009594	2460	2477
srpg_mc0012065	4001	4001
srpg_mc0019977	5713	5713
srpg_mc0063684	18101	18101
srpg_mc0067337	16385	16667
srpg_mc0068359	18174	18353
srpg_mc0080065	21483	21483
srpg_mc0095504	25742	25787
srpg_mc0104714	28275	28275
srpg_octa_mc0000082	22	22
srpg_octa_mc0000297	79	79
srpg_octa_mc0000322	93	94
srpg_octa_mc0000784	145	148
srpg_octa_mc0001206	295	297
srpg_octa_mc0001647	469	469
srpg_octa_mc0001693	498	498
srpg_octa_mc0002605	553	556
srpg_octa_mc0007035	1856	1856
srpg_octa_mc0011620	2668	2668
srpg_octa_mc0024691	5368	5368
srpg_octa_mc0030848	7660	7660
srpg_octa_mc0083376	19658	19658

srpg_octa_mc0097331	21189	21189
srpg_octa_mc0107884	26714	26714
srpg_smo_mc0005962	1174	1179
srpg_smo_mc0022826	3326	3326
srpg_smo_mc0023438	4966	4966
srpg_smo_mc0027976	5535	5535
srpg_smo_mc0040970	5878	5878
srpg_smo_mc0048732	9827	9827
srpg_smo_mc0062031	9115	9115
srpg_smo_mc0064487	9442	9442
srpg_smo_mc0089743	20261	20261
srpg_smo_mc0094408	15167	15167
srpg_smr_mc0006333	1312	1312
srpg_smr_mc0014427	2793	2793
srpg_smr_mc0015995	2677	2677
srpg_smr_mc0023805	4771	4771
srpg_smr_mc0041883	8607	8607
srpg_smr_mc0070381	9236	9236
srpg_smr_mc0075643	11043	11043
srpg_smr_mc0097788	17401	17752