Robust Geometric Computing and Optimal Visibility Coverage

Eli Packer
Preface

This book presents work in two of the most popular areas of geometric computation: Robust Geometric Computing and Optimal Visibility Coverage. Both areas have been extensively researched in recent years and attracted the attention of many researchers and developers in both the academic and the industrial worlds. The work we present here aims to contribute to this effort by providing efficient tools that can be used to solve important problems and challenges that arise in the areas specified above. We focus on both the theoretic and practical aspects of the problems we solve; we theoretically justify their efficiency and correctness and give experiential evidence, using applications that were developed for that purpose, to show their practicality and usefulness.

Robust Geometric Computing

Implementation of geometric algorithms is generally difficult because one must deal with both precision problems and degenerate input. While these issues are usually ignored when describing geometric algorithms in theory, overlooking them in practice often result in program crashes and incorrect results. Chapters 1, 2 and 3 are devoted to three algorithms that approximate arrangements of line segments in order to make them more robust for further manipulation and computation. We describe the algorithms in detail, prove important properties that they hold and present extensive experimental results obtained with their implementation.

In Chapter 1 we present an algorithm, Iterated Snap Rounding [52], to convert an arrangement of line segments into a more robust representation for algorithms that further manipulate the arrangement. More specifically, the algorithm converts each line segment into a polygonal chain of segments such that the vertices of the new arrangement have integer coordinates. The main goal of the algorithm is to provide an efficient alternative to the well known Snap Rounding algorithm: While in a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding, Iterated Snap Rounding rounds the arrangement such that each vertex is at least half-the-width-of-a-pixel away from any non-incident edge. By doing so, Iterated Snap Rounding produces more robust results that are safer to further manipulate.

We note that in Iterated Snap Rounding the deviation of the output arrangement from the input arrangement may be huge. In Chapter 2 we present another Snap Rounding variant, Iterated Snap Rounding with Bounded Drift [73, 75, 76], that comes to improve the output quality of Iterated Snap Rounding in which the deviation of the output arrangement from the input arrangement may be huge. Iterated Snap Rounding with Bounded Drift guarantees bounded deviation specified as a input parameter.

Chapter 3 describes another algorithm for converting arrangements of line segments into a more
robust representation using the technique of Controlled Perturbation [72]. While the goal of this algorithm is very similar to that of Snap Rounding, it produces a completely different output that can be more useful than the output produced by Snap Rounding in different cases. In the core of this work we present a novel technique to decrease the perturbation magnitude.

**Optimal Visibility Coverage**

The optimal visibility coverage problem, often called the "art gallery problem", is one of the classic geometric problems. The input for the problem is a geometric domain and the challenge is to minimize the number of guards that cover, or see, the domain. This topic is motivated by problems in surveillance, sensor networking, illumination and more. In the previous decades, numerous versions of this problem have been proposed, analyzed and solved. Since many variants are NP-hard, most research has been focused on heuristics and approximation methods.

In Chapters 4 and 5, we present techniques for optimizing visibility coverage. Given a polygonal domain, we address two different coverage problems in which the goal is to see all of the interior of the domain using stationary or mobile point guards. The first coverage problem (Chapter 5) is known as the classic *Art Gallery* problem; it deals with stationary guards, and the goal is to minimize the number of guards in a set that completely covers (sees) the domain. The second coverage problem (Chapter 6), known as the *Mobile Watchmen* problem, allows guards to move along predefined routes that we are to determine.

Chapter 5 gives an experimental analysis of algorithms we developed for the *Art Gallery* problem [8]. Given a polygon in the plane, the objective is to find a minimum-cardinality set of point guards inside the polygon that cover (or “see”) the entire polygon. This problem is known to be NP-hard; only limited results are known about efficient approximation algorithms. We design and implement several efficient heuristics for the problem; additionally, in order to compute lower bounds on an optimal solution, we implement heuristics for finding a set of visibility-independent “witness points”. Our experiments show that the *Art Gallery* problem can, in practice, be solved to optimality or near-optimality, with guarantees provided by the upper and lower bounds given by the heuristics.

Finally, Chapter 6 extends the work of Chapter 5 to support mobile guards [71]. In this model, the number of mobile guards are given and the goal is to minimize the lengths of their routes. We consider two measures of solution quality: the length of the longest route, and the sum of the lengths of the routes. We show that these problems are hard to approximate. We develop heuristics for optimizing the route lengths, describe their implementation, and report an extensive set of experimental results to evaluate them. We conclude that in many cases the results obtained are close to optimal.
Acknowledgements

I am grateful to my Master thesis adviser, Prof. Dan Halperin and to my PhD adviser, Prof. J.S.B Mitchell. Spending many hours together over the years, and collaborating with them towards research publications, were crucial for my ability to conduct and promote research efforts and for the success of the work that I present in this book.

I also thank Prof. Esther Arkin, Prof. Michael Bender, Prof. Alon Efrat, Dr. Esther Ezra, Prof. Jie Gao, Dr. Valentin Polishchuk, Prof. Steve Skiena and Dr. Avishay Traeger for useful and fruitful collaboration.

Finally, I deeply thank my parents Michael and Ziva, my sister Orna, my brothers Avi and Mickey, my wife Merav and my children Ronel, Sherel and Ariel, for being there for me over the years.
Contents

I Robust Geometric Computing

1 Iterated Snap Rounding

1.1 Introduction .......................................................... 13
1.2 The distance between a vertex and a non-incident Edge ......................... 15
1.3 Iterated Snap Rounding ........................................... 16
1.4 Algorithmic Details and Complexity analysis .................................. 19
1.5 c-Oriented kd-Trees ............................................. 21
1.6 Rounding Examples: SR vs. ISR .................................... 23
   1.6.1 Congestion Data ........................................ 24
   1.6.2 Triangulation Data .................................... 24
   1.6.3 Geographic Data ....................................... 25
1.7 Conclusions ...................................................... 25

2 Iterated Snap Rounding with Bounded Drift

2.1 Introduction ....................................................... 29
   2.1.1 Our contribution ................................... 30
2.2 Preliminaries and Key Ideas .................................... 30
2.3 Algorithm ......................................................... 33
   2.3.1 Heating New Hot Pixels ................................. 34
   2.3.2 Removing Redundant Degree-2 Vertices ............... 36
   2.3.3 Illustrative Example ................................ 37
2.4 Implementation Details and Complexity Analysis .......................... 38
   2.4.1 Output Complexity .................................. 38
   2.4.2 Implementation Issues ................................. 43
   2.4.3 Analysis ............................................... 45
   2.4.4 Determining the Maximum Deviation Parameter ............ 46
2.5 Desirable Properties ........................................... 46
### 2.6 Experimental Results
- 2.6.1 Randomized Input
- 2.6.2 Visibility Graph of a Polygon
- 2.6.3 Experimental Conclusions

### 2.7 Extending ISR and ISRBD
- 2.7.1 Dynamic Rounding
- 2.7.2 Line Segments in $\mathbb{R}^3$

### 2.8 Conclusions and Future Work

### 3 Controlled Perturbation of Sets of Line Segments in $\mathbb{R}^2$ with Smart Processing Order

#### 3.1 Introduction

#### 3.2 Main Ideas and Algorithm

#### 3.3 Computing the Perturbation Radius
- 3.3.1 Computing Forbidden Loci induced by Endpoints
- 3.3.2 Computing Forbidden Locus Induced by $v_i$ and a Non-incident Line Segment
- 3.3.3 Computing Forbidden Loci Induced by Intersections of Segments and Line Segments incident to $v_i$
- 3.3.4 Computing Forbidden Loci Induced by $V_{i-1}'$ and Segments incident to $v_i$
- 3.3.5 A Lower Bound on Another Kind of Distance Between an Intersection Point and a Line Segment
- 3.3.6 Computing $\delta$

#### 3.4 Ordering the Endpoint Processing
- 3.4.1 Minimizing the Sum
- 3.4.2 Minimizing the Maximum

#### 3.5 Optimization Techniques
- 3.5.1 Partition into Grid Cells
- 3.5.2 Decreasing the Perturbation magnitude

#### 3.6 Comparison with Alternative Schemes
- 3.6.1 Snap Rounding
- 3.6.2 Exact Arithmetic
- 3.6.3 Advantages of CPLS

#### 3.7 Experiments

#### 3.8 The MIN-MAX Problem With Negative Weights

#### 3.9 Conclusions and Future Work
5.4 Experiments ................................................................. 128
5.5 Conclusions and future work ......................................... 131
Part I

Robust Geometric Computing
Chapter 1

Iterated Snap Rounding

Snap Rounding is a well known method for converting arbitrary-precision arrangements of segments into a fixed-precision representation. We point out that in a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. We propose and analyze an augmented procedure, *Iterated Snap Rounding*, which rounds the arrangement such that each vertex is at least half-the-width-of-a-pixel away from any non-incident edge. Iterated Snap Rounding preserves the topology of the original arrangement in the same sense that the original scheme does. We describe an implementation in which we substitute an intricate data structure for segment/pixel intersection that is used to obtain good worst-case resource bounds for Iterated Snap Rounding by a simple and effective data structure which is a cluster of kd-trees. Finally, we present rounding examples obtained with the implementation.

1.1 Introduction

Geometric algorithms are typically described in the infinite-precision “real RAM” model of computation and under the assumption of general position, namely that the input is degeneracy-free. These assumptions raise great difficulties in implementing robust geometric algorithms. A variety of techniques have been proposed in recent years to overcome these difficulties [87, 81].

One approach to robust computing produces a finite-precision approximation of the geometric objects in question; for a survey of finite-precision approximation algorithms, see e.g. [79]. Snap Rounding is a method of this type for converting an arrangement of segments into a low-precision representation.

Given a finite collection $S$ of segments in the plane, the arrangement of $S$, denoted $A(S)$, is the subdivision of the plane into vertices, edges, and faces induced by $S$. A vertex of the arrangement is either a segment endpoint or the intersection of two segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, Snap Rounding (SR, for short) proceeds as follows [47, 56]. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge $e$ is replaced by the polygonal chain through the centers of the hot pixels met by $e$, in the same order as they are met by $e$. See figure 1.1 for an illustration.
In the process, vertices and edges of the original arrangement may have collapsed. However, the rounded arrangement preserves certain topological properties of the original arrangement: The rounding can be viewed as a continuous process of deforming curves (the original segments into chains) such that no vertex of the arrangement ever crosses through a curve [48]. The rounded version \( s' \) of an original segment \( s \) approximates \( s \) such that \( s' \) lies within the Minkowski sum of \( s \) and a pixel centered at the origin.

SR makes the vertices of the arrangement well separated. We would expect that in the rounded arrangement a vertex \( v \) and an edge \( e \) not incident to \( v \) will also be well separated, namely, that the minimum separation between a vertex and a non-incident edge will be at roughly the same scale as the minimum separation between vertices. However, as we show in the next section, this is not the case and the distance between a vertex and non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding.

We propose an augmented procedure, Iterated Snap Rounding, ISR for short, which rounds the arrangement such that each vertex is at least half a unit away from any non-incident edge. ISR preserves the topology of the original arrangement in the same sense as the original scheme does. However, the guaranteed quality of the approximation degrades and the chain may be further away from the segment it approximates than the corresponding chain produced by SR. Thus each scheme may be suitable in different settings. We also show that the maximum combinatorial complexity, namely, the maximum overall number of vertices in all the chains as well as the maximum complexity of the rounded arrangements, is the same for SR and ISR.

We present a conceptually simple algorithm for computing ISR (as well as SR), Whose only non-trivial component is a data structure to answer segment intersection queries on a given collection of (hot) pixels. To provide asymptotically good worst-case resource bounds we use multi-level partition trees for this structure. In our implementation, however, we use a simple alternative which we call c-oriented kd-trees. We present below rounding results obtained with our implementation of SR and ISR.

Throughout the paper we use the following notation and terminology. The input \( S \) consists of \( n \) line segments \( s_1, \ldots, s_n \). The rounding schemes transform each input segment into polygonal chain. We call each straight line segment of an output chain between two hot pixels’ centers a link. The output of SR for an input segment \( s \) is denoted by \( s' \) and the output of ISR for \( s \) is denoted by \( s^* \).
Figure 1.2: A vertex becomes very close to a non-incident edge after snap rounding. (a) Input (b) Snap Rounding output

**Related Work.** Greene and Yao [46] proposed a rounding scheme that preceded Snap Rounding. Several algorithms were proposed to implement Snap Rounding [38, 45, 48, 55, 56]. They were mainly devoted to improve asymptotic time complexity of the process, to present on-line algorithms and to extend Snap Rounding to 3D. Recently, de Berg et al.[29] introduced an improved output-sensitive algorithm for Snap Rounding and an output optimization which is used with changes in our work. Eigenwillig et al.[36] extended Snap Rounding to Bezier curves.

The rest of the chapter is organized as follows. In the next section we show that in SR a vertex and a non-incident edge of the rounded arrangement can be very close to one another. In section 1.3 we describe the augmented procedure ISR, prove its main properties and outline an algorithm for computing it. In section 1.4 we fill in the algorithmic details of our algorithm and analyze its complexity. Section 1.5 is devoted to the implementation of the algorithm using c-oriented kd-trees. Rounding examples are given in Section 5.4. We conclude in Section 5.5 by pointing out possible directions for future work.

### 1.2 The distance between a vertex and a non-incidence Edge

Consider the two segments $s$, $t$ displayed in Figure 1.2 before and after SR. We denote the right endpoint of $s'$ by $s'_r$. (Recall that $u'$ is the rounded version of $u$.) After rounding, $t'$ penetrates the hot pixel containing $s'_r$, but it does not pass through its center.

We can modify the input segment $t$ so that $t'$ become very close to $s'_r$: we move the left endpoint of $t$ arbitrarily close to the top right corner of the pixel containing it. We vertically translate the right endpoint of $t$ far downwards—the farther down we translate it, the closest it will be to $s'_r$.

We cannot make $t'$ arbitrarily close to $s'_r$. If they are not incident then there is lower bound on the distance between them. The distance, however, can be rather small. Let $b$ denote the number of bits in the representation of the vertex coordinates of the output chains of SR. We tile a bounding square of the arrangement with $2^b \times 2^b$ unit pixels. In this setting the distance between $t'$ and $s'_r$ can be made as small as $1 / \sqrt{(2^b - 1)^2 + 1} \approx 2^{-b}$.

One could argue that although SR produces near-degenerate output, it is still possible, during the rounding process, to determine the correct topology of the rounded arrangement in the hot pixel.
containing $s'_r$. However, this requires that the output of SR should include additional information beyond the simple listing of polygonal chains specified by their rounded vertices, making it more cumbersome to use and further manipulate.

### 1.3 Iterated Snap Rounding

We augment SR to eliminate the near-degeneracies mentioned above. Our procedure, which we call *Iterated Snap Rounding* (ISR, for short), produces a rounded arrangement where an original segment is substituted by a polygonal chain each vertex of which is at least 1/2 a unit distant from any non-incident edge.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be the collection of input segments whose arrangement we wish to round. Recall that a pixel is hot if and only if it contains a vertex of the input arrangement. Let $H$ denote the set of hot pixels included by the arrangement induced by $S$ (we denote the arrangement by $A(S)$).

Our goal is to create chains out of the input segments such that a chain that passes through a hot pixel is re-routed to pass through the pixel’s center. The difficulty is that once we reroute a chain it may have entered other hot pixels and we need to further reroute it, and so on.

Our rounding algorithm consists of two stages. In a preprocessing stage we compute the hot pixels (by finding all the vertices of the arrangement) and prepare a segment intersection search structure $D$ on the hot pixels to answer queries of the following type: Given a segment $s$, report the hot pixels that $s$ intersects. In the second stage we operate a recursive procedure, **Reroute**, on each input segment. We postpone the algorithmic details of the preprocessing stage to the next sections and concentrate here on the rerouting stage.

**Reroute** is a "depth-first" procedure. As we show below, the rerouting that we propose does not add more hot pixels, so whenever we refer to the set of hot pixels we mean $H$. The input to **Reroute** is a segment $s \in \mathcal{S}$. The output is a polygonal chain $s^*$ which approximates $s$. Whenever $s^*$ passes through a hot pixel, it passes through it center. See Figure 1.3 for an illustration.

We next describe the ISR algorithm. The routine **Reroute** will produce an output chain $s^*$ in
Figure 1.4: The tree $T_1$ corresponding to **RE_ROUTE** of Figure 1.3. Nodes denoted by full-line circles contain segments with which we query the structure $D$. The dashed-line circle denotes a node containing an exact copy of the segment of its parent.

the global parameter **OUTPUT_CHAIN** as an ordered list of links. If a segment is contained inside a single pixel, the chain degenerates to a single point.

**Iterated Snap Rounding**

**Input:** a set $S$ of $n$ segments  
**Output:** a set $S^*$ of $n$ polygonal chains; initially $S^* = \emptyset$

/* stage 1: preprocessing */
1. Compute the set $H$ of hot pixels  
2. Construct a segment intersection search structure $D$ on $H$

/* stage 2: rerouting */
3. for each input segment $s \in S$
4. initialize **OUTPUT_CHAIN** to be empty  
5. **RE_ROUTE**($s$)  
6. add **OUTPUT_CHAIN** to $S^*$

**RE_ROUTE**

/* $s$ is the input segment with endpoints $p$ and $q$ */
1. query $D$ to find $H_s$, the set of hot pixels intersected by $s$
2. if $H_s$ contains a single hot pixel /* $s$ is entirely inside a pixel */
3. then add the center of the hot pixel containing $s$ to **OUTPUT_CHAIN**
4. else
5. let $m_1, m_2, \ldots, m_r$ be the centers of the $r$ hot pixels in $H_s$ in the order of the intersection along $s$
6. if ($r = 2$ and $p, q$ are centers of pixels)
7. then add the link $m_1m_2$ to **OUTPUT_CHAIN**
8. else
9. for $i = 1$ to $r - 1$
10. **RE_ROUTE**(m$_i$m$_{i+1}$)
We next discuss the properties of the procedure.

We fix an orientation for each input segment and its induced chains: it is oriented in lexicographically increasing order of its vertices. Thus, a non-vertical segment for example is oriented from its left endpoint to its right endpoint. (the orientation of a chain is well defined since, as is easily verified, a chain is (weakly) x-monotone and (weakly) y-monotone.) We represent the operation of REROUTE on a segment \( s_i \), as a tree \( T_i \). The root contains \( s_i \). The leaves of the tree contain the output polygonal chain \( s_i^* \), one link in each node, ordered from left to right where the first link is in the leftmost leaf. Each internal node \( \nu \) together with its children represent one application of REROUTE (without recurrence): the segment \( s_\nu \), which passes through the hot pixels with centers \( m_1, m_2, \ldots, m_r \), is transformed into the links \( m_q m_{q+1}, q = 1, \ldots, r - 1 \) which are placed in the children of \( \nu \) ordered from left to right to preserve the orientation of the chain. We denote all the segments in the nodes at the \( j \)’th level from left to right by \( s_j, s_{j+1}, \ldots, s_{j+l_{i,j}} \), where \( l_{i,j} \) denotes the number of nodes in this level. We denote the chain consisting of all the links at level \( j \) ordered from left to right by \( s_j^* \). Thus \( s_0^* = s_i \). We denote by \( k_i \) the depth of the tree for \( s_i \) and let \( k := \max_{i=1,\ldots,n} k_i \). For notational convenience, if a leaf \( \lambda \) is at level \( k_\lambda < k \) then we add a linear path of \( k_i - k \) artificial nodes descending from \( \lambda \) and all containing the same link that \( \lambda \) contains (we denote it differently at any level according to the level). See Figure 1.4 for an illustration of the tree \( T_1 \) corresponding to segment \( s_1 \) of Figure 1.3. We denote by \( s(\nu) \) the segment (or link) that is contained in the node \( \nu \).

The next lemma gives an alternative view of ISR.

**Lemma 1.1.** Given a set of segments \( S \), the output of ISR is equivalent to the final output of a finite series of applications of SR starting with \( S \), where the output of one SR is the input to the next SR.

**Proof.** Once we determine the hot pixels \( H \), snap rounding an input segment \( s \) (i.e, by the standard SR) can be done independently of the other segments. That is, the information necessary for rounding is in \( H \). Notice that the chains \( s_j^*, i = 1, \ldots, n \) are the result of applying SR to the original input segments \( S \).

The crucial observation is that SR does not create new hot pixels. It can break a segment into two segments that meet at the center of an existing hot pixel, but it cannot create a new endpoint nor a new intersection point (with another segment) which are not at the center of an existing hot pixel—this would violate the topology preservation properties of SR [48].

It follows that with the same set \( H \) of hot pixels, the chains \( s_{i+1}^*, i = 1, \ldots, n \) are the result of applying SR to the links in the chains \( s_j^*, i = 1, \ldots, n \) and so on.

The process terminates when the link in each leaf of the tree has its endpoints in the center of hot pixels and it does not cross any other hot pixel besides the hot pixels that contain its endpoints.

The tree continues to grow beyond level \( j \) only as long as at least one node \( \nu \) in level \( j \) when we query with \( s(\nu) \) we discover a new hot pixel through which \( s(\nu) \) passes. We claim that a hot pixel is not discovered more than once per tree. This is so since, as already mentioned, each chain \( s_j^* \) is (weakly) x-monotone and (weakly) y-monotone. Since there are at most \( O(n^2) \) hot pixels, the process will stop after finite number of steps.

The lemma’s algorithmic interpretation is inefficient, but it is useful for proving some of the following properties.
**Corollary 1.2.** ISR preserves the topology of the arrangement of the input segments in the same sense that SR does.

*Proof.* The topological properties that are preserved by SR can be summarized by viewing SR as a continuous process of deforming curves (the original segments into chains) such that no vertex of the arrangement ever crosses through a curve [48]. Since SR does not create new vertices, the assertion follows from Lemma 1.1. □

**Lemma 1.3.** (i) If an output chain of ISR passes through a hot pixel then it passes through its center. (ii) In the output chains each vertex is at least $1/2$ unit away from any non-incident segment.

*Proof.* Claim (i) follows from the definition of the procedure *Reroute*. Since all the vertices of the rounded arrangement are centers of hot pixels, claim (ii) is an immediate consequence of (i). □

A drawback of ISR is that an output chain $s_i^*$ can be farther away from the original segment $s_i$ compared with the chain produced for the same input segment by SR. Recall that $k_i$ denotes the depth of the recursion of *Reroute*($s_i$).

**Lemma 1.4.** A final chain $s^*$ lies in the Minkowski sum of $s_i$ and a square of side size $k_i$ centered at the origin.

*Proof.* In SR a rounded segment $s'$ lies inside the Minkowski sum of the input segment $s$ and a unit square centered at the origin. Since the ISR is equivalent to $k_i$ applications of SR, the claim follows. □

This deviation may be acceptable in situations where the pixel size is sufficiently small or when $k := \max_{i=1,...,n}k_i$ is small.

### 1.4 Algorithmic Details and Complexity analysis

Let $I$ denote the number of intersection points of segments in the original arrangement $\mathcal{A}(S)$. We first compute the set $H$ of hot pixels. For that we use an algorithm for segment intersection. This could be done with a plane sweep algorithm, or more efficiently in $O(I + n \log n)$ time by more involved algorithms [13, 22]. To compute the hot pixels, the algorithm should also be given a pixel’s width $w$ and a point $p$ that will be assigned the coordinates $(0, 0)$. The plane will be tiled with pixels that we will consider to be of unit width, and their centers will have integer coordinates. We denote the number of hot pixels by $N$. Notice that $N$ is at most $O(n + I)$.

**Remark.** One could alternatively detect the hot pixels by the SR algorithm of Goodrich et al. [45] and thus get rid of the dependence of the running time of the algorithm on the number of intersections $I$. Notice however that for this step alone (namely for detecting the hot pixels) and for certain inputs (e.g., the input depicted in Figure 1.5 and described below) this alternative is costly.

Next we prepare the data structure $D$ on the hot pixels $H$ to answer segment intersection queries. We construct a multi-level partition tree [4] on the vertical boundary segments of the hot pixels, and an analogous tree for the horizontal boundary segments. The partition trees report the segments intersected by a query segment $s$ from which we deduce the hot pixels intersected by $s$. Each tree
requires $O(M^{1+\varepsilon})$ preprocessing time when allowed $M$ units of storage for $N \leq M \leq N^2$. A query takes $O(N^{1+\varepsilon}/\sqrt{M} + g)$ time, where $g$ is the number of hot pixels found [4].

How many times do we query the structure $D$ for segment intersection?

**Lemma 1.5.** If an output chain $s_i^*$ consists of $l_i$ links then during REROUTE($s_i$) the structure $D$ is queried at most $2l_i$ times.

**Proof.** During REROUTE($s$) when we query with a link (line 1 of REROUTE) either we do not find new hot pixels (new for the rounded version of $s_i$) in which case we charge the query to the link which is then a link of the final chain, or we charge it to the first new hot pixel (recall that we assigned an orientation to each segment and to each link). Each final link is charged exactly once and each vertex of the final chain is charged at most once, besides the last vertex is never charged. The bound follows.

Let $L$ denote the overall number of links in all the chains output by ISR. We summarize the performance bounds of ISR in the following theorem.

**Theorem 1.6.** Given an arrangement of $n$ segments with $I$ intersection points, the ISR requires $O(n \log n + I + L^{2/3}N^{2/3+\varepsilon} + L)$ time for any $\varepsilon > 0$ and $O(n + N + L^{2/3}N^{2/3+\varepsilon})$ working storage, where $N$ is the number of hot pixels (which is at most $2n + I$) and $L$ is the overall number of links in the chains produced by the algorithm.

**Proof.** To find the intersection of the input segments we use Balaban’s algorithm which requires $O(n \log n + I)$ time and $O(n)$ working storage. When an intersection is found we simply keep its corresponding hot pixel. For constructing and querying the multi-level partition trees (by Lemma 1.5 we perform at most $2L$ queries overall) we use a standard trick that balances between the preprocessing time and the overall query time, and does not require that we know the number of queries in advance. See e.g., [30].

We conclude this section with combinatorial bounds on the maximum complexity of the rounded arrangements. Interestingly, as shown next, there is no difference between the maximum asymptotic complexity of the rounded arrangements between SR and ISR.

**Theorem 1.7.** Given an arrangement of $n$ segments in the plane, in its rounded version: (i) the maximum number of hot pixels through which a single output chain passes is $\theta(n^2)$, and (ii) the maximum overall number of incidences between output chains and hot pixels is $\theta(n^3)$. (iii) The number of segments in the rounded arrangement (namely without counting multiplicities) is $\theta(n^2)$, and if the input segments induce $N$ hot pixels then this number is $\theta(N)$. All these bounds apply both to SR and to ISR.

**Proof.** The upper bounds in claims (i) and (ii) are obvious. To see that these bounds are tight consider the following construction (see figure 1.5). We take $n/2$ long horizontal segments spanning row of $n^2/4$ pixels. Next we take $n/2$ short, slightly slanted segments, each spanning $n/2$ pixels such that overall each pixel in the row is intersected by exactly one short segment. The short segments are slanted such that in each pixel that they cross they intersect exactly one of the long segments. Each pixel in the row is now a hot pixel, and each of the long segments crosses all the hot pixels. The rounding obtained with both SR and ISR is the same.
The construction yields a degenerate rounded arrangement. Each of the output chains is in fact a horizontal line segment. This construction can be slanted so that each rounded version of a long segment is a chain with "true" $\Omega(n^2)$ links. In the slanted version we use $n^2/2$ pixels arranged in $n^4/4$ rows. In each row at least one pixel is hot. See figure 1.6 for an illustration.

Finally, we ignore the chains, and ask how complex can the rounded arrangement be, that is, we ignore multiplicities (overlap) of chains. Obviously, the rounded arrangement can have $\Omega(n^2)$ complexity. But this is also an upper bound since the (rounded) arrangement has $N$ vertices and it is planar graph. Therefore the number of edges can be at most $O(N)$. $N$ can be at most $O(n^2)$. Again, our arguments do not depend on how the rounding was done (by SR or ISR).

1.5 \textit{c-Oriented kd-Trees}

In our implementation we use a plane sweep algorithm to find the intersections between segments in $\mathcal{S}$ and thus we identify the hot pixels. The non-trivial part to implement is the search structure $D$ with which we answer segment/pixel intersection queries. In the theoretical analysis we use partition trees for $D$, as these lead to asymptotically good worst-case complexity. In practice, (multi-level) partition tress are difficult to implement. Instead, we implemented a data structure consisting of several kd-trees. Next we explain the details.

\textbf{Observation 1.8.} A segment $s$ intersects a pixel $p$ of width $w$, if and only if the Minkowski sum of $s$ with a pixel of width $w$ centered at the origin contains the center of $p$.

We could use observation 1.8 in order to answer segment intersection queries in the following way: build a range search structure on the centers of the hot pixels. Let $s$ be the query segment and $M(s)$ be its Minkowski sum with pixel centered at the origin. Then query the structure with range $M(s)$. Unfortunately, the known data structures for this type of queries are similar to the multi-level data structures that we have used in the previous section.
Figure 1.7: The bounding box of the Minkowski sum of a segment with a pixel size centered at the origin. The shaded area is the redundant range.

Instead we use kd-trees as an approximation of this scheme. A kd-tree answers range queries for axis-parallel rectangles [31]. Its guaranteed worst-case query time is far from optimal but it is practically efficient. A trivial solution would be to query with the axis-parallel bounding box of $M(s)$, which we denote by $B(s)$; see Figure 1.7. This may not be sufficiently satisfactory since the area of $B(s)$, which we denote by $|B(s)|$, may be much larger than the area of $M(s)$.

If we rotate the plane together with $M(s)$ the (area of the) axis-parallel bounding box changes whereas $M(s)$ remains fixed. The difference between the bounding boxes for the two different rotations can be huge. Our goal is to produce a number of rotated copies of the set of centers of hot pixels so that for each query segment $s$ there will be one rotation for which the area of the bounding box is not too much different from the area of $M(s)$. Notice that if a segment $s$ is rotated by $\pi/2$ radians, the size of the relevant bounding box remains the same. Since the determination of which rotation to choose is dependant only on the size of the respective bounding box, the range of rotations should be the half-open interval $[0 : \pi/2)$.

We construct a collection of kd-trees each serving as a range search structure for a rotated copy of the centers of hot pixels. We call this cluster $c$-oriented kd-trees. Let $c$ be a positive integer and let $\alpha_i := (i - 1)\frac{\pi}{2c}$ for $1 \leq i \leq c$. The structure consists of $c$ kd-trees such that the $i$-th kd-tree, denoted by $kd_i$, has the input points rotated by $\alpha_i$. Let $R_i(s)$ be the segment $s$ rotated by $\alpha_i$. For each query with segment $s$ we do the following: for each $kd_i$, $1 \leq i \leq c$, we compute $|B(R_i(s))|$. Let $1 \leq h \leq c$ be the serial number of the kd-tree for which $|B(R_h(s))| = \min_{i=1,\ldots,c}|B(R_i(s))|$. Then we use the $h$-th kd-tree to answer the query with the segment $s$ rotated by $\alpha_h$. Finally, we discard all the points for which the segment does not intersect the respective hot pixels.

We next discuss a few important issues regarding the implementation and usage of this structure.

**Exact rotations.** We used exact arithmetic to implement ISR. Unfortunately, the available exact arithmetic data-types do not support the calculations of sines and cosines which are necessary for calculating rotations. Instead we use only angles for which the sines and cosines can be expressed as rational numbers with small enumerator and denominator [18]. We keep an array $Z$ of approximations to the sines of integer degree angles between $0 - 89$. We emphasize that once we fix an angle $\beta$ we have the exact sine and cosine of $\beta$. What we cannot do is obtain the exact values of the trigonometric functions of a prescribed arbitrary angle. Since our choice of rotation angles is heuristic to begin with, the precise angle is immaterial, and the angle we use is never more than one degree off the prescribed angle. Moreover, there are techniques to achieve better approximation
How big should $c$ be? There are advantages and drawbacks in using few kd-trees, say even one kd-tree compared to using many. When using one kd-tree, we are prone to get many false points in the range queries, resulting in more time to filter out the results. When using many kd-trees, we need to invest time in their construction and a little more time per query to find the best rotation. Our experiments show that in many cases a small number of trees suffices. Consider for example the numerical table "different number of kd-trees" in Figure 1.8. (the rounding example in this figure as well as the other examples are explained in detail in the next section; here we only refer to the number of kd-trees used in their computation.) The first column shows how many kd-trees were used and the last column shows how much time the overall rerouting stage took compared with the time when using only one kd-tree (the full legend is given in Table 4.2). The best performance is obtained when we use 7 kd-trees. The time savings in this case is 17% over using a single kd-tree. The analogous table in the next example (Figure 1.9) shows that in that example there is no benefit in using more than one kd-tree. In the next paragraph we present a heuristic improvement of the number of kd-trees. However, we leave the computation of the best number of kd-trees together with the best rotation angles of each one for further research.

Skipping $kd_i$’s. Since $c$ should be small, we expect most of the links of a certain input segment to have the same rotation as the input segment, since they should all have nearby slopes. Let $J_i$ be the number of input segments that are rotated by $\alpha_i$. If $J_i$ is very small, it is not effective to create the respective kd-tree. Thus we fix a lower limit $\tau$, and construct a kd-tree $kd_i$ only if $J_i \geq \tau$. Obviously $\tau$ should be a function of $c$, and be sufficiently small to ensure that at least one kd-tree will be constructed. We chose to use $\tau = \frac{n}{2c}$. In the examples of Figures 1.8 and 1.9, $\tau$ is always greater than $\frac{n}{2c}$. In other examples, such as the one in Figure 1.10, not all $c$ trees are always constructed—in this example, when the algorithm is given $c > 7$ it chooses to skip some of the $kd_i$’s. Also in this example, using more than one kd-tree is wasteful: since the map is relatively sparse, most of the segments are relatively small compared to the whole map and the bounding box of their Minkowski sum with a unit pixel does not intersect many hot pixel centers.

1.6 Rounding Examples: SR vs. ISR

To give the flavor of how the output of ISR differs from that of SR we present the rounding results for three input examples; see Figures 1.8, 1.9 and 1.10. For each example we display the input, the SR result and the ISR result. Then we zoom in on a specific area of interest in these three drawings—an area where the rounding schemes differ noticeably. A square near a drawing represents the actual pixel size used for rounding. Then we provide two tables of statistics. The first one refers to the best number of kd-trees as related to the discussion in the previous section. The second table summarizes the differences in the rounding for different pixel sizes. The abbreviations we use in these two tables are explained in Table 4.2. The deviation of a chain from its inducing segment $s$ is the maximal distance of a point on the chain from $s$.

The running time indicated in the tables is in seconds while using arbitrary precision rational arithmetic. The pixel size in the first example is 1 and in the second example is 15.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inkd</td>
<td>input number of kd-trees</td>
</tr>
<tr>
<td>nkd</td>
<td>actual number of kd-trees created</td>
</tr>
<tr>
<td>nfhp</td>
<td>overall number of false hot pixels in all the queries</td>
</tr>
<tr>
<td>tt</td>
<td>total time relative to using one tree</td>
</tr>
<tr>
<td>md</td>
<td>maximum deviation over all chains</td>
</tr>
<tr>
<td>ad</td>
<td>average deviation</td>
</tr>
<tr>
<td>mnv</td>
<td>maximum number of vertices in an output chain</td>
</tr>
<tr>
<td>anv</td>
<td>average number of vertices in an output chain</td>
</tr>
<tr>
<td>mdvs</td>
<td>minimum distance between a vertex and a non-incident edge</td>
</tr>
<tr>
<td>ncv</td>
<td>number of pairs of a vertex and a non-incident edge that are less than half the width of a pixel apart</td>
</tr>
<tr>
<td>ps</td>
<td>pixel size</td>
</tr>
<tr>
<td>nhp</td>
<td>number of hot pixels</td>
</tr>
</tbody>
</table>

Table 1.1: Abbreviations

1.6.1 Congestion Data

The data contains 200 segments with 18,674 intersections. (for clarity, the pictures in Figure 1.8 depict a similar example with only 100 segments.) The bottom left part of the arrangement is zoomed in.

Both rounding schemes will collapse thin triangles that have two corners close by. However, not allowing proximity between vertices and non-incident edges, ISR collapses ‘skinny’ faces of the arrangement that SR does not (see the bottom of the zoomed-in area), for example triangles that have one corner close to the middle of the opposite edge.

For pixel size 1, SR and ISR are very different and the number of vertices that are less than half a unit away from a non-uniform edge in the SR output is in the hundreds. The average deviation in ISR in this example is never more than 2.5 times that of the corresponding SR output. For pixel size greater than 1 the average deviation of a chain in ISR is almost the same as in SR. However, for pixel size smaller than 1, the average deviation is larger in the ISR output than in the SR output.

In terms of combinatorial complexity the results are similar and the average number of vertices per chain is roughly the same in both outputs. This is a phenomenon we have observed in all our experiments.

1.6.2 Triangulation Data

Figure 1.9 shows a set of input points and a triangulation of this set. The triangulation consists of 906 segments. The zoomed in pictures show a part of the triangulation for which there is considerable difference between SR and ISR.

Again ISR collapses thin polygons that SR does not collapse. The second table in Figure 1.9 shows that in this case the average deviation of a chain in both schemes does not differ by much. The maximum deviation in ISR is always less than twice the pixel width. Here also the average number of links per chain is almost the same for the output of SR and ISR.
1.6.3 Geographic Data

We ran both schemes on several geographic maps of countries and cities which are less cluttered than the examples above. The experiments for this type of data typically show little difference between the SR and ISR results. Figure 1.10 depicts the result for a map of the USA. The data contains 486 segments intersecting only at endpoints.

The second table in Figure 1.10 shows the difference of using SR and ISR. In most of the tests, there are occasional cases in which the distance between a vertex and a non-incident segment is shorter than half the size of a pixel. Thus there are differences between the SR and the ISR output. These differences are however minor. In the ISR output the maximum deviation is no more than twice that of the SR output. The average deviation in both the SR and ISR output is similar.

1.7 Conclusions

We presented an augmented snap rounding procedure which rounds an arbitrary precision arrangement of segments in $\mathbb{R}^2$ with the advantage that each vertex in the rounded arrangement is at least half a unit away from any non-incident edge. The new scheme makes the rounded arrangement more robust for further manipulation with limited precision arithmetic than the output that the standard Snap Rounding algorithm produces. We implemented ISR using exact arithmetic.

We propose several directions for further research: (1) Can detecting all the hot pixels through which an output chain passes be done more efficiently? (2) Extend the scheme to non-linear curves. (3) The rounded arrangement can have at most $O(n^2)$ segments, whereas our algorithm (as well as the known algorithms for SR) may produces $\Omega(n^3)$ output links. The task here is to devise an output sensitive algorithm where the output size is the size of the rounded arrangement and not the overall complexity of the chains. (4) Improve the heuristics for choosing the directions of the kd-trees.
Different number of kd-trees

<table>
<thead>
<tr>
<th>nkd</th>
<th>nhp</th>
<th>tt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>613477</td>
<td>100% = 213.2 s</td>
</tr>
<tr>
<td>2</td>
<td>513551</td>
<td>87.2%</td>
</tr>
<tr>
<td>3</td>
<td>474997</td>
<td>83.6%</td>
</tr>
<tr>
<td>4</td>
<td>478749</td>
<td>84%</td>
</tr>
<tr>
<td>5</td>
<td>479507</td>
<td>84.3%</td>
</tr>
<tr>
<td>6</td>
<td>463025</td>
<td>83.4%</td>
</tr>
<tr>
<td>7</td>
<td>456882</td>
<td>83%</td>
</tr>
<tr>
<td>8</td>
<td>456269</td>
<td>84%</td>
</tr>
<tr>
<td>9</td>
<td>455334</td>
<td>84.8%</td>
</tr>
<tr>
<td>10</td>
<td>456196</td>
<td>86.3%</td>
</tr>
</tbody>
</table>

ISR and SR comparison

Figure 1.8: Congestion data
<table>
<thead>
<tr>
<th>nkd</th>
<th>nhp</th>
<th>tt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4872</td>
<td>100% = 15.4 s</td>
</tr>
<tr>
<td>2</td>
<td>4789</td>
<td>103.2%</td>
</tr>
<tr>
<td>3</td>
<td>4852</td>
<td>102.6%</td>
</tr>
<tr>
<td>4</td>
<td>4597</td>
<td>101.9%</td>
</tr>
<tr>
<td>5</td>
<td>4487</td>
<td>102.6%</td>
</tr>
<tr>
<td>6</td>
<td>4349</td>
<td>103.2%</td>
</tr>
<tr>
<td>7</td>
<td>4349</td>
<td>102.6%</td>
</tr>
<tr>
<td>8</td>
<td>4399</td>
<td>102.6%</td>
</tr>
<tr>
<td>9</td>
<td>4419</td>
<td>103.2%</td>
</tr>
<tr>
<td>10</td>
<td>4358</td>
<td>102.6%</td>
</tr>
</tbody>
</table>

Different number of kd-trees

<table>
<thead>
<tr>
<th>ps</th>
<th>nhp</th>
<th>isr md</th>
<th>isr ad</th>
<th>isr mnv</th>
<th>isr anv</th>
<th>isr mdvs</th>
<th>isr ncv</th>
<th>sr md</th>
<th>sr ad</th>
<th>sr mnv</th>
<th>sr anv</th>
<th>sr mdvs</th>
<th>sr ncv</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>306</td>
<td>2.231</td>
<td>0.825</td>
<td>6</td>
<td>2.219</td>
<td>1.223</td>
<td>0</td>
<td>1.341</td>
<td>0.812</td>
<td>6</td>
<td>2.198</td>
<td>0.318</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>9.804</td>
<td>2.691</td>
<td>7</td>
<td>2.625</td>
<td>3.14</td>
<td>0</td>
<td>3.494</td>
<td>2.442</td>
<td>6</td>
<td>2.48</td>
<td>0.741</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>249</td>
<td>17.194</td>
<td>5.18</td>
<td>9</td>
<td>2.761</td>
<td>5.368</td>
<td>0</td>
<td>7.028</td>
<td>4.847</td>
<td>7</td>
<td>2.637</td>
<td>1.414</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>162</td>
<td>32.207</td>
<td>7.614</td>
<td>9</td>
<td>2.621</td>
<td>11.767</td>
<td>0</td>
<td>13.914</td>
<td>7.19</td>
<td>8</td>
<td>2.532</td>
<td>4.85</td>
<td>45</td>
</tr>
</tbody>
</table>

ISR and SR comparison

Figure 1.9: Triangulation data
ISR and SR comparison

Figure 1.10: Geographic data
Chapter 2

Iterated Snap Rounding with Bounded Drift

Snap Rounding and its variant, Iterated Snap Rounding, are methods for converting arbitrary-precision arrangements of segments into a fixed-precision representation (we call them SR and ISR for short). Both methods approximate each original segment by a polygonal chain, and both may lead, for certain inputs, to rounded arrangements with undesirable properties: in SR the distance between a vertex and a non-incident edge of the rounded arrangement can be extremely small, inducing potential degeneracies. In ISR, a vertex and a non-incident edge are well separated, but the approximating chain may drift far away from the original segment it approximates. We propose a new variant, Iterated Snap Rounding with Bounded Drift, which overcomes these two shortcomings of the earlier methods. The new solution augments ISR with simple and efficient procedures that guarantee the quality of the geometric approximation of the original segments, while still maintaining the property that a vertex and a non-incident edge in the rounded arrangement are well separated. We investigate the properties of the new method and compare it with the earlier variants. We have implemented the new scheme on top of CGAL, the Computational Geometry Algorithms Library, and report on experimental results.

2.1 Introduction

As described in the previous chapter, in Snap Rounding both the deviation of the input is very small and certain (limited) topological properties are preserved (for instance, no segment ever crosses completely over a vertex of the arrangement). However, in a a Snap Rounding output, a vertex of the output may be extremely close to a non-incident edge\(^1\), inducing potential degeneracies. Thus, queries that involve such elements may have non-robust answers, leading to possible inconsistencies. In the previous chapter we proposed an augmented procedure, Iterated Snap Rounding (ISR, for short), to eliminate this undesirable property. ISR rounds the arrangement differently from SR, such that any vertex is at least half-the-width-of-a-pixel away from any non-incident edge. Figure 2.6 depicts an example in which SR introduces a short distance between a vertex and a non-incident edge. This distance is increased by ISR as illustrated in this example.

\(^1\)The distance between a vertex and a non-incident edge can be made as small as \(1/\sqrt{(2^b - 1)^2 + 1} \approx 2^{-b}\), where \(b\) is the number of bits in the representation. The SR/ISR tiling contains \(2^b \times 2^b\) unit pixels.
ISR, however, may round segments far from their origin. Figure 2.6(c) illustrates long drift from the input. In this example, the approximating segment is $\Theta(n^2)$ units away from the original segment, degrading the close proximity property of Snap Rounding.

### 2.1.1 Our contribution

We propose a new algorithm, Iterated Snap Rounding with Bounded Drift (ISRBD, for short), which rounds the segments such that both the distance between a vertex and a non-incident edge is at least half-the-width-of-a-pixel, and the deviation is bounded by a user-specified parameter. Thus ISRBD avoid both undesirable properties of SR and ISR.

ISRBD can be viewed as a modification of ISR. What we do is take the ISR algorithm and plug in two new procedures. In the first ($\text{GenerateNewHotPixels}$), new hot pixels are introduced. These hot pixels are used to bound the rounding magnitude. This procedure is the heart of this chapter. We show that in theory, and practice, the space required for the output is not significantly larger than the one required by ISR. ISRBD, therefore, eliminates the undesirable feature of ISR (namely the possible large deviation), which in turn maintains a half-unit distance between vertices and non-incident edges.

The second procedure ($\text{RemoveDegree2Vertices}$) comes to improve the output quality by removing some of the degree-2 vertices. It follows an idea presented in [29].

Both procedures are fairly simple to understand and implement and use well known geometric data structures (the simplicity depends on having these data structure available).

We augment a list of desirable properties presented in [29] and show that ISRBD is the only method to satisfy all. Thus, we believe that ISRBD may be a good option to choose when a snap-rounded–like arrangement of segments is required.

The rest of this chapter is organized as follows. In Section 2.2 we present the main ideas of this work. In Section 2.3 we describe our algorithm. Implementation details and complexity analysis of our algorithm are presented in Section 2.4. In Section 2.5 we present desirable properties that a snap-rounded–like arrangement should satisfy and fit them to ISR and ISRBD. Experiments performed with our implementation are presented in Section 2.6. We extend ISR and ISRBD to $\mathbb{R}^3$ and to support dynamic rounding in Section 2.7. We conclude and present ideas for future research in Section 2.8.

### 2.2 Preliminaries and Key Ideas

In order to make our description and analysis more clear, we normalize the input such that the pixel edge size is \textit{unit length}. We use the following notations. $S = \{s_1, s_2, \ldots, s_n\}$ is the set of input segments. Let $A(S)$ be the arrangement of $S$ with output $A^*(S)$. Let $S^*$ be the list of output chains. Let $H = \{h_1, h_2, \ldots, h_m\}$ be the set of hot pixels. Since $H$ is dynamic in ISRBD in a sense that hot pixels can be inserted and removed on the fly, $H$ refers to the set that exists at the time of reference, unless otherwise stated. Let $s \in S$ be a segment in the input and $h \in H$ be a hot pixel. From now on, whenever we refer to $s$ or $h$ without explicitly defining them, we mean that $s$ is any segment of $S$ and $h$ is any hot pixel of $H$. Let $h(p)$ be the hot pixel containing a point $p$. For any pixel $h$, let $h_c$ be its center with coordinates $h_x$ and $h_y$. We denote the output of $s$ by $\lambda(s)$. Let
$d(s)$ be the Hausdorff distance between $s$ and $\lambda(s)$. Our goal, therefore, is to bound $d(s)$. For any set of one or more geometric objects, denoted by $X$, let $B(X)$ be the axis-aligned bounding box of $X$. We denote the triangle with vertices $x, y$ and $z$ by $\Delta(x, y, z)$.

For each hot pixel $h$ and segment $s$ such that $h$ properly intersects $B(s)$, we define a right triangle $\Delta(s, h)$ (the definition applies when $s$ has a negative slope and it is to the right of $h$; other cases are defined analogously). Let $v_1$ be the upper-right corner of $h$, $\ell$ be the infinite line containing $s$ and $\vec{v}$ be the vector $[\frac{1}{2}, \frac{1}{2}]$. Let $\ell'$ be the line parallel to $\ell$, shifted by $\vec{v}$ units from $\ell$. Let $v_2$ and $v_3$ be the intersections between $\ell'$ and the horizontal and vertical rays from $v_1$ towards $\ell'$. Then $\Delta(s, h) \equiv \Delta(v_1, v_2, v_3)$. See Figure 2.1 for an illustration.

![Figure 2.1: $\Delta(s, h)$ is the triangle whose edges are marked by bold lines.](image)

We define the following two predicates:

- For any segment $s$ and a hot pixel $h$, $\Xi(s, h)$ is true if and only if $\Delta(s, h)$ contains no center of hot pixels.
- For any two hot pixels $h_1$ and $h_2$, $\Phi(h_1, h_2)$ is true if and only if their centers have either the same $x$-coordinate or the same $y$-coordinate.

We continue by introducing a few properties that hold for ISR (Lemmata 2.1-2.3).

**Lemma 2.1.** $\lambda(s)$ is weakly-monotone.

*Proof.* Suppose that $\lambda(s)$ is not weakly-monotone. Recall that ISR is equal to a series of SR applications. It follows that during at least one of the iterations of SR, a link $\ell$ is snapped by a hot pixel $h_2$, creating two links that are not weakly-monotone. Let $h_1$ and $h_3$ be the hot pixels containing the endpoints of $\ell$. We get a contradiction since it follows that a straight segment $\ell$ intersects both $h_1$, $h_2$ and $h_3$, while the chain through their centers is not weakly-monotone. □

**Lemma 2.2.** $\lambda(s)$ lies within $B(h(p)_c, h(q)_c)$ where $p$ and $q$ are the endpoints of $s$.

*Proof.* Suppose that $\lambda(s)$ is not entirely within $B(h(p)_c, h(q)_c)$. Then there is a hot pixel, $h$, such that the polygonal chain through $h(p)_c, h_c$ and $h(q)_c$ is not weakly-monotone. It follows that $\lambda(s)$ is not weakly-monotone in contradiction to Lemma 2.1. □
For the next lemma we restrict ourselves to hot pixels that properly intersect $B(s)$. Others are irrelevant for this discussion since $\lambda(s)$ does not contain their centers as Lemma 2.2 indicates.

**Lemma 2.3.** For any segment $s \in S$ and hot pixel $h \in H$, $h_c$ cannot belong to $\lambda(s)$ if $\Xi(s,h)$ is false.

**Proof.** Without loss of generality, assume that $h$ lies to the left of $s$, and that $s$ has a negative slope. Suppose the claim is false and $\Delta(s,h)$ contains a center of a hot pixel $h'$ and $\lambda(s)$ contains $h_c$. Let $p$ and $q$ be the endpoints of $s$. According the definition of $\Delta(s,h)$, $h_c$ and $h'_c$ cannot be located within different sides of $s$. Since the rounding magnitude of each iteration is at most $\sqrt{2}$ units, the output link cannot snap to $h_c$ without snapping to $h'_c$ as well (or in other words, it cannot bypass $h'_c$). It follows that $h_c$, $h'_c$, $h(p)_c$ and $h(q)_c$ are all vertices of $\lambda(s)$. However, these four vertices cannot form a weakly-monotone subsequence in $\lambda(s)$, in contradiction to Lemma 2.1. See Figure 2.2 for an illustration.

![Figure 2.2](image)

Figure 2.2: A segment $s$ (the solid line) and a possible chain through the centers of hot pixels $h(p)$, $h'$, $h$ and $h(q)$ (the dashed polygonal chain). Such a chain through these hot pixels is never weakly-monotone.

The main idea of our work is to bound the deviation of the output of ISR. Let $\delta$ be the maximum deviation allowed, given as a parameter. We require that $\delta > \frac{3\sqrt{2}}{2}$ for a reason described in Section 2.3.1. For each segment $s \in S$ with endpoints $p$ and $q$, $\delta$ defines a domain $D(s, \delta)$ that must contain $\lambda(s)$ (see Figure 2.3). Since $\lambda(s)$ lies inside $B(h(p)_c, h(q)_c)$ (Lemma 2.2), $D(s, \delta)$ is the intersection of $B(h(p)_c, h(q)_c)$ with the Minkowski sum of $s$ and a circle with radius $\delta$ centered at the origin. We ignore the case where $s$ is either horizontal or vertical, since in this case $d(s) \leq \frac{\sqrt{2}}{2}$, maintaining a deviation that is clearly smaller than $\delta$ (see Section 2.4.4 for more details).

![Figure 2.3](image)

Figure 2.3: The domain $D(s, \delta)$ (shaded) and forbidden locus $F(s, \delta) = F_l(s, \delta) \cup F_r(s, \delta)$. The input $s$ is the thick segment crossing $D(s, \delta)$. The two small squares are the hot pixels containing the endpoints of $s$. 
Our goal, therefore, is to bound $\lambda(s)$ to $D(s, \delta)$. Let $F_l(s, \delta)$ and $F_r(s, \delta)$ be the two trapezoids, lying to the left and right of $D(s, \delta)$, respectively. $F_l(s, \delta)$ is defined as follows. One edge of $F_l(s, \delta)$ is the non-isothetic left edge of $D(s, \delta)$, denoted by $e$ (see Figure 2.3). Let $\vec{v}$ be the vector $\left[\frac{1}{2}, \frac{1}{2}\right]$. The opposite edge of $e$ in $F_l(s, \delta)$ is parallel to $e$ and shifted $\vec{v}$ units from $e$; its endpoints lie on $B(h(p)_c, h(q)_c)$. The two edges are connected by segments of $B(h(p)_c, h(q)_c)$. $F_r(s, \delta)$ is defined similarly to the right of $D(s, \delta)$. See Figure 2.3 for an illustration. Let $F(s, \delta) = F_l(s, \delta) \cup F_r(s, \delta)$ and $F(S, \delta) = \{F(s, \delta) | s \in S\}$. The following claims refer only to the area lying to the left of $s$, but are symmetric and can be applied to the right.

We mentioned that ISR is equivalent to the final output of a finite series of applications of SR. Consider some segment $s$ for which the output is obtained after $N$ applications of SR. Let $\phi_i(s)$ be the temporary chain of $s$ after the $i$-th application of SR. Since links are rounded to centers of hot pixels, the Hausdorff distance between $\phi_i(s)$ and $\phi_{i+1}(s)$ is at most $\frac{sN}{2}$ units for any $1 \leq i \leq N - 1$. Notice that if $\lambda(s)$ is snapped during one of the applications of SR to the left (right) beyond $D(s, \delta)$, then the center of the hot pixel responsible for that snapping must lie within $F_l(s, \delta)$ ($F_r(s, \delta)$). Following the construction of $F_l(s, \delta)$ ($F_r(s, \delta)$), other hot pixels would be too far to snap any segment inside $D(s, \delta)$. Thus, in order to bound $\lambda(s)$ to $D(s, \delta)$, it is sufficient to make sure that no hot pixels whose centers are contained inside $F(s, \delta)$ ever snap $s$. This, in turn, is achievable if we make sure that for any hot pixel $h$ and segment $s$, if $h_c$ lies inside $F(s, \delta)$, then $\Delta(s, h)$ contains at least one center of a hot pixel.

The above discussion leads to the main idea ISRBD: for any segment $s$ and hot pixel $h$, if $h_c \in F(s, \delta)$ and $\Xi(s, h)$ is true, we heat a pixel $h'$ whose center lies within $\Delta(s, h)$. As a result, $\Xi(s, h)$ becomes false. Clearly, this process must be performed before the rounding stage. Since the output of the rounding stage can be illustrated as a continuous deformation of pixels [48], heating pixels as we propose may change the output, but not the topology as defined in [48] (see Property 5.2 for more details). This process may cascade as the new hot pixel may lie within a forbidden locus of another segment.

The following corollary establishes the condition that is sufficient for bounding the deviation of $\lambda(s)$ from $s$:

**Corollary 2.4.** If $\Xi(s, h)$ is false for each segment $s \in S$ and a hot pixel $h \in H$ whose center lies within $F(s, \delta)$, then the deviation of ISRBD is bounded by $\delta$.

We distinguish between pixels that are heated because they contain vertices of $A(S)$ and pixels that are heated to bound the deviation as explained above. We refer to the pixels in the first group as the *original* hot pixels, as they are defined in ISR, and to the ones in the second group as the *new* hot pixels, as they are defined in this work for the first time.

In the remainder of this work we have situations in which a hot pixel $h$ blocks another hot pixel $h'$ from rounding a segment $s$. This is because $h_c$ is contained within $\Delta(s, h')$. Next, we formalize this situation. Let $h$ and $h'$ be two hot pixels and $s$ a segment. The predicate $\zeta(h, h', s)$ is true if and only if $h_c$ is contained inside $\Delta(s, h')$.

### 2.3 Algorithm

The main idea of our algorithm follows the discussion in the previous section. The idea is to heat pixels iteratively until the condition in Corollary 2.4 is met for all hot pixels and segments, namely
for each segment \( s \) and a hot pixel \( h \) whose center lies within \( F(s, \delta) \), \( \Xi(s, h) \) is false. As a result, the output drift obtained in the rounding stage will be bounded as required.

The algorithm for computing ISR has three stages [52]. In the first, the hot pixels are detected and stored. In the second, a range search data structure is built for answering queries that report the hot pixels that a given segment intersects. The third is the rounding stage. ISRBD applies all three stages of ISR, as well as plugging in two new procedures. In the first, \( \text{GenerateNewHotPixels} \), pixels to bound the drift are heated. The procedure is executed after the first stage of ISR (detecting the set of original hot pixels). It is executed after this stage because the list of original hot pixels has to be available. It is executed before the other two stages of ISR because clearly they require a complete list of hot pixels (original and new). In the second procedure, \( \text{RemoveDegree2Vertices} \), some of the degree-2 vertices of the output are removed to improve the output quality. It is executed after the third stage of ISR (the rounding stage), because the output chains of all input segments have to be available. We describe both procedures in this section.

The following is a high level pseudo-code of ISRBD.

**Iterated Snap Rounding with Bounded Drift**

Input: a set \( S \) of \( n \) segments and maximum deviation \( \delta \)
Output: a set \( S^* \) of \( n \) polygonal chains

1. Compute the set \( H \) of hot pixels
2. Call \text{GenerateNewHotPixels}
3. Construct a segment intersection search structure \( D \) on \( H \) /* \( H \) here includes the new hot pixels heated in \text{GenerateNewHotPixels} */
4. Perform the rounding stage
5. Call \text{RemoveDegree2Vertices}

### 2.3.1 Heating New Hot Pixels

In this section we discuss the procedure \text{GenerateNewHotPixels}. For any segment \( s \) and hot pixel \( h \), let \( s' \) be any segment that intersects \( \Delta(s, h) \). Let \( I(s', s, h) \) be the intersection of \( s' \) and \( \Delta(s, h) \). Suppose a center of a hot pixel \( h_c \) lies within \( F(s, \delta) \) where \( \Xi(s, h) \) is true. We need to heat a pixel whose center is contained inside \( \Delta(s, h) \). Next we explain which pixel we heat. Without loss of generality, we assume that \( h \) lies to the lower-left of \( s \), where \( s \) has a negative slope. Other cases are similar. We first check if there are segments that penetrate the pixel \( h' \) with center coordinates \((h_x + 1, h_y + 1)\); \( h \) cannot be in their forbidden loci since \( \delta > \frac{3\sqrt{2}}{2} \) (this is the reason for constraining \( \delta \) to be at least this magnitude; also see Figure 2.4 for an illustration). If there are such segments, we heat this pixel (see Figure 2.5(a)). Otherwise, let \( t \) be the first segment to the right of \( h \) which intersects \( \Delta(s, h) \). We heat the pixel that contains the middle point of \( I(t, s, h) \). Figure 2.5(b) illustrates this situation in which a pixel on \( s \) is heated. Figure 2.5(c) illustrates the same case, but here we heat a pixel on a segment \( s' \), which is closer to \( h \) than \( s \). (In Lemma B.5(a) we show that \( s \) and \( t \) cannot intersect inside \( \Delta(s, h) \).) We choose this technique for heating pixels in order to have certain claims hold (see Section 2.4). Once a pixel is heated, it is possible that it is located inside a forbidden locus of another segment. Thus, this process is performed for each hot pixel, whether original or new.

Our algorithm proceeds as follow. For each hot pixel \( h \in H \) (original or new; the new hot pixels
are inserted to $H$ during this process), we locate the set of segments $S'$ such that $h$ is within $F(s, \delta)$ for each $s \in S'$. For each $s \in S'$, we check if $\Delta(s, h)$ is empty of centers of hot pixels. If it is empty, we heat a pixel as explained above.

![Diagram](image)

Figure 2.4: $h'$ is contained inside the quarter-circle centered at $h_c$ with radius $\frac{3\sqrt{2}}{2} \times \delta$. Since $t$ intersects $h'$, $h$ cannot be in a forbidden locus of $t$.

![Diagram](image)

Figure 2.5: Heating a new hot pixel, $h'$ as a result of hot pixel $h$ being within the forbidden locus of $s$. $\Delta(s, h)$ is depicted with thin edges in all sub-figures. (a) Heating the upper-right neighbor of $h$. (b) Heating on $s$ (c) Heating on a segment that intersects $\Delta(s, h)$.

We use a trapezoidal decomposition of forbidden loci (denoted by $\Gamma_1$), for all input segments. This data structure helps us to efficiently identify in which forbidden loci each hot pixel is located. We also use a trapezoidal decomposition of input segments (denoted by $\Gamma_2$) to be able to query segments which intersect $\Delta(s, h)$ efficiently. The purpose is to locate the segment on which we heat a pixel as described above. In order to traverse the hot pixels (original and new), we use a hot pixel queue $Q$, initialized to the original hot pixels. We dequeue hot pixels from $Q$ and process them as explained above. When heated, new hot pixels are queued to be processed later. For querying the emptiness of $\Delta(s, h)$, we use a dynamic simplex range searching data structure (denoted by $\Psi$). It is initialized to the original hot pixels and updated whenever a pixel is heated. We use $\Psi$ also in RemoveDegree2Vertices; see Section 2.3.2. In Section 2.4.2, we provide details about $\Gamma_1$, $\Gamma_2$ and $\Psi$.

The following theorem summarizes the quality of the geometric approximation obtained with ISRBD.

**Theorem 2.5.** The polygonal chain $c(s) \in S^*$ of any segment $s \in S$ lies inside the Minkowski sum of $s$ with a disc of radius $\delta$, centered at the origin.
Proof. It immediately follows from our algorithm: whenever there is a hot pixel \( h \) in \( F(s, \delta) \), we make sure that \( s \) never snaps to the center of \( h \) since we guarantee that \( \Xi(s, h) \) is false after processing GenerateNewHotPixels. Thus, \( s \) never snaps to the centers of hot pixels which are beyond \( D(s, \delta) \). It follows that only hot pixels within distance at most \( \delta \) from \( s \) may be snapping points for \( s \) and the claim follows.

The following is a pseudo-code of \texttt{GenerateNewHotPixels}.

\begin{verbatim}
\textbf{GenerateNewHotPixels}
\begin{enumerate}
\item Build \( \Gamma_1 \) and \( \Gamma_2 \)
\item Build \( \Psi \)
\item Initialize a queue, \( Q \), with \( H \)
\item while \( Q \) is not empty
\begin{enumerate}
\item let \( h = \text{dequeue}(Q) \)
\item foreach segment \( s \) for which \( F(s, \delta) \) contains \( h_c \)
\begin{enumerate}
\item if \( \Xi(s, h) \) is true
\begin{enumerate}
\item Heat a pixel \( h' \) whose center is contained inside \( \Delta(s, h) \) (as described in Section 2.3.1)
\end{enumerate}
\item Queue(\( Q, h' \)) and insert \( h'_c \) to \( \Psi \)
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{verbatim}

\subsection{Removing Redundant Degree-2 Vertices}

In this section we discuss the procedure \texttt{RemoveDegree2Vertices}. De Berg et al. [29] proposed to modify \( A^*(S) \) by removing degree-2 vertices that do not correspond to endpoints. It is done as a post-processing step by removing them with their adjacent links and connecting their two neighboring vertices with a new link. They proved that important properties of SR still hold after this process. However, this idea will not work as is in ISRBD. Consider Figure 2.6(d). The center of the new hot pixel (denoted by \( h \); it is the pixel with bold edges) is a degree-2 vertex that would be removed if the above method is applied. However, removing \( h \) will cause the output to penetrate one of the hot pixels and not pass through its center. It is also possible that this removal will cause the output drift to be too large.

Nevertheless, applying a method in this spirit can be very beneficial in ISRBD since intuitively many of the new hot pixel centers are degree-2 vertices. Thus, we modify this method such that it will retain the properties of ISRBD. What we do is constrain the removal to only non-endpoint degree-2 vertices whose removal does not violate the following properties of ISRBD: (a) The new link passes through a hot pixel and not through its center (b) The Hausdorff distance between the new link and the original segment is bigger than \( \delta \). Such a removal makes the output simpler and more efficient. In Section 2.4.1 we analyze the output of ISRBD and show that there are examples for which this method significantly improves the quality of the output.

We denote by \( K \) the list of degree-2 vertices in the output that do not correspond to endpoints. Let \( \kappa = |K| \). \( K \) can be computed easily by traversing the output arrangement. We traverse all hot pixels in \( K \) and remove hot pixels which do not violate the properties of ISRBD as listed above. Let \( \beta(h) \) be the link obtained when removing a hot pixel \( h \). It is possible that some hot pixel \( h \) which has already been processed without being removed will be eligible for removal later. There
are two possible cases for that. The first is after the removal of one of its neighbors (where one of the links attached to \( h \) is changed). The second may occur if \( \beta(h) \) penetrates one or more hot pixels and not passing through their centers, and then those hot pixels are removed, allowing \( h \) to be removed too. For each hot pixel, we hold a list of other hot pixels that may be redundant after its removal, as described above. We denote this list by \( P \); each hot pixel will have its own instance. Thus, when removing a hot pixel, we take care of reprocessing the hot pixels in its instance of \( P \).

For each \( h \in K \), we check whether its removal violates the properties of ISRBD. If it does not, we remove it from \( H \) and update \( S^* \) and \( \Psi \) (\( \Psi \) was defined in Section 2.3.1) accordingly. The first test is done by computing the Hausdorff distance between the potential new link and the corresponding input segment and then comparing it to \( \delta \). The second test is done by checking if the potential new link does not penetrate any hot pixel without passing through its center. In order to query the hot pixels penetrated by segments (the second test), we need a dynamic range search data structure. It must be dynamic because hot pixels are removed in this routine. We use the data structure, called \( \Psi \), which is defined and used in Section 3.1.

Remark. Since this routine can be applied in ISR as well, it would be interesting to describe how it is adjusted for the sake of ISR. The variant of ISR is very similar to the one of ISRBD, with the only difference that the check for large drift is omitted. Other than that, the algorithm and its analysis are similar.

The following is a pseudo-code of \textit{RemoveDegree2Vertices}.

\begin{verbatim}
REMOVEDEGREE2VERTICES
1. Initialize every hot pixel to be \textit{active} and a list \( P \), one for each hot pixel, to contain the pixel neighbors
2. \textbf{foreach} \( h \in K \)
3. \hspace{1em} \textbf{if} \( h \) is removable (as described in Section 2.3.2)
4. \hspace{2em} \textbf{foreach} hot pixel \( h' \) in \( h.P \)
5. \hspace{3em} \textbf{if} \( h' \) is \textit{non-active}
6. \hspace{4em} \textbf{set} \( h' \) to be \textit{active}
7. \hspace{5em} \textbf{move} \( h' \) to the end of \( K \) /* done in order to recheck \( h' \) for redundancy */
8. \hspace{2em} \textbf{end if}
9. \hspace{1em} \textbf{end foreach}
10. \textbf{remove} \( h \) from \( K \) and update \( \Psi \) and the output \( S^* \) accordingly
11. \textbf{else}
12. \hspace{1em} \textbf{mark} \( h \) as \textit{non-active}
13. \hspace{1em} \textbf{foreach} degree-2 hot pixel \( h'' \) which is penetrated by the potential link obtained by the removal of \( h \), not through the center of \( h'' \)
14. \hspace{2em} \textbf{push} \( h \) to \( h''.P \)
15. \hspace{1em} \textbf{end foreach}
16. \hspace{1em} \textbf{end if}
17. \textbf{end foreach}
\end{verbatim}

2.3.3 Illustrative Example

Figure 2.6 illustrates the differences among SR, ISR, and ISRBD, for an input set of segments (Figure 2.6(a)) where ISR incurs large drift (on one of the segments). The input is in the spirit of
Figure 2.6: Results of SR, ISR and ISRBD of an example with large drift. All squares are hot pixels. The edges of new hot pixels and the ‘interesting’ segment are marked by bold lines. (a) Input segments (b) SR output (c) ISR output (d-e) ISRBD output for two different values of δ.

the large-drift example given in [74]. In this example, only one segment s will be modified in the rounding, and it is drawn in bold line. In the SR output (Figure 2.6(b)), λ(s) penetrates a hot pixel but does not go through its center, making the polygonal chain rather close to the endpoint of the other segments in that pixel—in general the polygonal chain and the endpoint of the other vertex may get extremely close to one another in the output of SR. Figure 2.6(c) demonstrates the output of ISR for the same input, where the drift of the segment is large—in general the approximating chain may drift Θ(n^2) pixels away in the output of ISR. The last two figures, Figure 2.6(d) and (e), show the output of ISRBD for two different values of δ: the drift is bounded as desired, and the chain and any other non-incident vertices are well separated.

2.4 Implementation Details and Complexity Analysis

2.4.1 Output Complexity

In this section we analyze the output complexity of ISRBD. We prove that for any segment s ∈ S, λ(s) may consist of O(n^2) links, resulting in an overall complexity of O(n^3). This result states that although we introduce new hot pixels, the output complexity of ISRBD is equivalent to SR and ISR. The idea of the proof is that although the output may contain O(n^3) new hot pixels, λ(s)
consists of \(O(n^2)\) centers of such hot pixels, for each \(s \in S\). It follows that the overall complexity is \(O(n^3)\). Since the lower bound is \(\Omega(n^3)\) due to an example presented in [52], the maximum output complexity is \(\Theta(n^3)\) (this example has the same output with SR, ISR and ISRBD—the proof for that claim would be immediate). We note that if we only consider the complexity of the rounded arrangement without counting the multiplicities of overlapping segments, the output complexity of SR and ISR becomes \(\Theta(n^2)\). Under this condition, we have not established tight bound for ISRBD yet. We show later that the lower bound is \(\Omega(n^3)\) due to an example presented in [52], the maximum output complexity is \(\Theta(n^3)\) (this example demonstrates the usefulness of RemoveDegree2Vertices).

For each hot pixel \(h\), we denote by \(f(h)\) the pixel that was the cause for heating \(h\) by being located within a forbidden locus of some segment (we also say that \(h\) is the child of \(f(h)\) or \(f(h)\) is the father of \(h\)). Since there is no reason to heat a pixel which is already hot, the relation induced by \(f\) defines a directed forest in which the root of each tree is an original hot pixel, and all of the pixels heated due to this pixel are below the root in this tree. Each edge in this data structure corresponds to a hot pixel and its child. For each original hot pixel \(h\), let \(T(h)\) be its corresponding tree and \(C(h)\) be the set of all the hot pixels in \(T(h)\). We divide \(C(h)\) into four sets corresponding to the four quadrants around \(h\) (we prove later that for any hot pixel \(h' \in C(h)\), \(\Phi(h, h')\) is false, namely the centers of \(h\) and \(h'\) do not lie on the same horizontal or vertical lines): we number them starting from the upper-right and go clockwise. Let \(C_1(h) \subseteq C(h)\) be the list of hot pixels to the upper-right of \(h\) and \(\{C_i(h)\mid i = 1 \ldots 4\}\) be the lists in clockwise order in space. In our analysis, we concentrate on the upper-right quadrant. The situation in other quadrants will be equivalent because of symmetry. We build our proofs by starting with an original hot pixel and explore its tree.

Consider an original hot pixel, \(h\), and a segment \(s\) that intersects \(h\). Let \(h'\) be a child of \(h\) and assume that \(h'\) is located to the upper-right of \(h\). Let \(s'\) be the segment on which \(h'\) was heated. It follows that \(s'\) has a negative slope (since \(h'\) is to the upper-right of \(h\)). Let \(h''\) be a child of \(h'\) (assume that such a pixel exists; even if there is no such hot pixel, all the proof that follow which does not depend on \(h''\) will hold) and \(s''\) be the segment on which \(h''\) was heated. We divide the plane into four quadrants around \(h'\) and explore the possibility and the results of placing \(h''\) in a different quadrant (see Figure 2.7). We first rule out two quadrants which cannot contain \(h''\).

![Figure 2.7: Different cases of pixel heating](image)

**Lemma 2.6.** No new hot pixel needs to be heated on the upper-left and lower-right quadrants.

**Proof.** If the slope of \(s''\) is negative, \(s''\) cannot be the source for \(h''\). The reason is that in this case
\(\Delta(s'', h')\) must be in the upper-right or lower-left quadrants. Thus, we assume that \(s''\) has a positive slope. Next we check all the possibilities for placing \(s''\). Clearly, \(s''\) cannot intersect triangle \(\Delta(s', h)\), otherwise \(\Xi(s', h)\) would be false and \(h'\) would not be heated. Consider Figures 2.7(b) and (d): \(s''\) must intersect the line containing \(s'\), because otherwise \(h'\) is not in its axis-aligned bounding box and forbidden locus (note that \(s''\) cannot intersect \(h'\)). Thus, there must be a pixel \(h_b\) on \(s'\) such that \(\zeta(h_b, h', s'')\) holds (\(h_b\) would correspond to either an endpoint of \(s'\) or to an intersection of \(s'\) with either \(s''\) or another segment).

We continue with ruling out another quadrant.

**Lemma 2.7.** No new hot pixel needs to be heated on the lower-left quadrant.

*Proof.* Analogously to Lemma 4.1, we can assume that \(s''\) has a negative slope. It also must lie to the left of \(h'\). Following the location of the new hot pixel (Section 2.3.1), there are no segments to the left of \(h'\) which intersect \(\Delta(s, h)\). Thus, consider the case shown in Figure 2.7(c): there is a segment \(s''\) which lies to the lower-left of \(h'\), its axis-aligned bounding box contains \(h'_c\) and whose forbidden locus contains \(h'\). Then \(\zeta(h, h', s'')\) holds and \(s''\) will never snap to \(h'\). Thus, no pixel has to be heated in this case. □

The next lemma provides another insight to the location of \(h'\). We prove that the center of \(h\) and \(h'\) cannot lie in the same horizontal or vertical line.

**Lemma 2.8.** \(\Phi(h, h')\) is false.

*Proof.* If we heat the upper-right neighbor of \(h\), it is clear that the claim is valid. Otherwise, let \(b = I(s', s', h)\) (the portion of \(s'\) on \(\Delta(s', h)\)). Since \(b\) does not intersect the upper-right neighbor of \(h\), it extends to more than two \(x\) and \(y\) units. Thus, its middle point cannot be contained inside a hot pixel \(h'\) in a way that \(\Phi(h, h')\) is true (see Figure 2.8). □

Following the last three Lemmata, we get that the only possible location for \(h''\) is to the upper-right of \(h'\). In this case we heat the appropriate pixel. This case is possible as illustrated in Figure 2.7(a) in which \(h'\) lies within \(F(s'', \delta)\). Clearly, this process of heating pixels may cascade further to the upper right in the same way. Next we prove that the constructed structure is a chain.

**Lemma 2.9.** The pixels of \(C_1(h)\) are arranged in a chain under \(h\) in \(T(h)\).

*Proof.* We prove that at most one pixel may be the child of \(h\) in the upper-right quadrant. If the upper-right neighbor of \(h\) is heated, it will clearly block any other segments in the upper-right quadrant from snapping to \(h\), thus there is no need for another child of \(h\). Otherwise, suppose that two distinct pixels are heated right below \(h\), on segments \(s_1\) and \(s_2\). Let \(I_1 = I(s_1, s_1, h)\) and \(I_2 = I(s_2, s_2, h)\). \(I_1\) and \(I_2\) cannot intersect—otherwise their intersection would heat a pixel \(h'\) such that \(\zeta(h', h, s_1)\) holds. Then, without loss of generality, let \(I_1\) be closer to \(h\) than \(I_2\). Following our algorithm, no pixel would be heated on \(I_2\), regardless of the order in which we process segments. This hot pixel (possibly on \(s_1\)) will block both \(s_1\) and \(s_2\) from snapping to \(h\). Thus, no pixel is heated on \(s_2\) in contradiction to our assumption.
In the same way, we can prove that any descendant of \( h \) to the upper-right has at most one child. Due to symmetry, this behavior is similar in other three quadrants. Thus, all the pixels of \( C_1(h) \) are contained in the above chain under \( h \).

Let \( T_{ur}(h) \) be this sequence of hot pixels, starting from \( h \) towards the upper right direction.

![Diagram](image)

Figure 2.8: \( j \), the center of \( I(s', s', h) \) (the thick segment; \( I(s', s', h) \) does not intersect pixel \( g \) which is not hot), is contained inside a pixel \( m \) such that \( \Phi(m, h) \) is never true.

We continue with two lemmata that constrain the output of each segment.

**Lemma 2.10.** Let \( s \in S \) be a segment that intersects pixel \( h \). Then \( \lambda(s) \) cannot pass through any child \( h' \) of \( h \).

**Proof.** If the slope of \( s \) is not positive, it cannot intersect \( h' \) (the weakly-monotone condition would be violated since \( h' \) is to the upper-right of \( h \)). Thus, assume that \( s \) has a positive slope. Since \( s \) intersects \( h \), it does not penetrate \( \Delta(s', h) \) (otherwise \( \Xi(s', h) \) would be false). Then it must be the case that if \( \lambda(s) \) penetrates \( \Delta(s', h) \), it will have to be non weakly-monotone by changing its direction towards another hot pixel \( h'' \) on \( s' \). \( h'' \) corresponds either to an endpoint of \( s' \) or to an intersection on \( s' \) (with \( s \) or another segment). We get that the only way for \( s \) to snap to \( h' \) is to surround \( \Delta(s', h) \). However, note that for arguments similar to Lemma 2.8, \( \Phi(h', h'') \) is false. Thus, \( \lambda(s) \) would not be weakly-monotone if it surrounds \( \Delta(s', h) \). It follows that \( \lambda(s) \) will not snap to \( h' \).

**Lemma 2.11.** For any segment \( s \in S \) and original hot pixel \( h \in H \), \( \lambda(s) \) can pass through at most one hot pixel in \( T_{ur}(h) \).

**Proof.** Lemma 2.10 can be easily generalized such that any segment \( s \) which passes through one of the pixels in \( T_{ur}(h) \), cannot pass through any other pixels in \( T_{ur}(h) \). Since the pixels in \( T_{ur}(h) \) are monotonely increasing in the \( y \)-direction, any segment with negative slope (whose output must be monotonely-decreasing in \( y \)) cannot pass through more than one of them. The claim follows.

We are now ready to describe the structure of \( T(h) \).

**Lemma 2.12.** For each original hot pixel \( h \), \( T(h) \) satisfies the following properties:
(a) The root has at most four children.
(b) Each child of the root is a root of a subchain.
(c) \( |T(h)| = O(n) \).
Proof. (a) and (b) are immediate consequences of Lemma 2.9, when applied for all directions. (c) Consider $T_{ur}(h)$. We heat pixels that intersect input segments. From Lemma 2.11, each segment $s \in S$ can intersect at most one pixel of $T_{ur}(h)$. Thus $s$ can be the source of heating at most one pixel in $T_{ur}(h)$. Then the size of $C_1(h)$ is $O(n)$. Since the situation in other quadrants is similar and from (a) and (b), $|T(h)| = O(n)$. 

We next bound the number of hot pixels created in our algorithm.

**Lemma 2.13.** The overall number of hot pixels generated in GenerateNewHotPixels is $O(n^3)$.

**Proof.** It immediately follows from Lemma 2.12(c), since there are $O(n^2)$ original hot pixels. 

We continue with a theorem that summarizes the output complexity.

**Theorem 2.14.** The maximum output complexity of ISRBD is $\Theta(n^3)$.

**Proof.** In [52] it is shown that the output may need $\Omega(n^3)$ space. The lower bound example need not heat any new hot pixels when using ISRBD (it is clear from the structure of the example; we omit the exact proof). Thus, this example provides a lower bound for ISRBD as well. Since for each segment $s \in S$, $\lambda(s)$ intersects $O(1)$ new hot pixels from $C(h)$ for each original hot pixel $h$ (Lemma 2.11), $\lambda(s)$ may intersect $O(n^2)$ new hot pixels. It can also intersect $O(n^2)$ original hot pixels. Thus, each output chain consists of $O(n^2)$ links. Then the overall output complexity of ISRBD is $O(n^3)$. The claim follows. 

![Figure 2.9: An example of an arrangement of segments which is similar to another arrangement in which $\Theta(n^3)$ pixels are heated in GenerateNewHotPixels.](image)

From Theorem 2.14, the output complexity is the same as SR and ISR. (Note again that if we do not count overlapping segments more than once, then the output complexity of SR and ISR decreases to $\Theta(n^2)$ while we do not know yet what the bound of ISRBD is.)

From Lemma 2.13, the overall number of pixels created in GenerateNewHotPixels is $O(n^3)$. Thus, the overall number of hot pixels is $O(n^3)$ as well. It is $\Omega(n^2)$ [52]. We hope to close this gap in future research. We experimented with many examples in an attempt to produce many new hot pixels, but the number of new hot pixels was always far less than the number of original hot pixels.

We next show that performing RemoveDegree2Vertices can be very useful.

**Lemma 2.15.** If RemoveDegree2Vertices is not called, the number of pixels created in GenerateNewHotPixels is $\Omega(n^3)$ and the maximum overall output complexity is $\Omega(n^3)$ even if overlapping segments are not counted more than once.
Proof. Consider Figure 2.9: its left part consists of a grid of \( n/2 \) segments which includes two groups with \( n/4 \) segments each. The segments in each group are parallel and each segment from the first group intersects each segment from the second one. Let \( s \) be the first segment to the right of the grid. Each hot pixel \( h \) within this structure that corresponds to an intersection is in a location such that \( \Delta(s, h) \) satisfies \( \Xi(s, h) \). It is possible to construct such an example. For example, imagine a similar situation where the left side is very thin and rotated counter-clockwise by 45 degrees, such that any intersection is far away from other intersections. Then the set of hot pixels will be monotonically decreasing along the \( x \)-axis and can satisfy \( \Xi(s, h) \) for each hot pixel induced in this grid. Furthermore, the centers of all of these hot pixels are within \( F(s, \delta) \). The second structure is composed of \( n/2 \) parallel segments, starting from \( s \) and continuing to the right. The distance between each two consecutive segments is such that all of the centers of all hot pixels on one segment will be within the forbidden locus of the segment to its right. It follows that \( \Theta(n^2) \) pixels are heated along \( s \) and each causes a series of \( \Theta(n) \) pixels to be heated on the segments to the right of \( s \). Thus, \( \Theta(n^3) \) new pixels are created. An immediate result is that the output chain of each segment in the right structure consists of \( \Theta(n^2) \) links. Since no output chain of such share any link with other output chains, the overall output complexity is \( \Omega(n^3) \). \( \square \)

However, since we call \textit{RemoveDegree2Vertices} (Section 2.3.2), we will remove all the new hot pixels on \( s \) and the segments to its right, leaving the output in the above example with \( \Theta(n^2) \) hot pixels. The reason is that the segments in the right structure are spread far enough such that it is impossible for one segment to snap to hot pixels lying on its neighboring segment. This example demonstrates that applying \textit{RemoveDegree2Vertices} is very useful for improving the output quality.

### 2.4.2 Implementation Issues

In this section we discuss the implementation of our algorithm. We use two geometric data structures: Trapezoidal decomposition and Dynamic simplex range searching.

#### Trapezoidal Decomposition

Recall that we use the trapezoidal decomposition data structure [31] twice. The first is with the forbidden loci \( F(S, \delta) \) (denoted by \( \Gamma_1 \)) and the second is with the input segments \( S \) (denoted by \( \Gamma_2 \)).

We build \( \Gamma_1 \), a trapezoidal decomposition of \( F(S, \delta) \) (see Section 2.3.1 for details). Each trapezoid in \( \Gamma_1 \) may be within a forbidden loci of one or more segments. For each trapezoid \( t \), we denote by \( t \rightarrow W \) the list of these segments. We need an efficient data structure that queries the corresponding segments for a given trapezoid. By using a trapezoidal decomposition, we can efficiently locate the trapezoid that contains a center of hot pixel. Once we retrieve the trapezoid \( t \), we perform queries between the hot pixel and the segments using \( t \rightarrow W \).

If we store \( t \rightarrow W \) explicitly, we will need \( O(n^3) \) space (since there are \( O(n^2) \) trapezoids, each of which contains a list of \( O(n) \) segments). To save space, we use the following technique. Each trapezoid holds only two pointers: one to another trapezoid (denoted by \( p_1 \)) and the other to a segment (denoted by \( p_2 \)). Then the segments for each trapezoid can be retrieved by tracing trapezoids with \( p_1 \) and collecting the segments pointed to by \( p_2 \). Let \( \Gamma'_i \) be the temporary trapezoidal decomposition, after inserting the forbidden locus of the \( i \)-th segment. After inserting forbidden
locus \((i + 1)\), we can partition the trapezoids into three groups. The first group, \(G_1\), contains trapezoids which did not undergo any change during iteration \((i + 1)\). The second group, \(G_2\), contains new trapezoids whose \(W\) does not contain the forbidden locus of segment \((i + 1)\). The final group, \(G_3\), contains new trapezoids whose \(W\) contains the forbidden locus of segment \((i + 1)\). Then the values of \(p_1\) and \(p_2\) of each trapezoid will be determined in the following way. The data of the trapezoids in \(G_1\) will not change. \(p_1\) of any new trapezoid in \(G_3\) will point to a trapezoid in \(G_2\) and their \(p_2\) will point to segment \((i + 1)\). The idea is that trapezoids of \(G_3\) contain the same forbidden loci as \(G_2\) plus the new one of segment \((i + 1)\), so a trapezoid in \(G_3\) will point to a trapezoid in \(G_2\), whose \(W\) are equivalent except from the new forbidden locus. For each new trapezoid in \(G_2\), there is a trapezoid of \(\Gamma_1\) such that both have the same \(W\). We copy the data from the trapezoid in \(\Gamma_1\) to its corresponding trapezoid in \(G_2\). Since maintaining the pointers \(p_1\) and \(p_2\) can be done locally, it does not increase the asymptotic time complexity.

To conclude, for each hot pixel \(h\), we locate the trapezoid that contains it and find the segments as described above. These will be the set of segments whose forbidden loci contain \(h\). The working storage is linear in the number of trapezoids. Since we build a trapezoidal decomposition of arrangement of \(2n\) input trapezoids \((F_l(s, \delta)\) and \(F_r(s, \delta)\) for each segment \(s)\), this data structure requires \(O(n^2)\) working storage, \(O(n^2 \log n)\) time for preprocessing, and each query will take \(O(\log n + k)\) time where \(k\) is the output size.

The second trapezoidal decomposition is \(\Gamma_2\)—trapezoidal decomposition of the input segments \(S\). The purpose of this data structure is to provide an efficient query mechanism that locates the segment that is the closest to a hot pixel \(h\) inside \(\Delta(s, h)\). Then, a pixel on this segment will be heated (see Section 2.3.1 for details). The working storage and running time are the same as required for \(\Gamma_1\).

### Dynamic Simplex Range Searching

We use this data structure (denoted by \(\Psi\)) for two purposes. The first is to check whether a triangle is empty or not, to decide if a pixel should be heated (see Section 2.2 for details). The second use of \(\Psi\) is to query whether a segment intersects hot pixels (in \texttt{RemoveDegree2Vertices}; see Section 2.3.2). The technique for this query was introduced in [52]: we query hot pixel centers with polygons defined as the Minkowski sum of the queried segment and a unit pixel centered at the origin. We need dynamicity, as we need to support insertions and deletions of hot pixels. Let \(I_1\) and \(I_2\) be the number of original and new hot pixels respectively. Let \(I = I_1 + I_2\). We use the data structure by Agarwal and Matoušek [2]. For any \(\varepsilon > 0\), it requires \(O(I^{1+\varepsilon})\) space, the preprocessing time is \(O(I_1^{1+\varepsilon})\) and each update takes \(O(I^{\varepsilon})\) time. A general query takes \(O(\log I + k) = O(\log n + k)\) where \(k\) is the query result size. Since we need to query emptiness, the query time becomes \(O(\log n)\).

**Remark.** In practice, this data structure is difficult to implement. Instead, we use a \(kd\)-trees [31]. Since \(kd\)-trees query axis aligned rectangles, our query will use the axis-aligned bounding box of the polygons being queried, and filter out points that are not contained inside the polygons. We found this alternative very efficient in practice. \(kd\)-trees were also used to simplify the implementation in [52].

\(^2\)We ignore degenerate cases for the sake of clarity. Degenerate cases have different behavior, but they can be maintained by the above data structure in the same way traditional methods that use trapezoidal decomposition do.
2.4.3 Analysis

In this section we analyze the running time and working storage ISRBD requires. Let us review the notations that we use. \(n\) is the number of input segments. \(I_1\) is the number of original hot pixels and \(I\) is the total number of hot pixels (original and new). \(K\) is the list of degree-2 vertices that do not correspond to endpoints with size \(\kappa\).

We analyze the complexities for the new two procedures of ISRBD.

**Theorem 2.16.** GenerateNewHotPixels takes \(O(n^3(\log n + I^\varepsilon))\) time and \(O(n^2 + I^{1+\varepsilon})\) working storage for any \(\varepsilon > 0\).

**Proof.** We itemize the work done in this procedure:

- Building \(\Gamma_1\) and \(\Gamma_2\) takes \(O(n^2 \log n)\) time and the data structures require \(O(n^2)\) working storage.
- Building \(\Psi\) takes \(O(I_1^{1+\varepsilon})\) time and requires \(O(I_1^{1+\varepsilon})\) working storage for any \(\varepsilon > 0\).
- For each hot pixel we locate the trapezoid of \(\Gamma_1\) that contains it. Since there are \(I\) hot pixels, the total time is \(O(I \log n)\).
- Each original hot pixel \(h\) can be the source of heating \(O(n)\) new hot pixels, and during this process the total number of times they are within a forbidden loci is at most \(O(n)\). The reason is that each segment can be detected here at most \(O(1)\) times as our analysis in Section 2.4.1 indicates. Since there are \(O(n^2)\) original hot pixels, in total there are \(O(n^3(\log n + I^\varepsilon))\) pixel-segment pairs for which we do the following work.
  - (a) For hot pixel \(h\) and segment \(s\) we check if \(\Xi(s, h)\) is true (time \(O(\log n)\)).
  - (b) If there is a need to heat a pixel, we find the segment on which the pixel is heated (time \(O(\log n)\)).
  - (c) Update \(\Psi\) and the hot pixels queue \(Q\) if a pixel is heated (time \(O(I^\varepsilon)\)). Thus, the total time required for this item is \(O(n^3(\log n + I^\varepsilon))\).

Together, GenerateNewHotPixels requires \(O(n^3(\log n + I^\varepsilon))\) time and \(O(n^2 + I^{1+\varepsilon})\) working storage for any \(\varepsilon > 0\) (note that \(I = O(n^3))\). \(\square\)

**Theorem 2.17.** RemoveDegree2Vertices takes \(O(\kappa^2 \log n + \kappa I^\varepsilon)\) time for any \(\varepsilon > 0\).

**Proof.** Each degree-2 hot pixel which does not correspond to endpoints can be processed either when first traversed in \(K\) or after another hot pixel, which contains it in its list \(W\), was removed. Thus, each such hot pixel may be tested \(\kappa\) times, resulting in \(O(\kappa^2)\) total tests. Each query takes \(O(\log n)\) time. Each removal of a pixel from \(\Psi\) takes \(O(I^\varepsilon)\) for any \(\varepsilon > 0\). There are at most \(\kappa\) such removals. Together, RemoveDegree2Vertices takes \(O(\kappa(\kappa \log n + I^\varepsilon))\) time for any \(\varepsilon > 0\). \(\square\)

The next theorem summarizes the work of ISRBD.

**Theorem 2.18.** Given an arrangement of \(n\) segments, Iterated Snap Rounding with Bounded Drift requires \(O(n^3(\log n + I^\varepsilon)) + \kappa(\kappa \log n + I^\varepsilon) + L^2 I^{3+\varepsilon} + L)\) time, and \(O(n^2 + I^{1+\varepsilon} + L^2 I^{3+\varepsilon})\) working storage for any \(\varepsilon > 0\), where \(L\) is the overall number of links in the chains produced by the algorithm, \(I\)
is the total number of pixels produced by the algorithm and \( \kappa \) is the number of degree-2 hot pixels which does not correspond to segment endpoints.

Proof. The complexities are obtained by summing up the time and working storage in Theorems 4.2 and 4.3 with the complexity of ISR [52].

### 2.4.4 Determining the Maximum Deviation Parameter

The maximum drift \( \delta \) is a user-specified parameter. However, it cannot be arbitrarily small; clearly it must be at least the maximum size a segment can be rounded each time \( (\sqrt{2}/2) \). However, we stated in Section 2.3.1 that \( \delta > \frac{3\sqrt{2}}{2} \), so this is the constrain a user must follow when choosing \( \delta \). Thus, the minimum value for \( \delta \) is about three times the maximum deviation of SR \( (\sqrt{2}/2) \).

Clearly, there is an advantage in using smaller \( \delta \) values with which the geometric similarity is better. However, our experiments (Section 6) indicate that bigger values usually reduce the number of new pixels being heated. This is logical, since larger values of \( \delta \) results in larger \( \Delta(s, h) \), which have a higher probability of containing hot pixel centers that block other segments from snapping into hot pixels. Thus, we have a trade off when determining \( \delta \). An interesting direction for future work is to establish solid optimization criteria for choosing \( \delta \).

### 2.5 Desirable Properties

Recall that \( A(S) \) is the arrangement of a set of planar segments \( S \), and we define \( A^*(S) \) to be the output obtained by either rounding method, depending on the context. Recently, de Berg et al. [29] listed four desirable properties for the output SR produces. We next explore these properties, fit them to ISR and ISRBD, and formulate a new one.

Two of the properties listed in [29] hold for the output of ISR and ISRBD as well. We list them below.

Property 2.19. Fixed-precision representation: All vertices of \( A^*(S) \) are at centers of grid squares.

Proof. It follows from the way the rounding stage performs (recall that the same rounding procedure is used for both ISR and ISRBD). Since its results are equivalent to a finite series of rounding steps of SR (in which this property clearly holds), all vertices of \( A^*(S) \) are at centers of grid squares.

Property 2.20. Topological similarity: There is a continuous deformation of the segments in \( S \) to their snap-rounded counterparts such that no segment ever crosses completely over a vertex of the arrangement.

Proof. Once the set of hot pixels is fixed (after GenerateNewHotPixels), the rounding stage will satisfy this property [48].

46
The other two properties are modified in the context of ISR and ISRBD. The first one holds for ISRBD only. It follows from Theorem 3.1 and the discussion in Section 2.3.1.

**Property 2.21.** Geometric similarity: For each input segment \( s \in S \), its output lies within the Minkowski sum of \( s \) and a disc with radius \( \delta \) centered at the origin. \( \delta \) is determined by the user and must be larger than \( \frac{3\sqrt{2}}{2} \) units.

Note that in ISR the distance may be as large as \( \Theta(n^2) \). The second modified property holds with modifications for both ISRBD and ISR. It differs in each. The correctness follows from the discussion in Section 2.3.2. We list them next.

**Property 2.22.** Non-redundancy (ISRBD version): Any degree-2 vertex corresponds to either an endpoint of an original segment or to a vertex whose removal violates either the property of the separation between a vertex and a non-incident edge or the geometric similarity property.

Since the geometric similarity in the output of ISR may be poor, we modify the above property for its sake.

**Property 2.23.** Non-redundancy (ISR version): Any degree-2 vertex corresponds to either an endpoint of an original segment or to a vertex whose removal violates the property of the separation between a vertex and a non-incident edge.

In addition, we formulate another property that holds for both ISR and ISRBD (but not for SR). It was proved in [52] for ISR and holds for ISRBD since the same rounding stage is performed in both. This property is desirable for robust usage of the rounded arrangement. (see Section 2.1 for more details on this issue.)

**Property 2.24.** Separation between a vertex and a non-incident edge: The distance between a vertex and a non-incident edge is at least half of a unit-pixel.

This property, together with the fixed-precision representation property, establish good robustness criteria for the output in order to avoid errors due to floating-point limitations.

To conclude, we see that ISRBD is the first method to satisfy both the desirable properties listed in [29] and the one we formulated above.

### 2.6 Experimental Results

We implemented the algorithm using the CGAL library [21]. We tested the algorithm with many examples and present two here. For each, we display the input, output and the new hot pixels. We also provide statistics that help with understanding the quality of the output. Our conclusions from the experiments follow.

#### 2.6.1 Randomized Input

The idea behind this experiment is to have congested input with many intersections—thus creating many hot pixels. The segment coordinates are chosen randomly inside a fixed box. Our tests
were controlled by the number of segments and $\delta$. We performed many tests for each such pair. Figure 2.10(a) and (b) illustrates a small example with 20 random segments. In this example, two pixels were heated in $\text{GenerateNewHotPixels}$. Note that the bottom new hot pixel was later removed by $\text{RemoveDegree2Vertices}$. The data in Table 1 imply the following. First, the number of new hot pixels was relatively small and decreased as $\delta$ increased. Second, we observed an increase of $29 - 42\%$ in the average total time in comparison to ISR. The largest average number of new hot pixels was produced with 300 segments. This is because more segments produced denser original hot pixel sets which blocked segments from potential snapping into hot pixels, while less segments had a sparse set of original hot pixels, thus less new ones were introduced.

### 2.6.2 Visibility Graph of a Polygon

We tested several examples of visibility graphs of polygons created randomly (we used the implementation of [11] for creating the polygons). We believe that such examples can be interesting since they may have many intersections in small area. Our tests were controlled by the number of edges in the polygons and $\delta$. Table 2 summarizes the results, and Figure 2.10(c) and (d) illustrates an example of a polygon with 31 segments. In this case, only one pixel was heated in $\text{GenerateNewHotPixels}$ (it was later removed by $\text{RemoveDegree2Vertices}$). Our conclusions from this experiment is similar to the random segment experiment in both the number of new hot pixels and the extra time needed in ISRBD in comparison to ISR.

### 2.6.3 Experimental Conclusions

We tested many examples, each with a variety of parameter values. Two experiment sets were presented here. In all examples, we observed similar behavior:

- The number of new hot pixels produced in $\text{GenerateNewHotPixels}$ was very small in comparison to the number of original hot pixels. This is a positive indication for the output of ISRBD, since one generally prefers less hot pixels and fewer links in the output.

- The extra time required for ISRBD in comparison to ISR did not increase the order of the processing time.

- The majority of the pixels heated by $\text{GenerateNewHotPixels}$ were removed in $\text{RemoveDegree2Vertices}$ (see Section 2.3.2). Thus, it is useful to perform this optimization since it improves the output quality.

### 2.7 Extending ISR and ISRBD

Some of the publications that were devoted to SR offered extensions. Guibas and Marimont [48] showed how to support dynamic rounding (the ability to insert and delete line segments after the Snap Rounding representation has been computed). Goodrich et al. [45] showed how to deal with arrangements of line segments in $\mathbb{R}^3$ and Fortune [38] generalized the input to polyhedral subdivisions.
Figure 2.10: Experiment snapshots obtained with our software: 20 random segments ((a) input, (b) output before removing redundant vertices, (c) final output) and a polygon with its visibility graph ((d) input, (e) output before removing redundant vertices, (f) final output). The new hot pixels are white and the hot pixels that are responsible for their heating are shaded.
Figure 2.11: Results of SR, ISR and ISRBD. All the squares in sub figures (b),(c), (d) and (f) are hot pixels. The line segment which demonstrates the difference between the methods is marked in bold. (a) Input (b) SR output (c) ISR output (d) ISRBD output. The effect of deleting/inserting a line segment is illustrated in sub figures (e) and (f). In (e), an input line segment is deleted. As a result, the output of the bold line segment changes, both in ISR and ISRBD. The result (same in ISR and ISRBD) is illustrated in (f). Alternatively, one can view this process as adding a new line segment if the original input is as in (e).
### Table 2.1: Results of random segment tests. Each result is in the form $X/Y$ where $X$ is the average number of new hot pixels in the output and $Y$ is the average extra time ISRBD needed in comparison with ISR (in %).

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>200</th>
<th>300</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.17/35</td>
<td>0.23/32</td>
<td>0.22/37</td>
<td>0.14/36</td>
</tr>
<tr>
<td>6</td>
<td>0.28/40</td>
<td>0.63/35</td>
<td>0.37/35</td>
<td>0.3/31</td>
</tr>
<tr>
<td>4</td>
<td>0.34/38</td>
<td>0.92/34</td>
<td>0.6/29</td>
<td>0.47/36</td>
</tr>
<tr>
<td>3</td>
<td>0.79/39</td>
<td>1.29/42</td>
<td>0.84/32</td>
<td>0.63/35</td>
</tr>
<tr>
<td>2</td>
<td>0.97/33</td>
<td>1.6/37</td>
<td>1.05/35</td>
<td>0.79/39</td>
</tr>
</tbody>
</table>

### Table 2.2: Results of polygon visibility graph tests. The format is equivalent to the one in Table 1.

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.5/30</td>
<td>1.2/35</td>
<td>2.3/35</td>
<td>3.5/38</td>
</tr>
<tr>
<td>6</td>
<td>0.6/35</td>
<td>1.5/37</td>
<td>2.6/34</td>
<td>3.7/33</td>
</tr>
<tr>
<td>4</td>
<td>1.0/41</td>
<td>1.8/40</td>
<td>2.0/32</td>
<td>4.0/33</td>
</tr>
<tr>
<td>3</td>
<td>0.8/39</td>
<td>1.6/40</td>
<td>1.8/38</td>
<td>4.2/41</td>
</tr>
<tr>
<td>2</td>
<td>0.5/33</td>
<td>1.4/38</td>
<td>1.8/40</td>
<td>3.8/39</td>
</tr>
</tbody>
</table>

In this section we propose similar extensions for both ISR and ISRBD. This work was presented in the 19th Canadian Conference on Computational Geometry [73].

#### 2.7.1 Dynamic Rounding

Guibas and Marimont [48] showed how to maintain hierarchical vertical cell decomposition of line segments in $\mathbb{R}^2$ (VCD) [66] with Snap-Rounded representation. They further showed how to support insertions and deletions of line segments. We use the VCD analogously, but will not attempt to describe how we use it in detail. This information is mainly technical and will be described in a detailed report.

Let $S$ be the set of input line segments and let $\mathcal{A}(S)$ be the output arrangement of $S$ (in either variant). We next show how ISR and ISRBD proceed.

**Dynamic Iterated Snap Rounding**

**Insertion.** Let $s$ be the line segment we insert. We find all the pixels that need to be heated as a result of inserting $s$: one or two for its endpoints and others that correspond to intersections with the input line segments (these are found with the VCD). We collect only the above pixels that are not hot yet. Let $H$ denote this set. Except of rounding $s$, we also round any link of $\mathcal{A}(S)$ that intersects the pixels in $H$; we find these links (using the VCD) and use the rounding routine of ISR to round links [52].

51
Deletion. We delete a line segment $s$ from $A(S)$. By doing so, it is possible that hot pixels cool down (cease to be hot). These are the hot pixels that would not have been heated if $s$ had not been inserted to $A(S)$. This information can be easily saved under the hot pixel data structure. Then we process each cooling of a hot pixel by modifying the output of some of the input line segments as follows. Let $s'$ be any line segment. We construct a hierarchical tree of links for $s'$, denoted by $T(s')$. In the first level, $T(s')$ contains $s'$. In the second, $T(s')$ contains the links through the center of hot pixels that $s'$ intersects in the order of intersections and so on. Note that the output of $s'$ is the links at the bottom of $T(s')$, ordered by the DFS. Let $h$ be a hot pixel that is cooled down. Let $l$ be the top link in $T(s')$ that intersects with $h$ but does not end there (assume that such a link exists). Consider the two links $l_1$ and $l_2$, one level below $l$, that are adjacent to $h$. We first delete both subtrees of $l_1$ and $l_2$ from $T(s')$. Let $h_1$ and $h_2$ be the hot pixels that contain the other endpoints of $l_1$ and $l_2$, respectively. We insert the link that connects the centers of $h_1$ and $h_2$ to $T(s')$ by calling the rounding routine of ISR. The new representation of $s'$ will correspond to the modified $T(s')$.

Let $A$ be complexity of $A(S)$. Let $H$ denote the set of hot pixels and let $W$ denote the set of pixels that either contain an endpoint of a vertical attachment in the VCD or is a neighbor of such a pixel. For any pixel $p$, its degree will be denoted by $|p|$. Note that this quantity may be larger in ISR than in SR, as links are broken to create multiple ones. The following is a theorem that bounds the total insertion time of $n$ line segments. It follows from the analysis in [48].

**Theorem 2.25.** Performing $n$ insertions using Dynamic Iterated Snap Rounding takes $O(n \log n + A + \sum_{h \in H} |h| \log |h| + \sum_{w \in W} |w|)$ time.

**Dynamic Iterated Snap Rounding with Bounded Drift**

**Insertion.** The insertion of line segments is similar to the insertion in the previous subsection, with the modifications that ISRBD requires (see Section 5.1). When inserting a line segment $s$, we proceed as follows. We find the hot pixels that are centered within the forbidden loci of $s$. For each, we heat a corresponding pixel as explained in [75]. This process may cascade as the heated pixel may be centered within a forbidden loci of another line segment. Then we round $s$ and the links that intersect with the hot pixels in the way ISR proceeds (see the previous section). Finally, we delete some of the degree-2 hot pixels similarly to ISRBD (see section 5.1 and [75] for details).

**Deletion.** When deleting a line segment $s$, we need to do both the work of the deletion that is presented in Section 2.1 and the work required by ISRBD which we describe next. For each line segment, we collect the new pixels that were heated before to constrain its deviation (denote this set by $\Xi$). The pixels of $\Xi$ have to be cooled down when deleting $s$, unless they were used to constrain the deviation of other line segments as well. As we mentioned above, the process of heating pixels may cascade. Thus, we also need to delete all the pixels that were heated as a result of cascading from the pixels of $\Xi$ (we save the corresponding data). After all of the hot pixels are cooled down, we proceed in the same way as the removal of ISR, namely reconstructing the hierarchy of the output of the line segments that are effected by this process.

Figure 2.11 illustrates the results of inserting and deleting a line segment.

The following theorem bounds the total insertion time of $n$ line segments. It is mostly dominated by the routines of ISRBD. In this case $H$ denotes the set of all hot pixels, including the ones ISRBD heats to constrain the deviation of the output.
Theorem 2.26. Performing $n$ insertions using Dynamic Iterated Snap Rounding with Bounded Drift takes $O(n^3(\log n + |\mathcal{H}|^2) + \kappa(\kappa \log n + |\mathcal{H}|^2) + \Sigma_{w \in \mathcal{W}}|w|)$ time for any $\varepsilon > 0$ where $\kappa$ is the number of degree two vertices that do not correspond to endpoints.

2.7.2 Line Segments in $\mathbb{R}^3$

Goodrich et al. [45] extended SR to line segments in $\mathbb{R}^3$. In $\mathbb{R}^3$, the space is partitioned into a set of voxels (unit-cubes) centered at points with integer coordinates. Similarly to $\mathbb{R}^2$, voxels which contain line segment endpoints are hot. The rest of the hot voxels are determined as follows. For each pair of line segments $s_1$ and $s_2$, find $l = (p_1, p_2)$, the shortest link that connects them with length $d$ in the $L_\infty$ metric. If $d \leq 1$, we heat the voxels that contain $p_1$ and $p_2$ (possibly the same voxel contains both). To test whether two line segments are close, they define a tube $\tau(s)$ to be the Minkowski sum of $s$ and an axis-oriented unit cube centered at the origin. It is easy to verify that two line segments are within an $L_\infty$ distance of 1 if and only if their tubes intersect. To report the tubes that intersect, they use a range searching technique for semi-algebraic varieties [3] (denoted by $\phi$). The main idea is to define a search structure of the faces of the tubes that finds all the segments of other tubes that penetrate the faces. We use $\phi$ for the same purpose. Then the output of any line segment is the polygonal chain through the centers of the hot voxels it meets, in the order of appearance. We next present how ISR and ISRBD proceed.

Iterated Snap Rounding in $\mathbb{R}^3$

We first find the hot voxels using $\phi$. Then we round links to hot voxels in the way ISR proceeds [52], with the analogous changes. To find the hot voxels that intersect a given line segment, we use the multi level partition tree data structure [4] on the hot voxels. We project the hot voxels onto the three planes $x = 0$, $y = 0$ and $z = 0$, and build the multi level partition tree for each. Then an $\mathbb{R}^3$ query is carried out by intersecting the query results of the three $\mathbb{R}^2$ trees; each queries the projection of the line segment.

The next theorem summarizes ISR in $\mathbb{R}^3$. It follows from analyzing the algorithm of ISR with the necessary modifications we described above.

Theorem 2.27. Iterated Snap Rounding of line segments in $\mathbb{R}^3$ can be computed in $O(n^{3+\varepsilon} + K + L^{3+\varepsilon} + N)$ time and $O(n^{3+\varepsilon} + K + L^{3+\varepsilon} + N)$ space for any $\varepsilon > 0$ where $K$ denotes the number of intersecting tubes, $L$ is the overall number of links in the chains produced by the algorithm and $N$ is the number of hot voxels.

Iterated Snap Rounding with Bounded Drift in $\mathbb{R}^3$

Let $\delta$ be the deviation bound. We first decide which voxels to heat in order to restrict the deviation of any line segment. We then cool down degree-two hot voxels which are both not associated with endpoints and will not cause line segments to drift too much if deleted (analogous to the work in [75]).

Note that in $\mathbb{R}^2$ we define forbidden loci to be the two stripes to the left and right of a line segment $s$, that necessarily contain the center of any hot pixel that may be the first to round $s$ too far. In $\mathbb{R}^3$, the forbidden loci are constructed analogously as follows. For any line segment $s$, let
Let $v_1$ and $v_2$ be the voxels that contain its endpoints. Let $B(s)$ be the bounding box of the centers of $v_1$ and $v_2$. Due to the weakly-monotone property, the output of $s$ will be located within $B(s)$. Let $C(s)$ be the Minkowski sum of $s$ with a ball of radius $\delta$ centered at the origin. It follows that the output of $s$ must be located within $D(s) = B(s) \cap C(s)$. Note that the maximum deviation any link may undergo during one rounding step is $\sqrt{3} \Delta_2$ units. It follows that if any hot voxel rounds the output of $s$ beyond $D(s)$, it must be centered at a point that is within distance $0 < d \leq \frac{\sqrt{3} \Delta_2}{2}$ from $D(s)$, and inside $B(s)$. We define $C'(s)$ to be the Minkowski sum of $s$ with a ball of radius $\delta + \frac{\sqrt{3} \Delta_2}{2}$ centered at the origin. It follows that the forbidden loci of $s$ is $F(s) = (C'(s) \setminus C(s)) \cap B(s)$ (see Figure 2.7.2 for an illustration).

The routine for heating voxels that restrict the deviation proceeds as follows. For each hot voxel $v$, we find the set of line segments whose forbidden loci contain its center. Let $s$ be a line segment in this set. We heat a voxel which both intersects $s$ and will guarantee that $s$ will not be rounded to $v$. There are three questions to address at this point.

First, which voxel do we heat on $s$? For each voxel $v$, we partition $\mathbb{R}^3$ into 8 octants using the three orthogonal walls through the center of $v$. Ignoring degenerate cases (which are not difficult to handle), $s$ will pass through three or four octants. Let $\Psi(s, v)$ be an octant that does not contain any of the endpoints of $s$ and let $s' = s \cap \Psi(s, v)$. We heat the voxel which contains the middle of $s'$. By heating it we are guaranteed that $s$ will never be rounded to $v$; otherwise it would contradict the weakly-monotone property\(^3\). Note however, that if there is already a hot voxel that intersects $s'$, such that its 3 coordinates differ from the corresponding coordinates of $v$, we need not heat any

\(^3\)Analogously to [75], we require $\delta > \frac{3\sqrt{3} \Delta_2}{2}$ to make this idea works. We postpone this discussion to the detailed report.
The second question is how to find the line segments whose forbidden loci contain the center of $v$. It would be desirable to use a range search data structure of $\{F(s) | s \in S\}$. However, to the best of our knowledge such a data structure is yet not available in the literature. So instead we will test each pair of line segment-hot voxel.

Next we ask what would be the complexity of the output. In [75], the selection of pixels to heat for restricting the deviation was such that the output complexity of ISRBD is not increased\(^4\). It is not clear how to do the same in $\mathbb{R}^3$ and whether it is possible. We leave this task for a future work. We believe that the output complexity does not increase unless the input is pathologic, if at all. We note that the number of new voxels is finite as we deal with finite input of line segments.

The next theorem summarizes ISRBD in $\mathbb{R}^3$. It follows from analyzing the algorithm of ISRBD in $\mathbb{R}^2$ with the necessary modifications we described above.

**Theorem 2.28.** Iterated Snap Rounding with Bounded Drift of line segments in $\mathbb{R}^3$ can be computed in $O(n^3 + K + \frac{L}{\varepsilon} N \frac{2}{3} + L + \kappa (\log n + N^\varepsilon))$ time and $O(n^3 + L^2 N \frac{2}{3} + N + K^{1+\varepsilon})$ space for any $\varepsilon > 0$. The parameters were defined in Theorem 3.

### 2.8 Conclusions and Future Work

We presented an algorithm that rounds an arrangement of segments in 2D which follows Snap Rounding (SR) and Iterated Snap Rounding (ISR). It augments ISR with two simple and efficient procedures. The contribution of this algorithm is that it prevents problematic output that may be produced by SR or ISR. These occur in SR when there is a very small distance between vertices and non-incident edges, and in ISR when there is a large deviation of an approximating polygonal chain from the corresponding original segment. We eliminate these problematic features by introducing new hot pixels to ISR. We showed that ISRBD produces efficient output both in theory and in experimental evaluations. We also showed that ISRBD satisfies all the properties discussed in the literature so far, plus another one we formulated in this chapter. Our experiments showed that relative to the number of original hot pixels, a very small number of new hot pixels is produced by our algorithm, and most of them are removed as they induced degree-2 vertices that meet certain conditions. The extra time required by our algorithm in comparison to ISR did not increase the order of the processing time. Based on these results, we believe that ISRBD is a good option for creating snap-rounded-like arrangements.

We propose a few directions for future research. First, we would like to further investigate the output complexity of our algorithm. We proved a tight output complexity, similar to SR and ISR, but there is still a gap when overlapping segments are counted only once. Since we experienced a small number of new hot pixels with our technique, it is also tempting to theoretically establish the upper bound and conditions for the amount of new hot pixels.

In Section 2.3.2 we showed how to remove redundant degree-2 vertices in order to simplify the output. It seems more efficient to avoid heating the corresponding pixels from the start. Thus, an interesting challenge would be to devise an efficient technique for identifying the redundancy of a pixel before heating it in *GenerateNewHotPixels*.

---

\(^4\)It is still open if this is the case when links that are shared by multiple polygonal chains are counted only once.
We explained the trade off in choosing $\delta$ (better geometric similarity vs. less new hot pixels). It would be interesting to establish good criteria for choosing $\delta$, and possibly choose the value automatically based on the input.
Chapter 3

Controlled Perturbation of Sets of Line Segments in $\mathbb{R}^2$ with Smart Processing Order

Controlled Perturbation is a framework for perturbing geometric sets to make the processes that use them more robust for fixed-precision manipulation. We present a Controlled Perturbation scheme for sets of line segments in $\mathbb{R}^2$ (CPLS, for short). CPLS iteratively perturbs the endpoints of the line segments to eliminate degeneracies that may cause round-off errors when using fixed-precision arithmetic. We implemented CPLS and provide experimental results.

In the core of this work, we present a novel method for decreasing the perturbation magnitude. The main idea behind our method is that different endpoint processing orders yield different perturbation quality. We devise several heuristics for deciding smart endpoint processing to decrease the perturbation. We implemented and experimented with them. Our experiments show a significant decrease in the perturbation magnitude.

3.1 Introduction

Geometric applications compute geometric predicates whose values are used as conditions in branch statements in the program (such as if-then-else statements)\(^1\). In the majority of cases, the sign of the computation (positive, negative or zero) determines the branching. It follows that fixed-precision arithmetic may be problematic when the computation values are near zero: in such cases, the fixed-precision round-off may change the sign of the computation, leading to wrong and inconsistent branching, and consequently to program failures.

A Degeneracy is defined as a case where a predicate evaluates to zero\(^2\). It follows that using fixed-precision arithmetic is risky at the presence of degenerate or close-to-degenerate cases. We refer to these cases as potential degeneracies since it is usually impossible to tell whether a case is degenerate or not with fixed-precision arithmetic.

\(^1\)For example, a predicate can ask whether a point lies to the right or left of a line and based on the result branches to the desired case.

\(^2\)Degeneracies can be viewed in this context as special geometric situations such as three lines intersecting at the same point or two line segments sharing an endpoint.
We present a scheme that approximates sets of line segments in $\mathbb{R}^2$ with fixed-precision number types. The main idea is to incrementally perturb the line segment endpoints to remove some of the potential degeneracies (and thus provide safer predicate evaluations). The idea follows a framework that has been presented in several publications [39, 51, 53, 63, 80] and was termed Controlled Perturbation (CP for short). In this framework, potential degeneracies that correspond to close proximities (such as a close proximity between a vertex and a non-incident line segment) are eliminated.

Applications that will work on our approximated sets will benefit from the efficiency of fixed-precision arithmetic and yet will produce more robust results since many potential degeneracies will be eliminated.

The input to our scheme is an undirected geometric graph in the Euclidean metric, $G = (V, E)$. $V$ is the set of endpoints and $E \subseteq V \times V$ is the set of endpoint pairs, that essentially defines the line segments. Our scheme perturbs some of the endpoints of $V$ so that all close-proximity potential degenerate cases but one are eliminated. The only degenerate case that we allow is the case when multiple line segments share an endpoint. Note that corresponding predicate evaluations give robust results with fixed-precision arithmetic: the test for line segment incidence is simply carried out by comparing the coordinates of endpoints; if the coordinates are equal, one will safely deduce that the case is degenerate. We use the term *definite-degeneracy* to refer to this degenerate case and do not regard it as a potential degeneracy. Note that by using this strategy we maintain the combinatorial structure of $G$ (all vertices continue to be incident to the same edges as before). Supporting line segment incidence potentially enables using our scheme with popular geometric data structures such as triangulations and Voronoi diagrams (see Section 3.7 for examples). We note that in previous CP schemes, definite-degeneracies were either not supported or supported partially.

We use the term *resolution bound* to define the minimum absolute values that evaluations of predicates must have in order to overcome round-off errors with the given (limited) precision (namely, still guarantee that round-off errors will not change the sign of the predicates). In our context, a resolution bound (with respect to a specific predicate) is defined as the minimum separation between features that guarantees safe predicate computation\(^3\). Note that there are different resolution bounds for different kinds of predicates. The different resolution bounds correspond to different kinds of potential degeneracies and usually depend on the machine-precision and the input. The main idea of our scheme is to perturb the endpoints of separate features to at least their corresponding resolution bound.

A major drawback of typical CP schemes is the possible large perturbation it may perform. It is evident that in complicated and congested regions, that naturally contain many potential degeneracies to resolve, the perturbation tends to increase. In such cases, the output may be unsatisfactory for further use. Thus, developing methods to constrain the perturbation magnitude is highly desired. In this work we focus on a novel method to decrease the perturbation magnitude. It is based on determining the processing order of the endpoints intelligently. We designed several variants which we call *sorting algorithms*. We implemented them and report related experimental results in Section 3.7.

\(^3\)In this work we consider predicates which can be formulated in terms of distances (as previous Controlled Perturbation did). It should be noted that in general a safe predicate evaluation is not automatically guaranteed if the separation between features is large enough.
Related Work. Earlier perturbation schemes can be found at [1, 9, 49, 69]. Halperin and Shelton [53] were the first to introduce Controlled Perturbation. They worked on arrangements of spheres in $\mathbb{R}^3$ that support geometric queries on molecular models. Raab [80] followed by proposing Controlled Perturbation of polyhedral surfaces in $\mathbb{R}^3$ to eliminate degeneracies in swept volume applications. Halperin and Leiserovich [51] described a framework for circles in $\mathbb{R}^2$. They were the first and the only one so far to actually compute the resolution bounds instead of setting them as parameters. Funke et al. [39] used Controlled Perturbation in randomized incremental constructions, and designed specific schemes for planar Delaunay triangulations and convex hulls in arbitrary dimensions. Mehlhorn et al. [63] extended this work and developed a general methodology for deriving quantitative relations between the amount of perturbation and the precision of the approximate arithmetic.

The rest of the paper is organized as follows. In the next section we discuss the main ideas of our scheme and present the algorithm. In Section 3.3 we present the potential degeneracy cases that we handle and compute an upper bound on the perturbation magnitude. In Section 3.4 we discuss the sorting algorithms. In Section 3.5 we present optimization techniques to decrease the perturbation and the running time. We devote Section 3.6 to compare with other geometric robustness techniques. In Section 3.7 we present experiments performed with our implementation. In Section 3.8 we investigate a graph problem that is related to the problems we handle in Section 3.4. We conclude and present ideas for future research in Section 3.9.

3.2 Main Ideas and Algorithm

Our perturbation scheme processes the endpoints of $V$, one at a time, by possibly perturbing them to eliminate potential degeneracies. The processing order is determined by the sorting algorithm we choose (see Section 3.4). Let $\Pi(V) = (v_1, v_2, \ldots, v_n)$ be the list of endpoints obtained with one of the sorting algorithms (where $n$ is the number of endpoints in the input).

The output of an endpoint $v_i : 1 \leq i \leq n$ is denoted by $v'_i$ and inserted to the intermediate endpoint output $V'_i = \{v'_1, v'_2, \ldots, v'_i\}$ by setting $V'_i = V'_{i-1} \cup \{v'_i\}$. A line segment $e \in E$ is considered processed only after both of its endpoints have been processed. Let $E'_i = \{[v'_j, v'_k] : j, k \leq i$ and $[v_j, v_k] \in E\}$ be the set of line segments that have been processed by the time $i$ endpoints are fully processed. Let $G'_i = (V'_i, E'_i)$. Let $A'_i$ be the arrangement induced by $G'_i$. Let $E(v_i) = \{[v_i, v'_j] : j < i$ and $[v_i, v_j] \in E\}$ be the incident line segments of $v_i$ whose other endpoint was processed before $v_i$. Let $E'(v_i) = \{[v'_i, v'_j] : j < i$ and $[v_i, v_j] \in E\}$. It follows that $v_i$ induces potential degeneracies if $v_i$ or any $e \in E(v_i)$ are involved in potential degeneracies with $A'_{i-1}$ (that is, either with vertices or edges of $A'_{i-1}$). In practice, we test if $v_i$ is initially placed such that potential degeneracies are induced with $A'_{i-1}$. If this is not the case, $v_i$ is simply inserted into $V'_i$ (by setting $V'_i = V'_{i-1} \cup \{v_i\}$). Otherwise, $v_i$ is perturbed to a degenerate-free placement and inserted into $V'_i$. After the processing of $v_i$ is complete, its placement is fixed and it will not be perturbed again.

In this fashion we incrementally build the intermediate output $G'_i$. By a simple induction argument, $G'_i$ contains no potential degeneracies for each $1 \leq i \leq n$. After processing all endpoints, we obtain the output $G' = (V', E')$ where $G' = G'_n$, $V' = V'_n$ and $E' = E'_n$.

We next describe the perturbation process of a single endpoint $v$ in detail. We subdivide $\mathbb{R}^2$ into two regions (each of which is not necessarily connected). The first, $F(v)$, contains all the points that are forbidden to $v$—if $v$ lands on them, potential degeneracies are induced as described above.
We call $F(v)$ the forbidden loci of $v$. The second region, $\bar{F}(v) = \mathbb{R}^2 \setminus F(v)$, contains valid locations for $v$ to land. We say that these locations are degeneracy-free with respect to $v$. After $v$ is processed, $v'$ must lie within $\bar{F}(v)$.

Each endpoint $v$ and a parameter $\delta$ (the perturbation radius) defines the perturbation disc $B_\delta(v)$. $B_\delta(v)$ is centered at $v$ and has a radius $\delta$. In order to resolve potential degeneracies (if they exist), $v$ is perturbed randomly inside $B_\delta(v)$ once or more, until a degeneracy-free placement is detected. In order to find degeneracy-free placement efficiently, $\delta$ should be large enough so that $B_\delta(v)$ will necessarily contain relatively large areas from $\bar{F}(v)$ (note that the area of $F(v)$ is bounded). On the other hand, we want to constrain the size of $\delta$ to minimize the perturbation magnitudes. Thus, $\delta$ is a trade-off between the perturbation magnitude and the computation efficiency. In Section 3.3 we compute a suitable $\delta$ for our scheme. See Figure 3.1 for an illustration of the perturbation step.

![Figure 3.1: Perturbing an endpoint $v$ to eliminate potential degeneracies. The large disc is $B_\delta(v)$. The portion of $F(v)$ inside $B_\delta(v)$ is shaded and corresponds to an endpoint $u'$ and a line segment $e'$. Since the initial placement of $v$ is inside a forbidden locus, it has to be perturbed to a degeneracy-free locus (one valid perturbation is arrowed).](image)

Let $F$ be an upper bound on the forbidden locus area induced by all the potential degeneracies that we handle (see Figure 3.2 for an illustration). $F$ is given by summing the cases of individual forbidden loci associated with the different types of degeneracies. Let $\varphi = \frac{|B_\delta(v)| - F}{|B_\delta(v)|}$ be a lower bound on the probability that $v$ will be placed within a degenerate-free placement, if perturbed randomly inside $B_\delta(v)$. Since our perturbations are independent, $Q = \frac{1}{\varphi}$ is an upper bound on the average number of trials that are required to find a potential degeneracy-free placement for $v$ (if $v$ needs to be perturbed). In this work we compute $\delta$ such that $\varphi = \frac{1}{2}$. Thus, if an endpoint is placed within a forbidden locus, we need at most two trials on average to find a degeneracy-free placement for $v$ and the probability for larger number of trials decreases exponentially. Section 3.3 contains a detailed description of the computation of $\delta$.

Following previous CP schemes, we handle potential degeneracy types that involve separation of geometrically close features. By doing that, we guarantee robust construction of the output [51] and make subsequent algorithms more robust. In our case we have two families of potential degeneracies: a potential degeneracy between a vertex (an endpoint or an intersection point) and a non-incident line segment and a potential degeneracy between two vertices. In section 3.3 we analyze all possible ways that these potential degeneracy families can be induced and compute an upper bound of their forbidden locus area.

Our algorithm consists of three steps. We first compute $\delta$ (see Section 3.3). Then we decide the endpoint processing order, $\Pi(V)$, using one of the sorting algorithms (see Section 3.4). In the last step, we process the endpoints to obtain the output. Let $D(v)$ be a predicate that is true if an endpoint $v$ induces potential degeneracies. (we compute $D(v)$ by testing all possible
Figure 3.2: The potential degeneracy cases that we eliminate. Each subfigure shows the degeneracy and a possible valid perturbation. The closed curve in each represents the corresponding forbidden locus. The result of the perturbation is illustrated with an arrow that indicates the object before and after perturbation. (a) A degeneracy between two endpoints (b) A degeneracy between an endpoint and a non-incident line segment (c) A degeneracy between an intersection point and a non-incident line segment. (d) A degeneracy between an endpoint and a non-incident line segment. Note that here one of the line segment endpoints is perturbed as opposed to the case in (b) where the single endpoint is perturbed. The case that is used depends on the endpoint processing order.

degeneracies that \( v \) may induce—those are listed in Section 3.3.) We are ready to introduce a high level pseudocode of CPLS.

**Controlled Perturbation of Sets of Line Segments in \( \mathbb{R}^2 \)**

**Input:** \( G = (V, E) \)

**Output:** \( G' = (V', E') \)

1. Compute the perturbation radius \( \delta \) (see Section 3.3)
2. Compute a processing order \( \Pi(V) \) (see Section 3.4)
3. \( V_0' = \emptyset, E_0' = \emptyset \)
4. **foreach** endpoint \( v_i \in V \) in the order of \( \Pi(V) \)
5. let \( v_i' = v_i \)
6. **while** \( D(v_i') \)
7. Perturb \( v_i' \) randomly inside \( B_\delta(v_i) \)
8. **end while**
9. \( V_i' = V_{i-1}' \cup \{v_i'\} \)
10. \( E_i' = E_{i-1}' \cup E'(v_i) \)
11. **end foreach**
12. \( E' = E_n' \)
13. \( V' = V_n' \)

### 3.3 Computing the Perturbation Radius

In this section we derive an upper bound on \( \delta \), the perturbation radius. Recall that this bound guarantees a possibility of at least \( \frac{1}{2} \) in finding degeneracy-free placement when perturbing an endpoint randomly. We emphasize that the computation of \( \delta \) is independent of the actual perturbation; our algorithm computes \( \delta \) before the perturbation step.
Since $\delta$ depends on the potential degeneracies of the model, we present the ones that we handle and analyze their corresponding forbidden loci. Similarly to previous CP schemes, we eliminate potential degeneracies that correspond to close proximities. In arrangements of line segments such potential degeneracies can be induced near vertices. Thus, we will be concerned with potential degeneracies that involve either two vertices or a vertex and a non-incident line segment. We make each pair well separated by considering all cases in which these potential degeneracies are induced. We denote by $n$ the number of endpoints in the input and by $m$ the number of line segments.

We emphasize that $\delta$ is a crude upper bound of the necessary perturbation radius. In practice, much smaller perturbation radii suffice for finding valid locations efficiently. We use this idea to improve the perturbation magnitude (see Section 3.5.2 for details).

Let $B$ be the length of the shortest line segment in $E$. We next formalize the relationship between the perturbation radius and the input.

**Definition 3.1.** For any $\lambda > 0$, a set of line segments is $\lambda$-approximated if and only if $\delta/B \leq \lambda$.

We need this definition later when we analyze our scheme. If the input turns out to be non-$\lambda$-approximated, one should resort to larger $\lambda$'s. We note that finding a suitable $\lambda$ could be determined by a binary search that checks the input or alternatively forced by the user. As the result of this section shows, $\delta$ is larger than all the resolution bounds. Thus, if the input is $\lambda$-approximated then for each resolution bound $\varepsilon$, $\frac{\delta}{B} \leq \lambda$.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the input endpoints, sorted by one of the sorting algorithms in Section 3.4. We denote by $v_i \in V$, $1 \leq i \leq n$, some endpoint. Let $v_i'$ be the final placement of $v_i$, possibly perturbed. Let $E_i$ and $V_i$ be the set of all input line segments and endpoints that are processed by the time $v_i$ is fully processed. Let $V_i' = \{v_1', v_2', \ldots, v_i'\}$ be the set of the final placements of the first $i$ input endpoints.

We maintain four resolution bounds. The reason for having multiple resolution bounds will be clarified later. We denote them by $\varepsilon_1 - \varepsilon_4$.

In the rest of this section we compute the value of $\delta$ by considering the different potential degeneracy kinds and computing upper bounds on the area that each can occupy. In our analysis we use a generic endpoint $v_i$. Thus, the computations below will hold for each endpoint of the input.

### 3.3.1 Computing Forbidden Loci induced by Endpoints

Let $\varepsilon_1$ be the minimum separation that two endpoints must maintain. Then, each $v_j' | j < i$ defines a forbidden disc of radius $\varepsilon_1$ for $v_i$. Since there is an upper bound of $n$ endpoints that may induce forbidden discs, we can bound the total forbidden loci of this potential degeneracy type by

$$F_1 = n\pi\varepsilon_1^2$$

(3.1)

For clarity, some of the definitions here repeat definitions from the previous section.
3.3.2 Computing Forbidden Locus Induced by $v_i$ and a Non-incident Line Segment

Let $\varepsilon_2$ be the minimum separation that an endpoint and a non-incident line segment must maintain (thus also the separation between an endpoint and vertices along the non-incident line segment). Then each $e \in E_{i-1}$ defines a forbidden locus for $v_i$. This region is the Minkowski sum of $e$ and a disc centered at the origin with radius $\varepsilon_2$. It is easy to show that the maximum area which the forbidden locus can cut from $B_\delta(v_i)$ is when $e$ passes through $v_i$ and intersects $\partial B_\delta(v_i)$ twice. This area is bounded by a rectangle whose area is $2\varepsilon_2 \times 2\delta$ (the rectangle $abcd$ in Figure 3.3). There is an upper bound of $m$ line segments that may induce such potential degeneracy. Let $F_2$ be the maximum total forbidden loci in this case. We get

$$F_2 = 4m\varepsilon_2 \delta$$

(3.2)

Figure 3.3: A forbidden locus induced by $v_i$ and $e \in E_{i-1}$.

3.3.3 Computing Forbidden Loci Induced by Intersections of Segments and Line Segments incident to $v_i$

Let $e$ be a line segment incident to $v_i$, such that the other endpoint of $e$ has already been processed. Let $u'$ be an intersection point of two line segments of $E'_{i-1}$ (non-incident to $v_i$). Let $L$ be the longest line segment in $E$. We maintain a separation of $\varepsilon_3$ between $e'$ (the output of $e$) and $u'$ (thus also between $u'$ and any vertex on $e'$). In this case, $e'$ must not penetrate the disc of radius $\varepsilon_3$ centered at $u'$. This is illustrated in Figure 3.4, where $e$ is the thick line whose first endpoint, $u''$, has already been processed. The intersection point, $u'$, must be at least $\varepsilon_3$ away from $e'$ in order not to induce potential degeneracies (we explain below why we place $u'$ at the intersection between $e$ and the disc of radius $\varepsilon_2$ around $u''$). It follows that in order to prevent $e'$ from penetrating the disc with radius $\varepsilon_3$ around $u'$, $v'_i$ must not be located inside the wedge $du''c$. It defines a quadrilateral which bounds the part of the forbidden locus inside $B_\delta(v_i)$ (quadrilateral $abcd$ in Figure 3.4). The quadrilateral has maximum area when $u'$ is located on $e$ and on the disc with radius $\varepsilon_2$ around $u''$ (the later disc contains no non-incident line segments since $u''$ is the result of processing an endpoint) and when the length of $e$ is maximum $(L + \delta)$. In this case $abcd$ is a trapezoid. This explains the placement of $u'$. Note that $g$ in Figure 3.4 is the place where the line segment $u''c$ is tangent to the disc centered at $u'$. We denote by $D_1$ the maximum area of the trapezoid $abcd$ (which bounds the forbidden
locus). Next we compute its magnitude.

\[
\Delta u''u'g \approx \Delta u''hb \approx \Delta u''lc \\
\frac{\varepsilon_3}{|u''g|} = \frac{|ab|/2}{L} = \frac{|dc|/2}{L + 2\delta} \\
|ab| = \frac{2\varepsilon_3 L}{|u''g|} \\
|dc| = \frac{2\varepsilon_3 (L + 2\delta)}{|u''g|} \\
D_1 = (|ab| + |dc|)\delta \\
D_1 = \frac{4\varepsilon_3 \delta (L + \delta)}{\sqrt{\varepsilon_2^2 - \varepsilon_3^2}} \quad (3.3)
\]

It should be clear from our construction that \( \varepsilon_3 \) must be much smaller than \( \varepsilon_2 \): if \( \varepsilon_3 \) is not much smaller than \( \varepsilon_2 \) then in Figure 3.4, \( \angle du''c \) may be sufficiently large to make the size of the trapezoid \( abcd \) unacceptably large. (Note that our assumption was that \( \varepsilon_2 > \varepsilon_3 \); otherwise Figure 3.4 would be wrong.) On the other hand, making \( \varepsilon_2 \) much larger than \( \varepsilon_3 \) indicates that \( \varepsilon_2 \) would be very large itself, resulting in a large perturbation for the first endpoint—see Equation 3.2.

We next coordinate between \( \varepsilon_3 \) and \( \varepsilon_2 \) in order to compute \( \delta \) in terms of the input parameters. Let \( R \) be the ratio \( \frac{\varepsilon_3}{\varepsilon_2} \). Then

\[
\varepsilon_2 = R \varepsilon_3 \quad (3.4)
\]

\( R \) is decided with a binary search that minimizes \( \delta \).

There is an upper bound of \( \binom{m}{2} = \frac{m(m-1)}{2} \) possible intersections that may take place here. Also, processing \( v_i \) may involve the processing of at most \( m \) incident line segments. Thus, the total forbidden loci is bounded by

\[
F_3 = \frac{2m^2(m - 1)\delta (L + \delta)}{\sqrt{R^2 - 1}}
\]

Since the input set has to be \( \lambda \)-approximated

\[
F_3 \leq \frac{2m^2(m - 1)\delta (L + \lambda B)}{\sqrt{R^2 - 1}} \quad (3.5)
\]

Since \( \varepsilon_2 \) must be larger than \( \varepsilon_3 \), the square root in equation 3.5 is real.

### 3.3.4 Computing Forbidden Loci Induced by \( V'_{i-1} \) and Segments incident to \( v_i \)

This case is similar to the case in the previous subsection and illustrated in Figure 3.5. The only difference is that we must maintain an empty disc of radius \( \varepsilon_2 \) around any \( u' \in V'_{i-1} \) for future
Figure 3.4: A forbidden locus of an intersection of two line segments. Note that $u'$ (the intersection) is the center of the small disc while $\varepsilon_3$ is its radius.

perturbations (note that in the previous subsection the disc around $u''$ with radius $\varepsilon_2$ had to be empty of non-incident line segments). Here we assume for simplicity that all such endpoints are incident to segments that have not been processed yet. Thus, there is a separation of at least $\varepsilon_2$ between $u'$ and both $e'$ and any vertex on $e'$. Recall that we maintain a separation of at least $\varepsilon_1$ between any pair of endpoints. The analysis of this case is similar to the one in the previous subsection. Let $D_2$ be the maximum area this forbidden locus can occupy. We next compute $D_2$.

$$
\Delta u''u'g \approx \Delta u''hb \approx \Delta u''lc
$$

$$
\frac{\varepsilon_2}{|u''g|} = \frac{|ab|/2}{L} = \frac{|dc|/2}{L + 2\delta}
$$

$$
|\overline{ab}| = \frac{2\varepsilon_2L}{|u''g|}
$$

$$
|\overline{dc}| = \frac{2\varepsilon_2(L + 2\delta)}{|u''g|}
$$

$$
D_2 = (|\overline{ab}| + |\overline{dc}|)\delta
$$

$$
D_2 = \frac{4\varepsilon_2\delta(L + \delta)}{\sqrt{\varepsilon_1^2 - \varepsilon_2^2}}
$$

(3.6)

There is an upper bound of $m$ possible incident line segments and the test for each may involve testing at most $n$ endpoints. Thus, the total area of the forbidden loci in this case is bounded by

$$
F_4 = \frac{4mn\varepsilon_2\delta(L + \delta)}{\sqrt{\varepsilon_1^2 - \varepsilon_2^2}}
$$

and also by

$$
F_4 \leq \frac{4mn\varepsilon_2\delta(L + \lambda B)}{\sqrt{\varepsilon_1^2 - \varepsilon_2^2}}
$$

(3.7)
Figure 3.5: Forbidden locus of an endpoint.

For simplicity, we require that the angles $du''c$ in Figures 3.4 and 3.5 be equal. This determines the value of $\varepsilon_1$. From triangle similarity we get

$$R = \frac{\varepsilon_1}{\varepsilon_2} = \frac{\varepsilon_2}{\varepsilon_3}$$

$$\varepsilon_1 = \varepsilon_2 R \quad (3.8)$$

Then by using equations 3.8 in equation 3.7 we get

$$F_4 \leq \frac{4mn\delta(L + \lambda B)}{\sqrt{R^2 - 1}} \quad (3.9)$$

### 3.3.5 A Lower Bound on Another Kind of Distance Between an Intersection Point and a Line Segment

Let $e_i$ be a line segment incident to $v_i$ whose other endpoint has already been processed. Let $e'_j$ and $e'_k$ be the output of two segments, where $e'_j$ intersects $e'_i$ at point $f$ (see Figure 3.6). We next argue that if $e'_i$ does not induce any of the potential degeneracies we discussed above (as our algorithm guarantees), then a potential degeneracy of type vertex-segment between $f$ and $e'_k$ cannot exist as well (and thus also no potential degeneracy between $f$ and any vertex on $e'_k$). We do so, by giving a lower bound on the distance between $f$ and $e'_k$; we denote this lower bound by $\varepsilon_4$.

We differentiate between two cases. The first one is when $e'_j$ and $e'_k$ do not intersect. Since the endpoints of $e'_j$ are at least within distance of $\varepsilon_2$ from $e'_k$, any point on $e'_j$ is within distance of $\varepsilon_2$ from $e'_k$ as well. Then the distance between $f$ to $e'_k$ is at least $\varepsilon_2$ and no potential degeneracy is induced. Thus, we ignore this case in this analysis.
Figure 3.6: The minimum distance between $f$ and $e'$ when $e_j'$ and $e_k'$ intersect.

The second case where $e_j'$ and $e_k'$ intersect is illustrated in Figure 3.6. $e_j'$ and $e_k'$ force $e_j'$ not to penetrate the disc $C$, centered at their intersection, $c$, with radius $\varepsilon_3$. Thus, $f$, which lies on $e_j'$, would be closest to $e_k'$ if it is placed on the boundary of $C$. Moreover, the smaller the angle $\angle bca$ (denoted by $\alpha$) is, the smaller $\varepsilon_4$ (the distance between $f$ and $e_k'$) is. In Figure 3.6, $\varepsilon_4$ is the size of $\overline{fh}$. $\alpha$ has minimum value when $e_j'$ and $e_k'$ have maximum length below $C$ (bounded by $L + 2\delta$) and the distance between their lower endpoints ($a$ and $b$ are these endpoints in the figure) is $\varepsilon_2$.

We next compute a lower bound on the length of the line segment $\overline{ef}$.

By triangle similarity

$$\frac{|ef|}{\varepsilon_2} = \frac{\varepsilon_3}{L + 2\delta}$$

Thus,

$$|ef| = \frac{\varepsilon_2 \varepsilon_3}{L + 2\delta}$$

Let $\sigma$ denote any resolution bound or perturbation radius. Then $\sigma \leq \lambda l$. Together with a simple trigonometric observation from Figure 3.6 we get

$$\cos\left(\frac{\pi}{2} - \frac{\alpha}{2}\right) = \frac{\varepsilon_2/2}{L + 2\delta}$$

$$\cos^2\left(\frac{\pi}{2} - \frac{\alpha}{2}\right) = \frac{\varepsilon_2^2}{4(L + 2\delta)^2}$$

$$\sin^2\left(\frac{\pi}{2} - \frac{\alpha}{2}\right) = 1 - \frac{\varepsilon_2^2}{4(L + 2\delta)^2} \geq$$

$$1 - \frac{(\lambda B)^2}{4(L + 2\delta)^2} \geq 1 - \frac{(\lambda B)^2}{4L^2}$$

$$\sin\left(\frac{\pi}{2} - \frac{\alpha}{2}\right) \geq \sqrt{1 - \frac{(\lambda B)^2}{4L^2}}$$

In the same figure we also get a lower bound on $\varepsilon_4$. 

67
\[ \sin\left(\frac{\pi}{2} - \frac{\alpha}{2}\right) = \varepsilon_4 / |ef| \]
\[ \varepsilon_4 \geq |ef| \sqrt{1 - \frac{(\lambda B)^2}{4L^2}} \]
\[ \varepsilon_4 \geq \frac{R\varepsilon_3^2 \sqrt{1 - \frac{(\lambda B)^2}{4L^2}}}{L + 2\delta} \]
\[ \varepsilon_4 \geq \frac{R\varepsilon_3^2 \sqrt{1 - \frac{(\lambda B)^2}{4L^2}}}{L + 2\lambda B} \quad (3.10) \]

For the same reasons it follows that \( \varepsilon_4 \) is also a lower bound on the distance between any pair of intersections along \( e'_i \).

The next lemma summarizes the relationship among the resolution bounds.

**Lemma 3.2.** \( \varepsilon_4 < \varepsilon_3 < \varepsilon_2 < \varepsilon_1 \)

**Proof.** Since we fix \( \varepsilon_3 < \varepsilon_2 < \varepsilon_1 \), we only have to prove that \( \varepsilon_4 < \varepsilon_3 \). Consider Figure 3.6. According to the construction above, the lengths of \( \overline{ca} \) and \( \overline{cb} \) are in the magnitude of \( L \), and the length of \( \overline{ab} \) equals \( \varepsilon_2 \). If the input set is \( \lambda \)-approximated, then \( \angle bca \) will be sufficiently small so that \( \varepsilon_4 \) (the length of line segment \( \overline{fh} \)) is smaller than \( \varepsilon_3 \) (the length of segment \( \overline{ve} \)). \( \square \)

Note that \( \varepsilon_4 \) will be the maximum between two resolution bounds which involve the separation of a vertex and a non-incident segment and a separation between two vertices. Given \( \varepsilon_4 \) as a parameter, we then compute the values of \( \varepsilon_1, \varepsilon_2 \) and \( \varepsilon_3 \).

Since we have a lower bound on \( \varepsilon_4 \), we can ignore the potential degeneracy we discuss in this section when perturbing endpoints. However, note that it will affect the magnitudes of the resolution bounds and the perturbation radius.

We next compute \( \varepsilon_3 \) as a function of the input parameters and \( \varepsilon_4 \). If we transfer inequality 3.10 to an equation, we obtain a relationship between \( \varepsilon_3 \) and \( \varepsilon_4 \).

\[ \varepsilon_3 = \frac{(L + 2\lambda B)\varepsilon_4}{\sqrt{R} \sqrt{1 - \frac{(\lambda B)^2}{4L^2}}} \quad (3.11) \]

### 3.3.6 Computing \( \delta \)

Recall that we want the perturbation disc area to be twice as large as the total area of all the forbidden loci. Thus, \( \pi \delta^2 = 2(F_1 + F_2 + F_3 + F_4) \). By using formulas 3.1, 3.2, 3.4, 3.5, 3.8,

\[ \text{A deep analysis of the resolution bounds (analogous to the work in [51]) would potentially set the resolution bounds based on the input and the machine precision. In addition, it would need to satisfy the relationships between them as described here.} \]

68
3.9 and 3.11 where we transform formulas 3.5 and 3.9 into equations, we get the following quadratic equation on $\delta$ which corresponds to a sufficiently large perturbation radius:

$$a\delta^2 + b\delta + c = 0$$

where $a = \pi, b = -8mR\varepsilon_3 - \frac{4m(L+\lambda B)(m(m-1)+2)n}{\sqrt{R^2-1}}, c = -2n\pi R^4\varepsilon_3^2$ and $\varepsilon_3 = \sqrt{\frac{(L+2\lambda B)\varepsilon_4}{R^4(1-\lambda B^2)}}$.

It is easy to verify that this equation has two real roots, one positive and one negative. Thus, the value of $\delta$ will be the positive one.

**Theorem 3.3.** When CPLS terminates, each vertex of the arrangement is at least $\varepsilon_4$ away from both all non-incident line segments and all other vertices.

**Proof.** Let $v$ be any endpoint. From Section 3.3.1, $v'$ is at least $\varepsilon_1$ units away from all other endpoints. Moreover, from Sections 3.3.2 and 3.3.4, $v'$ is at least $\varepsilon_2$ units away from all non-incident segment. Let $u'$ be any intersection point in the output. $u'$ is at least $\varepsilon_3$ away from all non-incident line segments that were processed after $u'$ was induced (from Section 3.3.3) and at least $\varepsilon_4$ away from all non-incident line segments that were processed before $u'$ was induced (from Section 3.3.5). From Lemma 3.2 the claim follows.

### 3.4 Ordering the Endpoint Processing

A major drawback of a typical Controlled Perturbation scheme is the potential large perturbation. In this section we describe a novel method to decrease it.

Recall that CPLS incrementally processes endpoints. Let $\Pi(V)$ be a permutation of $V$ that determines the endpoint processing order. For each endpoint $v \in V$, let $\Pi(v)$ denote its place (or index) in $\Pi(V)$. Let $v_1, v_2 \in V$ be two endpoints such that both $\Pi(v_1) < \Pi(v_2)$ and $[v_1, v_2] \in E$. It follows that processing $v_2$ involves testing potential degeneracies for the line segment $e' = [v'_1, v_2]$. However, since $v_2$ is processed after $v_1$, no test that involves $e$ is performed while processing $v_1$. On the other hand, if $\Pi(v_1) > \Pi(v_2)$, potential degeneracies with $e'' = [v_1, v'_2]$ would be tested when processing $v_1$. Since different forbidden loci (with potentially different area sizes) are involved in the above two different cases, each may result, on average, in different perturbation magnitudes.

Let $E(v) = \{[v, u']|v, u \in E, \Pi(u) < \Pi(v)\}$. (this definition is slightly different from the one in Section 3.2.) Then processing $v$ involves eliminating potential degeneracies induced by the line segments of $E(v)$.

Consider Figure 3.7 which contains four input line segments. The three thin solid line segments are the intermediate output of CPLS (denote them by $S$) and the endpoints $a$ and $b$ are the only ones that have not been processed yet. Let $e$ be the line segment $[a, b]$. The discs around some of the endpoints of the line segments of $S$ are their forbidden loci. Note that the next endpoint to process (either $a$ or $b$) does not have to be perturbed since it is sufficiently far from the three line segments. However, when processing the last endpoint we need to test $e'$ ($e$ with the first endpoint processed), which penetrates two forbidden discs.

We denote the union of the three discs of Figure 3.7 by $J$. Let $\gamma(b)$ and $\gamma(a)$ be the wedges that originate from $b$ and $a$ respectively, are tangent to $J$ and contain $a$ and $b$ respectively. $\gamma(b)$ and $\gamma(a)$
are drawn with dashed lines in the figure. Let $F_J(a) \subset F(a)$ and $F_J(b) \subset F(b)$ be the forbidden loci of $a$ and $b$, respectively, that correspond to the three discs in the figure. Then $F_J(a) \subset \gamma(b)$ and $F_J(b) \subset \gamma(a)$. Let $\zeta(a)$ ($\zeta(b)$) be the part of $\gamma(b)$ ($\gamma(a)$) to the right (left) of $J$. We note that in this example $\delta$ is small enough such that $B_\delta(a)$ and $B_\delta(b)$ are disjoint from $J$ (both perturbation disks are not depicted in Figure 3.7 for clarity). Thus, $F_J(a) = \zeta(a) \cup J$ and $F_J(b) = \zeta(b) \cup J$.

Let $\bar{F}_\delta(v)$ be the degeneracy-free locus of $v$ within $B_\delta(v)$. Assuming that $B_\delta(a)$ and $B_\delta(b)$ are disjoint from other forbidden loci introduced by $S$, we get that $\bar{F}_\delta(a) = B_\delta(a) \setminus F_J(a)$ and $\bar{F}_\delta(b) = B_\delta(b) \setminus F_J(b)$. Since the angle of $\gamma(a)$ is smaller than the angle of $\gamma(b)$, $|\bar{F}_\delta(b)| < |\bar{F}_\delta(a)|$. Hence, it would be a better choice to process $a$ before $b$ in order to obtain larger forbidden-free loci.

Based on this idea, we develop and use efficient heuristics to improve the quality of the output by predetermining the processing order of the endpoints. The idea is to analyze the two potential forbidden loci that affect the placement of each line segment—one for each possible processing order of its endpoints. Note that since we sort the endpoints before perturbing them, we consider their initial positions. We use two measurements to evaluate the efficiency of the processing order:

- MIN-SUM: Minimize the perturbation sum.
- MIN-MAX: Minimize the maximum perturbation.

We define two different directed graphs with weights, one for the MIN-SUM problem and the other for MIN-MAX. We will denote both graphs by $G(V, E)$. Information corresponding the forbidden locus area associated with the segments is embedded in their weights.

In the rest of this section, we define the optimization problems for MIN-SUM and MIN-MAX and describe their corresponding heuristics.

---

With a slight abuse of notations, we refer by $F$ and $F_J$ to the forbidden loci that are induced if the corresponding endpoint is processed last.
3.4.1 Minimizing the Sum

The goal is to minimize the perturbation magnitude sum. Given \( v \in V \) and an incident line segment \( e \), let \( F_e(v) \) be the forbidden locus areas induced inside \( B_\delta(v) \) as the result of the forbidden loci that \( e \) involves\textsuperscript{7}. These areas include forbidden loci induced by endpoints and intersections. The directed graph \( G \) is defined as follows\textsuperscript{8}. Let \( v_1 \) and \( v_2 \) denote a pair of endpoints connected by an edge of \( E \). We define a directed edge \( e \in E \) that corresponds to the line segment that connects \( v_1 \) and \( v_2 \) as follows. If \( F_e(v_2) > F_e(v_1) \), then it would likely be beneficial (ignoring forbidden loci that do not involve \( e \)) to order \( v_2 \) before \( v_1 \). The reason, following the discussion above, is that in this way we deal with smaller forbidden loci. Thus, the direction of \( e \) is from \( v_2 \) to \( v_1 \) and its weight is \( w(e) = F_e(v_2) - F_e(v_1) \), reflecting the benefit of placing \( v_2 \) before \( v_1 \). (Note that \( w(e) \) is positive.) If \( F_e(v_2) \leq F_e(v_1) \) the direction of \( e \) and its weight will be analogous. In this computation, we consider for simplicity all input endpoints and intersections.

Given a permutation \( \Pi(V) \) on the endpoints, an edge \( e \) is a forward edge if it complies with the direction of \( e \) in \( G \) (namely its source appears before its destination in \( \Pi(V) \)). Similarly, an edge is a backward edge (or backedge) if its direction does not comply with \( \Pi(V) \). Note that if an edge is a backedge, its corresponding forbidden locus is larger than the forbidden locus it would have if it was a forward edge. Thus, we are interested in minimizing the sum of weights associated with backedges.

Let \( E_\Pi = \{ e \in E | \text{The source of } e \text{ appears after its target in } \Pi(V) \} \). We define the graph version of MIN-SUM as follows.

**MIN-SUM:** Given a directed graph \( G = (V, E) \) with positive weights on \( E \), find a permutation \( \Pi(V) \) that minimizes \( \sum_{e \in E_\Pi} w(e) \).

It turns out that the decision problem associated with MIN-SUM is NP-complete by observing that it is equivalent to the Feedback Arc Set problem (FAS) \textsuperscript{40}. In the FAS problem, we are given a directed graph \( G = (V, E) \) with positive weights on \( E \). The goal is to find a subset \( H \subseteq E \) with minimal weights such that \( G' = (V, E \setminus H) \) is acyclic. Since \( G' \) is a directly acyclic graph, it defines a topological order on the vertices of \( V \). Edges which comply with this order are included in \( E \setminus H \) since the goal is to maximize the total weight of this set. Others are included in \( H \). It follows that FAS is identical to our problem in which the backedges, whose sum we want to minimize, are exactly the set \( H \) in the FAS problem.

There are several heuristics in the literature to approximate FAS. We applied three \textsuperscript{6, 27, 32} and adapt them to our MIN-SUM version. We refer to these algorithms by MIN-SUM-1, MIN-SUM-2 and MIN-SUM-3, respectively. We next briefly introduce them and describe how we adapt them to our needs.

- **MIN-SUM-1.** An approximation algorithm \textsuperscript{32} that achieves an approximation ratio bounded by the length, in terms of number of arcs, of the longest simple cycle in the graph. The algorithm consists of two phases. In the first, cycles are identified iteratively. For each edge in each cycle, the minimum edge weight in the cycle is subtracted from the weights of the edges.\textsuperscript{9}

---

\textsuperscript{7}In practice, to simplify the computation, we use approximating polygons in a similar way to our computation in Section 3.3.3.

\textsuperscript{8}By a slight abuse of notation, we use the notation we used to describe the input here too.

---

71
Edges that get weight 0 are removed. The first phase terminates when the graph becomes acyclic. In the second phase, the algorithm adds the deleted edges that do not close cycles. Finally, the set of the removed edges is the approximated feedback arc set. We refer to these edges as the backedges in our optimization problem.

- **MIN-SUM-2.** An approximation algorithm, termed FAS-TOURNAMENT, that works on restricted set of graphs [27]. FAS-TOURNAMENT works on complete graphs with probability constraints (for every two nodes, the sum of the weights of the directed halfedges from one to the other equals 1). The algorithm simply sorts the vertices by increasing order of their incoming weights and defines the feedback arc set as the edges that are directed opposite to the sort (those correspond to the backedge set in our problem). It achieves an approximation ratio of 5. In order to use this algorithm, we modify our graph in the following way. For each pair of nodes, \( u \) and \( v \), we compute the forbidden locus area in both directions. If there is either no forbidden locus in both directions or they have equal area, we assign a cost 0.5 for each corresponding directed edge to reflect that the direct relationship between \( u \) and \( v \) gives us no information on which vertex should come before the other. Otherwise, we normalize the weights according to the corresponding forbidden locus area. It is done in the same way as the process we described in the beginning of Section 3.4, but here we assign the two directed half-edges with the corresponding weights instead of one directed edge whose weight is the forbidden locus area difference.

- **MIN-SUM-3.** An approximation algorithm [6] that works on a complete directed graph with no weights. The algorithm picks a random vertex \( p \) as the pivot. Then the nodes are split into two groups: one of the vertices which have out-edges to \( p \) and the other is of the vertices which have in-edges from \( p \). Then the algorithm recurses on both groups. The nodes of the first group appear before the nodes of the second group in \( \Pi(V) \). The algorithm achieves an average approximation ratio of 3. As in MIN-SUM-2, the permutation defines the approximating feedback arc set as the edges with opposite direction to the permutation (these are the backedges of our problem). In order to use the algorithm we make two modifications to \( G \). First, we connect any two non-adjacent nodes by an edge whose direction is randomly chosen (note that the order between these two with respect to direct relationship in CPLS is immaterial). Then we remove all weights and remain with a complete graph with no weights.

In order to improve the results, we implemented and used the following procedure before applying the FAS approximation algorithms, but after determining the direction of the edges. Let \( S(G) = (S_1, S_2, \ldots, S_k) \) be the strongly connected components (SCC) of \( G \), topologically ordered. Let \( G_i = (S_i, E_i) \) be the subgraph induced by taking the nodes that belong to \( S_i \), \( 1 \leq i \leq k \), and the edges that connect two nodes of \( S_i \) (denoted by \( E_i \)). Let \( \Pi_i \) be an optimal permutation of \( S_i \) for the MIN-SUM problem of \( G_i \).

The next theorem claims that an optimal solution for \( G \) is in the form \((\Pi_1, \Pi_2, \ldots, \Pi_{k-1}, \Pi_k)\).

**Theorem 3.4.** Let \( S(G) = (S_1, S_2, \ldots, S_k) \) be the strongly connected components of \( G(V, E) \), in their topological order. Let \( G_i \) be the subgraph that includes the vertices of \( S_i \) and the edges connecting them. An optimal permutation for the MIN-SUM problem is in the form \((\Pi_1, \Pi_2, \ldots, \Pi_{k-1}, \Pi_k)\) where \( \Pi_i \) is an optimal permutation of \( G_i \).

**Proof.** Let \( SCC(v) \) be the strongly connected component (SCC) of any \( v \in V \). Let \( \Pi' \) be an optimal permutation for the FAS problem and suppose it is not of the form \((\Pi_1, \Pi_2, \ldots, \Pi_{k-1}, \Pi_k)\).
We iteratively swap adjacent nodes in $\Pi'$, $v_1$ and $v_2$, where $v_1$ appears before $v_2$ in $\Pi'$, but $SCC(v_1)$ is topologically after $SCC(v_2)$. By doing so, we are guaranteed that we do not harm $\sum_{e \in E} w(e)$, because there is no directed edge from $v_1$ to $v_2$. We continue in this fashion until there are no such adjacent nodes to swap. It follows that the final permutation is in the form $(\Pi'_1, \Pi'_2, \ldots, \Pi'_{k-1}, \Pi'_k)$ where $\Pi'_i$ is a permutation of subgraph $G_i$. Note that at this point there are no backedges between any two subgraphs. Thus, there are no cycles that consist of nodes of more than one strongly connected component. Hence, we can analyze each subgraph independently. If $\Pi'_i$ is not an optimal permutation of $G_i$, then we replace it with an optimal one, $\Pi_i$, and the result may only be improved. The claim follows. 

Following this theorem, we divide the graph into strongly connected components. Then we apply the approximation algorithm on each subgraph to get sets of sorted endpoints. Finally, we merge these sets by placing them in the order of $S(G)$ such that all edges connecting two different strongly connected components will be forward.

### 3.4.2 Minimizing the Maximum

The definition of $G(V, E)$ in this case is somewhat different from the definition in Section 3.4.1. Given an edge $e(a, b)$, the definitions of $F_e(a)$ and $F_e(b)$ are the same as in the MIN-SUM problem. We define two weighted halfedges in $G$: $h(a, b)$ with the weight $F_e(b)$ and $h(b, a)$ with the weight $F_e(a)$. Let $w(u, v)$ be the weight of the halfedge $h(u, v)$. Let $\Pi(v)$, $v \in V$, be the place (index) of $v$ in $\Pi(V)$. We use the following formula which sums up the weights of some of the incoming halfedges to a vertex. The halfedges that are used here are the ones which comply with the permutation $\Pi(V)$.

$$w_{\Pi}(v) = \sum_{\{u|v,u\in E, \Pi(u)\leq \Pi(v)\}} w(u, v)$$

It follows that $w_{\Pi}(v)$ measures the actual forbidden locus area induced by the edges that are directed to $v$.\footnote{For simplicity, we ignore intersections between corresponding forbidden loci that decrease the actual forbidden locus area, and simply sum up the forbidden locus areas.} Thus, our objective is to minimize $\max\{w_{\Pi}(v)|v \in V\}$. In other words, considering only halfedges that comply with $\Pi(V)$, our goal is to minimize the maximum sum of weights of incoming halfedges to a vertex.

It turns out that this problem is polynomially solvable. For each $v \in V$, let $w(v) = \sum_{u \neq v} h(u, v)$. The following is a pseudocode for optimally solving the MIN-MAX problem using a greedy approach in polynomial time.

```plaintext
For simplicity, we ignore intersections between corresponding forbidden loci that decrease the actual forbidden locus area, and simply sum up the forbidden locus areas.
```
MinimizeMaximumPerturbation

**Input:** a directed graph $G = (V, E)$ with positive weights on the edges.

**Output:** a permutation $\Pi(V)$ that minimizes $\max\{w_\Pi(v) | v \in V\}$.

1. foreach $v \in V$, compute $w(v)$
2. sort $V$ with respect to $w(v)$
3. while $V$ is not empty
4. let $v$ be the vertex with the minimum $w(v)$ in $V$
5. foreach $u \in V$
6. $w(u) = w(u) - w(v, u)$
7. remove $v$ from $V$ and insert it to the front of $\Pi(V)$
8. end while

We next prove that our greedy algorithm finds an optimal solution. Let $\Pi_i(V)$ be the last $i$ elements in $\Pi(V)$ and $\Pi_i(V)$ be the first $n - i$ elements of $\Pi(V)$, where $n = |V|$.

**Lemma 3.5.** The values of $w_\Pi(v) : v \in \Pi_i(V)$ do not depend on the order of $\Pi_i(V)$.

**Proof.** According to the greedy algorithm, the elements of $\Pi_i(V)$ are chosen before the elements of $\Pi_i(V)$, and thus appear after them in $\Pi_i(V)$. Therefore, the values of $w_\Pi(v) : v \in \Pi_i(V)$ include all the weights of the halfedges directed from the elements of $\Pi_i(V)$. Moreover, the order of the elements of $\Pi_i(V)$ is set by the time the algorithm works on the elements of $\Pi_i(V)$. From the above two observations, the values of $w_\Pi(v) : v \in \Pi_i(V)$ are finalized by the time the algorithm determines the order of the elements in $\Pi_i(V)$. The claim follows.

**Corollary 3.6.** The optimal solution contains within it optimal solutions to subproblems (the Optimal Substructure property).

For any permutation $\Pi(V)$, let $\Xi(\Pi(V)) = \max\{w_\Pi(v) | v \in V\}$. In other words, $\Xi(\Pi(V))$ is the value of the MIN-MAX problem. We are now ready to prove the main idea.

**Theorem 3.7.** MinimizeMaximumPerturbation finds an order which minimizes the maximum $w_\Pi(v)$ for all $v \in V$.

**Proof.** Let $OPT$ denote an optimal permutation and $GREEDY$ denote the permutation obtained with our algorithm. Our goal is to prove that $\Xi(OPT) = \Xi(GREEDY)$. We prove it inductively by showing that whenever an element of $OPT$ does not satisfy the greedy selection, we can safely replace the corresponding element by the element that would have been chosen with a greedy selection and still be optimal. Suppose $OPT$ satisfies the greedy selection for its last $i$ elements, $1 \leq i \leq n - 1$ (the first $i$ elements to be chosen). Further, suppose that it differs from the greedy algorithm in the $(n - i)'$th element (the one just before the last $i$ elements). Let $OPT_i$ be the first $n - i$ elements of $OPT$. Following Corollary 3.6, $OPT_i$ is an optimal solution for the subproblem of $OPT$ that involves the first $n - i$ elements. Let $v'$ be the last element of $OPT_i$ and let $v_{\min}$ be the element that would be chosen by our greedy algorithm instead of $v'$. Let $OPT'$ be defined as follows. Its last $i$ elements are identical to the last $i$ elements of $OPT$ (in the same order). Its $(n - i)'$th element is $v_{\min}$. Finally, its first $n - i - 1$ elements (we denote this list by $\Psi$) are the same as the element of $OPT_i$, except of $v_{\min}$ which is removed (again, in the same order). From the construction of $OPT'$ we get that
\[ w_{OPT}(v_{\min}) < w_{OPT_i}(v') \leq \Xi(OPT_i) \quad (3.12) \]

Since \( \Psi \) is obtained by removing \( v_{\min} \) from \( OPT_i \), \( w_\Psi(v) \leq w_{OPT_i}(v) : v \in \Psi \). The reason is that by removing \( v_{\min} \), the sum of the weights of the endpoints are decreased by the weights of halfedges whose source is \( v_{\min} \). We get that

\[ \Xi(\Psi) \leq \Xi(OPT_i) \quad (3.13) \]

From the construction of \( OPT' \) and Formulas 3.12 and 3.13, we get that \( \Xi(OPT') \leq \Xi(OPT) \). Since OPT is an optimal permutation, \( \Xi(OPT') = \Xi(OPT) \). Thus, we can arrive at an optimal solution by choosing the greedy choice and choosing an optimal permutation of the subproblem of the first \( n - i - 1 \) elements. By induction on the elements of \( OPT \), we get that we can safely make the greedy choice at each iteration and obtain an optimal solution. The claim follows.

\[ \Box \]

**Running time.** The two loops that start in lines 3 and 5 imply that the running time of the algorithm is \( O(n^2) \), where \( n \) is the number of vertices. There is no need to use auxiliary memory besides \( V \), thus the space is linear.

Since we use only one algorithm to solve the MIN-MAX problem, we denote it by MIN-MAX too. We implemented all of the above algorithms (MIN-SUM[1-3] and MIN-MAX) and analyze their performance experimentally in Section 3.7.

**Remark.** Since the sorting algorithms are performed prior to the perturbation process, they do not comply with a Controlled Perturbation version of a randomized incremental algorithm. However, there are certain input sets that can still use the ideas of our sorting algorithms with a randomized incremental algorithm. One important example is a set of separate line segments. In this case, for each line segment we would use our ideas to decide the processing order of its vertices on-line.

The next theorem summarizes CPLS.

**Theorem 3.8.** Given an arrangement of \( n \) line segments in \( \mathbb{R}^2 \) with \( n \) endpoints, CPLS produces a perturbed arrangement in \( O(n^3 + \Psi(n)) \) expected time and \( O(n^2 + \Phi(n)) \) working storage where \( \Psi \) and \( \Phi \) are the time and the space the sorting algorithm consumes. The output is linear in the size of the input. In the perturbed arrangement, each vertex is at least \( \varepsilon_A \) away from both all non-incident line segments and all other vertices\(^{10}\).

**Proof.** Note that each atomic test for potential degeneracy takes \( O(1) \) time as it involves a constant number of operations. Then the time is derived immediately by analyzing the pseudocode presented in Section 3.2. Since we maintain the incidences in \( G \), the output will differ from the input only by the Euclidean coordinates of the vertices, and the combinatorial structure will not change. The claim follows.

\[ \Box \]

\(^{10}\)The last part of the theorem is copied from Theorem 3.3
In comparison, constructing an arrangement of line segments in the plane takes $O(n^2)$ time and space. We implemented an optimization technique to decrease the actual processing time significantly (see Section 3.5.2 for details).

We note that an asymptotic cubic running time is inherent in CP schemes where each pair of objects intersect at most a constant number of times. In that case, each perturbation (there are $O(n)$ of them) involves $O(n^2)$ vertices to test. However, we emphasize that in practice the asymptotic running time is much smaller in all cases we experimented. It is still unknown to us whether the cubic running time is tight. In all our experiments, the running times were linear to quadratic. We believe that except possibly rare cases, most input sets require linear to quadratic running time too.

We finally note that the sorting algorithms that we use do not increase the overall asymptotic time and space complexity.

### 3.5 Optimization Techniques

We implemented and used two optimizations to improve the quality and the performance of our software. Both use similar ideas from previous Controlled Perturbation work [53].

#### 3.5.1 Partition into Grid Cells

The idea is to partition the plane into grid cells with edge length slightly longer than the longest line segment. Then, in order to detect potential degeneracies, we test with features that lie within either the same cell or neighboring cells (other features will be too far to induce degeneracies). Thus, if the input is $\lambda$-acceptable, then with an edge length that is longer than $(1 + 2\lambda)L$, it is sufficient to test the cell that contains the endpoint and its immediate neighbors when processing an endpoint. This technique is useful when the line segments are much shorter than the model diameter, and when the distribution is relatively uniform. Several classic Computational Geometry data structures, such as Triangulations, Voronoi Diagrams and GIS data usually satisfy these properties when the input is uniformly distributed.

#### 3.5.2 Decreasing the Perturbation magnitude

The goal of this technique is to decrease the perturbation. We modify the perturbation step as follows. If we need to perturb an endpoint, we start with a very small perturbation radius $r$ (we choose $r = 2\varepsilon_1$; recall that $\varepsilon_1$ is the largest resolution bound), and try to find a valid placement. If we fail to find a valid placement after $C$ times ($C$ is a small constant), we double $r$ and try again. We continue in this fashion until we find a valid placement or until $r > \delta$. At this point we fix $r = \delta$ and find a valid solution after at most 2 trials on the average (we fix $r$ to satisfy our perturbation analysis). With this technique, we are likely to decrease the perturbation significantly, since $\delta$ is usually a crude upper bound. It follows that the average number of trials we need to perform if an endpoint has to be perturbed is upper bounded by $O(\log(\frac{2}{r})C)$. We note that in all our experiments we needed no more than a small constant number of trials to find a valid placement, when an endpoint had to be perturbed.

The following is a high level pseudocode for determining $r$. 

76
Determine $r$

static counter = 1, $r^{11}$
1. if the vertex is perturbed for the first time
2. $r = 2\varepsilon_1$
3. else
4. if $r$ is not fixed /* It is fixed in line 8*/
5. if counter $= C$
6. counter $= 0$
7. $r = r \times 2$
8. if $r > \delta$
9. fix $r = \delta$
10. end if
11. counter $= counter + 1$
12. end if
13. end if

3.6 Comparison with Alternative Schemes

We compare our scheme to alternative techniques whose target is to provide robust geometric computing as well. The first, Snap Rounding, was introduced in the previous Chapter. The second, Exact Arithmetic, is a well known method that is supported and used as the recommended datatype in CGAL [21].

3.6.1 Snap Rounding

While Snap Rounding shares the main goal of Controlled Perturbation (namely to yield approximated data that can be further more robustly processed using only limited precision), it holds several differences. We next compare several characteristics of SR and CP.

(a) It is not easy to extend both techniques to new settings. As of today, CP has been adapted for several settings (see Section 5.1). However, each requires deep degeneracy analysis to determine the perturbation magnitudes. It also seems that computing the real resolution bounds (done so far only for one settings [51]) is a tedious work too. Extending SR seems difficult too. The main reason is that it is not easy to develop SR variants that snap line segments to centers of grid cells while satisfying the plausible properties that current SR variants hold (see below). So far SR has been adapted to a few settings (see above).

(b) There are significant differences between the output of SR and CP. We discuss here the $\mathbb{R}^2$ line segment version. Both make the vertices of the arrangement well separated. However, SR breaks line segments into multiple links. It has been proven that by doing so it guarantees two important properties: a vertex of the arrangement never passes through a line segment and the

\[11\text{We use the C language semantic for static variables: their scope is the entire routine and they maintain their values throughout the entire process.}\]
approximation is bounded by a small constant. However, vertices may collapse to one vertex and thus introduce new degeneracies that do not occur in the original set. Another drawback of the output is that vertices may lie very close to non-incident segments. Halperin and Packer [52] and Packer [77] modified SR to eliminate this property. As opposed to SR, CP does not guarantee small constant approximation. The approximation usually depends on the input and the machine precision. In this work we describe techniques to decrease the perturbation (Sections 3.4 and 3.5.2). However, in general it is not clear how large the perturbation would be (especially for more complicated settings). As opposed to SR, in CP vertices may cross non-incident line segments. Moreover, most CP schemes do not guarantee any incidence preservation, namely that two incident features remain incident after perturbation (note this property is maintained in SR). An exception is the work of Raab [79] which preserves incidences partially and this work in which we preserve all incidences. Figure 3.8 illustrates the differences between CPLS and SR.

(c) SR requires exact arithmetic while CP output can be computed with limited precision as long as the resolution bounds are large enough to overcome precision errors.

This discussion implies that CPLS provides an alternative scheme for constructing robust line segment sets in \( \mathbb{R}_2 \). Each scheme may be suitable in different situations.

### 3.6.2 Exact Arithmetic

A completely different approach to provide robust computation is to use Exact Arithmetic datatypes. This technique should guarantee exact and reliable computation. On one hand, this technique is much more general than Finite Precision Approximation techniques and could be adapted more easily to many models. On the other hand, the computation model of Exact Arithmetic is more complex than build-in datatypes (e.g integer, floating-point) and requires more time and space resources. Both Exact Arithmetic (EA, for short) and Finite-Precision Approximation (FTA, for short) require extra time and space resources. Since they use completely different methodologies, it is not easy to compare their performance. Intuitively, as the application that should work on the line segment set becomes heavier, the more plausible it is to use FTA. Let \( AT_A(I) \) be the time it takes to approximate the input \( I \) to \( A(I) \) with an FTA scheme. Let \( RT_A(P,A(I)) \) be the time program \( P \) works on \( A(I) \) with finite-precision arithmetic and let \( RT_E(P,I) \) be the running time
of $P$ on $I$ with EA. It follows that using FTA would be more time consuming than using EA if $RT_A(P, A(I)) + AT_A(I) > RT_E(P, I)$.

### 3.6.3 Advantages of CPLS

Each of the schemes we discussed above may be more suitable in different situations than the others. We give hints on where CPLS is strong and under which scenarios it has advantages in various aspects. We start by explaining why in many cases CPLS produces better results that its theoretic guarantees might imply and continue by comparing with SR and EA.

**Discussion: CPLS Behavior**

(a) While the topology is maintained when using EA and also maintained in some sense with SR (the order of intersection along line segments are not reversed), in CPLS (and also in other CP schemes), there are no topology preservation guarantees. In particular, line segments may cross vertices and the order of intersections along line segments may not be preserved. However, this situation may be acceptable if the perturbation that CPLS generates is sufficiently small. In such cases, topological violations will occur in the vicinity of very close features. In these cases, there is a good chance that the reliability of the topology is not precise to begin with, due to physical measurement errors that have good chances to deviate the input more than CPLS. In such cases CPLS would not degrade the quality of the input, and moreover will create more consistent data for further manipulation.

(b) Although we stated that the asymptotic running time of CPLS is cubic, in practice the running time order is significantly lower. Cubic running time can be achieved if on average each line segment is very close to a quadratic number of vertices. While we did not find any data that satisfies this condition, in most of the realistic data we tested line segments are never very close to numerous vertices: in many cases they are close to constant number of vertices, and only in highly congested data this number is linear (and thus the running time is in the order of $O(n) - O(n^2)$).

**Advantages over Snap Rounding**

(a) CP algorithms, and CPLS in particular, are easy to implement and does not require special data structures. We found the implementation of the sorting algorithms we chose easy to implement as well. Note that the user should be oblivious to the special theory behind the algorithm, and should use the optimization we described in Section 3.5.2 to initialize the perturbation radius instead. As of today, Snap Rounding (SR for short) algorithms that guarantee better asymptotic time than the one we implemented are not easy to implement and even the more simple ones seem more complicated to implement than CPLS. As far as we know, the more time efficient algorithms have not been implemented.

(b) CPLS (as other CP algorithms) does not require exact arithmetic for computation, while as far as we know, all SR algorithms in the literature must use exact arithmetic to provide reliable results. Thus, CPLS would be more plausible to use in terms of modularity and does not require availability of exact arithmetic libraries.
The output of CPLS seems more compatible with proceeding applications. While SR transforms line segments into polygonal chains, CPLS does not change the structure of the data. This could have the following significant consequences. When using SR, the application that follows should provide special treatment to polygonal chains (where each link may be shared by multiple line segments) and needs to associate each polygonal chain to its corresponding line segment. This will make the application more complicated and probably result in inferior performance. The situation is even worse when the application that uses the Snap Rounding representation comes as a black box and cannot be modified. In this case it will be more difficult, if possible at all, to associate polygonal chains with their original line segments.

Another result of the special output of SR is that the size of the output may increase significantly. While the output representation of CPLS is equivalent to the input representation, the output of SR may be quadratic in size if links that are shared by multiple line segments are counted once, and even cubic if not [52].

Advantages over Exact Arithmetic

As we discuss above, the ratio between the preprocessing time that CPLS requires and the work that the proceeding application needs to do on the data determines which method will be faster. As the later takes more time in comparison with the first, it becomes more plausible to use CPLS over EA. We give as an example two scenarios in which CPLS has good chances to beat EA. In the first (which is a typical computational geometry pattern), the input should be processed in order to provide a data structure to efficiently answer queries. If we expect a long stream of queries, it would potentially be beneficial to use CPLS because the majority of the time will be spent answering the queries—doing so with finite precision will save time. Another scenario is when multiple applications have to be executed on the same input since CPLS will need to approximate the input only once.

Since exact arithmetic results in large data, there is a risk of data explosion. Using CPLS will avoid it in many cases.

Some of the mathematic operations are not available with exact arithmetic (such as logarithm, trigonometry). If the proceeding process uses them, then the user would need to use work-arounds and tricks (if at all possible), which complicates his work.

The requirement to use EA makes the software less portable and available for different systems.

3.7 Experiments

We have implemented CPLS on a PC with Microsoft Visual C++ .NET (version 7.1) using CGAL [21]. The experiments were performed on a Microsoft Windows XP workstation with an Intel Pentium 4 3.2 GHz CPU and 2GB of RAM.

We performed many experiments to evaluate CPLS experimentally. Figures 3.9,3.10 and 3.11 depict different input sets before and after perturbation (for clarity, the examples we depict are much smaller than the data that we experimented with). We chose input sets that contain multiple intersections in small areas in order to obtain many potential degeneracy cases.
Figure 3.9: Experiment snapshots obtained with our software on different input sets. The input were created with a random generator.
Figure 3.10: Experiment snapshots obtained with our software on different input sets. Subfigure (a) shows an input to a robot motion planning problem. Subfigures (b) and (c) show the output of an intermediate step in the algorithm (the Minkowski sum of the robot and the obstacles). The obstacle triangles were created with a random generator.
Figure 3.11: Experiment snapshots obtained with our software on different input sets.
We implemented and experimented with the four sorting algorithms we presented in Section 3.4 (MIN-MAX, MIN-SUM-1, MIN-SUM-2 and MIN-SUM-3). By a slight abuse of notation, we refer to any CPLS program by the sorting algorithm it uses. For simplicity, we also use the term OPT to refer to any of the programs that uses one of the sorting algorithms. In order to evaluate the efficiency of the sorting algorithms, we compared their results with a CPLS program that uses random permutation to order the input (it would be a possible and popular choice when no sorting algorithm is available\textsuperscript{12}). We denote this program by RAN. On a whole, the perturbation magnitude obtained with all four sorting algorithms were very similar in all experiments. It turned out that for minimizing the maximum perturbation, MIN-MAX performed a little better than the other three on average. MIN-SUM-3 showed a slight advantage for minimizing the sum in some experiments. However, the differences were very small and not systematic. Hence, we do not discuss this specific issue further. More importantly, they all improved the results we obtained with RAN significantly as we show later.

There are two opposite factors on the processing time when comparing RAN with OPT. On one hand, OPT consumes time to sort the endpoints. On the other hand, since it results in less perturbations to perform (as a result of smaller forbidden loci), OPT requires less time to perform the perturbation step than RAN. It turned out that MIN-MAX was always substantially faster than the programs with the other sorting algorithms (usually the sorting step with the later ones took 50% to 100% more time in total). We note that we did not observe any other consistent behavior when comparing the running time of the sorting algorithms. Thus, MIN-MAX should be preferred over the other sorting algorithms if the processing time is important, as the results of all four are similar. However, if time allows, the user may only benefit from running all programs and take the best results. Figure 3.12 compares the average time obtained with MIN-MAX against RAN. While in the random line segment experiments\textsuperscript{13}(Figure 3.12(a)), MIN-MAX increased the total processing time, it decreased the total processing of the robot motion planning and triangulation overlay experiments (Figure 3.12(b) and 3.12(c), respectively). The differences were smaller than 40% and usually much smaller. We note that as the tests had more involved potential degeneracies to solve and thus their total processing time increased, MIN-MAX became more effective than RAN in terms of time (this is well illustrated in Figure 3.12). Note that MIN-MAX actually decreased the time of long computations.

Based on the discussion above, we used MIN-MAX to show the improvements in the perturbation magnitudes against RAN. The graphs (a), (b) and (c) in Figure 3.13 correspond to the experiments illustrated in Figure 4.15(c-d), Figure 4.15(a-b) and Figure 4.16, respectively. The graphs show by how much MIN-MAX decreased the perturbation that was achieved with RAN. The graphs show that we experienced improvements, especially for decreasing the maximum perturbation. The improvements became more significant as the tests became denser, thus having more involved degeneracies to solve and larger perturbations. We emphasize that this is a desirable characteristic, as it becomes more important to decrease the perturbation magnitudes when they become large. It worth noting that the improvement can be much bigger. For example, consider the case in Figure 4.17(a-b) where all line segments intersect in one point. As the intersection gets closer to the bottom or the top of the structure, it becomes more important to use a sorting algorithm. Suppose the intersection is located very close to the top. Then it becomes very important to process the bottom endpoints first (see Section 3.4 for the rational of this claim). All our sorting

\textsuperscript{12}Random order is used in many algorithms to eliminate "bad" input instances.

\textsuperscript{13}A random line segment is simply a line segment whose endpoints were chosen randomly inside a rectangle using uniform distribution.
Figure 3.12: Time as a function of the input size. The data correspond to the average of 10 executions. We report the average total running time for RAN and MIN-MAX. For the latter, we also show the time it took to sort the endpoints.
algorithms sort the endpoints of this example accordingly. As expected, RAN generated much larger perturbation than OPT, especially as the intersection point gets closer to the bottom or the top. Just for an illustration, with an input of 50 line segments where the ratio of the distance from the intersection to the top and the distance from the intersection to the bottom was 1:10, using OPT improved the average perturbation by a factor of roughly two hundredths. Another example with such a drastic improvement is depicted in Figure 4.17(c-d). The improvement is achieved because of many intersection of three line segments at the lower left of the model.

Table 4.2 shows the effect of different resolution bound sizes on the perturbation magnitudes. Not surprisingly, as the input and the resolution bounds become larger, there are more potential degeneracies to solve that result in larger perturbations. The running time increases accordingly. From the table we learn that for most input sets the perturbation was too large when $\varepsilon_1 = 0.5$ units (the segments in the input lie on a grid of 100X100 units). Other values of $\varepsilon_1$ that we tested produced relatively satisfactory results and were $\lambda$-approximated for small $\lambda$ values. Note that using different resolution bounds is similar to scaling the data up/down while maintaining constant resolution bounds. Based on our results, it might be beneficial to scale the input up.

Table 4.5 shows the results of the random line segments experiment with $\varepsilon_1 = 0.15$. It shows that the forbidden locus of Section 4.4 occurs the most. The reason is that there are $O(n^2)$ intersection points that may induce potential degeneracies. Two more conclusions from the table are: (1) As the number of line segments increases, the fraction of the endpoints that need to be perturbed increases too. (2) Increasing the input size causes the increase of the number of perturbations per endpoint (the ratio between the third and the second column).

<table>
<thead>
<tr>
<th># Line Segments</th>
<th>0.5</th>
<th>0.15</th>
<th>0.05</th>
<th>0.015</th>
<th>0.005</th>
<th>0.0015</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.61</td>
<td>0.033</td>
<td>0.002</td>
<td>0.000281</td>
<td>3.48e-05</td>
<td>0.015</td>
</tr>
<tr>
<td>200</td>
<td>0.115</td>
<td>0.103</td>
<td>0.099</td>
<td>0.094</td>
<td>0.092</td>
<td>0.09</td>
</tr>
<tr>
<td>300</td>
<td>0.074</td>
<td>0.0055</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td>400</td>
<td>0.666</td>
<td>0.082</td>
<td>0.000281</td>
<td>0.000281</td>
<td>0.000281</td>
<td>0.000281</td>
</tr>
<tr>
<td>500</td>
<td>0.42</td>
<td>0.013</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
</tr>
<tr>
<td>600</td>
<td>0.24</td>
<td>0.273</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
</tr>
<tr>
<td>700</td>
<td>0.42</td>
<td>0.013</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
</tr>
<tr>
<td>800</td>
<td>0.24</td>
<td>0.273</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
</tr>
<tr>
<td>900</td>
<td>0.42</td>
<td>0.013</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
<td>0.00085</td>
</tr>
<tr>
<td>1000</td>
<td>0.24</td>
<td>0.273</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
</tr>
</tbody>
</table>

Table 3.1: Results obtained with MIN-MAX on random line segments where the input size and $\varepsilon_1$ change (other resolution bounds were set accordingly). The data is in the format $X/Y/Z$ where $X$ is the average perturbation, $Y$ is the average number of perturbed endpoints and $Z$ is the average running time.
Figure 3.13: Results of using MIN-MAX against RAN. The X-axis corresponds to the size of the input and the Y-axis corresponds to the factor of improvement that MIN-MAX achieved, compared to RAN. We performed 10 tests for each case. The graphs in (c) correspond to tests with overlay of triangulations of polygons with 100 endpoints. We illustrate the maximum perturbation and average perturbation improvements. For any kind of experiment, let $k$ be the number of tests we performed. Let $A_i$ and $M_i$ be the average and maximum perturbation obtained in the $i$-th test, respectively. Then $MM = max_{1 \leq i \leq k} M_i$, $MA = max_{1 \leq i \leq k} A_i$, $AM = \frac{\Sigma_{1 \leq i \leq k} M_i}{k}$ and $AA = \frac{\Sigma_{1 \leq i \leq k} A_i}{k}$. 

87
<table>
<thead>
<tr>
<th># Line Segments</th>
<th>Number of perturbed endpoints</th>
<th>Number of perturbations</th>
<th>Number of induced forbidden loci Section 4.1</th>
<th>Number of induced forbidden loci Section 4.2</th>
<th>Number of induced forbidden loci Section 4.3</th>
<th>Number of induced forbidden loci Section 4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>25.5</td>
<td>164.5</td>
<td>0</td>
<td>16.2</td>
<td>0</td>
<td>187.5</td>
</tr>
<tr>
<td>200</td>
<td>64.7</td>
<td>450.1</td>
<td>1.5</td>
<td>105.6</td>
<td>0</td>
<td>480.7</td>
</tr>
<tr>
<td>300</td>
<td>130</td>
<td>1138.6</td>
<td>4.7</td>
<td>286.9</td>
<td>2.3</td>
<td>1171</td>
</tr>
<tr>
<td>400</td>
<td>214.1</td>
<td>2085.9</td>
<td>7</td>
<td>666.9</td>
<td>3.2</td>
<td>2151.1</td>
</tr>
<tr>
<td>500</td>
<td>296.1</td>
<td>3788.5</td>
<td>31.1</td>
<td>1283.5</td>
<td>6.9</td>
<td>3788.8</td>
</tr>
</tbody>
</table>

Table 3.2: Results of sets of random line segments of different sizes with $\varepsilon_1 = 0.15$. We note that in order to obtain fair statistics we checked for all possible potential degeneracies in each test. (In our program we perturb an endpoint as soon as one potential degeneracy is detected.)

### 3.8 The MIN-MAX Problem With Negative Weights

In Section 3.4.2 we gave a polynomial time algorithm for MIN-MAX.

Interestingly, if we allow negative weights on the edges, the problem becomes NP-complete as we prove in this section. Note that this variant could be viewed as a special case of the Feedback Arc Set problem. We start by formalizing the decision problem.

**INSTANCE:** A directed weighted graph $G = (V, E)$ with negative weights and a constant $H$.

**QUESTION:** Does $G$ embed an order of vertices $\Pi(V)$ such that for each vertex $v \in V$, $w_\Pi(v) < H$ ($w_\Pi$ is defined in Section 3.4.2).

**Theorem 3.9.** The MIN-MAX problem with negative weights is NP-complete.

**Proof.** The problem is clearly in NP: for every node $v \in V$, we simply compute $w_\Pi(v)$ by traversing $G$ and $\Pi$ in polynomial time and compare it with $H$.

In order to prove that the problem is NP-hard, we use a reduction from 3SAT. Suppose a formula $F$ contains $k$ clauses and $m$ variables. For each clause, we hold a node $C_i$, $1 \leq i \leq k$. The gadget for each variable $v_j$, $1 \leq j \leq m$, contains two nodes that correspond to the literals $v_j$ and $\bar{v}_j$ and other four nodes, $a_i$, $b_i$, $c_i$ and $d_i$ as illustrated in Figure 3.14(a) where the weights on the edges are shown as well. We denote the gadget of variable $v_i$ by $G_i$ and use it as a black box in Figures 3.14(b) and 3.14(c). Each clause node $C_i$ is connected to each $G_j$ through the nodes $v_j$, $\bar{v}_j$ and $a_i$ as illustrated in Figure 3.14(b). Finally, each clause is connected from the three literals that correspond to the negations of its literals. The weight of each such edge is $1/3$ (See Figure 3.14(c)). We let $H = 1$. A complete example is shown in Figure 3.15.

We next prove that $F$ is satisfiable if and only if the maximum weight is smaller than 1. We denote by $a \Rightarrow b$ the relation that forces $a$ to be selected before $b$. Such order is necessary to prevent $w_{\Pi}(v) : v \in V$ from exceeding 1. Immediate orders that are necessary (based on our construction) are: $b_i \Rightarrow a_i$, $c_i \Rightarrow b_i$, $d_i \Rightarrow c_i$ and $a_i \Rightarrow C_j$. Other orders that are forced are the following. By transitivity, we have $d_i \Rightarrow b_i$. Thus, $b_i$ gains a weight of 1 from this relation. To compensate, at least one of $v_i \Rightarrow b_i$ or $\bar{v}_i \Rightarrow b_i$ must hold. Any order between $v_i$ and $\bar{v}_i$ causes the other vertex to have a weight of $k$. It must be compensated by all $C_j$ in order to decrease the total weight below 1. Thus, any $C_j$ must precede at least one of $v_i$ and $\bar{v}_i$ (either $C_j \Rightarrow v_i$ or $C_j \Rightarrow \bar{v}_i$ must hold). Note also that all variable gadgets are independent. Thus, any orders between vertices of different
Figure 3.14: The reduction from 3SAT

Figure 3.15: An example of the reduction for the formula \((\overline{v}_l \cup v_k \cup \overline{v}_i) \cap (v_l \cup v_k \cup v_i)\)
gadgets is irrelevant. Consider any clause $C_j$ and any gadget $G_i$. All the above constraints leave us with four possibilities on the order of the vertices: $(d_i, c_i, \bar{v}_i, a_i, C_j, v_i)$, $(d_i, c_i, b_i, a_i, C_j, \bar{v}_i)$, $(\bar{v}_i, d_i, c_i, b_i, a_i, C_j, v_i)$ and $(v_i, d_i, c_i, b_i, a_i, C_j, \bar{v}_i)$. The important observation from these orders is that in all exactly one of $v_i$ or $\bar{v}_i$ proceeds $C_j$, which in turn proceed the other literal node. The remaining condition for the weights to be less than 1 is that the weights directed to $C_j$ be less than 1. It means that for each $C_j$, at most two literals can precede it. Note that it is the same as to say that in $F$, at least one literal is true, satisfying the clause. So if the formula is satisfied, we choose all the literals with false values before all $C_j$. Then we are guaranteed that at most two literal nodes precede any clause and as a result the overall minimum weight is less than 1. On the other hand, if the maximum weight is less than 1, then it must be the case that at most two literal nodes precede each clause. It follows that in $F$ at least one literal satisfies the corresponding clause.

3.9 Conclusions and Future Work

We presented a Controlled Perturbation scheme for sets of line segments in $\mathbb{R}^2$. The idea of the scheme is to approximate sets of line segments to make subsequent algorithms more robust when working on the data. The goal of the approximation is to eliminate potential degeneracies that are major source of errors when using finite-precision arithmetic. We implemented our scheme and reported experiments. Based on our results and the simple output representation that CPLS produces, we believe that it can be used as an efficient Finite Precision Approximation algorithm.

Possible large perturbation is one of the main drawbacks of Controlled Perturbation schemes. We presented a novel method to decrease the perturbation. The idea behind our method is that different input processing orders yield different perturbation magnitude. Our method applies geometric graph heuristics to sort the endpoints processing order intelligently before the perturbation step. We designed and implemented several variants for that purpose. Our experiments show that they decrease the perturbation significantly in many cases.

We emphasize that the improvement factor (that is by how much the perturbation decreases when using our sorting method instead of random order) increases when the potential degeneracies become more involved and require more perturbation. Such cases that result in large perturbations are problematic for any Controlled Perturbation scheme. As we observed, using our method can be the difference between a good and poor approximation. Our experiments show that the penalty of using one of the ordering heuristics (MIN-MAX) does not affect the total processing time much and in many cases even decreases it. We believe that based on our results, it may be useful to consider our ideas in other Controlled Perturbation schemes, in order to decrease their perturbation magnitude.

Our sorting method depends on the order of two adjacent endpoints. It would be interesting to design more sophisticated sorting heuristics that consider more involved relationships (for example, the relationship among three intersecting line segments).

We believe that our scheme can be used for real-world applications that need to separate geometric features. As an example we sketch the following problem from the Graph Drawing field. We are given a graph $G$ and we wish to draw it with captions surrounded by rectangles on the endpoints of $G$. For visualization purposes, the goal is to guarantee that no non-adjacent edges crosses the caption rectangles. By defining any such case as degenerate, we believe that our scheme can be adapted to achieve this goal.
An interesting direction for future research is to develop global perturbation algorithms that achieve some provable quality. The idea is either to maximize the separation while using congruent perturbation discs or to minimize the perturbation while fixing the minimum separation. A more simple variant of these problems was proven to be NP-complete [17].

In this work we theoretically compared CPLS with Snap Rounding and Exact Arithmetic. We hope to perform experimental comparison among these, and possibly with other techniques, in the future. We foresee several challenges in carrying out such comparison as we discuss next. One major challenge would be to determine reasonable parameter values for CPLS (resolution bounds) and SR (hot pixel size). Since the values of these parameters effect the performance significantly, it is important to select values that reflect the input and the machine precision in order to obtain realistic results. Thus, the comparison should follow deep theoretical analysis of the schemes (see [51] for a corresponding CP analysis on circles in $\mathbb{R}^2$). The situation may be even more complicated when comparing with Exact Arithmetic because the later embeds a completely different methodology. Thus, it would be challenging to construct a variety of experiments that reflect different possible scenarios. We finally note that comparing with Exact Arithmetic could be a part of a comprehensive work that cover also other CP schemes that have been developed and implemented (see Section 5.1 for details on these schemes).
Part II

Optimal Visibility Coverage
Chapter 4

Locating Guards for Visibility Coverage of Polygons

We propose heuristics for visibility coverage of a polygon with the fewest point guards. This optimal coverage problem, often called the “art gallery problem”, is known to be NP-hard, so most recent research has focused on heuristics and approximation methods. We evaluate our heuristics through experimentation, comparing the upper bounds on the optimal guard number given by our methods with computed lower bounds based on heuristics for placing a large number of visibility-independent “witness points”. We give experimental evidence that our heuristics perform well in practice, on a large suite of input data; often the heuristics give a provably optimal result, while in other cases there is only a small gap between the computed upper and lower bounds on the optimal guard number.

4.1 Introduction

The art gallery problem was introduced in 1973 when Klee asked how many guards are sufficient to “guard” the interior of a simple polygon having \( n \) vertices. Although it was shown that \( \lceil \frac{n}{3} \rceil \) guards are always sufficient and sometimes necessary [26, 37], and such a set of guards can be computed easily, such solutions are usually far from optimal in terms of minimizing the number of guards for a particular input polygon. Moreover, it was shown that determining an optimal set of guards is NP-hard, even for simple polygons [62]. Deshpande et al. presented a pseudo polynomial time algorithm that achieves an \( O(\log^2 n) \)-approximation factor. As far we we know, this algorithm have not been implemented and tested.

Approximation algorithms with logarithmic approximation ratios are known for somewhat restricted versions of the problem, e.g., requiring guards to be placed at vertices or at points of a discrete grid[35, 42, 44]. Constant-factor approximations are known for guarding 1.5D terrains and monotone polygons[15, 61, 68], and exact methods are known for the special case of rectangle visibility in rectilinear polygons [86].

Our Contribution. We propose heuristics for computing a small set of point guards to cover a given polygon. While we are not able to prove good worst-case approximation bounds for our methods (indeed, each can be made “bad” in certain cases), we conduct an experimental analysis of
their performance. We give methods also to compute lower bounds on the optimal number of guards for each instance in our experiments, using another set of implemented heuristics for determining a set of visibility-independent witness points. (The cardinality of such a set is a lower bound on the optimal number of guards.) We show that on a wide range of input polygons, our heuristics work well in practice, often matching our computed lower bounds (in which case the solution is provably optimal), and always yielding at most 2 times the lower bound (for the randomly generated instances). To our knowledge, ours is the first attempt to conduct a systematic experimentation with guard placement heuristics, together with methods to compute lower bounds that give provable performance bounds in terms of approximation ratios.

Since the art gallery problem is related to many problems in surveillance, sensor networking, robot planning and more, we believe that our heuristics can be useful as an intermediate step in their solutions. Packer[71] used the heuristics and the code of this work to implement heuristics for multiple watchman routes.

Our implementation is built on top of the CGAL arrangement package. Our experiments are conducted on a variety of polygons, including many generated “randomly” using the RPG software of Auer and Held[11].

Related Work. Surveys on the art gallery problem are given in [5, 70, 82, 85]. A related problem to computing a minimum guard cover is the problem of computing optimal partitions of polygons, e.g., into the fewest convex or star-shaped subpolygons. These problems are theoretically much easier than coverage problems; polynomial-time algorithms based on dynamic programming are known[12, 60]. Of course, if a polygon can be partitioned into $k$ convex or star-shaped subpolygons, it can be guarded using at most $k$ guards; thus, partitioning can serve as a basis for one type of heuristic for guard coverage. We note, however (see Section 4.5.1), that there are simple examples for which a constant number of guards are sufficient to cover, while the best partition is of linear size.

4.2 Preliminaries

The input is a (possibly multiply connected) polygon $P$ having a total of $n$ vertices on its boundary. We let $h$ be the number of holes in $P$; if $h = 0$, $P$ is said to be simple. For points $p, q ∈ P$, we say that $p$ and $q$ are visible to each other if the segment $pq$ is contained in $P$. For $p ∈ P$, we let $V(p)$ denote the visibility polygon of $p$, i.e., set of all points $q ∈ P$ that are visible to $p$. Clearly, $V(p)$ is star-shaped and $p$ belongs to its kernel. Similarly, for a set of points $P ⊆ P$ we denote by $V(P) = \bigcup_{p \in P} V(p)$ the union of all of the visibility polygons of the points of $P$. We say that a set $G ⊆ P$ of points is a guarding set of $P$ if $V(G) = P$. We let $g(P)$ be the number of guards in a minimum-cardinality guarding set of $P$.

We say that a finite set, $I ⊆ P$, of points in $P$ is a visibility-independent set of witness points if the visibility polygons $V(p) : p ∈ I$ are pairwise-disjoint: $\forall p, q ∈ I V(p) \cap V(q) = \emptyset$. We let $i(P)$ denote the independence number of $P$, defined to be the number of witness points in a maximum-cardinality visibility-independent set. Clearly, $g(P) \geq |I|$ for any visibility-independent set $I$, since no single guard is able to see more than one point of $I$. Thus, if we find a visibility-independent set $I$ and a guarding set $G$ such that $|I| = |G|$, then $G$ is an optimal guarding set (e.g., see Figure 4.15(e)). Not all polygons admit two such sets; indeed, there can be an $Ω(n)$ gap between $i(P)$ and $g(P)$ (see
Section 4.5.1). We say that a polygon for which \( i(P) = g(P) \) is a \textit{perfect polygon} (in analogy with the concept of perfect graphs).

In our experiments, our goal is find small guarding sets \( G \) and large visibility-independent sets \( I \); the set \( G \) we produce approximates the optimal guard number, \( g(P) \) with approximation ratio \( |G|/|I| \).

4.3 Algorithms

4.3.1 Greedy Algorithms.

A natural approach to placing guards is to do so greedily: Add guards one by one until coverage is achieved, choosing at each step a guard from among a set of candidates in order to maximize its contribution to the coverage (e.g., the “amount” of \( P \) that it sees that was not previously covered). Greedy methods are well known in set cover problems, as they yield logarithmic approximation bounds (which are best possible in some sense). We formulate this process as having two phases: First, we construct a set \( S \) of candidate guards that serve as a cover of \( P \): \( P = \mathcal{V}(S) \). Then, we greedily select a (much smaller) subset, \( G \subset S \), of the candidates that also serves to cover \( P \).

- Constructing a candidate set: The challenge of this step is to come up with a “good” candidate set \( S \) from which a good guarding set can be extracted.
- Criteria for greedily choosing guards \( G \subset S \): From \( S \) we choose a smaller set \( G \) that also covers \( P \). The challenge here is to derive a good heuristic for choosing guards that results in a small set \( G \).

Constructing a Candidate Set.

We experiment with three choices of candidate sets.

The first one, \( V(P) \), is essentially the set \( V \) of vertices of \( P \); it is easy to see that \( V \) guards all of \( P \). For implementation purposes, we actually used points arbitrarily close to the vertices of \( P \) from inside, in order to avoid degenerate visibility on the adjacent edges – see Section 4.2. The exact position is on the ray that bisects the interior angle of the corresponding vertex. We also made sure that each point is actually inside \( P \).

In our second choice of candidate set, we place the candidate points at the centers (centers of mass) of the convex polygons in a decomposition of \( P \) induced by an arrangement of certain line segments. We consider two different arrangements. One arrangement is defined by edge extensions, extending each edge that is incident to a reflex vertex \( v \), extending it beyond \( v \) until it intersects the boundary of \( P \) at some other point; see Figure 4.1(a). The second arrangement is defined by extensions of visibility graph edges that are incident to at least one reflex vertex, \( v \): If \( v \) sees vertex \( u \), then we construct a segment \( vw \) that extends along the line through \( u \) and \( v \), away from \( u \), until it first encounters a point \( w \) on the boundary of \( P \). See Figure 4.1(b). The arrangement of these visibility extensions also give rise to an arrangement having convex faces within \( P \). Finally, \( C(P) \) is defined to be the set of centers of mass for the convex faces in the arrangement (either of edge extensions or visibility extensions). It is easy to see that \( C(P) \) guards \( P \). Note that there
are $O(n)$ edge extensions (thus, their arrangement has worst-case complexity $O(n^2)$) and there are $O(n^2)$ visibility extensions (with arrangement complexity $O(n^4)$). The intuition behind using these partition-based candidate sets is that different subpolygons will contain points with different combinatorial type with respect to visibility within $P$. For instance, in Figure 4.1(a), points in different subpolygons see different subsets of the reflex angle edges: points in the left subpolygon see the left edge incident to the reflex vertex, points in the right subpolygon see the right edge, and points in the middle subpolygon see both. It is clear in this example that the best choice is to select a guard from the “middle” subpolygon. We later try to construct criteria that reflect this intuition.

Finally, the third choice of candidate set is simply the union: $V(P) \cup C(P)$.

**Extracting a Small Guarding Set.**

We experiment with the three candidate sets, $V(P)$, $C(P)$ and $V(P) \cup C(P)$, defined in the previous subsection. In the descriptions below, we use $W(P)$ to denote the candidate set chosen. Our goal is to generate a guarding set $G \subseteq W(P)$ that is as small as possible; in general, we expect $|G|$ to be much less than $|W(P)|$.

We greedily add “good” candidates to the guarding set $G$, until the entire polygon is covered. The next guard is selected to maximize a certain measure, among all of the remaining candidates. We considered the following algorithms, labeled “$A_1$” through “$A_{13}$” (another algorithm, $A_{14}$, is presented in Section 4.3.2):

- **$A_1$.** The candidates are $W(P) = V(P) \cup C(P)$, with $C(P)$ based on the arrangement of edge extensions. The score, $\mu(c)$, for a candidate $c \in W(P)$ is the number of points of $W(P)$ that are seen from $c$ but not from any point of $G$. At each iteration of the algorithm, the candidate $c$ with the highest score $\mu(c)$ is added to $G$, and the scores $\mu(\cdot)$ are updated accordingly.

- **$A_2$.** $A_2$ is similar to $A_1$ with the following modification: With each candidate $c$ added to $G$, the arrangement is augmented with the edges of $\mathcal{V}(c)$ that are not on the boundary of $P$. The idea is to enrich the set of candidates to reflect the still uncovered portion of the polygon. See Figure 4.2.

- **$A_3$.** $A_3$ is similar to $A_1$, but the score $\mu(c)$ of candidate $c$ is the area seen by $c$ that is not yet seen by $G$. This algorithm requires that after each candidate is added to $G$, we update the
(unguarded) visible area for each remaining unused candidate.

- **A4.** A4 is similar to A1, but uses a score \( \mu(c) \) that weights the candidates \( c' \) that are not-yet-covered by \( c \), by the (precomputed) area of the cell corresponding to \( c' \).

- **A5.** A5 is the same as A4, but the weight associated with \( c' \) is the length of the boundary of the cell in common with \( \partial P \), instead of its area.

- **A6.** A6 is the same as A4, but the weight associated with \( c' \) is the fraction of the perimeter of the cell that is in common with \( \partial P \).

- **A7.** A7 is like A1, but with candidates \( W(P) = V(P) \) (vertices of \( P \)).

- **A8.** A8 is like A1, but with candidates \( W(P) = C(P) \).

- **A9.** A9 is like A1, but with the score \( \mu(c) \) defined to be the number of not-yet-covered vertices seen by \( c \).

- **A10.** A10 is like A1, but with the score \( \mu(c) \) defined to be the number of not-yet-covered cell centers seen by \( c \).

- **A11.** A11 is like A1, but with the arrangement based on visibility extensions. As a result, the number of candidates becomes much larger (worst-case \( O(n^4) \), versus \( O(n^2) \)).

- **A12.** A12 is a combination of A2 and A11, using dynamically added segments, and arrangements of visibility extensions.

- **A13.** A13 is a probabilistic algorithm, based on the Brönnimann-Goodrich framework [16, 35]. Each candidate is assigned a weight dynamically, proportional to its chances to be selected. Initially, each point is assigned weight 1. In each iteration, a guard is selected randomly. Then a random point \( p \) that is still unguarded is selected. We find all non-guarding candidates that see \( p \) and double their weights, improving the chances of guarding \( p \) in the next iterations. This process gives an \( O(\log \Phi) \)-approximation on average, where \( \Phi \) is the optimal number of guards selected from \( W(P) \); Efrat and Har-Peled applied this strategy for candidates defined by a grid[35].

All of the above heuristics may result in a set \( G \) that is not minimal – i.e., it may be possible to remove one or more guards while still covering \( P \). Thus, we apply a post-processing step in which we iteratively remove redundant guards until we are left with a minimal guarding set.

**Remark.** As reflected in our list of algorithms, we formulated several parameters that control the behavior. Together, they can be combined to yield numerous variants, making it tedious and impractical to test all of them. Instead, we defined a basic variant (A1), which we believed would give good results and be relatively simple and time efficient. Other variants differed in one or two parameters from A1. We were interested both in the effect of this modification and in the possibility to modify parameters further. As our experiments showed (see Section 5.4), the only positive influences were obtained with A2 and A11. Thus, we also tried a combination of the two, A12.
Figure 4.2: Using algorithm $A_2$: (a). The polygon and the first guard to be selected (shaded). (b). The visibility polygon of the guard (large disk) caused the addition of 8 new candidates (small disks).

4.3.2 Methods Based on Polygon Partition.

A very different approach is to base the guarding on partitioning the polygon into pieces, each with an assigned guard:

- $A_{14}$. Partition the polygon into star-shaped pieces, and place one guard within the kernel of each piece. While a polynomial-time algorithm is known for partitioning a simple polygon into the fewest star-shaped pieces, we opt instead to apply a simpler (and faster) heuristic, which applies also to polygons with holes: Triangulate $P$, and then iteratively delete diagonals (according to a heuristic priority), merging two adjacent faces as long as the resulting new face is still star-shaped (this is similar to the Hertel-Mehlhorn’s heuristic that 4-approximates the minimum convex partition).

Since the results of this heuristic were relatively poor (see Section 5.4), we did not explore other variants of the algorithm further. Note too that for some “spike box” polygons, any partition-based algorithm gives very poor results compared with greedy coverage methods (see the discussion in Section 4.5.1).

4.3.3 Algorithms for Visibility-Independent Sets.

As we described in Section 5.1, a visibility-independent set $I$ provides a lower bound on the guarding number. Since maximizing $|I|$ is NP-hard, we resort to heuristics for finding independent sets. (While there are effective combinatorial optimization methods for exactly computing maximum independent sets, we opted here to use faster heuristics.) We apply a greedy heuristic similar to our guard placement strategies: We start with an initial set $S$ of candidates (not independent) and iteratively add visibility-independent points to a set $I$ (initially empty), adding at each step a point that sees the least number of points in $S$. After adding a point to $I$, we remove all of the points in $S$ that see it. We stop when $S$ is empty.

We note that if a point $p \in \partial P$ is perturbed slightly into the interior of $P$, its visibility polygon usually enlarges; see Figure 4.3(a). Since in our greedy algorithm, we want to choose points that see
as little as possible, a natural heuristic is to choose points of \( \partial P \) as candidates. Also, when perturbing a point at a convex vertex along \( \partial P \), the visibility polygon usually enlarges; see Figure 4.3(b). Thus, we include convex vertices in the candidate set. For reflex vertices, however, a perturbation away from the vertex along \( \partial P \) generally causes the visibility polygon to become smaller; see Figure 4.3(c). This motivates using a candidate set that includes two kinds of candidates: convex vertices (denoted by \( I_1 \)) and midpoints of edges incident to two reflex vertices (denoted by \( I_2 \)). Let \( I_3 = I_1 \cup I_2 \). We experimented with independent sets built using candidates of all 3 types. Not surprisingly, it turned out that \( I_3 \) gives the best results, as it has a richer set of candidates.

Figure 4.4 shows a test on a 44-vertex polygon. The guarding candidates set, the guarding set, and the visibility-independent set are shown.

### 4.4 Implementation Details

#### 4.4.1 Computing Extension Edges.

In all cases except vertex guard candidates, we compute a subdivision of the polygon into cells, as described in Section 4.3.1, by extending certain line segments that connect pairwise visible (possibly adjacent) vertices. If \( P \) has no holes, we can apply the algorithm of Hershberger[54], using time \( O(n + k) \); otherwise, we can compute the visibility graph in time \( O(k + n \log n) \) (e.g., [43]). Here, \( k \) is the output size and is the number of possible visibility extensions (at most \( O(n^2) \)). If we use only edge extensions, then there are \( O(n) \) such extensions, and each can be determined using a ray-shooting query (\( O(\log n) \) time query). The extensions, together with \( \partial P \), constitute the input segments for the arrangement computation.

#### 4.4.2 Using the Arrangement to Compute a Candidate Set.

The polygon boundary edges and the edge- or visibility-extensions are the segments for which we compute an arrangement, \( A(P) \), of complexity \( O(n^2) \) (for edge-extensions) or \( O(n^4) \) (for visibility-
Figure 4.4: The result of running a test on a polygon with 44 vertices with heuristic $A_1$, independent candidate set $I_3$, and using edge extensions for the polygon subdivision. The large disks are the guards, the squares are the visibility-independent points, the light small disks are the remaining candidates, and the segments inside the polygon are the edge extensions.

Each (convex) cell of $A(P)$ contributes one candidate, which we take to be the center of mass of the cell.

We use the CGAL arrangement package to compute $A(P)$. We found that using exact arithmetic is extremely slow; thus, we opted to use floating point, with some small perturbations of segment endpoints to avoid robustness difficulties. Since the endpoints of extensions are imprecise, we may erroneously combine two adjacent cells if we are not careful. To avoid this error, we push the extension endpoint slightly beyond the intersection with $\partial P$. In this way, the interior cells are separated and either the unbounded cell or a hole has “notches” (see Figure 4.5). Although those cells are affected, they are irrelevant since they are outside of $P$. We did not attempt to compute the minimum perturbation to be robust; rather, we assumed that a small extension by a prespecified $\epsilon$ was sufficient for our purposes (and never had an issue arise from this assumption). We were careful, however, to check that this extra extension did not result in reentering the interior, which may happen if two edges are very close. In this case, we decrease the extra extension accordingly and still assume that it is sufficient for robust computation. However, we did not encounter polygons with this rare feature in our experiments.

After computing the full arrangement $A(P)$, we compute the candidates of each cell. Then we compute the pairwise visibility among the candidates. If the polygon is simple, one can apply the algorithm of Ben-Moshe et al. to do so in time $O(n^2 \log^2 n + k)$ or $O(n^4 \log^2 n + k)$ (using space $O(n^2)$ or $O(n^4)$), for edge- and visibility-extensions, respectively[14]. In some of our heuristics we need to extract features of the cells (see Section 4.3.1); this takes time linear on the complexity of the cells and thus linear in $|A(P)|$ (worst-case $O(n^2)$ or $O(n^4)$).

### 4.4.3 Computing a Guarding Set.

We iteratively choose a guard and make the necessary updates until the polygon is covered. We next describe the different routines of this process.
Choosing the Next Guard.

This routine depends on the algorithm we use. We use two kinds of algorithms: the greedy and the probabilistic.

Greedy. We check each candidate that has not yet been selected. According to the scoring criteria of the algorithm, we score each candidate and select the one with the highest score to be the next guard. The work for each candidate is constant, except in the case of heuristic $A_3$, for which we need to update the arrangement for each new guard ($O(n \log n)$ time for each).

Probabilistic. We simply select the guard chosen randomly with respect to the weights. Then, we remove its weight from the total weights.

It is worth mentioning that in both types of algorithms it is possible that all candidates are visible to the temporary list of guards, yet the polygon is not yet covered. In this special case we identify the area that is not covered (using the data structure in Section 4.4.3), compute its center of mass and find a candidate that sees this point by using point location to find in which cell of $A(P)$ it is located.

Updating the Data Structures.

Once a candidate guard is selected, we need to update the relevant data structures before choosing the next guard. Again, it differs for each kind of algorithm we use, as we describe below.

Greedy. We update the weights of the candidates as follows. Let $g$ be the new guard and $K(g)$ be the visible candidates from $g$. Among these, there may be candidates that are newly guarded and thus need not be considered any more. For each such candidate, $c$, we again take all of its
visible candidates. For each, we update its weight by removing the effect of $c$. All of the information about the visibility among points was precomputed, as described in Section 4.4.2.

**Probabilistic.** We find a point that is still not guarded (if there are still such points), and double the weights of all of the candidates that see it (including the point itself).

**Checking Coverage.**

We maintain the region, $M$, that is guarded by the current guarding set, $G$, as follows. We initialize $M$ to be empty. For each new guard $g$ we perform $M = M \cup V(g)$ by performing a union of polygons. At each step we check whether $P = M$. If so, the polygon is fully covered. If the polygon is simple, then any of the above union computations takes linear time. (Note that $M$ never has holes.) If $P$ has holes, then $M$ can have holes and can have quadratic complexity; thus, each computation may take quadratic time. In order to compute visibility polygons we implemented the linear-time algorithm of Joe and Simpson[58], for visibility in simple polygons, and we use a radial sweep around the guard, for visibility in polygons with holes (time $O(n \log n)$).

**4.4.4 Removing Redundant Guards.**

By using the several heuristics we described above, we decrease the guarding set significantly from the initial set of candidates. However, it may still be the case that more guards can be removed while maintaining full coverage. It may happen that after selecting a guard $g$, other guards are selected such that they cover the parts covered alone by $g$ when it was chosen.

In order to remove redundant guards, we use the following routine. We maintain an arrangement of visibility polygons of the guarding set. We traverse the guards and for each guard do the following. We remove the guard’s visibility polygon edges from the arrangement. Then, we check whether the new faces are covered by the remaining guards. If they are, the guard is removed from the guarding set. Otherwise, we return the removed visibility polygon edges to the arrangement. By the time the routine is completed, the guarding set is clearly minimal.

**4.4.5 Computing Visibility-Independent Sets.**

In Section 4.3.3 we explained how we construct the independent candidate set and how we select the next independent point. We use an arrangement, $A$, of visibility polygon edges. For each point being selected, we insert the corresponding visibility polygon into $A$ and check whether it intersects previous visibility polygons. If it does not, we insert the point to the independent set. Otherwise we remove its visibility polygon edges from $A$.

**4.5 Experiments**

We have implemented our various algorithms on a PC using openGL and CGAL (version 3.1) libraries. Our software works with Microsoft Windows XP with Visual .Net compiler. The tests
were performed on a Microsoft Windows XP workstation with an Intel Pentium 4 CPU 3.20GHz, 2.00 GB of RAM.

We have performed extensive experiments with the algorithms we described in Section 4.3.1. In this section we report our results and conclusions from our experiments.

We found \textit{A}_1 (the basic algorithm) to be very useful in the sense that both the guarding was efficient and the time and space required were relatively reasonable. The results obtained with algorithm \textit{A}_2 were satisfactory and overall showed to be a little better than \textit{A}_1. However, we paid for a longer processing time. Although the algorithm \textit{A}_3 seems useful for testing, it was extremely slow—too slow for being a candidate here. Thus, we omit its results here. Algorithms \textit{A}_4, \textit{A}_5 and \textit{A}_6 gave reasonable results, but \textit{A}_1 turned out to be at least as good or better in most tests we performed. Algorithms \textit{A}_7, \textit{A}_8, \textit{A}_9 and \textit{A}_{10} were usually worse or equal to \textit{A}_1. \textit{A}_7 had very bad results for the spikes box (Figure 4.15(f)). Algorithm \textit{A}_{11} gave slightly better results than \textit{A}_1 and was comparable to \textit{A}_2. It was always slower than \textit{A}_1 and usually slower than \textit{A}_2. Naturally, it required more space than both \textit{A}_1 and \textit{A}_2. The algorithm \textit{A}_{12} gave the same results as \textit{A}_{11}. Since it is more complicated than \textit{A}_{11}, there is no reason to use it instead. The results of algorithm \textit{A}_{13} were not as satisfactory as \textit{A}_1. Also, the processing time was significantly higher, as the number of redundant guards was enormous, resulting in a lot of time for both computing the guarding set and removing some of them. The results of \textit{A}_{14} were the worst among all algorithms we tried. We implemented and tested this algorithm only with simple polygons. Based on the results, we found it useless to implement and test this algorithm for polygons with holes.

Our tests include both interesting cases produced manually and randomly generated polygons[11]. Figures 4.15, 4.16 and 4.17 shows some of the interesting cases we tested. They show the guarding and independent sets. Table 4.5 (see Section 4.5.3) shows statistics obtained with the algorithms we used on 40 input sets. As we show later, the best results were obtained with \textit{A}_2 and \textit{A}_{11} (ignore Algorithm \textit{A}_{12} for the moment). Those results were a little better overall than \textit{A}_1. Tables 4.1, 4.2 and 4.4 and the graphs in Figures 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11 provide statistics on the guarding quality and the time of \textit{A}_1, \textit{A}_2 and \textit{A}_{11}. While the guarding quality of \textit{A}_1 was usually equal or slightly worse than the other two, it took less time and space. \textit{A}_1 was faster than \textit{A}_2 which was faster than \textit{A}_{11} in general. Both (especially \textit{A}_{11}) also required more space than \textit{A}_1. \textit{A}_{11} was much slower and much more space costly with polygons whose vertex visibility graph has many edges (such polygons are depicted in Figure 4.15(a) and (f)). The good guarding results of \textit{A}_2 and \textit{A}_{11} motivated the testing of their combination, with dynamically inserted edges and visibility extensions (heuristic \textit{A}_{12}). However, the results with \textit{A}_{12} were almost identical to the results with \textit{A}_{11}. Thus, we found no use in considering \textit{A}_{12}. Based on all these results, we decided to concentrate on algorithms \textit{A}_1, \textit{A}_2 and \textit{A}_{11}. Figures 4.18 and 4.19 shows a few 100-vertex polygons we tested, with each of the algorithms \textit{A}_1, \textit{A}_2 and \textit{A}_{11}. As shown in this figure, the different heuristics were better for different examples. The three heuristics imposed a trade-off between guarding quality and time/space.

In order to explore their performance further and get more concrete information, we tested these three heuristics with more input sets, which strengthened our conclusions of the trade-off we described. This trade-off implies that when the input is small enough, or when time and space are not an issue, using \textit{A}_2 and \textit{A}_{11} would be recommended. As time and space become restricted, the use of \textit{A}_2 and especially \textit{A}_{11} becomes less desirable and motivates the use of \textit{A}_1.

In order to get a quantitative measure on the quality of the guarding sets computed, we computed visibility-independent sets too. The ratio between the best guarding and the best visibility-
an independent set (obtained with $I_3$) never exceeded 2, for randomly generated polygons, and was usually close to 1. That implies that our results were always a 2-approximation to the optimal guarding, and usually were much better. Table 4.6 shows the results of the three chosen algorithms for guarding and the three independent set heuristics we applied. The last column shows the maximum ratio between the minimum upper bound (guarding set size) and the maximum lower bound (independent set). For polygons with no edges with two reflex vertices, the set of candidates of $I_2$ is empty, thus no independent set can be computed and we placed X to represent that fact. Based on the table, we can see that the ratio never exceeds 2 and was usually smaller.

We analyze the time taken by the three algorithms. With those statistics, we want to show the correlation between the parameters and the time, and also to compare the running time of the three algorithms. Table 4.1 shows the number of guards as a function of the number of vertices. It shows that there is a slight correlation between the two, but generally it shows that the number of guards is almost independent of the number of vertices. It should be clear that generally there is no correlation between the two, as any convex polygon requires only one guard, while “comb” polygons (see Figure 4.16(c)) requires $n/3 - O(1)$ guards. Figures 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11 show the time as a function of different parameters for various kinds of polygons. Extensive tests on random and spike box polygons are given in Tables 4.2 and 4.4 respectively. The figures and tables show that although there is a correlation between the parameters and the time, this correlation is not strong and clearly there is a lot of noise. Thus, it would be hard to predict the time based on the parameters of the input. This observation can be explained also by the observation above about vertices vs. guards. Clearly both sizes should affect the running time and, since they are not highly correlated, the time is not correlated with each.

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>Number of guards $a_1$</th>
<th>Number of guards $a_2$</th>
<th>Number of guards $a_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>43</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>44</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>44</td>
<td>14</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>46</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>52</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>52</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>56</td>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>58</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>61</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>68</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>68</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>68</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>68</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>73</td>
<td>18</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>76</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>88</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>14</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of the guarding quality of $A_1$ (basic), $A_2$ (dynamic) and $A_{11}$ (full visibility).
Figure 4.6: (a). Comparison of the time as a function of guardung set size, comparing $A_1$ (basic), $A_2$ (dynamic) and $A_{11}$ (full visibility).

Figure 4.7: Comparison of the running time as a function of the extensions size, comparing $A_1$ (basic), $A_2$ (dynamic) and $A_{11}$ (full visibility).
Figure 4.8: A zoom-in of the plot in Figure 4.7

Figure 4.9: Time as a function of the number of vertices in the polygon.
4.5.1 Input Datasets.

We tried to explore an interesting and broad variety of input datasets. We tested the randomly generated polygons whose results are given in Table 4.2. We also constructed many more examples. We present some of them below.

Well. This interesting example demonstrates a case in which more than one guard “collaborates” in guarding “wells” (see Figure 4.15(a)). The challenge in guarding such polygons is that there is dependency among guards: if one moves, another will have to move too in order to compensate for the loss of visibility.

Spike box. This example is composed of a rectangular box with many spikes coming out of it (see Figure 4.15(f) and 4.17(d)). This example demonstrates how inefficient a partition technique can be. In Figure 4.15(f), two guards are sufficient to guard all spikes (and the rectangular area too). However, if we order the spikes around the rectangle perimeter and mark each spike according to the guard who sees it, we will have no consistency in this sequence. Let e be the number of guard changes along this sequence. If e is large, it is easy to see that any partition will subdivide the polygon into many pieces. In the worst case, a constant number of guards are sufficient to guard the polygon, while a linear number of guards is necessary with partition techniques. Figure 4.17(d) is an example in which the upper bound was much greater than the lower bound (here, a ratio of 4 was obtained). In general, this ratio can be $\Theta(n)$.

Experiment results of spike boxes with different size is give in Table 4.4

Special art gallery variants. Figure 4.15(c) shows an example for simulating line segments guarding (a rectangular polygon with skinny rectangular holes). Other variants of interest are the guarding of the boundary of the polygon from the outside (Figures 4.15(h), 4.16(a) and 4.17(a))
Figure 4.11: (a) Time as a function of the number of vertices in the polygon, plus 10 times the number of guards. (b) A zoom-in of the plot above.
and the prison yard problem, which is a combination of guarding the boundary from the inside and outside (see Figure 4.16(h)). Note, however, that while each of these examples were constructed for instances involving guarding only of the boundary, here we produce guards that guard the entire free space. However, it would not be hard to modify the software to support boundary (surface) guarding; indeed, the problem becomes easier than area guarding. Another variant that is well researched is the case of orthogonal polygons, which usually have different combinatorial art gallery theorems (see Figures 4.17(b) and (c)). Two more variants are rectangular rooms (Figure 4.17(e)) and spiral polygons, which are polygons composed of two chains – one with convex vertices and the other with reflex vertices (Figure 4.15(d) and 4.16(e)).

4.5.2 Covering Areas.

Maximizing the area seen by a set of $k$ guards has been studied (see, e.g.,[23]). The motivation is to guard a large portion of the polygon, while using a small and efficient guarding set. Since algorithm $A_3$ greedily finds the most uncovered area seen by the next guard, this algorithm may be expected to perform provably well. However, we did not explore this algorithm much as it was found to be extremely slow. Nevertheless, it would be interesting to evaluate the performance of
Let $l$ be the maximum lower bound obtained on the guard number of $P$; let $b > l$ be the upper bound obtained. We are interested to examine how the coverage area varies as the size of the guard set constructed by our algorithm varies from $l$ to $b$. We implemented this experiment, and the results with $A_1$ are illustrated in Figure 4.13.

Let $k$ be the size of an optimal guarding set of polygon $P$. Suppose we check how much area we cover with $k - 1$ guards or less. The results can be arbitrarily bad, as illustrated in Figure 4.14. The polygon in this figure consists of a big triangle from which a branch, $B$, of complicated small regions spikes from one of its edges. Using $A_1$, we first choose guards to cover $B$, and might only guard the region $P \setminus B$ with the very last guard. Now suppose our bounds are not tight; then, using the first $k - 1 \geq l$ guards gives us an arbitrarily bad area coverage, as $B$ gets arbitrarily small.

In our experiments, however, most of the time we covered at least 80% of the area of the polygon using $l$ guards. In the worst example, only 40% was covered. Interestingly, there is a steep increase with one of the guards that is added after the $l$th guard. This indicates that the situation tends to be similar to what we described in the example above, where one of the “late” added guards contributes significantly to the total coverage area. We also observe that in most of the cases where the area covered was relatively small with $l$ guards, adding a few more guards usually increased the area covered substantially. Thus, a good heuristic is to choose a number of guards that is not much more than $l$. According to our experiments, this method is likely to give better results.

### 4.5.3 Main Table of Results.

Let $A$ denote the set of all heuristics we use, and let $m$ be the number of input sets we tested. For each heuristic $A_i \in A$, let $G_{A_i}(j)$ be the number of guards obtained on some input number $j$. Let $G_{\text{min}}(j)$ be the minimum number of guards obtained with all heuristics on input $j$. For

<table>
<thead>
<tr>
<th>N</th>
<th>NE</th>
<th>NG</th>
<th>NGE</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>53</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>32</td>
<td>160</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>53</td>
<td>334</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>52</td>
<td>72</td>
<td>526</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>64</td>
<td>89</td>
<td>796</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>76</td>
<td>114</td>
<td>1081</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>88</td>
<td>125</td>
<td>1443</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>97</td>
<td>144</td>
<td>1731</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>106</td>
<td>159</td>
<td>2043</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>120</td>
<td>183</td>
<td>2539</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4.4: Results obtained with spike box polygons. See Table 4.3 for the abbreviations.
Figure 4.13: The area covered as a function of number of guards. The area in this graph corresponds to guarding sizes that range between the lower and the upper bounds we obtained. Each polygonal chain represents a different input. The area of the polygons was normalized (area 1 for each).

Figure 4.14: An example in which the guard that sees the most area is not chosen in the first place.
heuristic $A_i$, let $K_i = \frac{\sum_{1 \leq j \leq m} (G_{A_i}(j) - G_{\text{min}}(j))}{|A_i|}$, the average of number of guards minus the best guarding obtained with all heuristics, and let $M_i = \frac{\sum_{1 \leq j \leq m} ((G_{A_i}(j) - G_{\text{min}}(j))/G_{\text{min}}(j))}{|A_i|}$ represents the deviation from minimum in percentage. Let $Q_i$ be the number of times the guarding obtained with $A_i$ was the best among all heuristics (ties among the best guarding were counted for each heuristic). Let $B_i$ be the number of tests that were actually completed, not counting those that either exploded the memory or whose candidate scores became zero values (and thus caused the algorithm to select candidates arbitrarily, until all points of the polygon are seen, thereby taking an unusually long time to complete). We use these notations (without subscripts) in Table 4.5.

<table>
<thead>
<tr>
<th></th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
<th>$A_8$</th>
<th>$A_9$</th>
<th>$A_{10}$</th>
<th>$A_{11}$</th>
<th>$A_{12}$</th>
<th>$A_{13}$</th>
<th>$A_{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>0.7</td>
<td>0.47</td>
<td>1.47</td>
<td>1.6</td>
<td>1.3</td>
<td>1.22</td>
<td>0.9</td>
<td>0.83</td>
<td>1.75</td>
<td>0.48</td>
<td>0.5</td>
<td>1.64</td>
<td>3.33</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>0.10</td>
<td>0.06</td>
<td>0.22</td>
<td>0.22</td>
<td>0.16</td>
<td>0.29</td>
<td>0.13</td>
<td>0.14</td>
<td>0.41</td>
<td>0.08</td>
<td>0.09</td>
<td>0.27</td>
<td>0.69</td>
<td></td>
</tr>
<tr>
<td>$Q$</td>
<td>16</td>
<td>17</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>15</td>
<td>15</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>30</td>
<td>29</td>
<td>39</td>
<td>38</td>
<td>39</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.5: Results obtained with our heuristics on 40 input sets.

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_{11}$</th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>Minimum ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>X</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>1.09</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>X</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>1.125</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>1.85</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>X</td>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>13</td>
<td>11</td>
<td>6</td>
<td>11</td>
<td>1.09</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>16</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>1.25</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>16</td>
<td>13</td>
<td>7</td>
<td>12</td>
<td>1.15</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>14</td>
<td>9</td>
<td>5</td>
<td>9</td>
<td>1.55</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>17</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>1.3</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
<td>14</td>
<td>7</td>
<td>14</td>
<td>1.07</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>8</td>
<td>14</td>
<td>1.21</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>16</td>
<td>13</td>
<td>7</td>
<td>11</td>
<td>1.23</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>1.09</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>16</td>
<td>12</td>
<td>5</td>
<td>11</td>
<td>1.16</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>1.125</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>1.14</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>1.125</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>1.25</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.6: Different guarding and independent sets
4.6 Conclusions

We have conducted the first experimental analysis of a broad class of heuristics for locating guards to cover polygons. We designed and implemented several heuristics for guarding polygons, possibly with holes. We also computed visibility-independent sets, allowing us to obtain provable bounds on how close our results are to optimal. We concluded that there are three recommended heuristics, and they trade off guarding quality, space and time. The guarding sets obtained with them were very satisfactory in the sense that they were always either optimal or close to optimal (within factor 2 for all randomly generated instances) in all of the cases we encountered.

Most of our methods extend naturally to practical variants of the guarding and sensor coverage problem in which there are constraints on the visibility, e.g., view distance, good view angles of walls to be observed, robustness of coverage, etc [44, 59]. The methods also can be generalized to three dimensions, though the implementation would be substantially more involved.

There are several directions for further research. Since our results were very promising and the general art gallery problem is NP-hard, our main theoretical objective is to obtain provable heuristics.
Figure 4.15: Results of manually generated polygons, while using heuristic $A_1$. The hollow disks are the guards and gray rectangles are the independent points. The black regions inside some of the polygons represent holes.
Figure 4.16: Results of manually generated polygons, while using heuristic A_1. The hollow disks are the guards and the rectangles are the independent points. The black regions inside some of the polygons represent holes.
Figure 4.17: Results of manually generated polygons, while using heuristic $A_1$. The hollow disks are the guards and rectangles are the independent points. The black regions inside some of the polygons represent holes.
Figure 4.18: Results of different 100-vertex polygons (each column is dedicated to one input), with heuristics $A_1$ (first row), $A_2$ (middle row) and $A_{11}$ (third row). The hollow disks are the guards and rectangles are the independent points.
Figure 4.19: Results of different 100-vertex polygons (each column is dedicated to one input), with heuristics $A_1$ (first row), $A_2$ (middle row) and $A_{11}$ (third row). The hollow disks are the guards and rectangles are the independent points.
Chapter 5

Computing Multiple Watchman Routes

We present heuristics for computing multiple watchman routes. Given a polygon (with or without holes) and a parameter $k$, we compute a set of $k$ routes inside the polygon such that any point inside the polygon is visible from at least one point along one route. We measure the quality of our solutions by either the length of the longest route or the sum of the route lengths, where the goal is to minimize each. We start by computing a set of static guards, construct $k$ routes that visit all the static guards and try to shorten the routes while maintaining full coverage of the polygon. We implemented the algorithm and present extensive results to evaluate our methods, including a comparison with lower bound routes based on the idea of visiting large number of visibility-independent “witness points”. Our experiments showed that for a large suite of input data our heuristics give efficient routes that are comparable with the optimal solutions.

5.1 Introduction

As opposed to the classic Art Gallery problem (see Chapter 3) which deals with static guards, the watchman route variant is concerned with guards that can translate along routes. The goal is similar: make any point inside the polygon visible by at least one point along one of the routes. This problem is motivated by many applications that involve security, surveillance, imaging, simulations and more. In this problem, as the number of guards is usually predetermined, the measure of the result is often the route lengths (the Euclidian metric is usually used here, but other metrics, such as the number of links in the routes, have been used too). Interestingly, the minimum watchman route (one guard) inside a simple polygon has an exact polynomial time solution [83]. Unfortunately, extending the problem to support holes inside the polygon or allowing more than one guard (when minimizing the longest route) make the corresponding decision problems hard (the hardness proofs use simple reductions from the TSP [24] and partition [65] problems respectively).

We study the $k$-watchman routes, $k \geq 1$, where multiple watchmen are allowed to translate inside a polygon (possibly with holes). Two natural goals in this case are to minimize the maximum length of any route and to minimize the sum of the route lengths. We denote the corresponding problems by $KWRP_m$ and $KWRP_s$ respectively ($KWRP$ stands for $k$-watchman routes inside a polygon). We note that these problems are well motivated: one motivation behind $KWRP_m$ is to minimize the time it takes to cover the polygon while the motivation of $KWRP_s$ could be to save the total energy or frames taken by the entire system. We show later that it is even hard to give any meaningful
Our Contribution. We propose heuristics for computing watchman routes that cover polygons (with or without holes), for both \(KWRP_m\) and \(KWRP_s\). While it is impossible to develop exact or even approximate polynomial time algorithms for both problems unless \(P = NP\), we conduct an extensive experimental analysis of their performance. We show that our heuristics work well in practice for many kinds of polygons, and in some cases compare to lower bounds obtained by the idea of independent witness point set. As far as we know, our work is the first attempt to conduct a systematic experimentation with watchman route heuristics.

As we mentioned above, the 1-watchman route problem has been optimally solved with a polynomial time algorithm. However, to the best of our knowledge, neither implementation nor experiments have ever been reported. Perhaps the reason is that the algorithm is not easy to implement. Further, it is not clear if this algorithm will suffer from robustness problems. Hence, our work is also targeted for this specific and important variant, as when we set \(k = 1\) we compute the 1-watchman route.

Related Work. Good references for the various art gallery problems are [70, 82, 85]. Two detailed reports that provide valuable information about the watchman route problem and present algorithms for restricted versions are [67, 70]. The 1-watchman route problem has been extensively studied. After a few publications that were found to have flaws, Tan et al. [84] gave an \(O(n^4)\) time algorithm where the starting point is given and finally Tan [83] presented an \(O(n^5)\) algorithm for the general case, which followed [19]. Other interesting variants that have been studied are the minimum-link watchman route [7, 10] and Pursuit-Evasion [78] and others [20, 41, 57].

The rest of this chapter is organized as follows. In the next section we provide background information. In Section 5.3, we present our algorithm and discuss its implementation and performance. In Section 5.4 we present our experiments with the software we have implemented. We conclude and propose ideas for future research in Section 5.5.

5.2 Preliminaries

Given a polygon \(P\) (possibly with holes)\(^1\) and a point \(p \in P\), we denote by \(\mathcal{V}(p)\) (\(P\) is omitted for simplicity) the set of points inside \(P\) that are visible from \(p\).\(^2\) It is easy to observe that \(\mathcal{V}(p)\) is a star-shaped polygon and it is termed the visibility polygon of \(p\). A set of points \(S \subset P\) is said to cover \(P\) if \(\bigcup_{p \in S} \mathcal{V}(p) = P\). The classic art gallery problem is to find a smallest such set.

Over the years numerous variations of this problem have been proposed and studied. One of these variations allows guards to translate inside the polygon along predefined routes. In this case, the guards are often termed watchmen or mobile guards and their routes are termed watchman routes. Let \(w\) be a watchman with route \(R_w\) inside a polygon \(P\) (\(P\) is omitted for simplicity). Let \(\mathcal{V}(w) = \bigcup_{p \in R_w} \mathcal{V}(p)\) be defined similarly to the visibility polygons of the static guards. The goal here is to cover \(P\) as well, namely to find a set of watchmen \(S\) such that \(\bigcup_{w \in S} \mathcal{V}(w) = P\). In

\(^1\)From now on, by \(P\) we refer to any polygon

\(^2\)Two points are visible to each other if the line segment that connects them does not intersect any edge of the polygon.
In this context, the size of $S$ is usually given (we denote it by $k$) and the measure (or quality) of the solution involves the length of the routes. Two popular measures are the length of the longest route (corresponds to $KW_{RP_m}$) and the sum of route lengths (corresponds to $KW_{RP_s}$). More formally, the measure of $KW_{RP_m}$ and $KW_{RP_s}$ are $\max_{w \in S} L(w)$ and $\Sigma_{w \in S} L(w)$ where $L(w)$ is the length of route $R_w$. Table 5.1 summarizes the complexity of computing watchman routes for simple polygons. Table 5.2 summarizes results for constrained polygons that were established in [67]. In these tables MinSum and MinMax refer to the problems of minimizing the sum and minimizing the maximum, respectively. Both tables are borrowed from [67], while we updated the complexity of the MinSum problem in Table 5.1 for two or more guards. We note that when the polygon may have holes, both MinSum and MinMax problems become NP-hard for any number of watchmen.

<table>
<thead>
<tr>
<th>Optimization criterion</th>
<th>Number of watchman routes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MinSum</td>
<td>P</td>
</tr>
<tr>
<td>MinMax</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 5.1: Complexity of computing sets of watchman routes of various sizes inside a simple polygon.

<table>
<thead>
<tr>
<th>Optimization criterion</th>
<th>Polygon classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spiral</td>
</tr>
<tr>
<td>MinSum</td>
<td>P</td>
</tr>
<tr>
<td>MinMax</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 5.2: Complexity of computing sets of watchman routes of any size, for some classes of polygons.

Next we show that the related decision problems of $KW_{RP_m}$ and $KW_{RP_s}$ cannot have any $k$-approximation for any $k < n$ (unless $P = NP$).

**Theorem 5.1.** $KW_{RP_m}$ and $KW_{RP_s}$ can have no polynomial approximation algorithms unless $P = NP$.

**Proof.** Suppose there is such a polynomial algorithm $A$ (for either $KW_{RP_m}$ or $KW_{RP_s}$) with running time $O(\Gamma)$. We show how to solve the classic art gallery problem (denoted by $AG$) optimally in polynomial time using $A$, thus contradicting the above assumption, unless $P = NP$. It can be easily verified that any polygon $P$ can be guarded by $k$ static guards if and only if $A$ returns routes of zero length, given $k$ as a parameter. Given a polygon $P$ with $n$ vertices, the art gallery theorem states that $\lceil \frac{n}{3} \rceil$ guards are always sufficient to guard $P$. It follows that the solution to $AG$ can be found by searching the minimum $k$ for which the solution of $A$ contains only routes of zero length. It would require $O(\log(n)\Gamma)$ time. It follows that unless $P = NP$, such a polynomial approximation algorithm cannot exist. \hfill \square

Given two points $p_1, p_2 \in P$, we say that $p_1$ and $p_2$ are **independent** if there is no point $g \in P$ such that both $p_1 \in \mathcal{V}(g)$ and $p_2 \in \mathcal{V}(g)$. Let $S$ be a set of pairwise independent points in $P$ of size $s$.

---

3 In this table two kinds of polygons are defined as follows: (1) A polygon is an alp if it is monotone and one of the chains in the partition is a line segment parallel to the $x$-axis. (2) A polygon is a street if its boundary can be partitioned into two chains, each of which are guard sets for the polygon.
It follows that $m$ static guards are necessary (but may not be sufficient) to guard $P$. Hence, computing independent sets is a convenient tool to find lower bounds for the art gallery problem and used in [8] for that purpose. We use an analogous idea in our work to compute lower bounds. Recall that $k$ is the number of watchmen. By considering all partitions of $S$ into $k$ groups, and then computing the routes that cover all of the points in each group while optimizing the given problem (either $KWRP_m$ or $KWRP_s$), we find lower bounds. Although we did not design or implement any polynomial heuristic to carry out this task in this work, we use this idea to find lower bounds, and use them in our experimental evaluation.

### 5.3 Algorithm

The following is a high-level description of our heuristics.

<table>
<thead>
<tr>
<th>Compute Watchman Routes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A polygon $P$ (possibly with holes), $k$ (number of watchmen) and an indication whether to perform $KWRP_m$ or $KWRP_s$</td>
</tr>
<tr>
<td><strong>Output:</strong> A set of $k$ watchman routes inside $P$ that cover its interior</td>
</tr>
<tr>
<td><strong>Measure:</strong> The length of the longest route (for $KWRP_m$) or the sum of lengths of the routes (for $KWRP_s$)</td>
</tr>
</tbody>
</table>

(a) Compute a static guard set $S$ with heuristic $A_1$ of [8]. ($A_1$ is one of the three proposed heuristics. It was found efficient in time and produced good results.)

(b) Construct the visibility graph $\mathcal{U}$ of $S \cup V$, where $V$ is the set of vertices of $P$.

(c) Using $\mathcal{U}$, compute the pairwise shortest paths between any $s_1, s_2 \in S$ inside $P$ (denoted by $\mathcal{Z}$).

(d) Construct the minimum spanning tree of $S$ inside $P$ (denoted by $\mathcal{T}$) where the distance between any pair of points is computed from $\mathcal{Z}$.

(e) Split $\mathcal{T}$ into $k$ subtrees, $\mathcal{T}_1$-$\mathcal{T}_k$.

From step (f) and on, we work on each subtree independently.

(f) Construct a Hamiltonian route (let $\mathcal{R}$ denote an arbitrary one).

(g) Substitute vertices along $\mathcal{R}$ with others that shorten the length of $\mathcal{R}$ and maintain full coverage.

(h) Remove redundant vertices of $\mathcal{R}$ by connecting their adjacent vertices (we say that a vertex is redundant if when we connect its two adjacent vertices with their shortest path, the polygon remains fully covered).
We continue with a detailed description of the steps. We also describe the data structures that we use and analyze the complexity of the heuristics.

5.3.1 Computing a Static Guard Set (step a)

We start by computing a static guard set \( S \) [8]. The idea is that routes cover the polygon if they visit the static guards. The time to find the static guard set using heuristic \( A_1 \) is \( O(n^3) \) where \( n \) denotes the size of the input [8].

5.3.2 Constructing the Visibility Graph (step b)

We construct the visibility graph in \( O(n^2) \) time, or in \( O(n \log n + b) \) time where \( b \) is the number of arcs of the visibility graph [43].

5.3.3 Computing the Pairwise Shortest Paths (step c)

We use the Floyd-Warshall algorithm to compute all pairs of the shortest paths in \( O(n^3) \) time [28].

5.3.4 Constructing the Minimum Spanning Tree (step d)

Using Prim’s algorithm, we compute the minimum spanning tree in \( O(n^2) \) time [28].

5.3.5 Splitting the Minimum Spanning Tree into \( k \) subtrees (step e)

We split \( T \) into \( k \) subtrees, \( T_1 - T_k \). If the problem is \( KWRP_s \) we do it by simply removing the longest \( k - 1 \) edges of \( T \). The heuristic is more involved if the problem is \( KWRP_m \). By a reduction from the partition problem [65], the corresponding optimization problem is hard. We partition \( T \) by removing edges. The goal is to minimize the weight of the heaviest subtree. We remove edges by using ideas from parametric search (see below).

However, in some cases we get better results if we make a different kind of split which we call *split around a vertex*. Consider the polygon in Figure 5.1.\(^4\) The results of our heuristic (without the optimization we describe here) for \( KWRP_m \) are shown in Figure 5.1(a) and (b) for the minimum spanning tree and the output respectively. Figure 5.1(c) shows a different way to split \( T \). In this case it is split around the top vertex to induce the two subtrees in black and grey. Working on these trees separately will produce the two routes depicted in Figure 5.1(d). Consider the generalization of this example to any number of very long ‘legs’ of the same length (denoted by \( x \)) and \( k \) guards.

\(^4\)Note that some of the routes in this figure and others are not simple close chains. Some routes or parts of them sometime degenerate to polygonal chain or contain chains that connect two parts of the routes. The idea is that the watchman walks on them in both directions. It follows that some of the vertices along the routes may be of degree \( d > 2 \). In order to remove any possible ambiguity, we use arrows (here in Figure 5.1(f)) to clarify. In Figure 5.1(b) one of the watchman route degenerates to a single point (the watchman essentially becomes static). We represent it with a disc around the guard and do the same in other figures as well.
KW RP$_m$ produces the following results. For any $k > 1$, it removes $k - 1$ legs and remains with a tree that results in a route of length roughly $(m - k + 1)x$ where $m$ is the number of legs. On the other hand, an optimal solution is obtained by splitting around $v$ to subtrees each of which with $O([m/k])$ legs, resulting in a maximum route of length $[m/k]x$. It follows that on this particular type of polygons, our heuristic converges to an approximation of factor $\Theta(k)$ to the optimum.

In order to decrease this ratio, and remain simple at the same time, we use the following method which modifies $T$. In case there is a vertex $v \in T$ of high degree whose neighbors have low degrees, we remove most of the adjacent arcs of $v$ and reconstruct $T$ while not choosing any edge that connects $v$ again. Our rule of thumb is to do so on a vertex if its degree is at least $\max(k/2, 5)$ larger than at least $d/2$ of its adjacent vertices where $d$ is the degree of $v$. In this case we remove all adjacent arcs but $\lfloor \frac{2d}{\max(k/2, 5)} \rfloor$. This method improves the results in the above example significantly, giving the tree and routes depicted in Figure 5.1(e) and (f) respectively. Of course, this method can usually be improved by tuning (automatically or manually) our rule of thumb.

Figure 5.1(g) shows another interesting example where an optimum is achieved by adding a Steiner vertex (the disc in the figure) and splitting around it.

We note that generalizing our heuristics by adding Steiner vertices and splitting around vertices in more complicated cases (while continuing to support the ideas that we use, namely removing edges), seems like a very challenging task which can be simply observed as a hard problem. It is still not clear to us how to develop efficient heuristics that will work together with our other ideas, and it is not clear how useful and time costly they will be. We hope to investigate this idea further in the future.

**Implementation Details.** If the problem is KW RP$_s$, we need to remove the longest $k - 1$ edges. Finding them takes $O(n \log n)$ time by sorting. If the problem is KW RP$_m$, we use parametric search for finding the subtrees. We perform a binary search on the values 0 to the weight of the tree (denoted by $W$; Note that $W = \Omega(2^n)$) and stop when the interval on which we search becomes very small (smaller than a predefined constant). Thus, we perform $O(\log W)$ iterations. For each iteration, we do the following. Suppose the current weight we test is $W'$. We visit the tree in a bottom-up fashion and remove edges once the tree below them has weight larger than $W'$. It is optimal for the current iteration because if the edge that we remove is below, then its removal creates a smaller subtree and is clearly wasteful. Each iteration traverses the tree in $O(n)$ time. Together, this step takes $O(n \log W) = O(n^2)$ time.

**5.3.6 Constructing Hamiltonian Routes (step f)**

We use the ideas from the algorithm of Christofides in order to approximate the optimal route in $O(n^{2.5} \log^4 n)$ time [25].

**5.3.7 Substituting Vertices (step g)**

We try to substitute some of the vertices along each route $R$ with others that shorten its length. The vertices of $R$ can be partitioned into two groups: the first (denoted by $R_g$) contains vertices that belong to the static guard set and the second (denoted $R_a$) contains the rest, namely the vertices of $P$ along the paths that connect two vertices of $R_g$. The idea here is to replace vertices
of \( \mathcal{R}_g \) by others in order to shorten the length of \( \mathcal{R} \). For any \( p_1, p_2 \in P \), let \( \alpha(p_1, p_2) \) be the shortest path from \( p_1 \) to \( p_2 \) inside \( P \). Let \( v \in \mathcal{R}_g \) and let \( u, w \in \mathcal{R}_g \) be the two vertices before and after \( v \) along \( \mathcal{R} \) (excluding the vertices of \( R_g \)). Note that if \( |R_g| = 2 \) then \( u = w \). Let \( z \in P \) be some point. The replacement of \( v \) by \( z \) (denoted by \( R(v, z) \)) is defined as modifying \( \mathcal{R} \) by replacing \( \alpha(u, v) \) and \( \alpha(v, w) \) from \( \mathcal{R} \) and inserting \( \alpha(u, z) \) and \( \alpha(z, v) \) instead. Note that \( \mathcal{R} \) remains closed after performing this operation.

Let \( H \) be the set of reflex vertices of \( P \). We extend the edges that are adjacent to the vertices of \( H \) into the interior of \( P \), until they hit the boundary of \( P \) (denoted by \( @P \)). See an example in Figure 5.2(g). We denote these extensions by \( Q \). Let \( \mathcal{G} \) be the arrangement of \((@P) \cap Q \). We call \( \mathcal{G} \) the extension arrangement of \( P \). For each vertex \( v \in \mathcal{R} \), let \( f(v) \) be the face of \( \mathcal{G} \) that contains it (if \( v \) is a vertex of \( \mathcal{G} \), \( f(v) \) will be the set of adjacent faces). Based on the properties of the cells in extension arrangements, any vertex \( w \in @f(v) \) (\( @f(v) \) is the boundary of \( f(v) \)) has a good chance to maintain full coverage while replacing \( v \) by performing \( R(v, w) \). In case we find vertices on \( @f(v) \) that both maintain full coverage when they replace \( v \) and shorten \( \mathcal{R} \), we modify \( \mathcal{R} \) by replacing \( v \) with the one that minimizes the length of \( \mathcal{R} \). We then replace \( v \) by the new vertex in \( \mathcal{R}_g \). We iterate this process and work on more cells of \( \mathcal{G} \), as long as any improvement is achieved.

**Implementation Details.** We maintain an arrangement \( \mathcal{B} \) for a union of visibility polygons and \( P \). Note that in such an arrangement we can easily mark faces as covered or not by the visibility polygons. We initialize \( \mathcal{B} \) with the set of visibility polygons of the vertices of \( \mathcal{R}_g \). Then for each \( v \in \mathcal{R}_g \) we find the face \( F(v) \) that contains it in the extension arrangement \( \mathcal{G} \) (using point location). We then test whether we can replace \( v \) by any vertex of \( v' \in F(v) \) in the following way. We remove \( \mathcal{V}(v) \) from \( \mathcal{B} \) and insert \( \mathcal{V}(v') \), while maintaining the information whether a face is covered by the visibility polygons of \( v' \) or any other vertex of \( \mathcal{R}_g - \{v\} \). We then check whether \( P \) is fully covered. If so and the corresponding route is shorter, we perform a replacement and update \( \mathcal{B} \) accordingly. We iterate this process until no improvement is detected. Since there are \( O(n) \) vertices on the routes, together they are tested for replacement with \( O(n^2) \) vertices of \( \mathcal{B} \). Since each test takes \( O(n^2) \) time, the total time is \( O(n^4) \).

### 5.3.8 Removing Vertices (step h)

Let \( u, v, w \in \mathcal{R}_g \) be defined as in step g above. The removal of \( v \) is defined as removing \( \alpha(u, v) \) and \( \alpha(v, w) \) from \( \mathcal{R} \), and inserting \( \alpha(u, w) \) instead. The idea of this step is to perform removal of vertices if this operation maintains full coverage (note that it necessarily shortens \( \mathcal{R} \)). Let \( T \) be the vertices obtained when intersecting \( \alpha(u, w) \) and \( \mathcal{G} \). We check whether \( \bigcup_{p \in T} \mathcal{V}(p) \) contains \( \mathcal{V}(v) \). If so, we perform the removal of \( v \) and maintain full coverage. We iterate this process until no vertices can be removed.

**Implementation Details.** The routes contain \( O(n) \) vertices. Each vertex can be removed at most once and each removal check requires a test with \( O(n) \) visibility polygons for coverage. To carry out this test, we use an arrangement data structure similarly to step g, and use similar ideas. The induced arrangement for each vertex is thus of complexity \( O(n^2) \). Hence, the total time is \( O(n^3) \).
Figure 5.2 illustrates the execution of all the steps of heuristic $KWRP_m$.\(^5\)

### 5.3.9 Total Complexity

Combining all steps, we get that the time of our heuristic is $O(n^4)$. The space requirement is dominated by the resources required to maintain the various arrangements, which results in an $O(n^2)$ space.

We note that by analyzing the performance of our experiments, we observed that the asymptotic time was always much smaller (bounded by $O(n^3)$ or less).

### 5.4 Experiments

We have implemented our heuristics on a PC with Microsoft Visual C++ .NET (version 7.1). We used the libraries openGL and CGAL [21]. The tests were performed on a Microsoft Windows XP workstation on an Intel Pentium 4 3.2 GHz CPU with 2GB of RAM. We have performed extensive experiments with our heuristics. In this section we report our results and conclusions from our experiments. Our tests include user-generated polygons and polygons generated by a random polygon generator [11].

Figures 5.3, 5.4 and 5.6 show the routes obtained by our software for different kinds of polygons with different values of $k$. Figure 5.5 compares the results of different number of watchmen on two polygons.

Since the $k$-watchman route problem is hard even to approximate, it is frequently difficult to evaluate the results by comparing it to optimal solutions or even solutions that approximate the optimum. Instead, we use the idea of independent witnesses (see Section 5.2). However, both finding the maximum independent set and finding the shortest routes that visit a set of points (for $k > 1$) are hard (the second by a simple reduction from the partition problem). Given a set of independent points, we need to find an optimal solution for routes that see all the independent points (by either minimizing the longest route or minimizing the sum or route length). We made this task easier by constructing polygons manually such that the partition of independent points to watchmen was evident. Of course, this restricts the shape of the available polygons. We note that in many cases the lower bound measure was smaller than the optimum. Figure 5.7 depicts two instances, each with lower and upper bounds.

We note that our heuristics were always within a factor of 5 from the lower bound. Most were even within a factor of 2 and many were within a factor of 1.5. We emphasis that in many cases the lower bound routes did not cover the polygons, thus were not tight.

Even in cases where we could not find lower bounds easily, we were usually satisfied with the results obtained with our software on many types of polygons. In many cases our results looked optimal or very close to be optimal.

---

5Static guards from step (a) are marked as discs. Note that some of the routes are not simple close chains. Some routes or parts of them sometime degenerate to polygonal chain or contain chains that connect two parts of the routes. The idea is that the watchman walks on them in both directions. It follows that some of the vertices along the routes may be of degree $d > 2$. In order to remove any possible ambiguity, we use arrows to clarify. In some cases some of the watchman routes degenerate to points (the watchmen essentially become static). We represent them with a disc around the points. Note that the above elements appear in most of the figures of this report.
Figure 5.1: Improvement for step (e). The circle around the guard in (b) indicates that it is a stationary mobile guard.
Figure 5.2: Illustration of all the steps of $KWRP_m$. For clarity, the data in subfigures (b) and (c) are partially presented. In Figure (g) we drew in thin segments some of the extension edges that affected the output (see Section 5.3). In Figure (h), the solid and the dashed routes represent the situation before and after performing step (h), respectively (so the final route consists of the top segment).
There are many parameters that affect the time taken to run our software. The main ones that can be quantified are the size of the polygons, the number of watchmen, the number of static guards and the size of the extension arrangement $G$. As for the latter, in polygons with wide areas (such as the polygon in Figure 5.2 (a)-(g)) $G$ is large because many extension edges intersect. On the other hand, polygons with narrow passages (as the common randomly-generated polygon) result in smaller sizes of $G$. We note that our experiments showed that the selection of $k$ (number of mobile watchmen) did not have any significant effect on the time: running times were always very close, and moreover, they had no correlation with $k$. Thus, we ignore this parameter here. In our results the time refers to an average of four runs, with $k = \{1, 2, 3, 4\}$. In Figures 5.8, 5.9 and 5.10 we plot the time as a function of several parameters. In each figure there are two graphs, for $KWRP_m$ and $KWRP_s$. Figure 5.8 shows the results of many kinds of polygons. We devote separate graphs for random polygons in Figure 5.9 (Figure 5.4(a) is a snapshot of a random polygon) and spike box polygons in Figure 5.10 (Figure 5.3(b) is a snapshot of a spike box polygon). These two classes of polygons are different in the following aspects. First, spike box polygons usually results in large $G$ while random polygons usually results in small $G$. Second, the size of independent sets in spike box polygons is usually much smaller than this size in random polygons. The graphs show that there is a correlation between the time and both the polygon’s size and the number of static guards (although there are exceptions). On the other hand, we could not find any correlation between the size of $G$ and the time. Note that spikes in one graph are usually a result of stronger dependency on another parameter. Finally, we note that the times for $KWRP_m$ and $KWRP_s$ were usually very similar and there is no evidence for either one to be faster than the other in any set of polygons.

5.5 Conclusions and future work

We presented heuristics for constructing $k$-watchman routes inside polygons, possibly with holes. As far as we know, this is the first attempt to develop heuristics for this problem. We implemented these heuristics and conducted experiments. We tested our software with many polygons and presented our results in detail. In limited cases we were even able to evaluate our results by comparing with lower bounds, and obtained a bound of factor-5 approximation. Moreover, many other results in many kinds of polygons look efficient and not far from optimal. We are currently investigating possible directions to improve our heuristics. Next we briefly summarize some. We note that these ideas seem to require solutions that are NP-hard, very challenging, do not have clear heuristics, and seem to take much processing time.

- Possibly start with a different set of static guards that cover the polygon (even if it is not optimal for the static problem). The idea is that if the radius of this set is small, it may lead to shorter routes.
- Locally change the minimum spanning tree such that the Hamiltonian routes become shorter.
- Try also different kinds of splits to the minimum spanning tree. We observed that multiple subtrees that share a vertex can improve the results in specific cases. Sometimes combining this idea with adding Steiner vertices can improve the results further.

Finally, we propose the following directions for future research.

- Find ways to improve the time bounds of the different steps of our heuristics.
• Explore ideas for practical implementations of lower bounds.
• Develop heuristics for other kinds of watchman route problems.
• Find a way to prove efficient lower bounds.
• Develop approximation algorithms for restricted versions.
Figure 5.3: Experiment snapshots obtained with our software on different kinds of polygons. Subfigures (a) and (b) show results for $k = 1$ and Subfigures (c)-(h) shows results of $KW RP_m$
Figure 5.4: Experiment snapshots obtained with our software on different kinds of polygons. Subfigures (a)-(d) shows results of $KWRP_m$ while subfigures (e)-(g) shows results of $KWRP_s$. 
Figure 5.5: Results for different number of watchmen.
Figure 5.6: Some results obtained with $KWRP_s$ (Compare with the results of $KWRP_m$ on the same polygons in Figures 5.3 and 5.4).
Figure 5.7: Comparing the results with lower bounds. Subfigures (a) and (c) depict upper bounds obtained with our software, while Subfigures (b) and (d) depicts corresponding lower bounds where the squares represent independent witness points.
Figure 5.8: Time as a function of different parameters for different polygons
Figure 5.9: Time as a function of different parameters for randomly generated polygons
Figure 5.10: Time as a function of different parameters for spike box polygons
Bibliography


