Institut für Betriebssysteme und Rechnerverbund
Technische Universität Braunschweig

Masterarbeit

Computational Aspects of MaxMin Triangulations

Winfried Hellmann

2013-08-01 (DRAFT)

Betreuer: Prof. Dr. Sándor Fekete

for my daughter—in the hope that she will understand sometime

Erklärung

Abstract

Maxmin length triangulations are just awesome.

Zusammenfassung

Maxmin Triangulationen sind einfach super.

Contents

1. Introduction

2. Integer Programming

This chapter focuses on some preliminaries around integer programming—which is a subarea of (mathematical) optimization. In that context we introduce three common optimization problems which can be solved using integer programming: SAT, Vertex Cover, and Independent Set. They are discussed later within this thesis (chapters 3 and 4) in association with the MaxMin Length Triangulation (MMLT) problem.

An integer program is basically a problem description as a system of (in-)equations involving only variables with integer values. As an addition there can be an objective function for modeling optimization problems. Throughout this thesis, we assume that every integer program has only linear (in-)equations which is referred to as a *linear integer program.*

Definition 2.1 ((Linear) Integer Program (IP))
An IP is a system of integer variables $x \in \mathbb{Z}^n (n \in \mathbb{N})$ with a set of constraints on them and optionally an objective function. We consider only the case where the constraints are linear with respect to x.

A natural standardized way of formulating (linear) IPs is the so called *canonical form.* It combines the coefficients of the variable vector $x \in \mathbb{Z}^n (n \in \mathbb{N})$ for all $m \in \mathbb{N}$ (in-)equations in a matrix $A \in \mathbb{Z}^{m \times n}$, all constant terms are summed up to a vector $b \in \mathbb{Z}^m$, and the objective function is represented as a multiplication of the variables x and a constant target vector $c \in \mathbb{Z}^n$. For readability, we slightly modify the canonical form, e.g. by allowing sums.

Problem 2.2 (IP in canonical form [39])

$$\begin{array}{lll} \text{maximize} & c^T x & \\ \text{subject to} & Ax \leq b & \\ & x \geq 0 & \\ & x \in \mathbb{Z}^n & \end{array} \quad \text{or} \quad \begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & x \leq 0 \\ & x \in \mathbb{Z}^n \end{array}$$

($\leq$ and $\geq$ here denote the row-wise comparison of two vectors)

Even though the structure of IPs looks very simple, it is a long known fact that solving them is not easy. Nevertheless they come in useful for various optimization problems—especially because there are many practical applications involving only integers. A common strategy to simplify the problem is leaving out the integrity restriction for retrieving bounds of the optimal solution.

Theorem 2.3
Solving IPs is NP-hard.

Proof:
Even the special case where there is no objective function, only binary variables, and only equality constraints is NP-complete [30]. Therefore the more general problem is NP-hard.

2.1. SAT

The (boolean) satisfiability problem (SAT) asks for a variable assignment which fulfills a logical term. It was the first problem to be proven NP-complete [11] and has since been a common choice for NP-hardness proofs—many times even more restricted such as the 3-SAT problem which allows only for three variables in each "sub-term" (clause).

Problem 2.4 (3-SAT)

Given: Set of boolean literals X, a formula in conjunctive normal form consisting of clauses C each involving exactly three literals (or their negations)

Sought: An assignment for X which lets all clauses in C evaluate to "true"

Shortly after the concept of NP-completeness was introduced in [11], Richard Karp applied the idea to several other well-known problems [30]. One of them was the 3-SAT problem which has more structure than the general boolean satisfiability problem and is therefore easier to embed into constructive NP-hardness or NP-completeness proofs.

Theorem 2.5 (NP-completeness of 3-SAT)
Problem 2.4 is NP-complete. [30, Satisfiability with at most 3 Literals per Clause]

As already mentioned earlier, the 3-SAT problem can be formulated as an IP. This easily confirms the NP-hardness of solving IPs. Additionally it allows for applying IP solvers (such as CPLEX, see also section 5.2.4) to 3-SAT problems. Usually however, SAT solvers are used which take advantage of the problem structure better and have therefore less overhead.

Problem 2.6 (IP formulation of 3-SAT)

$$\begin{aligned}
&(\text{minimize} && 0^T x) \\
&\text{subject to} && \forall c \in C : \sum_{x_i \in c} x_i + \sum_{\neg x_i \in c} (1 - x_i) = 1 \\
& && \forall x_i \in X : \qquad\qquad\qquad x_i \in \{0, 1\}
\end{aligned}$$

($x_i \in c$ (or $\neg x_i \in c$) herein denotes that x_i (or $\neg x_i$) is part of the clause c)

For geometric and graph problems an even more restricted class of 3-SAT problem has been established, the *Planar 3-SAT*. It asks for the connection graph of all clauses and their variables to be embeddable in the plane.

Problem 2.7 (Planar 3-SAT)
An instance of the 3-SAT problem with literals X and clauses C which can be represented by a planar graph $G = (V, E)$ such that

$$\begin{aligned}
V &= \{v : v \in X \cup C\} \\
E &= \{\{x, c\} : x \in X,\ c \in C,\ (x \in c) \vee (\neg x \in c)\}
\end{aligned}$$

Figure 2.1 shows an example of a 3-SAT instance with a planar variable-clause-connection graph. There is a vertex for every clause (mint colored squares) and every variable (red circles) and edges in between if a variable (or its negation) is part of a clause.

It was shown that even this restricted class of satisfiability problems is NP-complete. That way problems can be proven to be NP-hard in the plane without having to deal with intersections. This result can then be used for higher dimensions (usually, problems become harder to solve if they are elevated to higher dimensions).

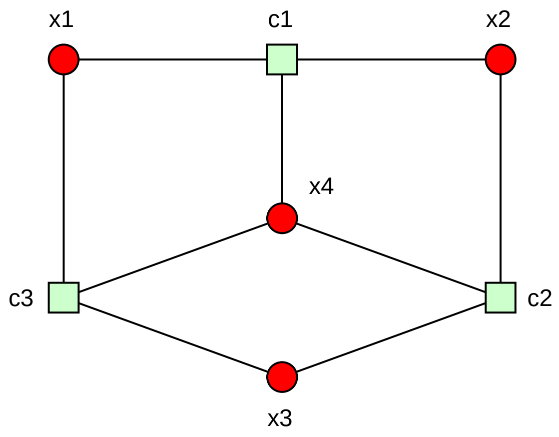


Figure 2.1.: Example of a Planar 3SAT instance which represents the term $\underbrace{(x_1 \vee x_2 \vee x_4)}_{c_1} \wedge \underbrace{(\neg x_2 \vee x_3 \vee \neg x_4)}_{c_2} \wedge \underbrace{(x_1 \vee \neg x_3 \vee \neg x_4)}_{c_3}$

Theorem 2.8 (NP-completeness of Planar 3-SAT)
Problem 2.7 is still NP-complete. [32]

2.2. Vertex Cover

Another problem in the collection of [30] is *Vertex Cover* which aims to cover (all) edges of a graph by their incident vertices. It is closely related to other graph problems such as Edge Cover, Clique, Independent Set (see section 2.3), Coloring, and the more general Set Cover.

Definition 2.9 (Vertex Cover)
Given an undirected graph $G = (V, E)$, a Vertex Cover $V_{\text{cover}} \subseteq V$ for G is a vertex set such that every edge $e \in E$ is incident to at least one vertex $v \in V_{\text{cover}}$:

$$\forall e \in E : \exists v \in V_{\text{cover}} : v \in e$$

A trivial Vertex Cover of any graph $G = (V, E)$ would be the complete vertex set V. In general, such a Vertex Cover is not useful but one asks for the covering vertex set to be minimal (roughly speaking without unnecessary vertices).

Definition 2.10 (Minimal Vertex Cover)
A Vertex Cover V_{cover} for an undirected graph $G = (V, E)$ is minimal if there is no vertex $v \in V_{\text{cover}}$ such that $V_{\text{cover}} \setminus \{v\}$ remains a Vertex Cover.

Still, *any* Minimal Vertex Cover is not always wanted as it needs not be unique—hence there can be a much smaller covering vertex set. An extreme example is a star-shaped graph where one vertex can cover all edges but the set of all other vertices would also be minimal. Therefore we distinguish between a Minimal Vertex Cover and a *Minimum* Vertex Cover, i.e. a smallest possible one. Figure 2.2 shows the difference between Minimal and Minimum Vertex Cover by example.

Definition 2.11 (Minimum Vertex Cover)
A Vertex Cover V_{cover} for an undirected graph $G = (V, E)$ is minimum if there is no other Vertex Cover V'_{cover} for G which has fewer vertices:

$$\forall\ V'_{\text{cover}} \text{ Vertex Cover} : |V_{\text{cover}}| \leq |V'_{\text{cover}}|$$

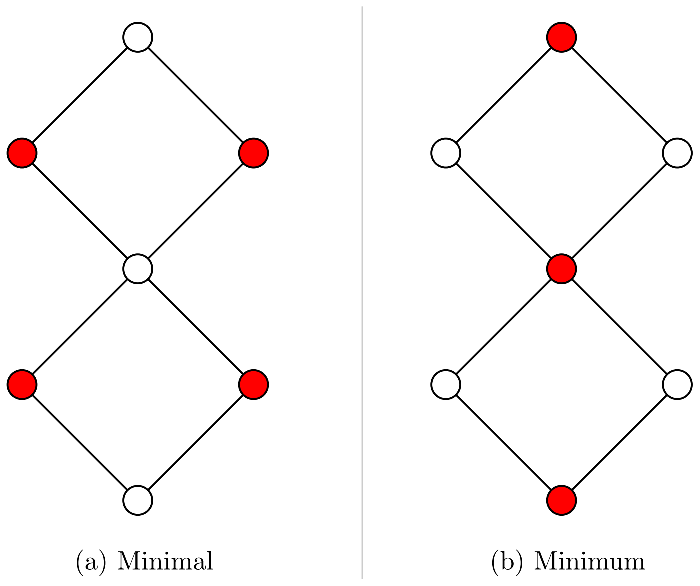

(a) Minimal (b) Minimum

Figure 2.2.: Example of Minimal vs. Minimum Vertex Cover, covering vertices in red

The IP formulation of Minimum Vertex Cover can be directly deduced from the definition: We minimize the number of covering vertices such that for every edge at least one vertex is selected. The optimal vertex set V_{cover} can then be retrieved from the IP solution vector by just collecting all selected vertices: $V_{\text{cover}} = \{v \in V : x_v = 1\}$

Problem 2.12 (IP Formulation of Minimum Vertex Cover)

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} x_v \\ \text{subject to } \forall \{v,w\} \in E : \quad & x_v + x_w \geq 1 \\ \forall v \in V : \quad & x_v \in \{0,1\} \end{aligned}$$

What follows is that the Minimum Vertex Cover is also NP-complete because it is a subset of the Set Cover problem [30] and clearly lies in NP as it can be (polynomially) reduced to solving IPs.

Theorem 2.13 (NP-completeness of Minimum Vertex Cover)
Minimum Vertex Cover is NP-complete. [30]

We now introduce the class of *well-covered* graphs which we will need in chapter 3. They combine all graphs in which there is no difference between Minimal and Minimum Vertex Cover. Minimal Vertex Covers can be calculated with significant less effort (i.e. in polynomial time) than Minimum Vertex Covers (e.g. by just iteratively removing vertices from the set until it would destroy the covering property). This leads to Minimum Vertex Cover being solvable in polynomial time for *well-covered* graphs.

A subset of *well-covered* graphs are connected bipartite graphs $G = (A \cup B, E \subseteq A \times B)$ with both vertex sets having the same size: $|A| = |B|$. Either one of A and B is a Minimal and a Minimum Vertex Cover for G. Clearly this graph class is not empty which already implies that the class of *well-covered* graph is not empty either.

Definition 2.14 (*Well-covered* Graph)
An undirected graph $G = (V, E)$ is *well-covered* iff every Minimal Vertex Cover for G is also a Minimum Vertex Cover for G. [40]

As a logical consequence of definition 2.14, every two vertex sets which are a Minimal Vertex Cover for a *well-covered* graph G have the same size. If one of them had more vertices than the other one, it could not have been a Minimum Vertex Cover for G—which contradicts G being *well-covered*.

Theorem 2.15
In a *well-covered* graph, all Minimal Vertex Covers have the same cardinality. [40]

2.3. Independent Set

Another prominent graph problem is the *Independent Set* which asks for a vertex set that is "independent", meaning that no two vertices are adjacent. It can be seen as the counter part of Vertex Cover (see section 2.2) because it requires *at most* one vertex for every edge while Vertex Cover demands *at least* one vertex for every edge. We consider this connection in theorem 2.19.

Definition 2.16 (Independent Set)
Given an undirected graph $G = (V, E)$, an independent set $V_{\text{IS}} \subseteq V$ is a vertex set such that no two vertices $v, w \in V_{\text{IS}}$ are incident to the same edge $\{v, w\} \in E$:

$$\forall v \in V_{\text{IS}} : \forall \{v, w\} \in E : w \notin V_{\text{IS}}$$

Again we can find a trivial Independent Set for every graph in constant time: the empty set. However, similar to Vertex Cover one usually asks for a "large" Independent Set as defined below:

Definition 2.17 (Maximal/Maximum Independent Set)
For an undirected graph $G = (V, E)$, an independent set $V_{\text{IS}} \subseteq V$ is maximal if there is no vertex $v \in V \setminus V_{\text{IS}}$ such that $V_{\text{IS}} \cup \{v\}$ remains an independent set. It is maximum if there is no independent set ${V_{\text{IS}}}'$ with larger cardinality.

The IP for Maximum Independent Set maximizes the number of selected vertices such that for every edge only one of the incident vertices is selected. Analogous to the IP formulation of Minimum Vertex Cover, the optimal V_{IS} solution can directly be retrieved from the IP solution: $IP = \{v \in V : x_v = 1\}$

Problem 2.18 (IP Formulation of Maximum Independent Set)

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} x_v \\ \text{subject to } \forall\{v,w\} \in E: \; & x_v + x_w \leq 1 \\ \forall v \in V: \; & x_v \in \{0,1\} \end{aligned}$$

We already mentioned the close connection between Vertex Cover and Independent Set earlier. Note that in fig. 2.2 the Minimal Vertex Cover is a Maximum Independent Set and the Minimum Vertex Cover a Maximal Independent Set. From the IP formulations it can also been seen that the problems behave the opposite way:

Theorem 2.19 (Independent Set and Vertex Cover)
For an undirected graph $G = (V, E)$, $V_{\text{IS}} \subseteq V$ is a Maximum Independent Set if and only if $V_{\text{cover}} = V \setminus V_{\text{IS}}$ is a Minimum Vertex Cover.

Proof:
Let V_{cover} be a Vertex Cover for G.

$$\begin{aligned} \forall e \in E : \exists v \in V_{\text{cover}} : v \in e &\iff \forall\{v,w\} \in E : v \in V_{\text{cover}} \lor w \in V_{\text{cover}} \\ &\iff \forall\{v,w\} \in E : \neg(v \notin V_{\text{cover}} \land w \notin V_{\text{cover}}) \\ &\iff \forall\{v,w\} \in E : \neg(v \in (V \setminus V_{\text{cover}}) \land w \in (V \setminus V_{\text{cover}})) \\ &\iff \forall v \in (V \setminus V_{\text{cover}}) : \forall\{v,w\} \in E : w \notin (V \setminus V_{\text{cover}}) \\ &\iff (V \setminus V_{\text{cover}}) \text{ independent set} \end{aligned}$$

Assume V_{cover} is a Minimum Vertex Cover for G and the independent set $V_{\text{IS}} = V \setminus V_{\text{cover}}$ is not maximum. Then there is an independent $V'_{\text{IS}} \subseteq V$ with $|V_{\text{IS}}| < |V'_{\text{IS}}|$. But then for the Vertex Cover $V'_{\text{cover}} = V \setminus V'_{\text{IS}}$ the following holds: $|V'_{\text{cover}}| < |V_{\text{cover}}|$—which is a contradiction to V_{cover} being minimum. The same argumentation applies in the other direction.

Now that the link between both problems is proven, we can apply properties of the Vertex Cover problem to Independent Sets. Firstly, it follows that the Maximum Independent Set problem is NP-complete as well because every solution can be transformed into a Minimum Vertex Cover and vice versa in polynomial time.

Theorem 2.20 (NP-Completeness of Maximum Independent Set)
Theorems 2.13 and 2.19 imply that Maximum Independent Set is NP-complete.

Additionally, we can adapt the concept of *well-covered* graphs to Independent Sets. The following theorem implies that for *well-covered* graphs, the complexity of finding a Maximum Independent Set is reduced to polynomial time.

Theorem 2.21 (Independent Set in *Well-covered* Graphs)
For a *well-covered* graph $G = (V, E)$, every maximal independent set has the same size and is therefore maximum.

Proof:
Theorem 2.21 follows directly from definition 2.14 and theorems 2.15 and 2.19.

Now we show a straightforward approach for finding Maximal Independent Sets in any undirected graph. It is similar to a method for Minimum Vertex Cover mentioned earlier. The idea consists in iteratively adding vertices to the set as long as they do not violate the Independent Set property until all vertices have been considered. Such an approach is called "greedy" because it constantly extends the solution until this is no longer possible—without removing part of the solution at any time.

Algorithm 2.1 : Greedy Algorithm for Independent Set

```
Input : Undirected graph $G = (V, E)$
Output : Maximal Independent Set $V_{\text{IS}} \subseteq V$ for $G$
1 Set $V_{\text{IS}} = \emptyset$
2 foreach $v \in V$ do
3     if $\forall \{v, w\} \in E : (w \notin V_{\text{IS}})$ then
4         Set $V_{\text{IS}} = V_{\text{IS}} \cup \{v\}$
5 return $V_{\text{IS}}$
```

This simple algorithm performs surprisingly well—to be seen in the following theorem. The Independent Set property can not be ensured without taking all edges into account which implies that the running time of algorithm 2.1 is even asymptotically optimal.

Theorem 2.22 (Correctness and Complexity of Algorithm 2.1)
Algorithm 2.1 always finds a Maximal Independent Set in $O(|E|)$ time.

Proof:
Because the vertices are processed sequentially, for every edge $\{v, w\} \in E$ at most one of v and w is added to V_{IS}. Therefore V_{IS} is an independent set. Additionally every vertex $v \in V$ is processed and if there is no $w \in V_{\text{IS}}$ with $\{v, w\} \in E$ then $v \in V_{\text{IS}}$. So V_{IS} is maximal.

The for-loop runs $|V|$ times but the if-statement is only executed twice for every edge $e \in E$. Every other statement runs in $O(1)$ time. Thus algorithm 2.1 needs $O(|E|)$ time.

Finally, we can take advantage of *well-covered* graphs when running algorithm 2.1 on them. It implicitly returns a Maximum Independent Set for *well-covered* graphs—without even comparing it to other Maximal Independent Sets.

Theorem 2.23 (Algorithm 2.1 in *well-covered* Graphs)
For a *well-covered* graph algorithm 2.1 always finds a Maximum Independent Set in $O(|E|)$ time.

Proof:
Theorem 2.23 follows directly from theorems 2.21 and 2.22.

3. Triangulations

In the following chapter, we introduce triangulations along with some of their different variants and mention related work that has been done in this field. Many triangulations are usually seen as a part of computational geometry—including the most popular one, the Delaunay Triangulation [2, Section 9.2]. However, the underlying structure that they share can also be interpreted as a combinatorial problem separated from geometric aspects. Thereby, we encounter some of the basics from chapter 2 again.

To begin with, we borrow a definition from graph theory: the *Complete Graph.* It is a simple (undirected) graph with the maximum number of edges. Often it is denoted as K_n where n is the number of vertices. We modify this definition slightly to let the graph be induced by a given vertex set:

Definition 3.1 (Complete Graph)
Given a vertex set V, the *Complete Graph* $K_V = (V, E)$ for V contains all possibles undirected edges between each pair of vertices in V:

$$E = \{e = \{v, w\} : v, w \in V \wedge v \neq w\}$$

Next, we develop the term of *conflicts.* This concept tries to abstract geometric properties of mutually exclusive objects (such as intersecting line segments). That way we can represent (some) geometric restrictions in combinatorial problems.

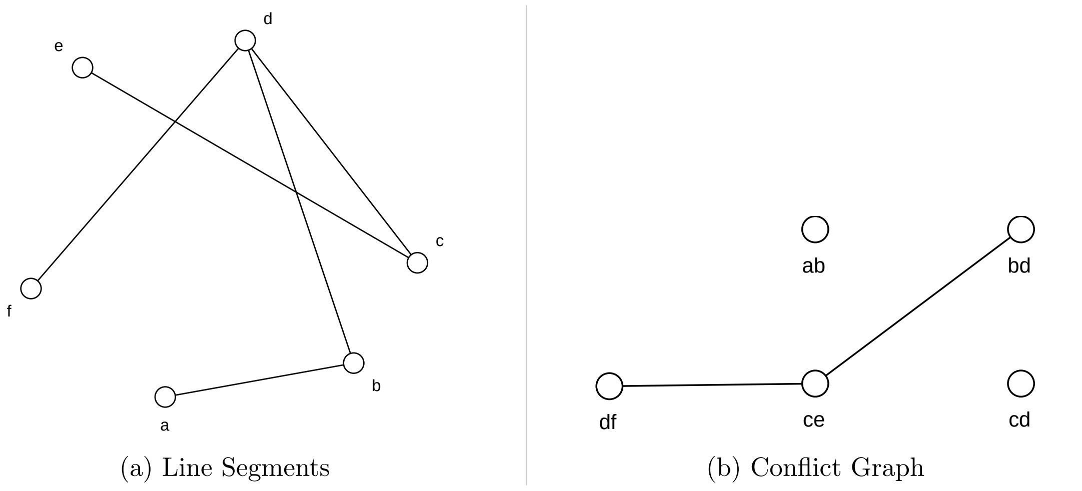
Definition 3.2 (Conflicts)
For a set of objects O, *conflicts* X are a set of unordered object pairs:

$$X \subseteq \{\{o_i, o_j\} : o_i, o_j \in O \wedge o_i \neq o_j\}$$

As we defined it, conflicts of objects are indistinguishable from edges of a simple graph having the objects as vertices. Below, we call such a graph the *Conflict Graph*:

Definition 3.3 (Conflict Graph)
The *Conflict Graph* $G_{\text{conf}}(O, X) = (O, X)$ for a set of objects O and a set of conflicts X is an undirected graph with O as vertices and X as edges.

Refer to fig. 3.1 for an example of the Conflict Graph with line segments being the objects and their conflicts representing all pairwise intersections.



(a) Line Segments (b) Conflict Graph

Figure 3.1.: Example of the Conflict Graph for a given set of line segments and the conflicts being all intersections

Now we can combine the previous definitions to characterize triangulations. Note that we start off with a combinatorial definition without any geometric components. In the course of this chapter, we show how to apply this formulation to geometric problems.

Definition 3.4 (Triangulation)
Given the Complete Graph $K_V = (V, E)$ for a vertex set V and a set of conflicts X for E such that the Conflict Graph $G_{\text{conf}}(E, X)$ is *well-covered*. A *Triangulation* $T(V, X) \subseteq E$ of V with respect to X is a maximum set of non-conflicting edges:

$$e_i \in T(V, X) \iff e_i \in E \land \forall e_j \in T(V, X) : \{e_i, e_j\} \notin X$$

Bringing the Maximum Independent Set from section 2.3 back to mind, we can see that a Triangulation is essentially the same problem. Apart from Triangulations operating on the Conflict Graph, both problems maximize the number of “independent” vertices.

Theorem 3.5 (Equality of Triangulation and Maximum Independent Set)
Every Triangulation $T(V,X)$ of a vertex set V with respect to conflicts X is a Maximum Independent Set for the Conflict Graph $G_{\text{conf}}(E,X)$ and vice versa. Herein E are the edges of the Complete Graph K_V.

Proof:
Theorem 3.5 follows directly from definition 2.17 and theorem 2.21.

Now we can make use of algorithm 2.1 from chapter 2 to gain a first bound on the time complexity of finding *any* Triangulation (for a given vertex set and conflicts). Later, we give more accurate time bounds for special classes of Triangulations.

Theorem 3.6 (Time Complexity of Triangulations)
From theorems 2.23 and 3.5 follows that finding a Triangulation $T(V,X)$ of a vertex set V with respect to conflicts X takes $O(|X|)$ time.

One important property of geometric objects has not been considered for our definition of Triangulations yet: *forbidden objects*. Besides conflicting objects from which only one can be part of the same Triangulation, there may as well be objects which are not wanted at all—as we consider the Complete Graph. This can be the case when a Triangulation is required to lie completely within a certain boundary or when one object overlaps multiple others.

Definition 3.7 (Triangulation with Forbidden Edges)
A *Triangulation with Forbidden Edges* $T(V,X,F)$ is a Triangulation of the vertex set V with respect to conflicts X which does not contain any of the edges in F:

$$\forall e \in F : e \notin T(V,X,F)$$

Even though definition 3.7 models geometric triangulation problems better, it is not solvable in polynomial time anymore (assuming P≠NP). Many geometric triangulations are solvable in polynomial time however (as we see later)—so this approach is clearly not the best one to solve them.

Theorem 3.8 (NP-completeness of Triangulation with Forbidden Edges)
The decision problem whether a Triangulation with Forbidden Edges $T(V, X, F)$ exists is NP-complete. [33, triangulation existence problem]

There is another approach to ensure that boundary is part of a Triangulation which we call *constraints*. Basically we require a Triangulation to contain certain objects, e.g. the boundary. This leaves the problem that objects outside the boundary can be part of the Triangulation but those can be removed from the Triangulation in polynomial time (because triangulations have polynomial size).

Definition 3.9 (Constrained Triangulation)
A *Constrained Triangulation* $T(V, X, C)$ is a Triangulation of the vertex set V with respect to conflicts X which contains the edge constrains C:

$$\forall e \in C : e \in T(V, X, C)$$

In contrast to the Triangulation with Forbidden Edges, Constrained Triangulations can be found in polynomial time. Therefore we slightly modify algorithm 2.1 to begin with the constraints as initial Independent Set and then proceed as before. This assumes that the constraints themselves are all valid and not conflicting. Even without that guarantee, an additional check before running the algorithm would require at most quadratic time with respect to the constraints—so the whole running time is still polynomial.

Theorem 3.10 (Time Complexity of Constrained Triangulations)
Using algorithm 2.1 a Constrained Triangulation $T(V, X, C)$ of the vertex set V with respect to conflicts X and constrains C can be calculated in $O(|C|^2 + |X|)$ time. In case C is guaranteed to have the Independent Set property, running time reduces to $O(|X|)$.

3.1. Point Set Triangulations

In this section we present the first geometric triangulation problem and draw the connection to our previous definition. The difference between a (topological) Triangulation as we have defined it earlier and a Point Set Triangulation which we get to in this section can be seen as the distinction between a graph and its embedding.

To make things clear, we start by defining some geometric terms which we use in the following. The equivalent of edges in a topological triangulation are line segments for a geometric triangulation in the plane. Since line segments are not restricted to the plane, we do not focus on two dimensions here. However according to some definitions, triangulations for higher dimensions contain also geometric objects of higher dimension (e.g. tetrahedra in three dimensions). For simplicity we consider those objects be implicitly defined by their bounding line segments.

Definition 3.11 (Line Segments)
A *line segment* $s = (p, q)$ is determined by its endpoints $p, q \in P$ with P being a point set (of arbitrary dimension). For compatibility with other definitions, s is directed from p to q, i.e. $(p, q) \neq (q, p)$ and contains all points m "between" p and q:

$$m \in s \iff \exists a \in [0, 1] : m = p + a \cdot (q - p)$$

Clearly, the counterpart of conflicting edges are intersecting line segments. Nevertheless, depending on the definition, intersection of two line segments with the same endpoint is either empty or the common endpoint. This is why we assume intersection of any geometric objects to be the set intersection of all (usually infinitely many) points contained in the objects and introduce the already widely used term *crossing*:

Definition 3.12 (Crossing)
Two line segments $s_i = (p_i, q_i)$ and $s_j = (p_j, q_j)$ with different slope are *crossing*, if their intersection is not empty and not an endpoint, i.e.

$$s_i, s_j \textit{ crossing} \iff (p = s_i \cap s_j) \wedge (|s_i \cap s_j| = 1) \wedge (p \notin \{p_i, q_i, p_j, q_j\})$$

Two line segments s_i and s_j are *non-crossing* if they are not *crossing*. A set S of line segments is *crossing* if at least two segments $s_i, s_j \in S$ are *crossing*. It is *non-crossing* if each pair $s_i, s_j \in S$ is *non-crossing*.

We do not require points to be in general position which is why we have to deal with degeneracies in the following. General position often preempts interesting instances—e.g. those which are heavily symmetric. Additionally, many man-made structures aim for collinearity, so extra effort has to be expended to make real world instances fit the general position requirements. The following definition identifies all unwanted line segments in case of collinear points. Refer to fig. 3.2 for examples of such degeneracies.

Definition 3.13 (Overlapping Line Segments)
Given a point set P and a line segment $s = (p, q)$ with $p, q \in P$. s is *overlapping* in P if and only if there is a point $p' \in P$ which lies in its interior:

$$s \text{ } overlapping \iff \exists p' \in P : (p' \in s) \wedge (p' \notin \{p, q\})$$

s is *non-overlapping* if it is not *overlapping*.

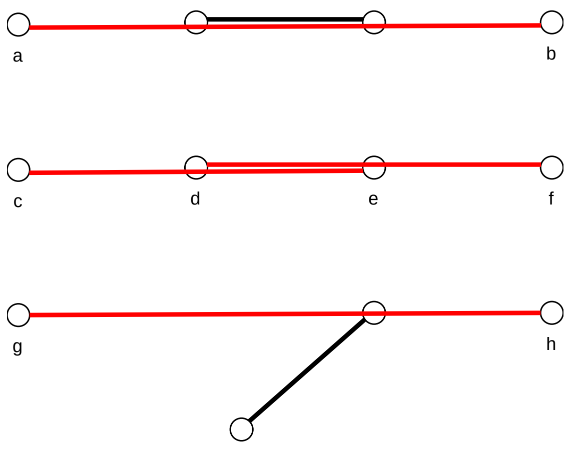


Figure 3.2.: Examples of overlapping line segments ($\{a, b\}, \{c, e\}, \{d, f\}, \{g, h\}$)

Earlier in this section, we already mentioned the connection between topological and geometric triangulations—and here comes the formal definition. Note that we distinguish between points (which have a geometry, i.e. coordinates) and vertices (which are solely topological objects).

Definition 3.14 (Topological Representation)
A vertex set $V(P)$ *represents* a point set P if there is exactly one vertex $v_p \in V(P)$ for each point $p \in P$ and v_p can be identified by p and vice versa.

After having all basic terms at hand, we now proceed with *Point Set Triangulations*. They are the geometric equivalent of topological triangulations for point sets. To emphasize that Point Set Triangulations without further restrictions can be calculated in polynomial time (compare theorems 3.6 and 3.16), we avoid the definition to be based on Triangulations with Forbidden Edges (see definition 3.7)—despite the fact that overlapping line segments are forbidden edges. Note in this case that finding a Triangulation involving overlapping line segments takes polynomial time. Afterwards they can be replaced by the line segments that they include in polynomial time (because there are polynomially many line segments).

Definition 3.15 (Point Set Triangulation)
The Triangulation $T(P)$ of a point set P is a triangulation $T(P) = T(V, X)$ where the vertex set $V = V(P)$ represents P, the conflicts X are all *crossing* line segments, and which contains no *overlapping* line segments:

$$\begin{aligned} (p,q),(p',q') \text{ crossing} &\iff \big\{\{v_p, v_q\}, \{v_{p'}, v_{q'}\}\big\} \in X \\ (p,q) \text{ overlapping} &\implies \{v_p, v_q\} \notin T(P) \end{aligned}$$

For convenience we define $s = (p,q) \equiv e = \{v_p, v_q\}$ such that $s \in T(P) \iff e \in T(P)$. See also [2, Section 9.1] for a slightly different yet equivalent definition of Point Set Triangulations.

The asymptotic time complexity of finding a Triangulation can be improved for the special case of Point Set Triangulations. For example by making use of sweep line algorithms which exploit that Triangulation properties can be ensured locally, the running time drops to $O(n \log n)$—as in [21].

Theorem 3.16 (Time Complexity of Planar Point Set Triangulation)
A planar point set $P \subseteq \mathbb{R}^2$ can be triangulated in $O(n \log n)$ time with $n = |P|$. [2, Theorem 9.12]

In definition 3.9 we already showed a way to force certain objects (e.g. boundaries) to be part of a Triangulation. The same concept can be transferred to Point Set Triangulations as in the following definition:

Definition 3.17 (Constrained Point Set Triangulation)
A *Constrained Point Set Triangulation* $T(P, C)$ of a point set P with line segment constraints C is a Point Set Triangulation $T(P, C) = T(P)$ such that $C \subseteq T(P, C)$.

The same time bound holds in the presence of constraints. Sweep line algorithms can be modified for taking constraints into account. Note that constraints can destroy desired properties of a Point Set Triangulation (such as large inner angles)—we come back to some of those additional requirements later in this chapter.

Theorem 3.18 (Constrained Point Set Triangulation)
A *Constrained Point Set Triangulation* $T(P, C)$ for a planar point set $P \subseteq \mathbb{R}^2$ and line segment constraints $C \subseteq P^2$ can be calculated in $O(n \log n)$ time. [6]

3.2. Intersection Graph

When we defined the Point Set Triangulation (definition 3.15), we assumed that the set of conflicts (i.e. all pairs of *crossing* line segments) is already given—which might not be they case. This section fills the gap by introducing the *Intersection Graph.* For consistency with other definitions, we kept the name even though it actually contains only *crossing* line segments (according to our previous definition of intersection).

Definition 3.19 (Intersection Graph)
For a set of line segments S the *Intersection Graph* $G_{\text{cross}}(S) = (V_S, X)$ consists of a vertex $v_s \in V_S$ for every line segment $s \in S$ and an edge $\{v_{s_i}, v_{s_j}\} \in X$ for every pair of *crossing* segments $s_i, s_j \in S$. It is the geometric equivalent of the Conflict Graph (see definition 3.3).

Generating the Intersection Graph for line segments in the plane can efficiently be done using an output sensitive sweep line algorithm. By output sensitive we refer to the (asymptotic) running time depending on the output size, i.e. how many line segments intersect.

Theorem 3.20 (Time Complexity of Intersection Graph)
Given a set of line segments S in the plane, the Intersection Graph $G_{\text{cross}}(S) = (V_S, X)$ can be calculated in $O(m \log m + i \log m)$ time where $m = |V_S| = |S|$ and $i = |X|$ is the number of intersections in S. [2, Lemma 2.3]

The referenced method performs very well for a small number of intersections but fails to do so when there are significantly more intersections than line segments. Let us first state how many intersections there are when all possible connections in a point set are considered:

Theorem 3.21 (Complexity of Point Set Intersections)
For a planar point set P with n points, the set of all line segments S with endpoints in P has $\Theta(n^4)$ intersection points. [36] Thus calculating the Intersection Graph for S takes $\Omega(n^4)$ time.

This bound on the number of intersections shows that the sweep line approach is not a got choice in our case—especially because it does not reach the lower time bound of $\Omega(n^4)$ (even though we have not yet shown that this is possible at all).

Theorem 3.22 (Non-Optimality of Sweep Algorithm for Point Sets)
The sweep algorithm presented in [2, Section 2.1] with the time complexity of theorem 3.20 is not optimal for finding all *crossing* line segments with endpoints in a given planar point set P of size $n = |P|$ as it takes $O(n^4 \log n)$ time.

Now let us consider the straightforward approach of simply checking all pairs of line segments if they are *crossing*. Algorithm 3.1 realizes this idea with a small optimization: To avoid checking each pair of line segments twice, we take the length $|s|$ of a line segment s into account. The same can be done by using a sorted list instead of a set for the line segments (which even works for line segments with the same length).

Algorithm 3.1 : Naive Intersection Algorithm

```
  Input : Set of line segments $S$
  Output : Intersection Graph $G_{\text{cross}}(S) = (V_S, X)$
1 Set $V_S = \{v_s : s \in S\}$
2 Set $X = \emptyset$
3 foreach $s \in S$ do
4     foreach $s_\times \in S$ with $|s| \leq |s_\times|$ do
5         if $s$ and $s_\times$ are crossing then
6             Add $\{v_s, v_{s_\times}\}$ to $X$
7 return $G_{\text{cross}}(S) = (V_S, X)$
```

We can directly deduce the asymptotic running time from the algorithm and its correctness is implied by the fact that every possible combination of potentially *crossing* line segments is considered.

Theorem 3.23 (Time Complexity and Correctness of Algorithm 3.1)
Algorithm 3.1 finds all *crossing* line segments in $O(m^2)$ time with $m = |S|$.

Again, the most simple approach is indeed asymptotically optimal for our application as already for algorithm 2.1 in chapter 2. Additionally algorithm 3.1 does not assume that the set of line segments is planar.

Theorem 3.24 (Optimality of Algorithm 3.1 for Point Set)
Algorithm 3.1 is asymptotically optimal for finding all *crossing* line segments with endpoints in a given planar point set P as it takes $O(n^4)$ time for $n = |P|$.

3.3. Polygon Triangulations

One case where boundary needs to be taken into account is when triangulating the interior of a polygon. We can reduce this case to the already defined Constrained Point Set Triangulation of the previous section 3.1:

Definition 3.25 (Polygon Triangulation)
A Triangulation $T(P)$ of a polygon P bounded by line segments are all boundary and interior edges of P in a Constrained Point Set Triangulation of the polygon vertices with the polygon boundary as constraints.

We can also reverse the reduction such that a Point Set Triangulation can be constructed by finding a Polygon Triangulation first:

Theorem 3.26 (Generalization of Point Set Triangulation)
Every triangulation of a point set P is a triangulation $T(\text{conv}(P) \cup P)$ of the polygon bounded by the convex hull $\text{conv}(P)$ of P and containing all inner points of P.

Proof:
First recall that the Triangulation $T(\text{conv}(P) \cup P)$ is a Constrained Point Set Triangulation $T_c(P, C)$ for the point set P with the constraints $C = \text{conv}(P)$ being all line segments on the convex hull of P. Now we only need to show that the choice of C does not exclude any Point Set Triangulation by proving that every line segment $s \in C$ has to be part of every Point Set Triangulation for P:

Assume that there is a Point Set Triangulation T' for P which does not contain a line segment $s \in C$. Because T' is a maximal set of *non-crossing* line segments by definition, there has to be a line segment $s' \in T'$ such that s and s' are *crossing*. This implies that the endpoints of s' have to be on opposite sides of s and since s is part of the convex hull, one of the endpoints of s' has to lie outside of the convex hull. This contradicts the definition of the convex hull.

3.4. Edge Flipping

Besides the already mentioned sweep algorithms for Triangulations of planar point sets there is another common approach called *Edge Flipping* which starts with any Triangulation and iteratively replaces an edge by another one until the Triangulation has a certain desired property (e.g. large inner angles).

Definition 3.27 (Edge Flip)
Given a triangulation $T(V, X)$ for a vertex set V with respect to a set of conflicts X, (e, f) with $e \in T(V, X)$ and $f \notin T(V, X)$ is an *edge flip* iff $T(V, X) \setminus \{e\} \cup \{f\}$ is a triangulation for V with respect to X.

An edge flip is not possible with any pair of edges. For planar point sets, both edges need to be diagonals of a convex quadrangle. More generally, they can only be conflicting edges:

Theorem 3.28 (Edge Flips are Conflicts)
Given a Triangulation $T(V, X)$ for a vertex set V with respect to a set of conflicts X, every edge flip (e, f) is a conflict: $\{e, f\} \in X$.

Proof:
Assume $\{e, f\} \notin X$. Further assume that

$$\neg \exists e' \in T(V, X) \setminus \{e\} : \{e', f\} \in X.$$

Then f can be added to $T(V, X)$ (without removing e) and therefore $T(V, X)$ is no triangulation—which is a contradiction. Now let $e' \in T(V, X) \setminus \{e\}$ such that $\{e', f\} \in X$. Then $e' \in T(V, X) \setminus \{e\} \cup f$ —which contradicts that $T(V, X) \setminus \{e\} \cup f$ is a triangulation. Therefore every edge flip (e, f) is a conflict.

Continuously applying edge flips to a Triangulation can be seen as traversing a path within the graph of all Triangulations. Such a graph with vertices for every Triangulation and edges for every possible edge flip is called the *Flip Graph*. An example can be seen in fig. 3.3.

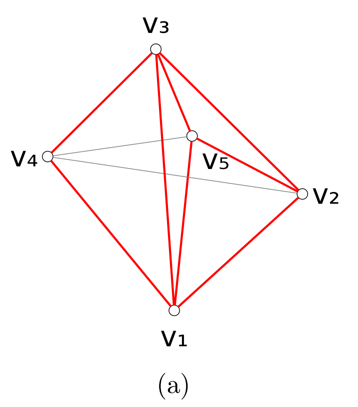
Definition 3.29 (Flip Graph)
The *Flip Graph* $G_{\text{flip}}(V, X) = (V_T, E_{\text{flip}})$ for a vertex set V and edge conflicts X contains a vertex $v \in V_T$ for every triangulation of V with respect to X and edges $e \in E_{\text{flip}}$ for every possible edge flip.

Changing any Triangulation into any other Triangulation using edge flips is only possible if the Flip Graph is connected, i.e. there is a path between every two vertices. This is the case for two dimensions but not yet completely explored for higher dimensions.

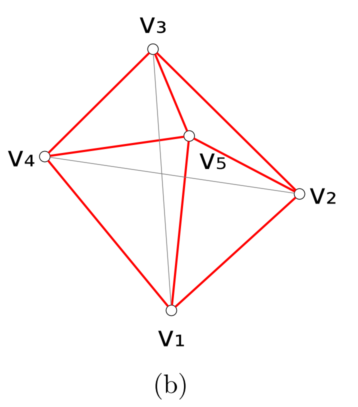
Theorem 3.30 (Connectivity of the Flip Graph)
The Flip Graph $G_{\text{flip}}(V, X)$ is connected in two dimensions [48, Behauptung 4] and has a diameter of at most $6n - 30$ for $n = |V|$. [4] Therefore every Triangulation $T(V, X)$ of a vertex set V with respect to edge conflicts X can be transformed into any other Triangulation of V with respect to X in $O(n)$ time.

For three dimensions, it is still an open problem whether the Flip Graph is connected. [15]

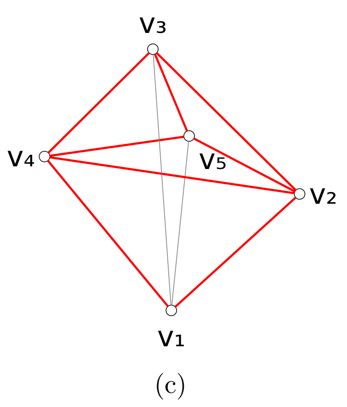
Another issue of edge flipping algorithms are Triangulations which have additional non-locally restricted objectives. If for every two edges in a potential edge flip it can be determined which one improves the Triangulation with respect to the objective, the



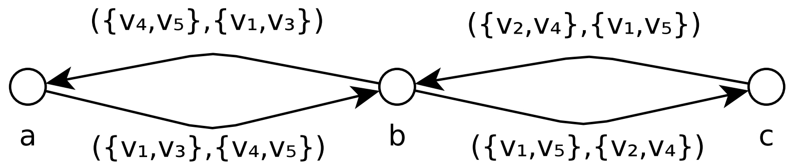
(a)



(b)



(c)



(d) Flip Graph

Figure 3.3.: Example triangulations and their flip graph

Flip Graph can be searched for a Triangulation for which all edge flips would worsen the solution with respect to the objective. This is possible for optimizing the inner angles of triangles. In chapter 4 we show that it is not possible for the MaxMin Length Triangulation (MMLT). Therefore the whole Flip Graph has to be considered—which has exponential size.

3.5. Related Work

In the following, we briefly mention some results for different kinds of Triangulations. These problems have kept researchers busy for over 100 years already [25] and several books have been written which cover the topic, e.g. [14].

The most famous kind is probably the Delaunay Triangulation [2, Section 9.2]. It forces every circumcircle of a triangle to be empty of other points and therefore maximizes the minimum angle [2, Theorem 9.9]. There is an edge flipping algorithm which calculates it in $O(n \log n)$ expected time using $O(n)$ space [2, Theorem 9.12].

The counterpart of a Delaunay Triangulation, minimizing the maximum angle, takes $O(n^2 \log n)$ time and $O(n)$ space [3]. The same approach can also produce triangulations which maximize the minimum height of a triangle. Finally, the same reference shows also that minimizing the maximum slope and minimizing the maximum eccentricity can both be done in $O(n^3)$ time and $O(n^2)$ space.

Optimizing the area of triangles has been studied in [47] resulting in $O(n^2 \log n)$ time and $O(n^2)$ space for minimizing or maximizing the area.

There are several results for optimal edge length Triangulations. One of the first publications [18] shows that minimizing the maximum edge length can be done in $O(n^2)$ time. Minimizing the edge length sum (also known as the Minimum Weight Triangulation) was proven NP-hard [38]. Maximizing the minimum edge length was stated as an open problem [18] but 20 years later it has been shown that it is NP-complete [20]. It remains NP-hard for polygons with holes and interior points [7] but can be solved in $O(n^3)$ time for simple polygons and even in linear time for convex polygons [26].

4. MaxMin Length Triangulation

Subsequently, we get on to the main problem of this thesis: the MaxMin Length Triangulation (MMLT). This chapter covers its complexity, mentions different potential approaches, and finally presents an algorithm for solving it. Let us begin with a formal definition of the problem by reducing it to the Point Set Triangulation (definition 3.15):

Problem 4.1 (MMLT)

Given: Set of points P, length function $|s|$ for each line segment s with endpoints in P (e.g. Euclidean distance of the endpoints)

Sought: Point Set Triangulation $T_{\text{opt}} = T(P)$ of P which maximizes $\min\limits_{s \in T_{\text{opt}}} |s|$

A first observation being made is that an optimal solution for MMLT needs not be unique, i.e. there can be multiple solutions T_{opt} with the same value for the shortest segment in T_{opt}. See also fig. 4.1 for an example.

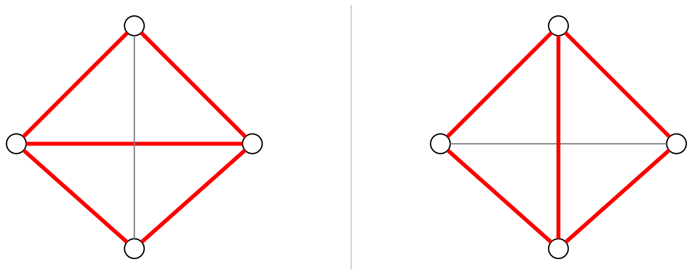


Figure 4.1.: Example of different optimal MMLT solutions for the same point set using Euclidean distance of the endpoints as length for each line segment

4.1. Complexity

As already mentioned in section 3.5, the MMLT problem was stated as an open problem by Edelsbrunner and Tan in 1991 [18] and has recently been proven to be NP-hard by Fekete [20]. In the following, we briefly sketch the proof.

Firstly, we adopt the definition of the Covering by Disjoint Segments (CDS) problem which has been part of [20] and which we will need afterwards:

Problem 4.2 (CDS)

Given: Set of line segments S, set of target points $T \subseteq \{s_i \cap s_j : s_i, s_j \in S\}$

Sought: Set of non-intersecting line segments $S_{\text{opt}} \subseteq S$ such that T is covered:

$$\forall p \in T : \exists s \in S_{\text{opt}} : p \in s$$

It can be shown that CDS is NP-complete—which we will skip at this point:

Theorem 4.3 (NP-completeness of CDS)
Problem 4.2 (CDS) is NP-complete.

Proof:
CDS can be reduced to Planar 3-SAT (problem 2.7) and vice-versa. Therefore we construct a CDS instance from the planar variable-clause-connection graph of a Planar 3-SAT instance. Each variable vertex gets replaced by a cycle of target points which are connected by line segments which alternating represent assigning either true or false to the variable. Additionally, the target points contain all clause vertices which are the intersection point of three long segments intersecting the respective assignment line segment of the variables taking part in the clause. See fig. 4.2 for an example of such a construction. Note that this construction can be done in polynomial time and the CDS can be solved if and only if there is an assignment to the Planar 3-SAT variables which lets all clauses evaluate to "true". For further details of the proof refer to [20].

Now we can proceed with the MMLT problem:

Theorem 4.4 (NP-hardness of MMLT)
Problem 4.1 (MMLT) is NP-hard—even for the case where P is planar.

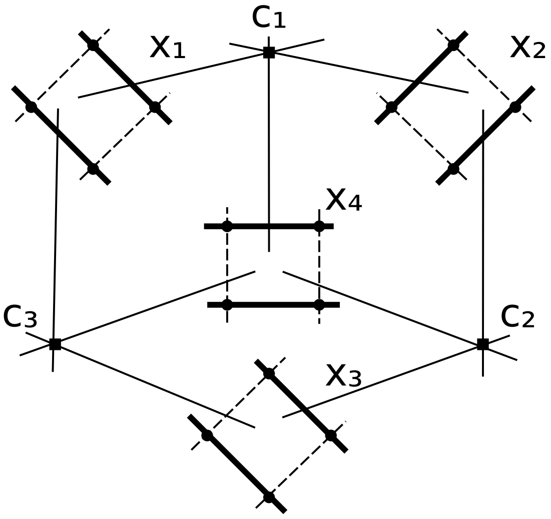


Figure 4.2.: Example of a CDS construction for the Planar 3-SAT instance from fig. 2.1, bold line segments represent assignment of variables to “true”, dashed line segments represent “false”, and long line segments connected clauses with the respective variable assignments

Proof:
We summarize the reduction from CDS to MMLT of [20]. Let us continue from the construction of the proof for theorem 4.3. We assume that three points are collinear only if they correspond to the endpoints of a line segment s and a target point covered by s (this can be achieved by perturbation). Furthermore, let δ be the minimum distance of any point to a line segment it is not part of and assume that every line segment has at least length δ (can be achieved by appropriate scaling). We now replace every target point of the CDS instance by a tuple of points with distance ε such that $\varepsilon \ll \delta$.

The shortest line segment in an optimal MMLT solution for the constructed instance has length ε if and only if the initial CDS has a solution. This is due to the fact that the only line segments of length ε are those which replaced the target points and every other line segment is longer. From the definition of MMLT follows, that every ε-segment is part of the solution if and only if there is no *crossing* line segment in the solution.

For the ε-segments which replaced the clause vertices, only the long line segments connecting a clause with its variables are *crossing*. This implies that the ε-segment for a clause it part of an optimal MMLT solution if and only if the clause can satisfied in the 3-SAT problem (or the target point at that position can be covered in the CDS problem).

In a variable cycle, ε-segments are only picked if neither all line segment representing an assignment to "true" nor all of those representing "false" can be part of the MMLT solution. This is the case when two clauses of the 3-SAT problem require the variable to have conflicting assignments.

Finally, let us point out that the above transformation of a CDS instance into a MMLT instance can be done in polynomial time such that we reduced CDS to MMLT—which implies MMLT being NP-hard.

4.2. Geometric Approaches

In chapter 3 we mentioned two efficient methods for finding Triangulations: sweep algorithms and edge flipping. Here we shortly explain why these approaches can not directly be applied to the MMLT problem.

Sweep algorithms compute solutions for geometric problems (usually in the plan) gradually and rely on the assumption that every part of the solution "behind" the current position of the sweep line is fixed and needs not to be changed anymore. For the MMLT problem, the last line segment reached by the sweep line may change for every line segment it crosses whether it is part of the solution—which may then force itself an update of other line segments. Therefore no part of the solution can be fixed until all line segments are considered which does not comply with the idea of sweep algorithms.

For the edge flipping strategy (traversing a path in the Flip Graph) we show an example in fig. 4.3 which requires an edge flip to worsen the MMLT before the optimal solution can be reached. This essentially requires that every edge flip of the whole Flip Graph has to be considered—which leads to exponential running time.

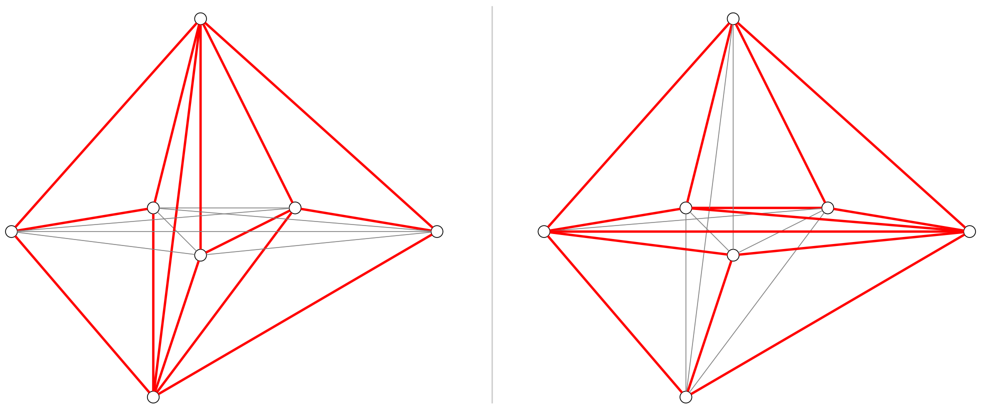


Figure 4.3.: Example of necessary locally non–optimal flips: The shortest edge needs to be flipped in before the optimal solution can be achieved.

4.3. Combinatorial Approach

In this section, we demonstrate an alternative way of finding optimal MMLT solutions based on the underlying combinatorial problem. We come up with another problem, the MaxMin Edge Length Index Triangulation (MELT), which is similar to MMLT yet solely combinatorial.

First of all, we define a new strict total order on the potential edges of a triangulation to circumvent the fact that two different line segments can have the same length which implies that two optimal MMLT solutions may have a different shortest line segment.

Definition 4.5 (Edge Length Order)
Given a set of edges E representing a set of line segments S and two edges $e_{s_i}, e_{s_j} \in E$ representing two line segments $s_i, s_j \in S$, respectively. Let $|s|$ for $s \in S$ be the segment length and $s_i < s_j$ the lexicographical order of $s_i, s_j \in S$. The *Edge Length Order* is defined as

$$e_{s_i} < e_{s_j} \iff |s_i| < |s_j| \vee ((|s_i| = |s_j|) \wedge (s_i < s_j)).$$

Furthermore, we define a short cut for the position that an edge has in a set sorted by the Edge Length Order which can be seen as a higher level length function.

Definition 4.6 (Edge Length Index)
Given a set of edges E representing a set of line segments S and an edge $e \in E$. The *Edge Length Index* $\mathrm{idx}(e)$ is the index of e in E sorted by Edge Length Order.

As a side effect of the Edge Length Order, we can now address every edge in a set uniquely by its Edge Length Index which we state in the following theorem.

Theorem 4.7 (Uniqueness of Edge Length Index)
For a set of edges E representing line segments S, there can be no two different edges $e, e' \in E$ with the same Edge Length Index.

Proof:
For two two segments $s, s' \in S$ the lexicographical order is unique—i.e. either $s < s'$ or $s > s'$ but not both. The same holds for the Edge Length Order.

Using the Edge Length Index for the Triangulation edges instead of the line segment length, we can now formulated the combinatorial equivalent of the MMLT problem:

Problem 4.8 (MELT)

Given: Set of points P

Sought: Point Set Triangulation $T_{\text{opt}} = T(P)$ of P which maximizes the minimum Edge Length Index $\min\limits_{e \in T_{\text{opt}}} \text{idx}(e)$.

Even though, there can still be arbitrary many optimal MELT solutions (as for MMLT) at least the "shortest" edge (meaning the one with minimum Edge Length Index) is the same in all of them. Because neither the MMLT nor the MELT problem ask for the whole vector of edges to be maximum, there can still be different combinations of edges with higher Edge Length Index.

Theorem 4.9 (Uniqueness of Optimal MELT Solutions)
From Theorem 4.7 follows directly that every optimal solution for problem 4.8 (MELT) has the same edge with minimum Edge Length Index.

Even though the link between MMLT and MELT may be obvious, the proof is yet to come and covered by the following theorem.

Theorem 4.10 (Equality of MELT and MMLT)
Every optimal MELT solution is an optimal MMLT solution.

Proof:
Let T_{opt} be an optimal MELT solution for a point set P and assume that that there is a MMLT solution T with $\min\limits_{s \in T} |s| < \min\limits_{s \in T_{\text{opt}}} |s|$. By definitions 4.5 and 4.6 $\text{idx}(\arg\min\limits_{s \in T} |s|) < \text{idx}(\arg\min\limits_{s \in T_{\text{opt}}} |s|)$—which contradicts T_{opt} being optimal.

Now the complexity of MELT is straightforward as a reduction from MMLT can easily be done considering theorem 4.10. Compare also section 4.1 for further details of the complexity of the MMLT problem.

Theorem 4.11 (NP-hardness of MELT)
From theorem 4.10 follows that MELT is NP-hard.

Despite the fact that MMLT instances can be solved through the corresponding MELT problem, there is one important difference between both problems that we will exploit in the following: Making use of the Edge Length Index, MELT instances contain only integers. This allows us to transform the problem into an Integer Program (IP) (see also chapter 2).

Problem 4.12 (IP Formulation of MELT)

$$\begin{aligned} E &: \textit{non-overlapping}\text{ line segments with endpoints in } P \\ X &: \text{pairs of }\textit{crossing}\text{ line segments} \\ T &: \text{MELT solution} \end{aligned}$$

$$\begin{aligned} &\text{maximize} \quad \min_{e \in T} x_e \cdot \mathrm{idx}(e) \\ &\text{subject to } \forall \{e_i, e_j\} \in X : \qquad x_{e_i} + x_{e_j} \le 1 \\ &\qquad\qquad \forall e_i \in E : x_{e_i} + \sum_{\{e_i, e_j\} \in X} x_{e_j} \ge 1 \\ &\qquad\qquad \forall e \in E : \qquad\qquad x_e \in \{0, 1\} \end{aligned}$$

4.4. Separators

The IP for MELT contains n^2 variables and $O(n^4)$ restrictions for n points—even though most of them are irrelevant for the optimal solution. Therefore we identify two groups of interesting edges: *Short Edges*, which are potential candidates for the smallest Edge Length Index in an optimal MELT solution, and *Separators*, which may avoid certain Short Edges.

Definition 4.13 (Short Edges)
Short Edges within a set of edges E are all edges with an Edge Length Index smaller than a certain threshold:

$$E_{\text{short}}(E, \ell) := \{e \in E : \text{idx}(e) < \ell\}$$

Definition 4.14 (Separators)
Given a set of edges E and edge conflicts $X \subseteq E^2$. The set of *Separators* $E_{\text{sep}}(E, X, e)$ for an edge $e \in E$ are all edges that improve the MELT solution, i.e. all e_{sep} with $\{e, e_{\text{sep}}\} \in X$ which have a higher Edge Length Index:

$$E_{\text{sep}}(E, X, e) := \{e_{\text{sep}} \in E : \text{idx}(e) < \text{idx}(e_{\text{sep}}) \wedge \{e, e_{\text{sep}}\} \in X\}$$

The preceding definitions 4.13 and 4.14 allow us to restrict the edges that may influences an optimal MELT solution T_{opt} such that we can formulate an upper bound on the minimum Edge Length Index in T_{opt}.

Theorem 4.15 (Upper Bound for MELT)
Given the optimal MELT solution T_{opt} for a point set P, let E be all *non-overlapping* line segments contained in the Complete Graph $K_{V(P)}$ where $V(P)$ represents P, and let X all conflicts in the Intersection Graph of P. Every edge $e \in E$ without Separators is an upper bound for T_{opt}:

$$\forall e \in E,\ E_{\text{sep}}(E, X, e) = \emptyset : \min_{e_{\text{min}} \in T_{\text{opt}}} \text{idx}(e_{\text{min}}) \leq \text{idx}(e)$$

which is equivalent to

$$\begin{aligned} \forall e \in E : \neg \exists e_{\text{sep}} \in E : \text{idx}(e) < \text{idx}(e_{\text{sep}}) \wedge\ \{e, e_{\text{sep}}\} \in X \\ \implies \min_{e_{\text{min}} \in T_{\text{opt}}} \text{idx}(e_{\text{min}}) \leq \text{idx}(e) \end{aligned}$$

Proof:
Assume

$$\exists e \in E,\ E_{\text{sep}}(E, X, e) = \emptyset : \min_{e_{\text{min}} \in T_{\text{opt}}} \text{idx}(e_{\text{min}}) > \text{idx}(e)$$

This implies $e \notin T_{\text{opt}}$ and

$$\forall e' \in E : \text{idx}(e') < \text{idx}(e) \implies e' \notin T_{\text{opt}}.$$

With $E_{\text{sep}}(E, X, e) = \emptyset$ it follows that $\forall \{e, e_\times\} \in X : e_\times \notin T_{\text{opt}}$. This means that for T_{opt} to be a Triangulation e has to be in T_{opt}—which is a contradiction.

From all the edges without Separators, the one with smallest Edge Length Index is clearly the one which restricts an optimal MELT solution in theorem 4.15 the most. Therefore we give it a name such that we can refer to it later:

Definition 4.16 (Shortest Non-separable Edge)
Given a set of edges E and edge conflicts $X \subseteq E^2$. The *Shortest Non-separable Edge* e_{nose} is the edge with the smallest Edge Length Index from all edges which has no Separators:

$$e_{\text{nose}} := \operatorname*{arg\,min}_{e \in E : E_{\text{sep}}(E,X,e)=\emptyset} \operatorname{idx}(e)$$

Figure 4.4 shows an example where the upper bound for an optimal MELT solution in theorem 4.15 is not tight for e_{nose}. However, we assume that considering e_{nose} is still of great help for reducing the problem size significantly—though we have only experimental evidence by now.

Theorem 4.17 (Index of e_{nose})
The Edge Length Index of the Shortest Non-separable Edge e_{nose} for a random point set P of size $n = |P|$ is at most $\operatorname{idx}(e_{\text{nose}}) \in O(n)$ on average.

Proof:
The proof is still unknown to us, yet we have experimental evidence for this to be true (see section 6.2).

The logical consequence of theorem 4.17 is that we only need to consider part of the edges. Our new task is to find Separators for Short Edges where possible, which is reflected in the Non-Conflicting Separators (NOCS) problem.

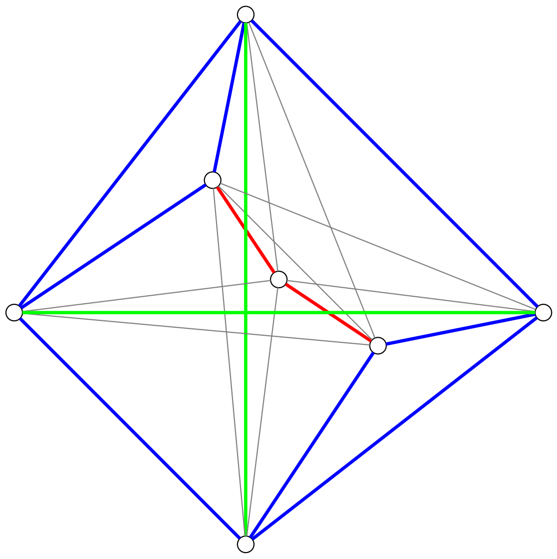


Figure 4.4.: Example where the upper bound from theorem 4.15 is not tight. One of the Short Segments (red) is part of an optimal MELT solution because their Separators (green) cross. Thus the shortest segment in the solution is shorter than the shortest one of all segments without Separators (blue).

Problem 4.18 (NOCS)

Given: Set of Short Edges E and their Separators E_{sep}, edge conflicts $X \subseteq (E \cup E_{\text{sep}})^2$

Sought: Set of non-conflicting edges

$$E_{\text{opt}} \subseteq E \cup \bigcup_{e \in E} E_{\text{sep}}(e)$$

which contains for every edge $e \in E$ either e itself or at least one Separator $e_{\text{sep}} \in E_{\text{sep}}(e)$ and maximizes the smallest Edge Length Index $\min\limits_{e \in E_{\text{opt}}} \text{idx}(e)$

An essential observation at this point is that MELT and NOCS pursue the same objective an lead to the same result—only that the NOCS solution does not contain edges which do not influence the minimum Edge Length Index. Therefore those “irrelevant” edges have to be added to any optimal NOCS solution to retrieve an optimal MELT solution.

Theorem 4.19 (Equality of MELT and NOCS)
Given a point set P along with all *non-overlapping* edges E of the Complete Graph $K_{V(P)}$, and all conflicts X from the Intersection Graph for E. Let e_{nose} be the shortest non-separable edge in E. An optimal MELT solution T_{opt} for P and an optimal NOCS solution S_{opt} for $E_{\text{short}}(E, \text{idx}(e_{\text{nose}}))$ have the same value:

$$\min_{e \in T_{\text{opt}}} \text{idx}(e) = \min_{e \in S_{\text{opt}}} \text{idx}(e)$$

Proof:
...

proof?

A direct consequence of the Shortest Non-separable Edge e_{nose} not being restricted in general (so it can be the edge with highest Edge Length Index in the worst case) is that both problems MELT and NOCS have the same complexity:

Theorem 4.20 (NP-hardness of NOCS)
For $E_{\text{short}}(E, \text{idx}(e_{\text{nose}})) \cup e_{\text{nose}} = E$, NOCS and MELT are the same problem. Therefore NOCS is also NP-hard.

Additionally, the IP formulation for NOCS is also the same as for MELT—besides the fact that we may have less variables and restrictions depending on the Shortest Non-separable Edge e_{nose}.

Problem 4.21 (IP Formulation of NOCS)

$$\begin{aligned} E &: \text{Short Edges} \\ E_{\text{sep}} &: \text{Separators} \\ X &: \text{edge conflicts} \\ S &: \text{NOCS solution} \end{aligned}$$

$$\begin{array}{lrr} \text{maximize} & \min_{e \in S} x_e \cdot \text{idx}(e) & \\ \text{subject to} & \forall \{e_i, e_j\} \in X : & x_{e_i} + x_{e_j} \leq 1 \\ & \forall e_i \in E : x_{e_i} + & \sum_{e_j \in E_{\text{sep}}(e_i)} x_{e_j} \geq 1 \\ & \forall e \in E \cup \bigcup_{e \in E} E_{\text{sep}}(e) : & x_e \in \{0, 1\} \end{array}$$

Assuming that theorem 4.17 is correct, the IP for NOCS is significantly smaller than for MELT as it only has $O(n)$ variables and $O(n^3)$ restrictions in case all the Short Edges conflict with every other edge—and we can assume that this does not happen for random point sets.

The presence of Separators which are not necessary (i.e. which only conflict with Short Edges already conflicting with other Separators) in a NOCS solution is not defined. This is what we intend to change by introducing *complete* NOCS solutions:

Definition 4.22 (Completion of NOCS)
Given an optimal NOCS solution T_{opt} for a set of Short Edges E, their Separators E_{sep}, and conflicts X, the *Completion* of T_{opt} is

$$T := T_{\text{opt}} \cup \left\{ e_{\text{sep}} \in \bigcup_{e \in E} E_{\text{sep}}(e) : \neg \exists e \in T : \{e, e_{\text{sep}}\} \in X \right\}.$$

If $T = T_{\text{opt}}$, T_{opt} is *complete.*

It is important to realize that Completion of an optimal NOCS solution does not destroy its optimality:

Theorem 4.23 (Optimality of NOCS Completion)
Given an optimal NOCS solution T_{opt} the Completion T of T_{opt} is also an optimal NOCS solution.

Proof:
...

proof?

After we defined that "unnecessary" Separators may as well be in an optimal NOCS solution if it is complete, we can exploit the newly acquired structure to recycle the Independent Set problem from chapter 2:

Theorem 4.24 (Connection of NOCS and Independent Set)
Every complete optimal NOCS solution T_{opt} for a set of Short Edges E, their Separators E_{sep}, and conflicts $X \subseteq (E \cup E_{\text{sep}})^2$ is a Maximal Independent Set (definition 2.17) for the simple graph $G = (E \cup E_{\text{sep}}, X)$.

Proof:
...

proof?

4.5. Algorithms

At last, we combine earlier concepts and observations to algorithms for solving MMLT instances. Chapter 5 describes our implementation and chapter 6 shows the results of running it on random point sets. Note that (according to the common assumption that P≠NP) polynomial time algorithms for NP-hard problems—and therefore also for MMLT—are unlikely to exist.

Transforming MMLT instances into MELT instances requires only for sorting the line segments by length and then by lexicographical order to determine the Edge Length Index for every edge. An algorithm for doing so is trivial and therefore we omit it here.

By making use of theorem 4.19, we can give a straightforward algorithm for solving MELT instances though reduction to NOCS.

Constructing the partial Intersection Graph and finding the Shortest Non-separable Edge at the same time can be done using a slightly modified version of algorithm 3.1 keeping the running time of $O(n^4)$ for n points.

Algorithm 4.1 : MELT Algorithm

```
  Input : Point Set $P$
  Output : Optimal MELT Solution $T_{\text{opt}}$ for $P$
1 Let $S$ be all non-overlapping line segments with endpoints in $P$
2 Let $E$ be edges representing all line segments in $S$ sorted by Edge Length Order
3 Find the Shortest Non-Separable Edge $e_{\text{nose}} \in E$ and compute the partial Intersection
  Graph $G_{\text{cross}}(S') = (E', X)$ for all line segments $S' \subseteq S$ represented by edges $E'$ with
  $E' = E_{\text{short}}(E, \text{idx}(e_{\text{nose}}))$
4 Find an optimal NOCS solution $E_{\text{opt}}$ for the Short Edges $E'$ with Separators
  $E_{\text{sep}}(E, X, e)$ for every $e \in E'$ and conflicts $X$
5 Compute a Constrained Point Set Triangulation $T_{\text{opt}}$ for $P$ with the constraints $E_{\text{opt}}$
6 return $T_{\text{opt}}$
```

Solving the NOCS instance can be done by directly using the IP formulation. In our implementation however, we run a binary search on the Edge Length Index starting with the interval 0 to $\text{idx}(e_{\text{nose}})$, and solve the corresponding decision problem for NOCS. This gives us the chance to abort the algorithm at any time gaining upper and lower bound for the optimal NOCS solution—and, if wanted, a "good" MELT solution instead of an optimal one.

5. Implementation

In this chapter, we will briefly describe the program components—roughly split into the geometry and the optimization part. To let the program run close to the hardware layer (which usually leads to fast execution times), the code was written in C++. First attempts to use Python instead (for the sake of clarity and better readability) stumbled over the non-readiness of the Computational Geometry Algorithms Library (CGAL) bindings and the lack of good alternatives. For the technical documentation, please refer to appendix A.

5.1. Geometry

The geometry part consists of number, point, and segment types, data structures for triangulation, convex hull, and bounding box, and it also handles the segment intersection. For most of it we made extensive use of CGAL, which will be introduced in section 5.1.1.

5.1.1. CGAL

CGAL [5] is an Open Source library (mainly) for computational geometry written in C++. It includes most of the common algorithms in the field and also offers efficient data structures. Through the use of C++ templates it is flexible and extendable: For example it is common to adjust the underlying number types to the application.

5.1.2. Kernel

A kernel in CGAL is something like a computational geometry operating system: It holds the basic type definitions like numbers, points, lines, and line segments. Basic operations such as intersection, angle calculations, or comparisons are also part of it.

In our application we use the built-in `Exact_predicates_inexact_constructions_kernel`[1] [19], which uses double as a number type and is not capable of constructing new geometric objects from existing ones accurately. Both properties lead to faster execution time yet do not produce wrong results in our case.

On top of the CGAL kernel there are two modifications: One is for printing points and segments without the need to use streams, the other one to output them to SVG (see also sections 5.3.2 and 5.3.6). Additionally segments are indexed by length and have the information whether they overlap with other segments attached to them.

[1]Thanks to Michael Hemmer for making me aware that I should use it!

5.1.3. Triangulation

CGAL brings along a constrained triangulation already [9] which triangulates a point set with respect to a given mandatory set of (non-crossing) edges. For this application the class was extended to be drawable to SVG and to find the shortest edge which is part of the triangulation.

5.1.4. Convex Hull

This class directly calls the `convex_hull_2` function of CGAL [10] which itself defaults to the algorithm of Akl & Toussaint [1]. The only extension is that the class serves as container which holds the output points and contains a function to find the shortest segment within.

For the algorithm this class is not necessary as its bound is worse than the one of the SAT solution. It is left in the implementation though as a measure of quality for the other bound.

5.1.5. Intersection Algorithm

To find all pairs of intersecting segments we use algorithm 3.1. For the intersection check itself we make use of the CGAL function `do_intersect` [16]. In contrast to the `intersecttion` function [28], it does not actually compute the intersection and therefore performs much better.[2]

5.1.6. Intersection Graph

The intersection graph (as defined in definition 3.19) stores for every segment the indices of all intersecting segments. This graph data structure (adjacency list) performs well for few edges (in this case intersections) per vertex (in this case line segment)—which we assume here. It may however in future versions of the implementation be replaced by a sparse adjacency matrix from the section 5.3.1 library.

5.2. Optimization

The optimization part itself consists mainly of data structures for SAT problems and solutions, an interface for SAT solvers and the solvers themselves (currently only CPLEX).

Only segment indices and intersections are passed to the SAT problem. This is because geometry does not influence the problem, but only topology. Also it is easier that way to keep track of which segments take part in the restrictions.

5.2.1. SAT problem

This class serves two purposes: To grant an interface to the relevant data for the SAT (i.e. segments and intersections) and to set the short segment range.

[2]Thanks again to Michael Hemmer!

5.2.2. SAT solution

The SAT solution class mainly just stores the segment indices derived from solving the SAT problem — which can be none if no feasible solution is found. Additionally it offers methods for drawing short segments and separators and for finding the shortest segment of the solution.

5.2.3. SAT solver

To unify the way solving the SAT problem is done, there are three interfaces: The base SAT solver and two derived interfaces for decision and optimization problems. They all share methods for adding forbidden segments, intersection and separation restrictions and for running the actual solving. Furthermore there is a method for binding short segments to the objective function for optimization problems.

5.2.4. CPLEX

IBM ILOG CPLEX [27] is a commercial optimization suite written in C. It contains a standalone tool for solving optimization problems and also includes libraries for being used in other programs or even other programming languages. According to [37] CPLEX is one of the two fastest MILP solvers.

In our application we use the Concert API for C++ [8] to access CPLEX. It allows for adding variables and restrictions to a model, extracting them to more efficient data structures and then running several solving algorithms on it.

5.3. Remaining Components

Besides the geometry and optimization parts, our implementation contains the following components: A controller class for the whole algorithm which combines all the other components and utility classes for reading input files, debug output and assertions, test case generation, and drawing certain states of the algorithm to SVG.

5.3.1. Boost

Boost [13] is a collection of free (as in freedom) C++ libraries containing tools for various tasks. It makes extensive use of the C++ pre-processor (mostly templates) and aims to extend the C++ standard library (STL).

In our implementation, we use the Spirit [24] library for parsing the JSON input files (see section 5.3.3), and the Program Options library [41] for parsing command line arguments and configuration files. Later versions of the implementation may as well use the Boost Graph Library [46] for the intersection graph, and the Geometry library [22] for creating SVG images (see section 5.3.6).

5.3.2. Qt

Qt [44] is a framework for cross-platform application development written in C++. It features some enhancements to the C++ standard library including an own string class `QString` [43] which supports a different formatting syntax than the `std::string`, offers a UI description format with integration into a even processing framework, and also comes with a simplified built system qmake which wraps platform dependent tools such as GNU make.

We make use of `QString` for our logger class (section 5.3.4), generate SVG images through the `QPainter` class, and build our applications using qmake (which also allows for integration in the C++ IDE Qtcreator).

5.3.3. JSON Parser

JSON Spirit [49] uses the Boost Spirit library [24] for parsing the input point files in JSON (JavaScript Object Notation) [29] format. As of writing, the built-in JSON support of Qt unfortunately has a bug [35] such that it is not compatible with the C++ standard library (and therefore also neither with CGAL nor with Boost). After parsing the input file, all points are stored in a sorted set to allow for fast lookup.

5.3.4. Logger

Our logger class supports different levels of verbosity (debug, info, error, print) and adds the current timestamp to each of them. Additionally, there are shortcut methods for output of measured times, and the current status of the algorithm. Debug output is only included in the programs if they are compiled in debug mode. For convenience, all methods accept `QString`.

5.3.5. Point Generator

For testing and analyzing the algorithm, we generated different instances of point sets and stored them for repeated runs. We hereby rely on the `Random_points_in_square_2` class [45] in combination with the `Creator_uniform_2` class [12]—both part of CGAL (section 5.1.1).

5.3.6. SVG Painter

Mainly for debugging purposes and to visualize different steps of the algorithm, we included the possibility to draw certain data structures to SVG using the `QPainter` class [42].

6. Results

. . .

3 pages

This chapter covers the results from running our implementation on a fine grid. Random sample points are uniformly selected within a square of side length $100\sqrt{n}$ where n is the number of points. This guarantees the same point density for all sample sizes and seemed to be a sufficient accuracy for real world instances. All coordinates are integer and transformed back to the unit square for analysis. For calculations (i.e. within the implementation itself) we use `double` however.

6.1. Technical Details

All experiments are run on a Intel® Core™ 2 Duo CPU E6850 with 2 GB RAM. For everything but the IP-solving with CPLEX only one core is used. We compile with the GCC and following options: `-frounding-math -std=c++11 -O3`. The libraries we compile against are Qt 5.0.0, CGAL 4.0.2, Boost 1.49, and JSON Spirit 4.06.

6.2. Segment Lengths

Figures 6.1 and 6.2 show the trend of the shortest non-separable segment in comparison with the shortest segment overall and the shortest segment of the MaxMin Edge Length Index Triangulation (MELT) solution. For each data point 100 instances were run–though some of them have been aborted after 30 minutes (refer to section 6.4 and tables B.2 and B.3).

Even though the specific progression of the shortest non-separable segment index is not clear from fig. 6.2, it can be assumed that it is sub-linear. From fig. 6.1, we can also see that the shortest segment length significantly drops below the shortest non-separable segment length—which results in more segments being in between for a uniform distribution.

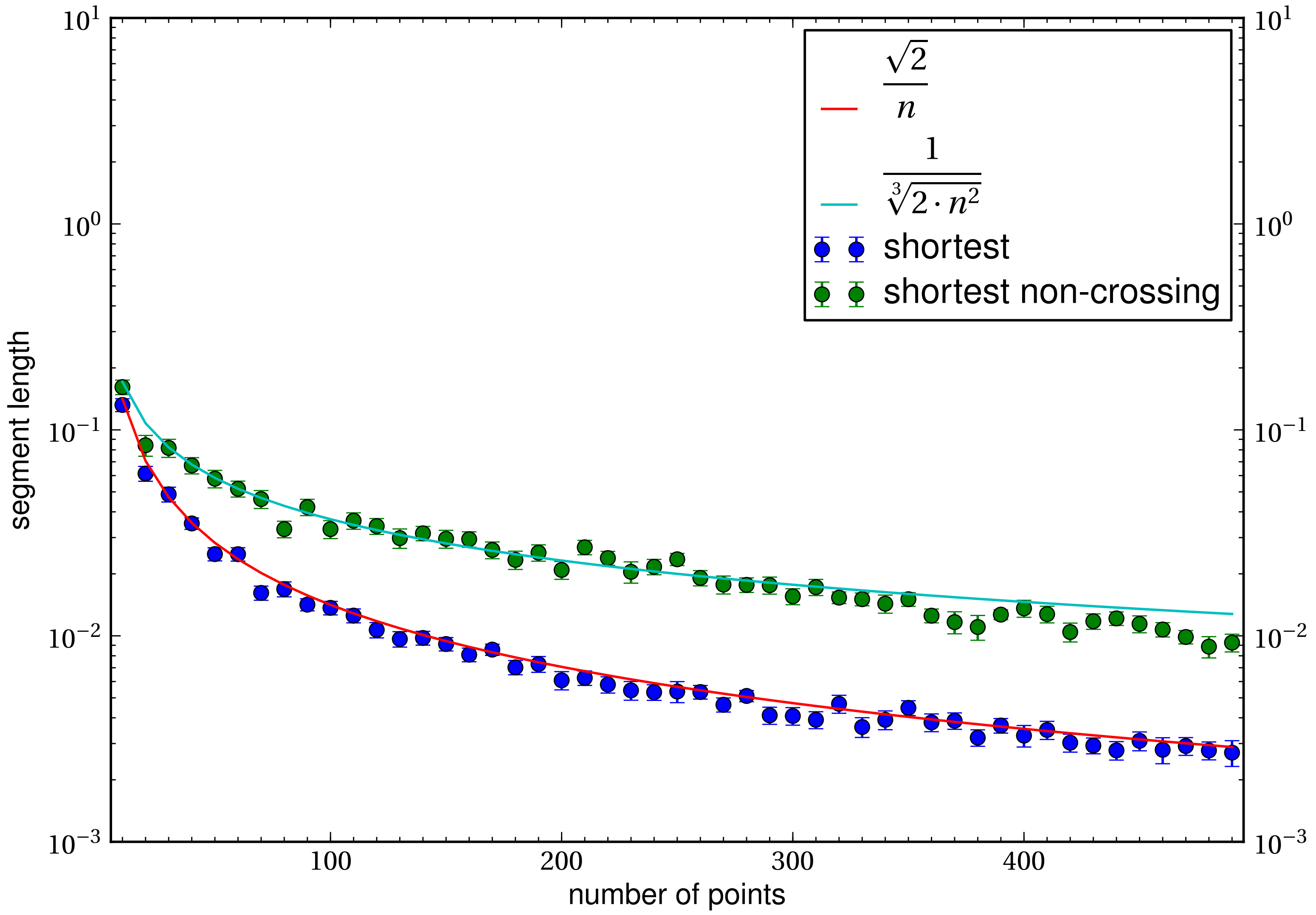


Figure 6.1.: Comparison of segment lengths

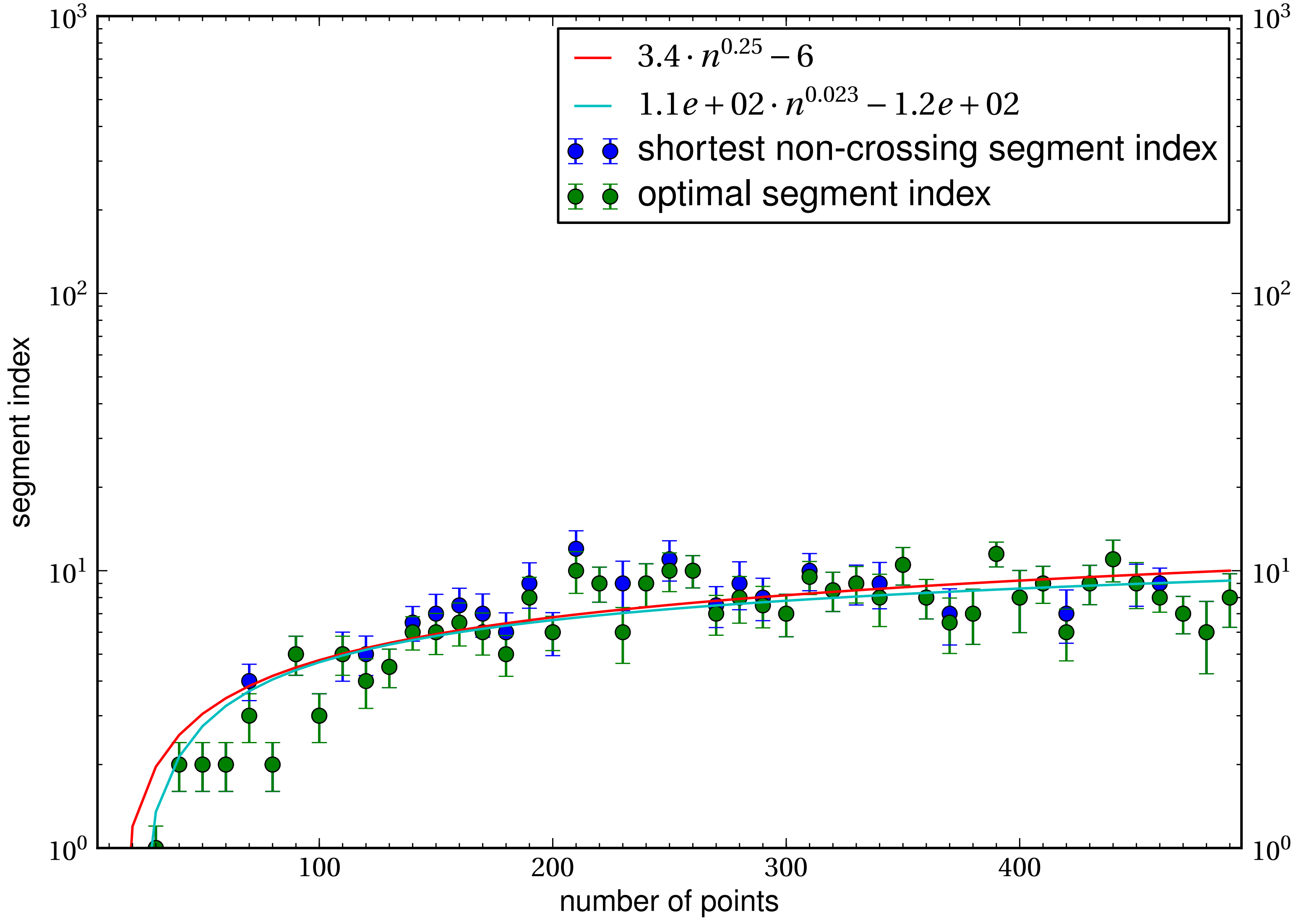


Figure 6.2.: Comparison of segment indices

6.3. Execution Time

We made different experimental analyses on the running time of our implementation. Again we let 100 instances run for each data point. The resulting times are real time (in contrast to user or system time)—i.e. execution time of non-related background processes is not excluded. This decision was made because it is usually impossible to guarantee that no other (system) tasks are running, so real time is more meaningful (yet less accurate).

Figure 6.3 shows which part of our algorithm takes what amount of time in comparison to the full execution execution time. As can

table B.1

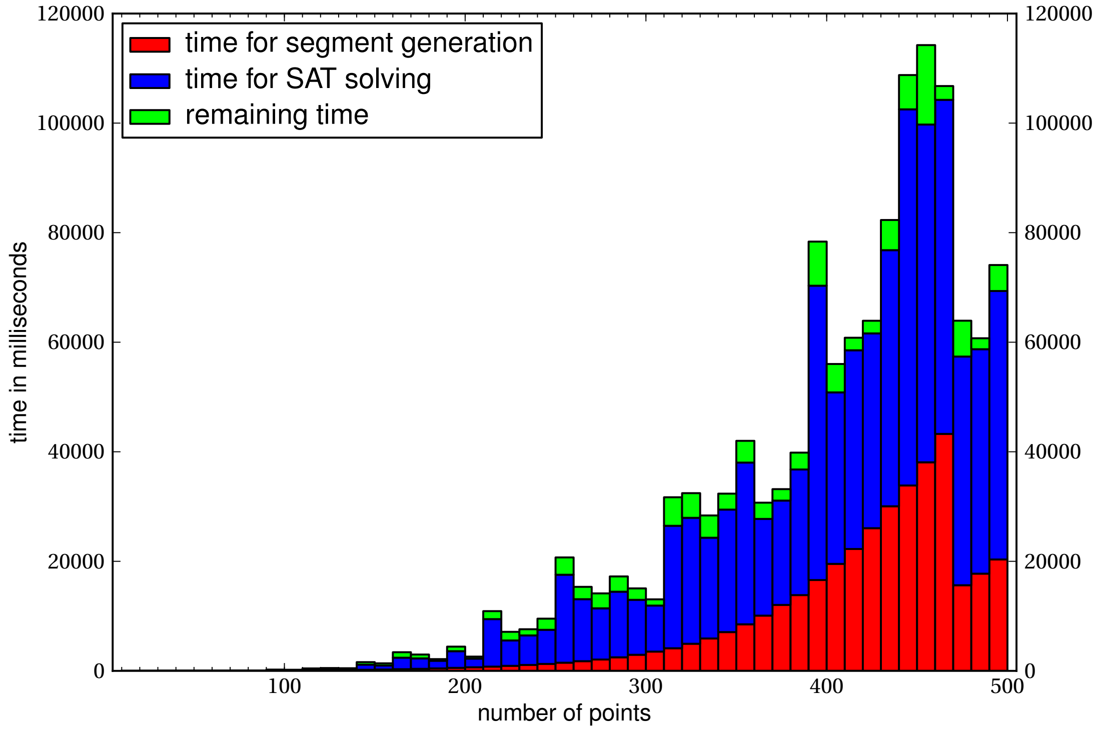
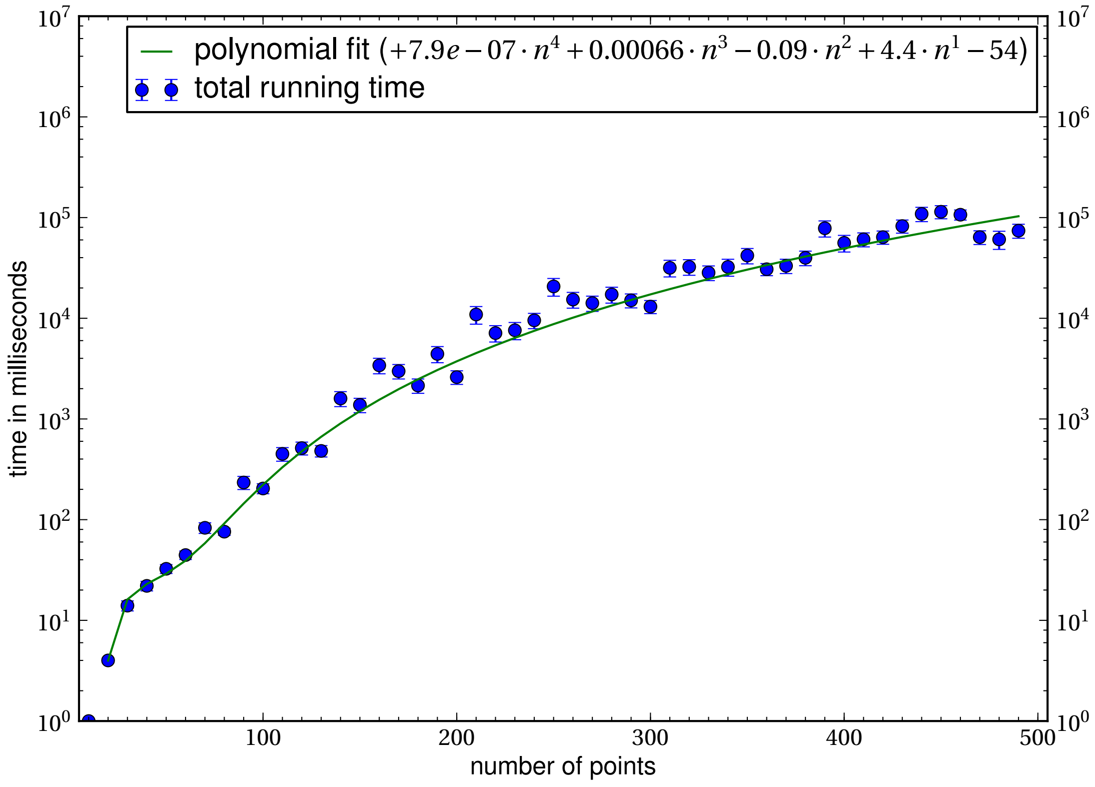
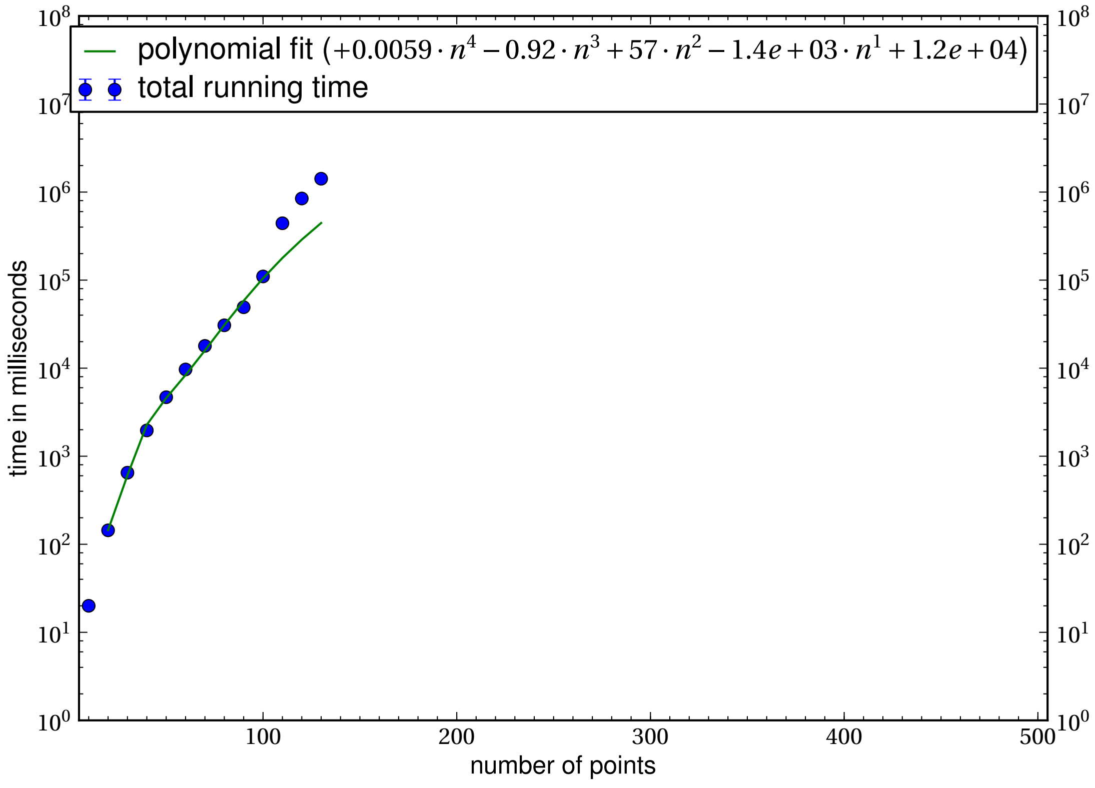


Figure 6.3.: Composition of execution times

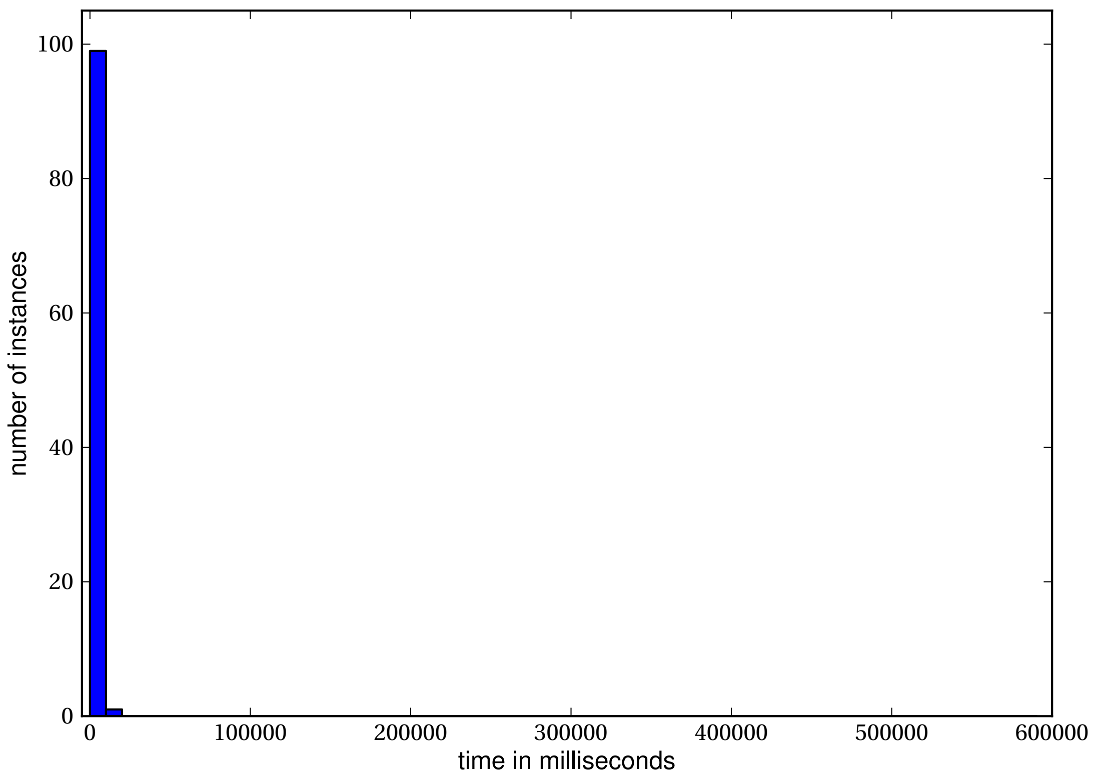


(a) Improved method

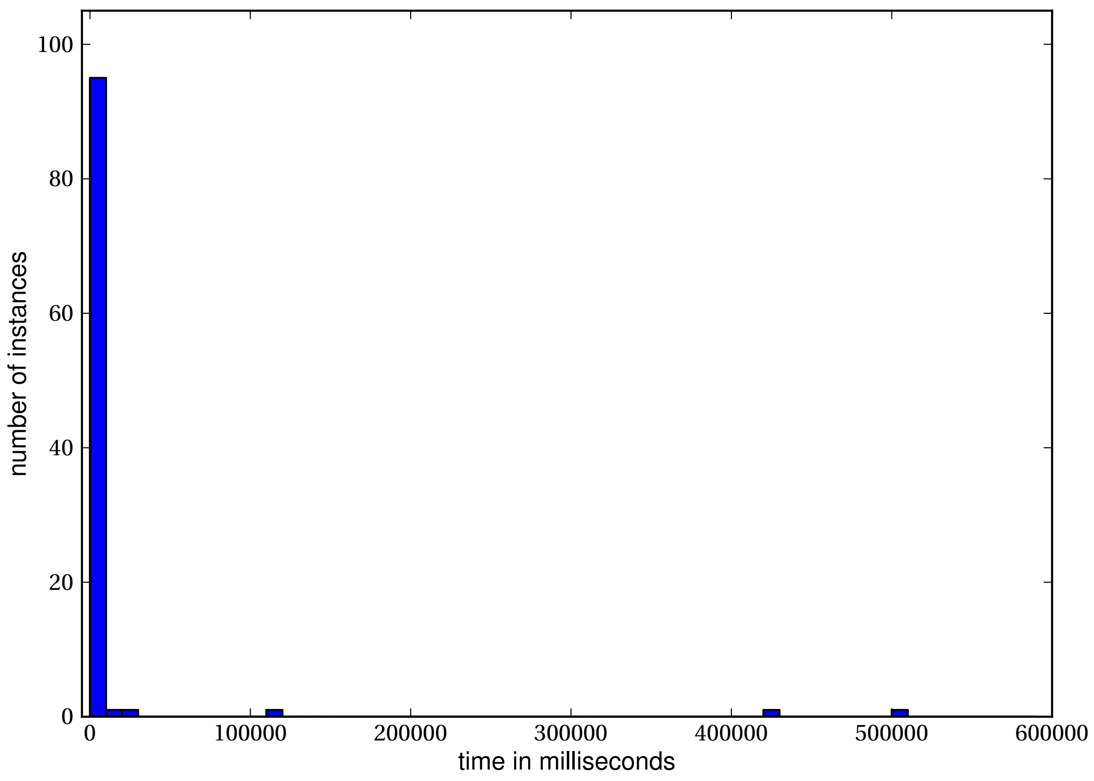


(b) Complete SAT

Figure 6.4.: Total execution time



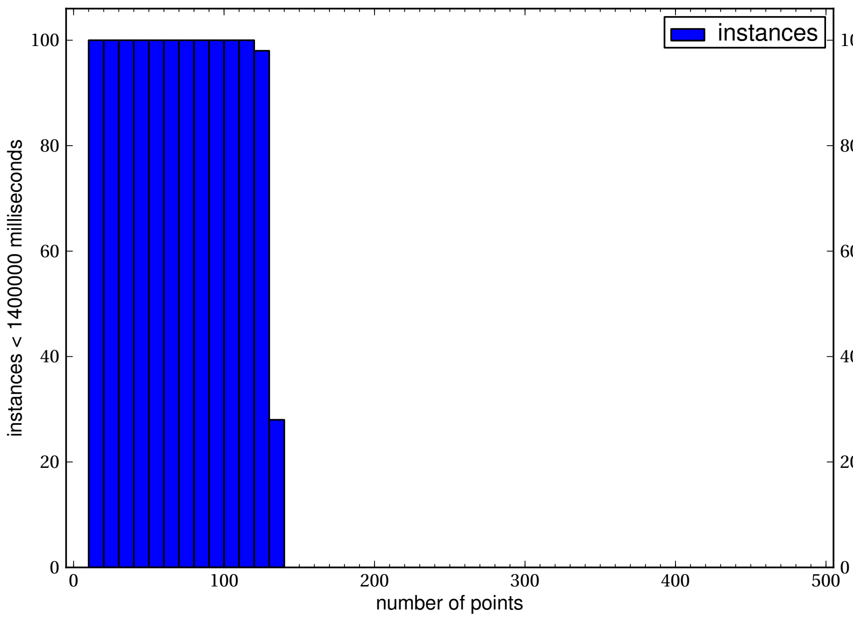
(a) 70 points



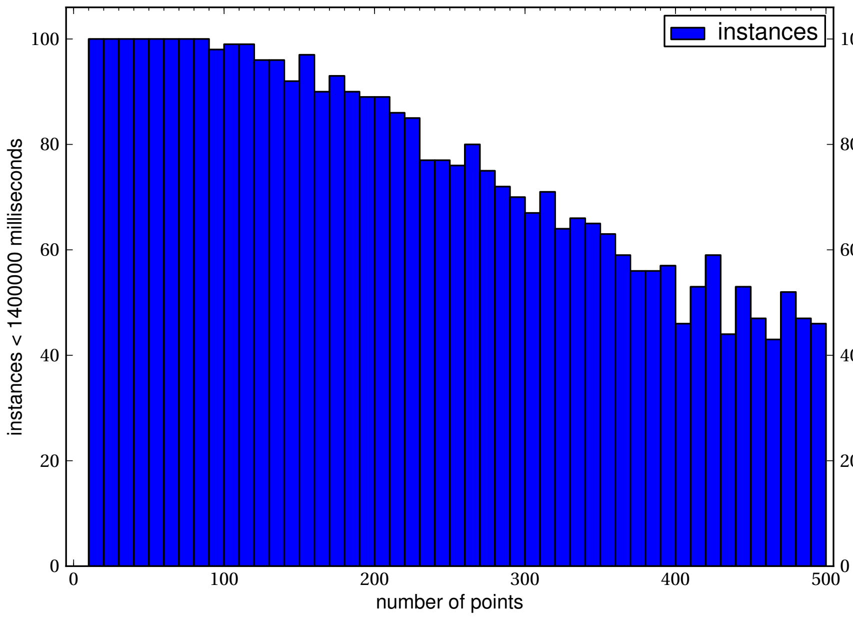
(b) 80 points

Figure 6.5.: Histogram of execution times

6.4. Aborted instances



(a) Complete SAT



(b) Improved method

Figure 6.6.: Instances with running time $<$ 23:20min

7. Conclusion

A. Documentation

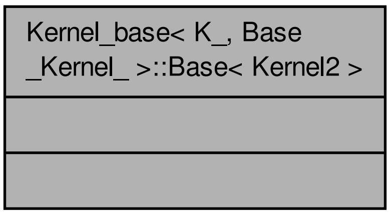
The following is the technical documentation of the program, including class structure and interfaces. It was generated using Doxygen [17]. For an overview of the program components see chapter 5. The source code itself is hosted at: `https://bitbucket.org/winniehell/mmlt`

Contents

1 Class Documentation

1.1 Kernel_base< K_, Base_Kernel_ >::Base< Kernel2 > Struct Template Reference

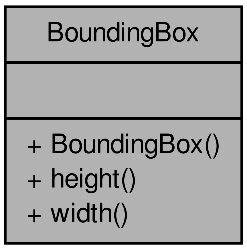
Collaboration diagram for Kernel_base< K_, Base_Kernel_ >::Base< Kernel2 >:



1.2 BoundingBox Class Reference

```
#include <bounding_box.h>
```

Collaboration diagram for BoundingBox:



Public Member Functions

- BoundingBox (const PointSet &points)
- Number height () const
- Number width () const

1.2.1 Constructor & Destructor Documentation

1.2.1.1 BoundingBox::BoundingBox (const PointSet & *points*) `[inline]`

1.2.2 Member Function Documentation

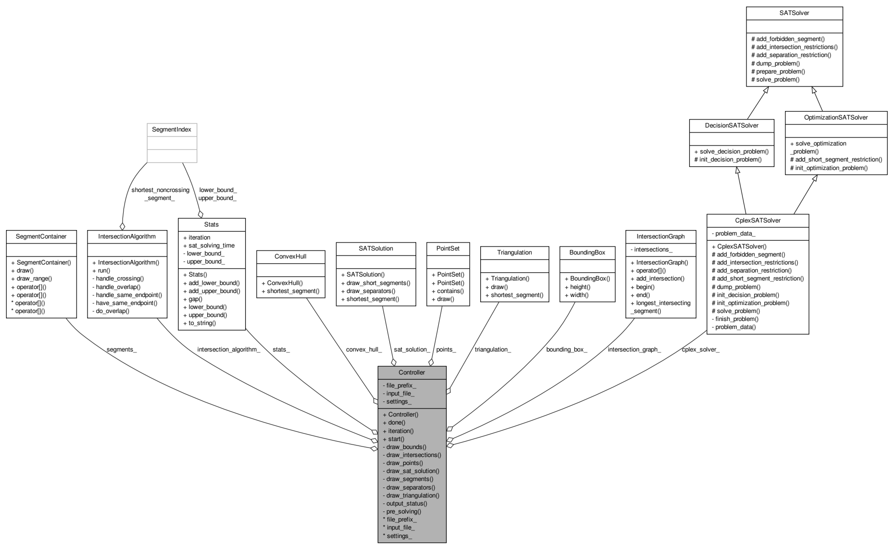
1.2.2.1 Number BoundingBox::height ( ) const `[inline]`

1.2.2.2 Number BoundingBox::width ( ) const `[inline]`

1.3 Controller Class Reference

```
#include <controller.h>
```

Collaboration diagram for Controller:



Public Member Functions

- Controller (const QString &file_prefix, QFile &input_file, const QSettings &settings)
- void done ()
- bool iteration ()
- bool start ()

Private Member Functions

- void draw_bounds () const
- void draw_intersections () const
- void draw_points (SVGPainter &painter) const
- void draw_sat_solution () const
- void draw_segments (SVGPainter &painter) const
- void draw_separators () const
- void draw_triangulation () const
- void output_status () const
- void pre_solving ()

Private Attributes

independent members

- CplexSATSolver cplex_solver_
- IntersectionAlgorithm intersection_algorithm_
- SATSolution sat_solution_
- Stats stats_

input parameters

- const QString & file_prefix_
- QFile & input_file_
- const QSettings & settings_

dependent on input parameter

- const PointSet points_

dependent on input points

- const BoundingBox bounding_box_
- const ConvexHull convex_hull_
- SegmentContainer segments_
- Triangulation triangulation_

dependent on segments

- IntersectionGraph intersection_graph_

1.3.1 Constructor & Destructor Documentation

1.3.1.1 Controller::Controller (const QString & *file_prefix*, QFile & *input_file*, const QSettings & *settings*)

1.3.2 Member Function Documentation

1.3.2.1 void Controller::done ( )

called after the algorithm finished

1.3.2.2 void Controller::draw_bounds ( ) const `[private]`

1.3.2.3 void Controller::draw_intersections ( ) const `[private]`

1.3.2.4 void Controller::draw_points (SVGPainter & *painter*) const `[private]`

1.3.2.5 void Controller::draw_sat_solution ( ) const `[private]`

1.3.2.6 void Controller::draw_segments (SVGPainter & *painter*) const `[private]`

1.3.2.7 void Controller::draw_separators ( ) const `[private]`

1.3.2.8 void Controller::draw_triangulation ( ) const `[private]`

1.3.2.9 bool Controller::iteration ( )

run next iteration

Returns

true if next iteration should be triggered

1.3.2.10 void Controller::output_status ( ) const `[private]`

dumps the current algorithm status

1.3.2.11 void Controller::pre_solving ( ) `[private]`

does some pre-processing

1.3.2.12 bool Controller::start ( )

start the algorithm

Returns

true if iteration should be triggered

1.3.3 Member Data Documentation

1.3.3.1 const BoundingBox Controller::bounding_box_ [private]

1.3.3.2 const ConvexHull Controller::convex_hull_ [private]

1.3.3.3 CplexSATSolver Controller::cplex_solver_ [private]

1.3.3.4 const QString& Controller::file_prefix_ [private]

1.3.3.5 QFile& Controller::input_file_ [private]

1.3.3.6 IntersectionAlgorithm Controller::intersection_algorithm_ [private]

1.3.3.7 IntersectionGraph Controller::intersection_graph_ [private]

1.3.3.8 const PointSet Controller::points_ [private]

1.3.3.9 SATSolution Controller::sat_solution_ [private]

1.3.3.10 SegmentContainer Controller::segments_ [private]

1.3.3.11 const QSettings& Controller::settings_ [private]

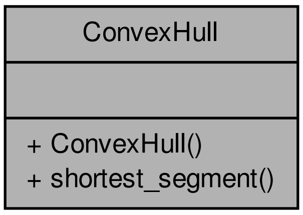
1.3.3.12 Stats Controller::stats_ [private]

1.3.3.13 Triangulation Controller::triangulation_ [private]

1.4 ConvexHull Class Reference

`#include <convex_hull.h>`

Collaboration diagram for ConvexHull:



Public Member Functions

- ConvexHull (const PointSet &points)
- const SegmentIndex & shortest_segment (const SegmentContainer &segments) const

1.4.1 Constructor & Destructor Documentation

1.4.1.1 ConvexHull::ConvexHull (const PointSet & *points*)

compute convex hull of given point set

1.4.2 Member Function Documentation

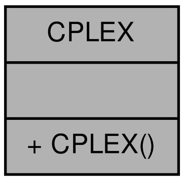
1.4.2.1 const SegmentIndex & ConvexHull::shortest_segment (const SegmentContainer & *segments*) const

find the convex hull segment with minimum length

1.5 CPLEX Class Reference

```
#include <concert.h>
```

Collaboration diagram for CPLEX:



Public Member Functions

- CPLEX ()

1.5.1 Detailed Description

ugly CPLEX code is not our fault helper class for CPLEX concert API

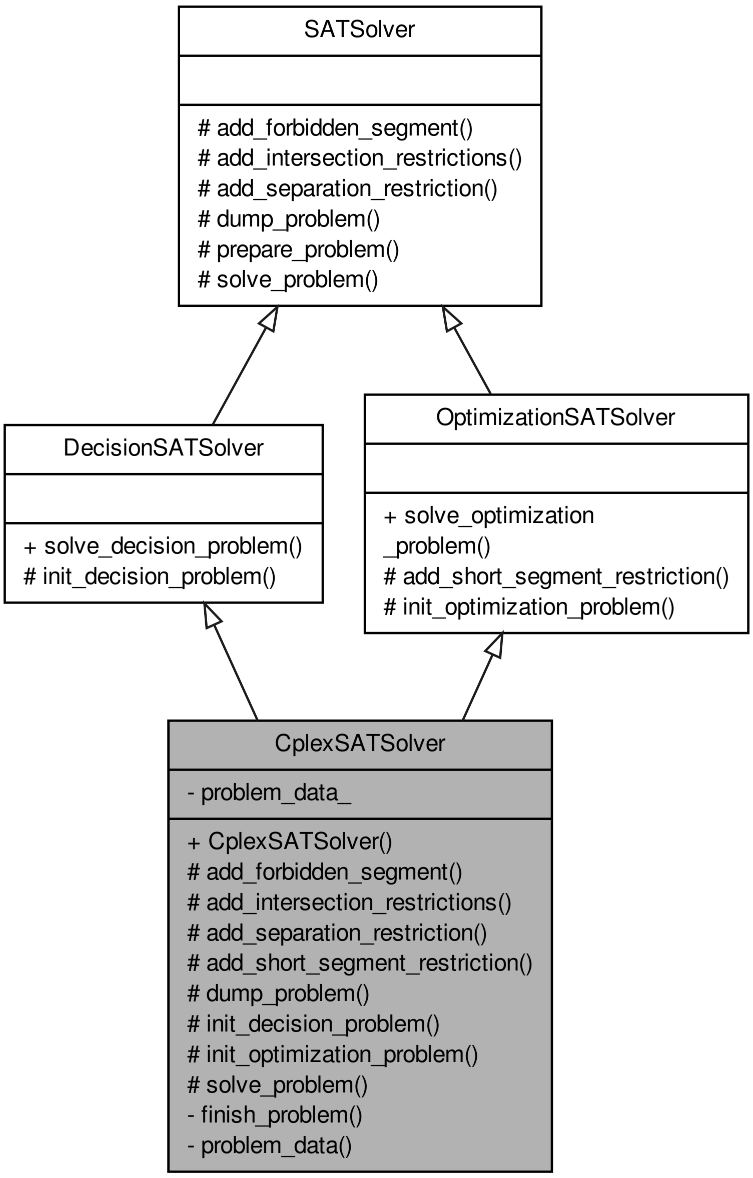
1.5.2 Constructor & Destructor Documentation

1.5.2.1 CPLEX::CPLEX ( ) [inline]

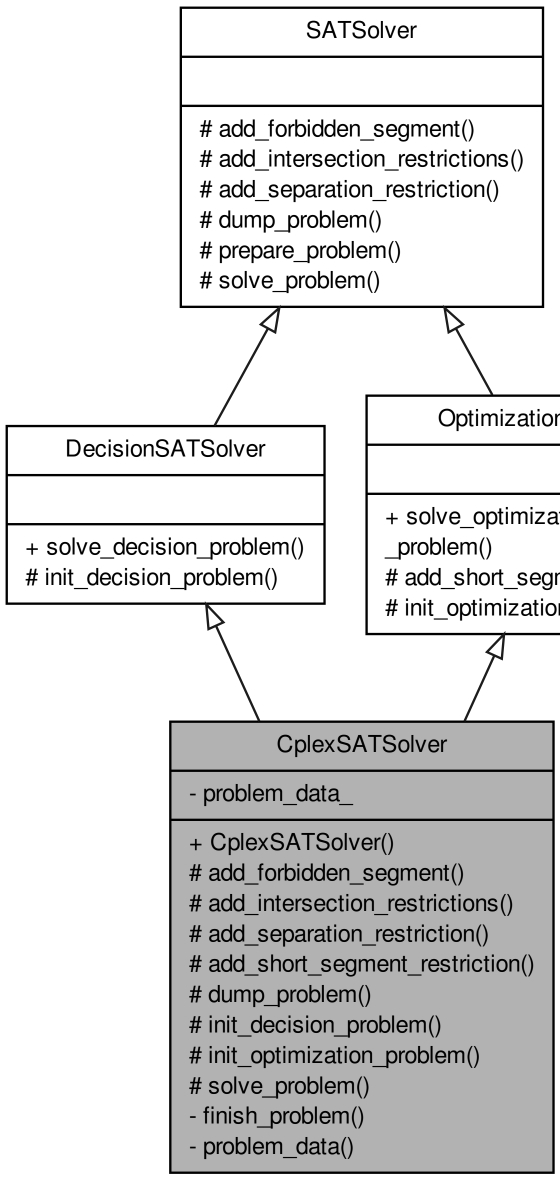
1.6 CplexSATSolver Class Reference

```
#include <cplex_sat_solver.h>
```

Inheritance diagram for CplexSATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
DecisionSATSolver
+ solve_decision_problem()
# init_decision_problem()
OptimizationSATSolver
+ solve_optimization _problem()
# add_short_segment_restriction()
# init_optimization_problem()
CplexSATSolver
- problem_data_
+ CplexSATSolver()
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# add_short_segment_restriction()
# dump_problem()
# init_decision_problem()
# init_optimization_problem()
# solve_problem()
- finish_problem()
- problem_data()

Collaboration diagram for CplexSATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
DecisionSATSolver
+ solve_decision_problem()
# init_decision_problem()
OptimizationSATSolver
+ solve_optimization _problem()
# add_short_segment_restriction()
# init_optimization_problem()
CplexSATSolver
- problem_data_
+ CplexSATSolver()
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# add_short_segment_restriction()
# dump_problem()
# init_decision_problem()
# init_optimization_problem()
# solve_problem()
- finish_problem()
- problem_data()

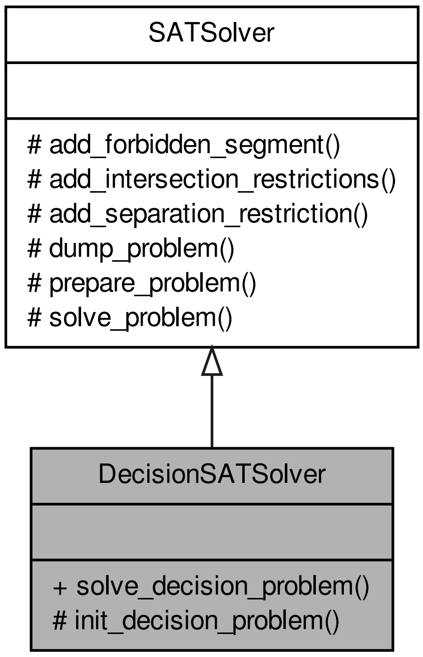
Classes

- struct ProblemData

Public Member Functions

- CplexSATSolver ()

Collaboration diagram for DecisionSATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
DecisionSATSolver
+ solve_decision_problem()
# init_decision_problem()

Public Member Functions

- void solve_decision_problem (const QSettings &settings, const QString &file_prefix, const SATProblem &problem, SATSolution &solution)

Protected Member Functions

- virtual void init_decision_problem (const SATProblem ∗problem)=0

1.6.1 Detailed Description

interface for decision SAT solvers

1.6.2 Member Function Documentation

1.6.2.1 virtual void DecisionSATSolver::init_decision_problem (const SATProblem ∗ *problem*) [protected], [pure virtual]

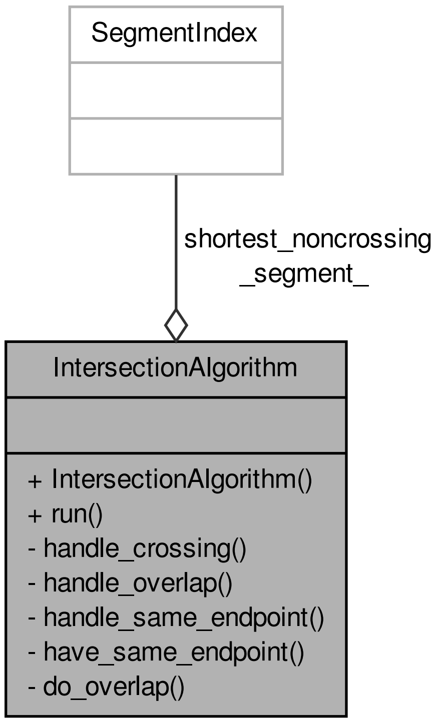
Implemented in CplexSATSolver.

1.6.2.2 void DecisionSATSolver::solve_decision_problem (const QSettings & *settings,* const QString & *file_prefix,* const SATProblem & *problem,* SATSolution & *solution*)

1.7 IntersectionAlgorithm Class Reference

```
#include <intersection_algorithm.h>
```

Collaboration diagram for IntersectionAlgorithm:



Public Member Functions

- IntersectionAlgorithm ()
- void run (IntersectionGraph &igraph, SegmentContainer &segments)

Public Attributes

- SegmentIndex shortest_noncrossing_segment_

Private Member Functions

- void handle_crossing (IntersectionGraph &igraph, const Segment &s1, const Segment &s2)
- void handle_overlap (IntersectionGraph &igraph, const Segment &s1, const Segment &s2)
- void handle_same_endpoint (const Segment &s1, const Segment &s2) const
- bool have_same_endpoint (const Segment &s1, const Segment &s2) const
- bool do_overlap (Segment &s1, Segment &s2) const

1.7.1 Constructor & Destructor Documentation

1.7.1.1 IntersectionAlgorithm::IntersectionAlgorithm ( )

1.7.2 Member Function Documentation

1.7.2.1 bool IntersectionAlgorithm::do_overlap (Segment & *s1*, Segment & *s2*) const `[private]`

checks if two segments overlap

Returns

the outer segment

1.7.2.2 void IntersectionAlgorithm::handle_crossing (IntersectionGraph & *igraph*, const Segment & *s1*, const Segment & *s2*) `[private]`

segments cross

1.7.2.3 void IntersectionAlgorithm::handle_overlap (IntersectionGraph & *igraph*, const Segment & *s1*, const Segment & *s2*) `[private]`

segments intersect but do not cross

1.7.2.4 void IntersectionAlgorithm::handle_same_endpoint (const Segment & *s1*, const Segment & *s2*) const `[private]`

segments have the same end point

1.7.2.5 bool IntersectionAlgorithm::have_same_endpoint (const Segment & *s1*, const Segment & *s2*) const `[private]`

checks if two segments share an endpoint

Returns

the endpoint

1.7.2.6 void IntersectionAlgorithm::run (IntersectionGraph & *igraph*, SegmentContainer & *segments*)

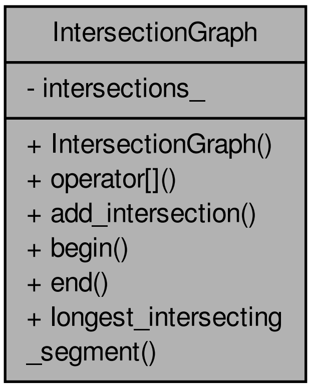
1.7.3 Member Data Documentation

1.7.3.1 SegmentIndex IntersectionAlgorithm::shortest_noncrossing_segment_

1.8 IntersectionGraph Class Reference

```
#include <intersection_graph.h>
```

Collaboration diagram for IntersectionGraph:



Public Member Functions

- IntersectionGraph (const SegmentIndex &size)
- const Intersections & operator[] (const SegmentIndex &index) const
- void add_intersection (const Segment &s1, const Segment &s2)
- IntersectionsVector::const_iterator begin () const
- IntersectionsVector::const_iterator end () const
- const SegmentIndex & longest_intersecting_segment (const SegmentIndex &index) const

Private Attributes

- IntersectionsVector intersections_

1.8.1 Constructor & Destructor Documentation

1.8.1.1 IntersectionGraph::IntersectionGraph (const SegmentIndex & *size*)

default constructor

1.8.2 Member Function Documentation

1.8.2.1 void IntersectionGraph::add_intersection (const Segment & *s1,* const Segment & *s2*)

add two intersecting segments to the graph

1.8.2.2 IntersectionsVector::const_iterator IntersectionGraph::begin ( ) const [inline]

1.8.2.3 IntersectionsVector::const_iterator IntersectionGraph::end ( ) const [inline]

1.8.2.4 const SegmentIndex & IntersectionGraph::longest_intersecting_segment (const SegmentIndex & *index*) const

1.8.2.5 const Intersections& IntersectionGraph::operator[] (const SegmentIndex & *index*) const [inline]

Returns

all intersecting segments for a segment

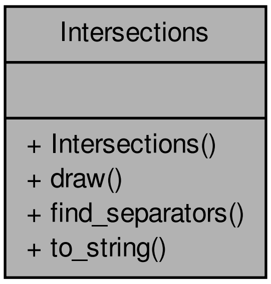
1.8.3 Member Data Documentation

1.8.3.1 IntersectionsVector IntersectionGraph::intersections_ `[private]`

1.9 Intersections Class Reference

`#include <intersections.h>`

Collaboration diagram for Intersections:



Public Member Functions

- Intersections ()
- void draw (QPainter &painter, const SegmentContainer &segments) const
- void find_separators (const SegmentIndex &segment_index, const SegmentContainer &segments, std::vector< SegmentIndex > &separators) const
- QString to_string (const SegmentContainer &segments) const

1.9.1 Detailed Description

sorted set of intersecting segments

1.9.2 Constructor & Destructor Documentation

1.9.2.1 Intersections::Intersections ( ) `[inline]`

default constructor

1.9.3 Member Function Documentation

1.9.3.1 void Intersections::draw (QPainter & *painter,* const SegmentContainer & *segments*) const

draws intersections using QPainter

1.9.3.2 void Intersections::find_separators (const SegmentIndex & *segment_index,* const SegmentContainer & *segments,* std::vector< SegmentIndex > & *separators*) const

finds all separators for a given segment and stores them in the passed container

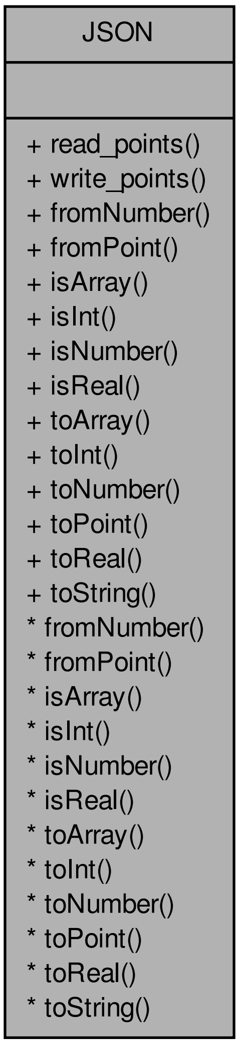
1.9.3.3 QString Intersections::to_string (const SegmentContainer & *segments*) const

output intersections to QString

1.10 JSON Class Reference

```
#include <json.h>
```

Collaboration diagram for JSON:



Static Public Member Functions

- template<typename OutputIterator >
  static bool read_points (QFile &file, OutputIterator output)
- template<typename Container >
  static bool write_points (const std::string &file_name, Container points)

helper functions

- static JSONValue fromNumber (const Number &value)
- static JSONArray fromPoint (const Point &point)
- static bool isArray (const JSONValue &value)
- static bool isInt (const JSONValue &value)
- static bool isNumber (const JSONValue &value)
- static bool isReal (const JSONValue &value)
- static const JSONArray & toArray (const JSONValue &value)
- static int toInt (const JSONValue &value)
- static Number toNumber (const JSONValue &value)
- static Point toPoint (const JSONValue &value)
- static double toReal (const JSONValue &value)
- static const std::string & toString (const JSONValue &value)

1.10.1 Member Function Documentation

1.10.1.1 JSON::JSONValue JSON::fromNumber (const Number & *value*) `[static]`

1.10.1.2 JSON::JSONArray JSON::fromPoint (const Point & *point*) `[static]`

1.10.1.3 bool JSON::isArray (const JSONValue & *value*) `[static]`

1.10.1.4 bool JSON::isInt (const JSONValue & *value*) `[static]`

1.10.1.5 bool JSON::isNumber (const JSONValue & *value*) `[static]`

1.10.1.6 bool JSON::isReal (const JSONValue & *value*) `[static]`

1.10.1.7 template<typename OutputIterator > static bool JSON::read_points (QFile & *file,* OutputIterator *output*) `[inline]`, `[static]`

1.10.1.8 const JSON::JSONArray & JSON::toArray (const JSONValue & *value*) `[static]`

1.10.1.9 int JSON::toInt (const JSONValue & *value*) `[static]`

1.10.1.10 Number JSON::toNumber (const JSONValue & *value*) `[static]`

1.10.1.11 Point JSON::toPoint (const JSONValue & *value*) `[static]`

1.10.1.12 double JSON::toReal (const JSONValue & *value*) `[static]`

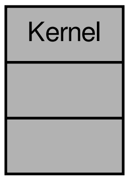
1.10.1.13 const std::string & JSON::toString (const JSONValue & *value*) `[static]`

1.10.1.14 template<typename Container > static bool JSON::write_points (const std::string & *file_name,* Container *points*) `[inline]`, `[static]`

1.11 Kernel Struct Reference

```
#include <kernel.h>
```

Collaboration diagram for Kernel:



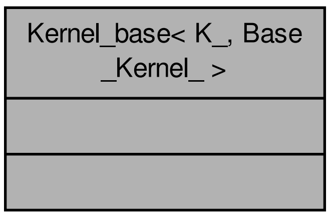
1.11.1 Detailed Description

customized kernel

1.12 Kernel_base$<$ K_, Base_Kernel_ $>$ Class Template Reference

`#include <kernel.h>`

Collaboration diagram for Kernel_base< K_, Base_Kernel_ >:



Classes

- struct Base

1.12.1 Detailed Description

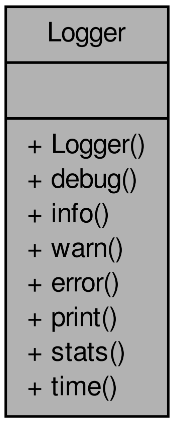
template<typename K_, typename Base_Kernel_>class Kernel_base< K_, Base_Kernel_ >

kernel base with customized PointC2 and SegmentC2

1.13 Logger Class Reference

`#include <logger.h>`

Collaboration diagram for Logger:



Public Member Functions

- Logger ()
- void debug (const QString &message) const
- void info (const QString &message) const
- void warn (const QString &message) const
- void error (const QString &message) const
- void print (const QString &message) const
- void stats (const Stats &stats) const
- void time (const QString &identifier, int milliseconds) const

1.13.1 Constructor & Destructor Documentation

1.13.1.1 Logger::Logger ( )

1.13.2 Member Function Documentation

1.13.2.1 void Logger::debug (const QString & *message*) const

1.13.2.2 void Logger::error (const QString & *message*) const

1.13.2.3 void Logger::info (const QString & *message*) const

1.13.2.4 void Logger::print (const QString & *message*) const

1.13.2.5 void Logger::stats (const Stats & *stats*) const

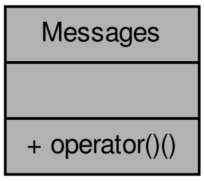
1.13.2.6 void Logger::time (const QString & *identifier,* int *milliseconds*) const

1.13.2.7 void Logger::warn (const QString & *message*) const

1.14 Messages Class Reference

```
#include <logger.h>
```

Collaboration diagram for Messages:



Public Member Functions

- QString operator() (const char ∗text)

1.14.1 Detailed Description

helper class for string literals

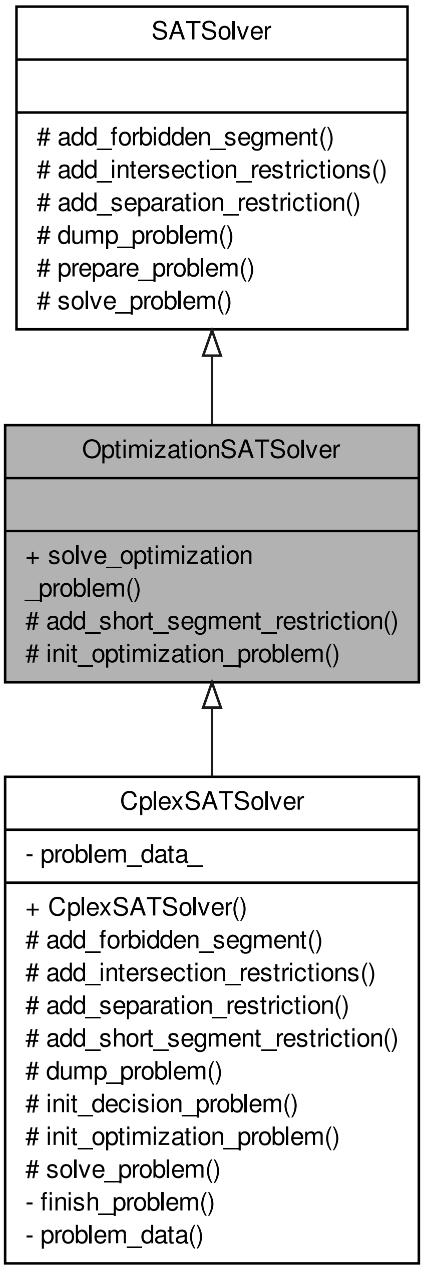
1.14.2 Member Function Documentation

1.14.2.1 QString Messages::operator() (const char ∗ *text*) [inline]

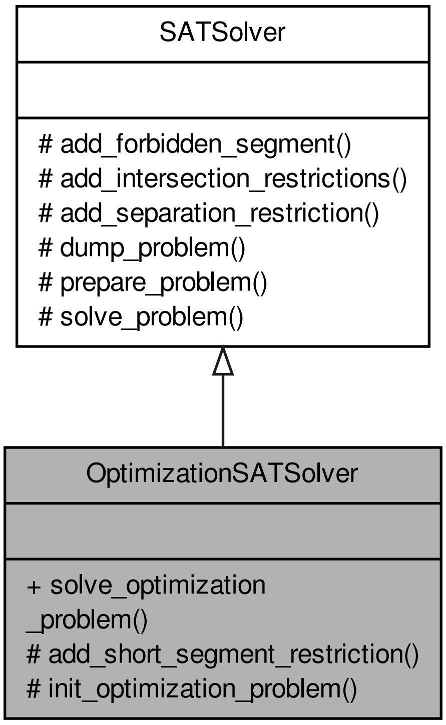
1.15 OptimizationSATSolver Class Reference

```
#include <sat_solver.h>
```

Inheritance diagram for OptimizationSATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
OptimizationSATSolver
+ solve_optimization _problem()
# add_short_segment_restriction()
# init_optimization_problem()
CplexSATSolver
- problem_data_
+ CplexSATSolver()
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# add_short_segment_restriction()
# dump_problem()
# init_decision_problem()
# init_optimization_problem()
# solve_problem()
- finish_problem()
- problem_data()

Collaboration diagram for OptimizationSATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
OptimizationSATSolver
+ solve_optimization _problem()
# add_short_segment_restriction()
# init_optimization_problem()

Public Member Functions

- void solve_optimization_problem (const QSettings &settings, const QString &file_prefix, const SATProblem &problem, SATSolution &solution)

Protected Member Functions

- virtual void add_short_segment_restriction (const SATProblem ∗problem, const SegmentIndex &index)=0
- virtual void init_optimization_problem (const SATProblem ∗problem)=0

1.15.1 Detailed Description

interface for optimization SAT solvers

1.15.2 Member Function Documentation

1.15.2.1 virtual void OptimizationSATSolver::add_short_segment_restriction (const SATProblem ∗ *problem,* const SegmentIndex & *index*) `[protected], [pure virtual]`

Implemented in CplexSATSolver.

1.15.2.2 virtual void OptimizationSATSolver::init_optimization_problem (const SATProblem * *problem*) [protected], [pure virtual]

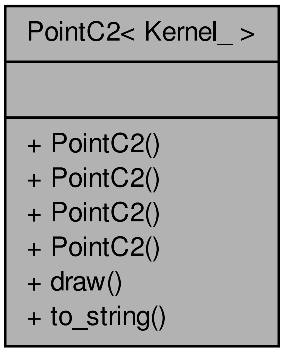
Implemented in CplexSATSolver.

1.15.2.3 void OptimizationSATSolver::solve_optimization_problem (const QSettings & *settings*, const QString & *file_prefix*, const SATProblem & *problem*, SATSolution & *solution*)

1.16 PointC2< Kernel_ > Class Template Reference

```
#include <point.h>
```

Collaboration diagram for PointC2< Kernel_ >:



Public Member Functions

- PointC2 ()
- PointC2 (const CGAL::Origin &origin)
- PointC2 (const FT &x, const FT &y)
- PointC2 (const FT &hx, const FT &hy, const FT &hw)
- void draw (QPainter &painter) const
- QString to_string () const

Private Types

- typedef Kernel_::FT FT
- typedef CGAL::PointC2< Kernel_ > PointBase

1.16.1 Detailed Description

template<class Kernel_>class PointC2< Kernel_ >

customized point type

1.16.2 Member Typedef Documentation

1.16.2.1 template<class Kernel_ > typedef Kernel_::FT **PointC2**< Kernel_ >::**FT** `[private]`

1.16.2.2 template<class Kernel_ > typedef CGAL::PointC2<Kernel_> **PointC2**< Kernel_ >::**PointBase** `[private]`

1.16.3 Constructor & Destructor Documentation

1.16.3.1 template<class Kernel_ > **PointC2**< Kernel_ >::PointC2 ( ) `[inline]`

empty constructor

1.16.3.2 template<class Kernel_ > **PointC2**< Kernel_ >::PointC2 (const CGAL::Origin & *origin*) `[inline]`

origin constructor

1.16.3.3 template<class Kernel_ > **PointC2**< Kernel_ >::PointC2 (const **FT** & *x*, const **FT** & *y*) `[inline]`

Cartesian constructor

1.16.3.4 template<class Kernel_ > **PointC2**< Kernel_ >::PointC2 (const **FT** & *hx*, const **FT** & *hy*, const **FT** & *hw*) `[inline]`

homogeneous constructor

1.16.4 Member Function Documentation

1.16.4.1 template<class Kernel_ > void **PointC2**< Kernel_ >::draw (QPainter & *painter*) const

draw segment using given QPainter

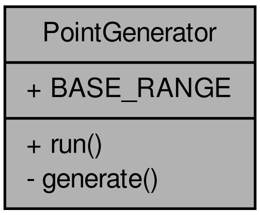
1.16.4.2 template<class Kernel_ > QString **PointC2**< Kernel_ >::to_string ( ) const

dump point to QString

1.17 PointGenerator Class Reference

`#include <point_generator.h>`

Collaboration diagram for PointGenerator:



Static Public Member Functions

- static void run (const QSettings &settings)

Static Public Attributes

- static const double BASE_RANGE = 100.0

Static Private Member Functions

- template<typename GeneratorType >
  static void generate (const QString &base_name, std::size_t num_points, std::size_t num_iterations)

1.17.1 Member Function Documentation

1.17.1.1 template<typename GeneratorType > static void PointGenerator::generate (const QString & *base_name,* std::size_t *num_points,* std::size_t *num_iterations*) `[inline], [static], [private]`

1.17.1.2 static void PointGenerator::run (const QSettings & *settings*) `[inline], [static]`

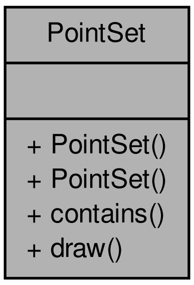
1.17.2 Member Data Documentation

1.17.2.1 const double PointGenerator::BASE_RANGE = 100.0 `[static]`

1.18 PointSet Class Reference

`#include <point_set.h>`

Collaboration diagram for PointSet:



Public Member Functions

- PointSet ()
- PointSet (QFile &input_file)
- bool contains (const Point &point) const
- void draw (QPainter &painter) const

1.18.1 Detailed Description

(sorted) set of points

1.18.2 Constructor & Destructor Documentation

1.18.2.1 PointSet::PointSet ( )

empty set

1.18.2.2 PointSet::PointSet (QFile & *input_file*)

read points from file

1.18.3 Member Function Documentation

1.18.3.1 bool PointSet::contains (const Point & *point*) const `[inline]`

shortcut for STL count()

Returns

true if point is in set

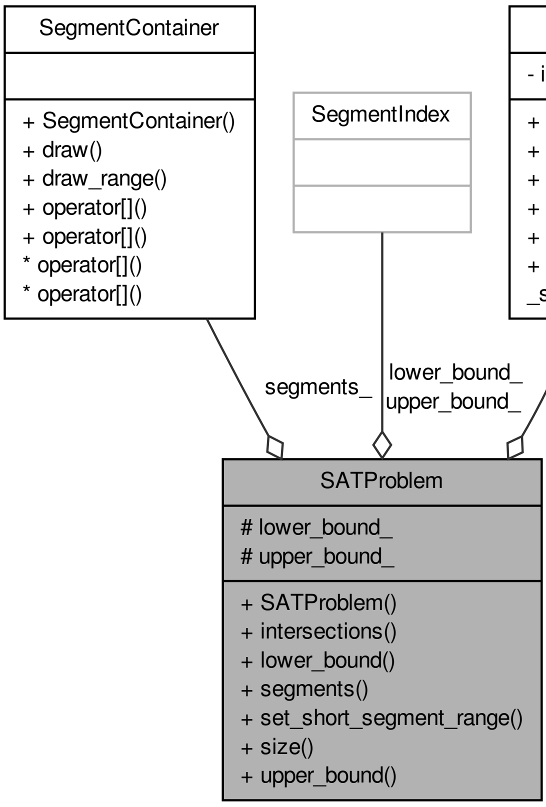
1.18.3.2 void PointSet::draw (QPainter & *painter*) const

output point set using QPainter

1.19 SATProblem Class Reference

```
#include <sat_problem.h>
```

Collaboration diagram for SATProblem:



Public Member Functions

- SATProblem (const IntersectionGraph &igraph, const SegmentContainer &segments)
- const Intersections & intersections (const SegmentIndex &index) const
- const SegmentIndex & lower_bound () const
- const SegmentContainer & segments () const
- void set_short_segment_range (const SegmentIndex &lower_bound, const SegmentIndex &upper_bound)
- SegmentIndex size () const
- const SegmentIndex & upper_bound () const

Protected Attributes

- const IntersectionGraph & igraph_
- const SegmentContainer & segments_
- SegmentIndex lower_bound_
- SegmentIndex upper_bound_

1.19.1 Constructor & Destructor Documentation

1.19.1.1 SATProblem::SATProblem (const IntersectionGraph & *igraph*, const SegmentContainer & *segments*)

default constructor

1.19.2 Member Function Documentation

1.19.2.1 const Intersections& SATProblem::intersections (const SegmentIndex & *index*) const [inline]

1.19.2.2 const SegmentIndex& SATProblem::lower_bound ( ) const [inline]

1.19.2.3 const SegmentContainer& SATProblem::segments ( ) const [inline]

1.19.2.4 void SATProblem::set_short_segment_range (const SegmentIndex & *lower_bound,* const SegmentIndex & *upper_bound*)

set range of segments to consider

1.19.2.5 SegmentIndex SATProblem::size ( ) const [inline]

1.19.2.6 const SegmentIndex& SATProblem::upper_bound ( ) const [inline]

1.19.3 Member Data Documentation

1.19.3.1 const IntersectionGraph& SATProblem::igraph_ [protected]

1.19.3.2 SegmentIndex SATProblem::lower_bound_ [protected]

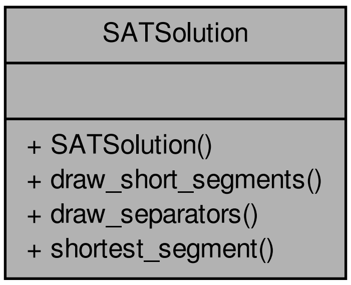
1.19.3.3 const SegmentContainer& SATProblem::segments_ [protected]

1.19.3.4 SegmentIndex SATProblem::upper_bound_ [protected]

1.20 SATSolution Class Reference

`#include <sat_solution.h>`

Collaboration diagram for SATSolution:



Public Member Functions

- SATSolution ()
- void draw_short_segments (QPainter &painter, const SegmentIndex &num_short_segments, const Segment-Container &segments) const
- void draw_separators (QPainter &painter, const SegmentIndex &num_short_segments, const Segment-Container &segments) const
- const SegmentIndex & shortest_segment () const

1.20.1 Constructor & Destructor Documentation

1.20.1.1 SATSolution::SATSolution ( ) [inline]

1.20.2 Member Function Documentation

1.20.2.1 void SATSolution::draw_separators (QPainter & *painter*, const SegmentIndex & *num_short_segments*, const SegmentContainer & *segments*) const

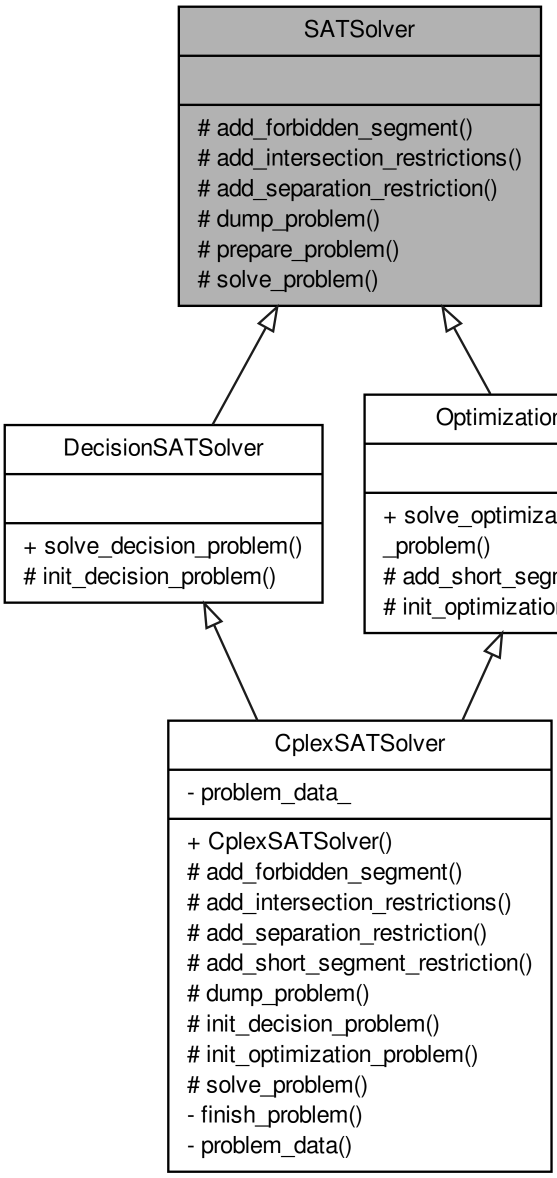
1.20.2.2 void SATSolution::draw_short_segments (QPainter & *painter*, const SegmentIndex & *num_short_segments*, const SegmentContainer & *segments*) const

1.20.2.3 const SegmentIndex & SATSolution::shortest_segment ( ) const

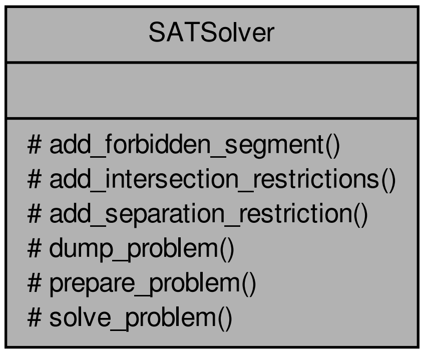
1.21 SATSolver Class Reference

```
#include <sat_solver.h>
```

Inheritance diagram for SATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()
DecisionSATSolver
+ solve_decision_problem()
# init_decision_problem()
OptimizationSATSolver
+ solve_optimization _problem()
# add_short_segment_restriction()
# init_optimization_problem()
CplexSATSolver
- problem_data_
+ CplexSATSolver()
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# add_short_segment_restriction()
# dump_problem()
# init_decision_problem()
# init_optimization_problem()
# solve_problem()
- finish_problem()
- problem_data()

Collaboration diagram for SATSolver:


SATSolver
# add_forbidden_segment()
# add_intersection_restrictions()
# add_separation_restriction()
# dump_problem()
# prepare_problem()
# solve_problem()

Protected Member Functions

- virtual void add_forbidden_segment (const SATProblem ∗problem, const SegmentIndex &index)=0
- virtual void add_intersection_restrictions (const SATProblem ∗problem, const SegmentIndex &index, const Intersections &igroup)=0
- virtual void add_separation_restriction (const SATProblem ∗problem, const SegmentIndex &index, const std::vector< SegmentIndex > &separators)=0
- virtual void dump_problem (const QString &file_prefix, const SATProblem ∗problem)=0
- void prepare_problem (const SATProblem &problem)
- virtual void solve_problem (const SATProblem ∗problem, SATSolution &solution)=0

1.21.1 Detailed Description

interface for SAT solvers

1.21.2 Member Function Documentation

1.21.2.1 virtual void SATSolver::add_forbidden_segment (const SATProblem ∗ *problem,* const SegmentIndex & *index*) [protected], [pure virtual]

Implemented in CplexSATSolver.

1.21.2.2 virtual void SATSolver::add_intersection_restrictions (const SATProblem ∗ *problem,* const SegmentIndex & *index,* const Intersections & *igroup*) [protected], [pure virtual]

Implemented in CplexSATSolver.

1.21.2.3 virtual void SATSolver::add_separation_restriction (const SATProblem ∗ *problem,* const SegmentIndex & *index,* const std::vector< SegmentIndex > & *separators*) [protected], [pure virtual]

Implemented in CplexSATSolver.

1.21.2.4 virtual void SATSolver::dump_problem (const QString & *file_prefix,* const SATProblem ∗ *problem*) [protected], [pure virtual]

Implemented in CplexSATSolver.

1.21.2.5 void SATSolver::prepare_problem (const **SATProblem** & *problem*) [protected]

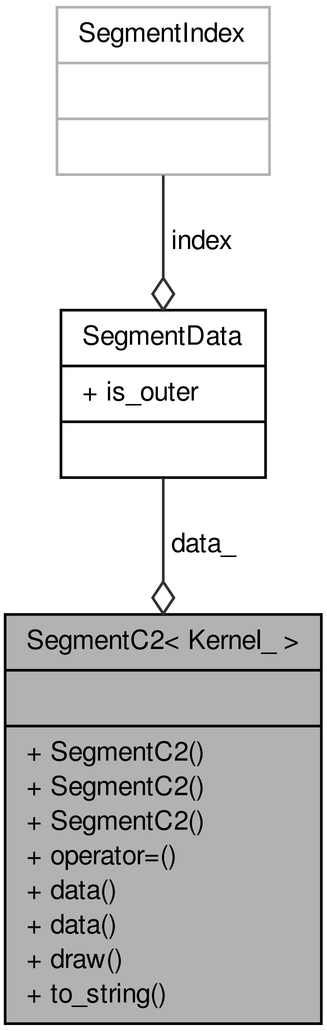
1.21.2.6 virtual void SATSolver::solve_problem (const **SATProblem** * *problem*, **SATSolution** & *solution*) [protected], [pure virtual]

Implemented in CplexSATSolver.

1.22 SegmentC2< Kernel_ > Class Template Reference

```
#include <segment.h>
```

Collaboration diagram for SegmentC2< Kernel_ >:



Public Member Functions

- SegmentC2 ()
- SegmentC2 (const Point_2 &source, const Point_2 &target)

- SegmentC2 (const SegmentC2 &other)
- SegmentC2 & operator= (const SegmentC2 &other)
- SegmentData & data ()
- const SegmentData & data () const
- void draw (QPainter &painter) const
- QString to_string () const

Private Attributes

- SegmentData data_

1.22.1 Detailed Description

template<class Kernel_>class SegmentC2< Kernel_ >

customized segment type

1.22.2 Constructor & Destructor Documentation

1.22.2.1 template<class Kernel_ > SegmentC2< Kernel_ >::SegmentC2 ( ) [inline]

empty constructor

1.22.2.2 template<class Kernel_ > SegmentC2< Kernel_ >::SegmentC2 (const Point_2 & *source*, const Point_2 & *target*) [inline]

base constructor

1.22.2.3 template<class Kernel_ > SegmentC2< Kernel_ >::SegmentC2 (const SegmentC2< Kernel_ > & *other*) [inline]

copy constructor

1.22.3 Member Function Documentation

1.22.3.1 template<class Kernel_ > SegmentData& SegmentC2< Kernel_ >::data ( ) [inline]

getter for attached data

1.22.3.2 template<class Kernel_ > const SegmentData& SegmentC2< Kernel_ >::data ( ) const [inline]

constant getter for attached data

1.22.3.3 template<class Kernel_ > void SegmentC2< Kernel_ >::draw (QPainter & *painter*) const

draw segment using given QPainter

1.22.3.4 template<class Kernel_ > SegmentC2& SegmentC2< Kernel_ >::operator= (const SegmentC2< Kernel_ > & *other*) [inline]

assignment operator

1.22.3.5 template<class Kernel_ > QString SegmentC2< Kernel_ >::to_string ( ) const

dump segment to QString

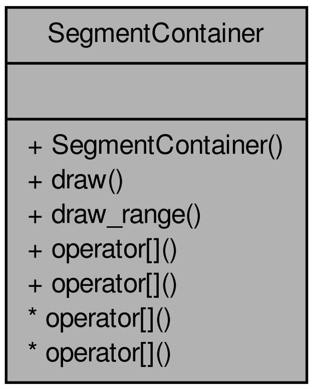
1.22.4 Member Data Documentation

1.22.4.1 template<class Kernel_ > SegmentData SegmentC2< Kernel_ >::data_ [private]

1.23 SegmentContainer Class Reference

`#include <segment_container.h>`

Collaboration diagram for SegmentContainer:



Public Member Functions

- SegmentContainer (const PointSet &points)
- void draw (QPainter &painter) const
- void draw_range (QPainter &painter, const SegmentIndex &lower_bound, const SegmentIndex &upper_-bound) const

access i-th shortest segment

these operators assume that the segment set is not changed after construction

- Segment & operator[] (const SegmentIndex &index)
- const Segment & operator[] (const SegmentIndex &index) const

1.23.1 Detailed Description

container of segments sorted by length

1.23.2 Constructor & Destructor Documentation

1.23.2.1 SegmentContainer::SegmentContainer (const PointSet & *points*)

construct segments for all point pairs from set

1.23.3 Member Function Documentation

1.23.3.1 void SegmentContainer::draw (QPainter & *painter*) const

draws all segments

1.23.3.2 void SegmentContainer::draw_range (QPainter & *painter,* const SegmentIndex & *lower_bound,* const SegmentIndex & *upper_bound*) const

draw a range of segments

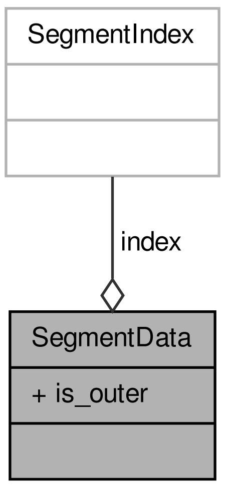
1.23.3.3 Segment & SegmentContainer::operator[] (const SegmentIndex & *index*)

1.23.3.4 const Segment & SegmentContainer::operator[] (const SegmentIndex & *index*) const

1.24 SegmentData Struct Reference

`#include <segment.h>`

Collaboration diagram for SegmentData:



Public Attributes

- SegmentIndex index
- bool is_outer

1.24.1 Detailed Description

data attached to a segment

1.24.2 Member Data Documentation

1.24.2.1 SegmentIndex SegmentData::index

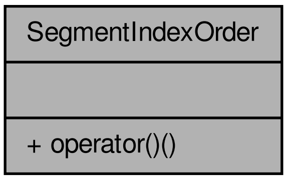
1.24.2.2 bool SegmentData::is_outer

true if the segment includes another

1.25 SegmentIndexOrder Struct Reference

`#include <orders.h>`

Collaboration diagram for SegmentIndexOrder:



Public Member Functions

- CGAL::Comparison_result operator() (const Segment &s, const Segment &t) const

1.25.1 Detailed Description

CGAL order for Segment by index

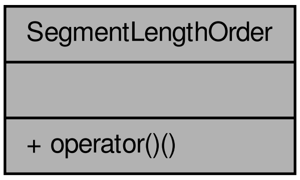
1.25.2 Member Function Documentation

1.25.2.1 CGAL::Comparison_result SegmentIndexOrder::operator() (const Segment & *s*, const Segment & *t*) const

1.26 SegmentLengthOrder Struct Reference

`#include <orders.h>`

Collaboration diagram for SegmentLengthOrder:



Public Member Functions

- CGAL::Comparison_result operator() (const Segment &s, const Segment &t) const

1.26.1 Detailed Description

CGAL order for Segment by length

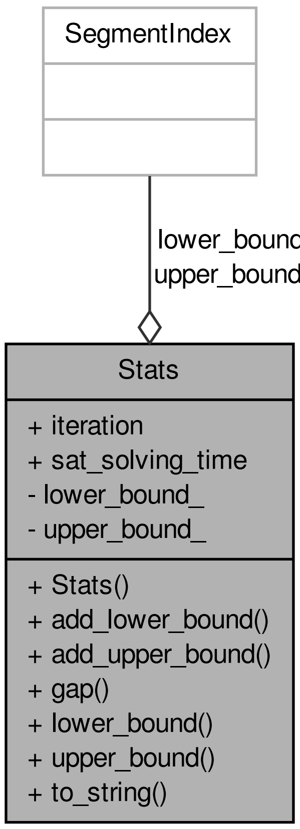
1.26.2 Member Function Documentation

1.26.2.1 CGAL::Comparison_result SegmentLengthOrder::operator() (const Segment & *s*, const Segment & *t*) const

1.27 Stats Class Reference

```
#include <stats.h>
```

Collaboration diagram for Stats:



Public Member Functions

- Stats ()
- void add_lower_bound (const SegmentIndex &bound)

- void add_upper_bound (const SegmentIndex &bound)
- SegmentIndex gap () const
- const SegmentIndex & lower_bound () const
- const SegmentIndex & upper_bound () const
- QString to_string () const

Public Attributes

- size_t iteration
- quint64 sat_solving_time

Private Attributes

- SegmentIndex lower_bound_
- SegmentIndex upper_bound_

1.27.1 Constructor & Destructor Documentation

1.27.1.1 Stats::Stats ( ) `[inline]`

1.27.2 Member Function Documentation

1.27.2.1 void Stats::add_lower_bound (const SegmentIndex & *bound*) `[inline]`

1.27.2.2 void Stats::add_upper_bound (const SegmentIndex & *bound*) `[inline]`

1.27.2.3 SegmentIndex Stats::gap ( ) const `[inline]`

1.27.2.4 const SegmentIndex& Stats::lower_bound ( ) const `[inline]`

1.27.2.5 QString Stats::to_string ( ) const `[inline]`

1.27.2.6 const SegmentIndex& Stats::upper_bound ( ) const `[inline]`

1.27.3 Member Data Documentation

1.27.3.1 size_t Stats::iteration

1.27.3.2 SegmentIndex Stats::lower_bound_ `[private]`

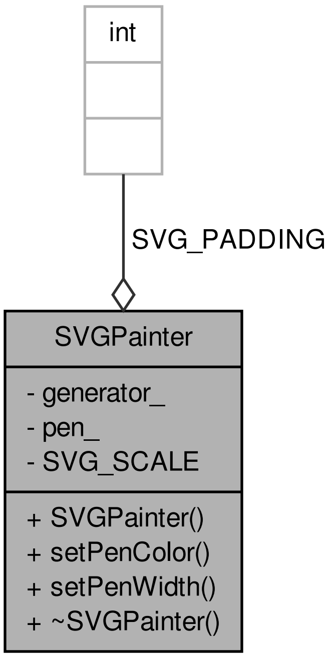
1.27.3.3 quint64 Stats::sat_solving_time

1.27.3.4 SegmentIndex Stats::upper_bound_ `[private]`

1.28 SVGPainter Class Reference

`#include <svg_painter.h>`

Collaboration diagram for SVGPainter:



Public Member Functions

- SVGPainter (const QString &file_prefix, const QString &file_name, const BoundingBox &bbox)
- void setPenColor (const QColor &color)
- void setPenWidth (int width)
- ∼SVGPainter ()

Private Attributes

- QSvgGenerator generator_
- QPen pen_

Static Private Attributes

- static const int SVG_PADDING = 10
- static const double SVG_SCALE = 4.0

1.28.1 Constructor & Destructor Documentation

1.28.1.1 SVGPainter::SVGPainter (const QString & *file_prefix,* const QString & *file_name,* const BoundingBox & *bbox*)

1.28.1.2 SVGPainter::∼SVGPainter ( )

1.28.2 Member Function Documentation

1.28.2.1 void SVGPainter::setPenColor (const QColor & *color*)

1.28.2.2 void SVGPainter::setPenWidth (int *width*)

1.28.3 Member Data Documentation

1.28.3.1 QSvgGenerator SVGPainter::generator_ [private]

1.28.3.2 QPen SVGPainter::pen_ [private]

1.28.3.3 const int SVGPainter::SVG_PADDING = 10 [static], [private]

padding for SVG images

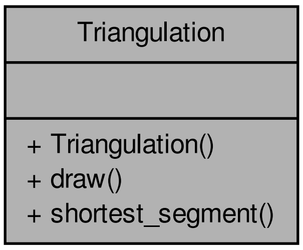
1.28.3.4 const double SVGPainter::SVG_SCALE = 4.0 [static], [private]

scale for SVG images

1.29 Triangulation Class Reference

`#include <triangulation.h>`

Collaboration diagram for Triangulation:



Public Member Functions

- Triangulation (const PointSet &points)
- void draw (QPainter &painter) const
- const SegmentIndex & shortest_segment (const SegmentContainer &segments) const

1.29.1 Constructor & Destructor Documentation

1.29.1.1 Triangulation::Triangulation (const PointSet & *points*)

default constructor

1.29.2 Member Function Documentation

1.29.2.1 void Triangulation::draw (QPainter & *painter*) const

draw triangulation segments using given QPainter

1.29.2.2 const SegmentIndex & Triangulation::shortest_segment (const SegmentContainer & *segments*) const

B. Result Data

replace

Table B.1.: Composition of execution times

replace

Table B.2.

replace

Table B.3.

Glossary

Glossary

e_{nose}	shortest non-separable edge
e_{sep}	separating edge
$s_{\times}$	crossing segment
CDS	Covering by Disjoint Segments
CGAL	Computational Geometry Algorithms Library
IP	Integer Program
MELT	MaxMin Edge Length Index Triangulation
MMLT	MaxMin Length Triangulation
NOCS	Non-Conflicting Separators

Properties

crossing	see definition 3.12
non-crossing	see definition 3.12
non-overlapping	see definition 3.13
overlapping	see definition 3.13
well-covered	see definition 2.14

Sets

E_{flip}	set of edge flips (see definition 3.27)
E_{sep}	set of separators (see definition 4.14)
E_{short}	set of short edges (see definition 4.13)
G_{conf}	conflict graph (see definition 3.3)
G_{cross}	intersection graph (see definition 3.19)
G_{flip}	flip graph (see definition 3.29)
S_{opt}	an optimal set of segments
T_{opt}	an optimal triangulation
V_{IS}	independent set (see definition 2.16)
V_{cover}	vertex cover (see definition 2.9)

Bibliography

[1] Selim G. Akl and Godfried T. Toussaint. „A Fast Convex Hull Algorithm“. In: *Information Processing Letters* 7.5 (1978), pp. 219–222.

[2] Mark de Berg et al. *Computational Geometry: Algorithms and Applications.* Third. Springer-Verlag, 2008, p. 386. URL: `http://www.cs.uu.nl/geobook/`.

[3] Marshall W. Bern et al. „Edge Insertion for Optimal Triangulations.“ In: *Discrete & Computational Geometry* 10 (1993), pp. 47–65. DOI: `10.1007/BF02573962`.

[4] Prosenjit Bose and Ferran Hurtado. „Flips in planar graphs“. In: *Computational Geometry: Theory and Applications* 42 (1 2009), pp. 60–80. DOI: `10.1016/j.comgeo.2008.04.001`.

[5] *CGAL - Computational Geometry Algorithms Library.* CGAL Open Source Project. 2013. URL: `https://www.cgal.org/` (visited on 2013-06-19).

[6] L. P. Chew. „Constrained Delaunay triangulations“. In: *Proceedings of the third annual symposium on Computational geometry.* SCG '87. ACM, 1987, pp. 215–222. ISBN: 0-89791-231-4. DOI: `10.1145/41958.41981`. URL: `http://doi.acm.org/10.1145/41958.41981`.

[7] Christiane Schmidt. „Maxmin Length Triangulation in Polygons“. In: *Proceedings of the 28th European Workshop on Computational Geometry.* 2012, pp. 121–124.

[8] *Concert Technology for C++ users.* Version 12.2. IBM Corporation. URL: `http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r2/topic/ilog.odms.cplex.help/Content/Optimization/Documentation/CPLEX/_pubskel/CPLEX197.html` (visited on 2013-06-25).

[9] *Constrained_triangulation_2<Traits,Tds,Itag>.* Version 4.2. CGAL Open Source Project. 2013-04-09. URL: `https://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Triangulation_2_ref/Class_Constrained_triangulation_2.html` (visited on 2013-06-19).

[10] *convex_hull_2.* Version 4.2. CGAL Open Source Project. 2013-04-09. URL: `https://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Convex_hull_2_ref/Function_convex_hull_2.html` (visited on 2013-06-19).

[11] Stephen A. Cook. „The complexity of theorem-proving procedures“. In: *Proceedings of the third annual ACM symposium on Theory of computing.* STOC '71. ACM, 1971, pp. 151–158. DOI: `10.1145/800157.805047`.

[12] *Creator_uniform_2<Arg, Result>*. Version 4.2. CGAL Open Source Project. 2013-04-09. URL: `http://www.cgal.org/Manual/4.2/doc_html/cgal_manual/STL_Extension_ref/FunctionObjectClass_Creator_uniform_2.html` (visited on 2013-07-22).

[13] Beman Dawes, David Abrahams, and Rene Rivera. *Boost C++ Libraries*. URL: `https://www.boost.org/` (visited on 2013-07-21).

[14] Jesús A. De Loera, Jörg Rambau, and Francisco Santos. *Triangulations. Structures for Algorithms and Applications*. Vol. 25. Algorithms and Computation in Mathematics. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-12970-4. DOI: `10.1007/978-3-642-12971-1`.

[15] E. Demaine et al. *Problem 28: Flip Graph Connectivity in 3D*. 2008-12-22. URL: `http://cs.smith.edu/~orourke/TOPP/P28.html` (visited on 2013-07-17).

[16] *do_intersect*. Version 4.2. CGAL Open Source Project. 2013-04-09. URL: `https://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Kernel_23_ref/Function_do_intersect.html` (visited on 2013-07-21).

[17] *Doxygen*. 2013-05-20. URL: `http://doxygen.org/` (visited on 2013-07-25).

[18] Herbert Edelsbrunner and Tiow Seng Tan. „A Quadratic Time Algorithm for the MinMax Length Triangulation". In: *SIAM J. Comput* 22 (1991), pp. 414–423.

[19] *Exact_predicates_inexact_constructions_kernel*. Version 4.2. CGAL Open Source Project. 2013-04-09. URL: `https://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Kernel_23_ref/Class_Exact_predicates_inexact_constructions_kernel.html` (visited on 2013-06-19).

[20] Sándor P. Fekete. „The Complexity of MaxMin Length Triangulation". In: *CoRR* abs/1208.0202 (2012).

[21] Steven Fortune. „A sweepline algorithm for Voronoi diagrams". In: *Algorithmica* 2.1-4 (1987), pp. 153–174. ISSN: 0178-4617. DOI: `10.1007/BF01840357`.

[22] Barend Gehrels et al. *Geometry*. 2013-06-25. URL: `https://www.boost.org/doc/libs/1_54_0/libs/graph/` (visited on 2013-07-21).

[23] Peter Greenberg. „Piecewise SL_2Z Geometry". In: *Transactions of the American Mathematical Society* 335.2 (1993), pp. 705–720. DOI: `10.2307/215440`.

[24] Joel de Guzman and Hartmut Kaiser. *Spirit*. Version 2.5.2. 2013-06-25. URL: `https://www.boost.org/doc/libs/1_54_0/libs/spirit/` (visited on 2013-07-21).

[25] David Hilbert. *Grundlagen der Geometrie*. 2nd ed. B. G. Teubner, 1903, p. 39.

[26] Shiyan Hu. „A linear time algorithm for max-min length triangulation of a convex polygon". In: *Inf. Process. Lett.* 101.5 (2007-03), pp. 203–208. ISSN: 0020-0190. DOI: `10.1016/j.ipl.2006.09.014`. URL: `http://dx.doi.org/10.1016/j.ipl.2006.09.014`.

[27] *IBM CPLEX Optimizer - United States*. Version 17. IBM Corporation. 2013-05-09. URL: https://www.ibm.com/software/commerce/optimization/cplex-optimizer/ (visited on 2013-06-25).

[28] *intersection*. Version 4.2. CGAL Open Source Project. 2013-04-09. URL: https://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Kernel_23_ref/Function_intersection.html (visited on 2013-07-21).

[29] *JSON*. 2013-07-02. URL: http://www.json.org/ (visited on 2013-07-21).

[30] Richard M. Karp. „Reducibility Among Combinatorial Problems“. In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, p. 94. ISBN: 0-306-30707-3.

[31] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 5th ed. Springer Publishing Company, Incorporated, 2012, p. 414. ISBN: 9783642244872.

[32] David Lichtenstein. „Planar Formulae and Their Uses“. In: *SIAM Journal on Computing* 11.2 (1982), pp. 329–343. DOI: 10.1137/0211025.

[33] Errol Lynn Lloyd. „On triangulations of a set of points in the plane“. In: *18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 228–240. DOI: 10.1109/SFCS.1977.21. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4567947.

[34] Sue Wheeler Lloyd. *Greg Likes Triangles for Lunch*. Authorhouse, 2011. ISBN: 978-1452098838.

[35] David Lonie. *QJson iterators broken – missing operator->*. 2013-02-08. URL: https://bugreports.qt-project.org/browse/QTBUG-29573 (visited on 2013-07-21).

[36] László Lovász et al. „Convex quadrilaterals and k-sets“. In: *Contemporary Mathematics Series, 342, AMS 2004*. 2004, pp. 139–148.

[37] H. Mittelmann. *Mixed Integer Linear Programming Benchmark (MIPLIB2010)*. 2013-05-25. URL: http://plato.asu.edu/ftp/milpc.html (visited on 2013-06-25).

[38] Wolfgang Mulzer and Günter Rote. „Minimum-weight triangulation is NP-hard“. In: *CoRR* abs/cs/0601002 (2006).

[39] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications Inc., 2000. ISBN: 978-0486402581.

[40] Michael D. Plummer. „Some covering concepts in graphs“. In: *Journal of Combinatorial Theory* 8.1 (1970), pp. 91–98. ISSN: 0021-9800. DOI: 10.1016/S0021-9800(70)80011-4. URL: http://www.sciencedirect.com/science/article/pii/S0021980070800114.

[41] Vladimir Prus. *Boost.Program_ options*. 2007-11-25. URL: https://www.boost.org/doc/libs/1_54_0/libs/program_options/ (visited on 2013-07-21).

[42] *QPainter Class*. Version 5.1. URL: https://qt-project.org/doc/qt-5.1/qtgui/qpainter.html (visited on 2013-07-22).

[43] *QString Class*. Version 5.1. URL: https://qt-project.org/doc/qt-5.1/qtcore/qstring.html (visited on 2013-07-22).

[44] *Qt Project*. URL: https://qt-project.org/ (visited on 2013-07-21).

[45] *Random_points_in_square_2<Point_2, Creator>*. Version 4.2. CGAL Open Source Project. 2013-04-09. URL: http://www.cgal.org/Manual/4.2/doc_html/cgal_manual/Generator_ref/Class_Random_points_in_square_2.html (visited on 2013-07-22).

[46] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. 2012-09-30. URL: https://www.boost.org/doc/libs/1_54_0/libs/graph/ (visited on 2013-07-21).

[47] Tzvetalin Simeonov Vassilev. „Optimal Area Triangulation“. PhD thesis. University of Saskatchewan, Saskatoon, 2005. URL: http://ecommons.usask.ca/bitstream/handle/10388/etd-08232005-111957/thesisFF.pdf.

[48] Klaus Wagner. „Bemerkungen zum Vierfarbenproblem“. In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 46 (1936), pp. 26–32.

[49] John W. Wilkinson. *JSON Spirit: A C++ JSON Parser/Generator Implemented with Boost Spirit*. 2013-05-23. URL: http://www.codeproject.com/Articles/20027/JSON-Spirit-A-C-JSON-Parser-Generator-Implemented (visited on 2013-07-21).

Liste der noch zu erledigenden Punkte