I/O-Efficient Construction of Constrained Delaunay Triangulations

Pankaj K. Agarwal[1][*]   Lars Arge[2,1][†]   Ke Yi[1][‡]

[1]Department of Computer Science, Duke University, Durham, NC 27708, USA.
{pankaj,large,yike}@cs.duke.edu
[2]Department of Computer Science, University of Aarhus, Aarhus, Denmark.
large@daimi.au.dk

Abstract

In this paper, we designed and implemented an I/O-efficient algorithm for constructing constrained Delaunay triangulations. If the number of constraining segments is smaller than the memory size, our algorithm runs in expected $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os for triangulating N points in the plane, where M is the memory size and B is the disk block size. If there are more constraining segments, the theoretical bound does not hold, but in practice the performance of our algorithm degrades gracefully. Through an extensive set of experiments with both synthetic and real data, we show that our algorithm is significantly faster than existing implementations, and is also insensitive to various data features such as point distributions and segment lengths.

1   Introduction

With the emergence of new terrain mapping technologies such as Laser altimetry (LIDAR), ground based laser scanning and Real Time Kinematic GPS (RTK-GPS), one can acquire millions of georeferenced points within minutes to hours. Converting this data into a digital elevation model (DEM) of the underlying terrain in an efficient manner is a challenging important problem. The so-called triangulated irregular network (TIN) is a widely used DEM, in which a terrain is represented as a triangulated xy-monotone surface. One of the popular methods to generate a TIN from elevation data—a cloud of points in $\mathbb{R}^3$—is to project the points onto the xy-plane, compute the Delaunay triangulation of the projected points, and then lift the Delaunay triangulation back to $\mathbb{R}^3$. However, in addition to the elevation data one often also has data representing various linear features on the terrain, such as rivers and road networks, in which case one would like to construct a TIN that is consistent with this data, that is, where the linear features appear along the edges of

the TIN. In such cases it is desirable to compute the so-called *constrained Delaunay Triangulation (CDT)* of the projected point set with respect to the projection of the linear features. Roughly speaking, the constrained Delaunay triangulation of a point set P and a segment set S is the triangulation that is as close to the Delaunay triangulation of P under the constraint that all segments of S appear as edges of the triangulation.

The datasets being generated by new mapping technologies are too large to fit in internal memory and are stored in secondary memory such as disks. Traditional algorithms, which optimize the CPU efficiency under the RAM model of computation, do not scale well with such large amounts of data. This has led to growing interest in designing I/O-efficient algorithms that optimize the data transfer between disk and internal memory. In this paper we study I/O-efficient algorithms for planar constrained Delaunay triangulations.

Problem statement. Let P be a set of N points in $\mathbb{R}^2$, and let S be a set of K line segments with pairwise-disjoint interiors whose endpoints are points in P. The points $p, q \in \mathbb{R}^2$ are *visible* if the interior of the segment pq does not intersect any segment of S. The *constrained Delaunay triangulation* $\text{CDT}(P, S)$ is the triangulation of S that consists of all segments of S, as well as all edges connecting pairs of points $p, q \in P$ that are visible and that lie on the boundary of an open disk containing only points of P that are not visible from both p and q. $\text{CDT}(P, \emptyset)$ is the Delaunay triangulation of the point set P. Refer to Figure 1. For clarity, we use *segments* to refer to the "obstacles" in S, and the term "edges" for all the edges in the triangulation $\text{CDT}(P, S)$.

We work in the standard external memory model [2]. In this model, the main memory holds M elements and each disk access (or I/O) transmits a block of B elements between main memory and continuous locations on disk. The complexity of an algorithm is measured in the total number of I/Os performed, while the internal computation cost is ignored.

Related results. Delaunay triangulation is one of the most widely studied problems in computational geometry; see [5] for a comprehensive survey. Several worst-case efficient $O(N \log N)$ algorithms are known in the RAM model, which are based on different standard paradigms, such as divide-and-conquer [12] and sweep-line. [13]. A randomized incremental algorithm with $O(N \log N)$ expected running time was proposed in [15]. This algorithm has received much attention in both theory and practice because of its simplicity. However, since it constructs the triangulation by incrementally inserting points in a random order, thus accessing intermediate triangulations in a non-local manner, it is very inefficient in modern memory hierarchies; Amenta et al. [3] gave techniques to improve its practical efficiency while preserving its optimal theoretical bound. By now
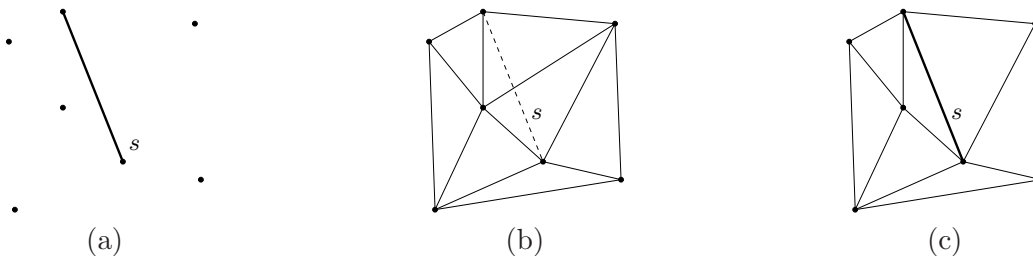

(a)   (b)   (c)

Figure 1: (a) The point set P of 7 points and segment set S of 1 segment s. (b) $\text{DT}(P) = \text{CDT}(P, \emptyset)$. (c) $\text{CDT}(P, S)$.

efficient implementations of many of the developed algorithms are also available. For example, the widely used software package `triangle`, developed by Shewchuk [22], has implementations of all three algorithms mentioned above. Both CGAL [7] and LEDA [19] software libraries also offer Delaunay triangulation implementations.

By modifying some of the algorithms for Delaunay triangulations, $O(N \log N)$ time RAM-model algorithms have been developed for constrained Delaunay triangulations [8, 21]. However, these algorithms are rather complicated and do not perform well in practice. A common practical approach for computing $\text{CDT}(P, S)$, e.g. used by `triangle` [22], is to first compute $\text{DT}(P)$ and then add the segments of S one by one and update the triangulation. A segment s is inserted by first removing all the triangles it intersects, and then retriangulating the two resulting polygons on either side of s. It is believed that this incremental approach works well when there are not too many segments and the segments are short. However, so far there has been no formal analysis for this incremental algorithm. In fact, it is an open question whether a randomized incremental algorithm with $O(N \log N)$ expected running time can be developed for computing the constrained Delaunay triangulation.

In the I/O-model, Goodrich et al. [14] gave an optimal $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O Delaunay triangulation algorithm, but it is too complicated to implement. Crauser et al. [11] extended the random incremental construction framework of Clarkson and Shor [10], obtaining an I/O-efficient Delaunay triangulation algorithm that runs in expected $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. A simplified version of this algorithm is later implemented by Kumar and Ramos [18]. To our knowledge, there has been no theoretical or practical study on I/O-efficient construction of constrained Delaunay triangulations.

Our results. By modifying the algorithm of Crauser et al. [11] we develop the first I/O-efficient constrained Delaunay triangulation algorithm. It uses $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os expected, provided that $|S| \leq c_0 M$, where c_0 is a constant. Although our algorithm falls short of the desired goal of having an algorithm that performs $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os irrespective of the size of S, it is useful for many practical situations. We demonstrate the efficiency and scalability of our algorithm through an extensive experimental study with both synthetic and real-life data. Compared with existing constrained Delaunay triangulation packages, our algorithm is significantly faster on large datasets, and is also insensitive to various data features, such as point distribution, segment lengths, etc. For example it can process 10GB of real-life LIDAR data using only 128MB of main memory in roughly 7.5 hours! As far as we know, this is the first implementation of constrained Delaunay triangulation algorithm that is able to process such a large dataset. Moreover, even when S is larger than the size of main memory, our algorithm does not fail, but its performance degrades quite gracefully.

An open question is whether there exists a randomized incremental algorithm for constructing $\text{CDT}(P, S)$ in $O(N \log N)$ expected time. An important step in the analysis of such an algorithm is bounding the expected number of structural changes during the incremental construction. In Appendix A we give an $\Omega(N \log K)$ lower bound on this quantity, where K is the number of segments. This is to be contrast with the well known $\Theta(N)$ bound for Delaunay triangulations. We also give an $O(N \log^2 K)$ upper bound. Closing this gap remains an open problem.

2 I/O-Efficient Algorithm

Let P be a set of N points in $\mathbb{R}^2$, and let S be a set of K segments with pairwise-disjoint interiors whose endpoints lie in P. Let E be the set of endpoints of segments in S. We assume that the points of P are in general position. For simplicity of presentation, we include a point p_∞ at infinity

in P. We also add p_∞ to E. Below we describe an algorithm for constructing $\text{CDT}(P, S)$ that follows the framework of Crauser et al. [11] for constructing Delaunay triangulations. However, we first introduce the notion of extended Voronoi diagrams, originally proposed by Seidel [21], and define conflict lists and kernels.

Extended Voronoi diagrams.   We extend the plane to a more complicated surface as described by Seidel [21]. Imagine the plane as a sheet of paper Σ with the points of P and the segments of S drawn on it. Along each segment $s \in S$ we "glue" an additional sheet of paper Σ_s, which is also a two-dimensional plane, onto Σ; the sheets are glued only at s. These $K + 1$ sheets together form a surface Σ_S. We call Σ the *primary* sheet, and the other sheets *secondary* sheets. P "lives" only on the primary sheet Σ, and a segment $s \in S$ "lives" in the primary sheet Σ and the secondary sheet Σ_s. For a secondary sheet Σ_s, we define its *outer region* to be the set of points that do not lie in the strip bounded by the two lines normal to s and passing through the endpoints of s.

Assume the following connectivity on Σ_S: When "traveling" in Σ_S, whenever we cross a segment $s \in S$ we must switch sheet, i.e., when traveling in a secondary sheet Σ_s and reaching the segment s we must switch to the primary sheet Σ, and vice versa. We can define a visibility relation using this switching rule. Roughly speaking, two points $x, y \in \Sigma_S$ are *visible* if we can draw a line segment from x to y on Σ_S following the above switching rule. More precisely, x and y are visible if: $x, y \in \Sigma$ and the segment xy does not intersect any segment of S; $x, y \in \Sigma_s$ and the segment xy does not intersect s; $x \in \Sigma, y \in \Sigma_s$ and the segment xy crosses s but no other segment; or $x \in \Sigma_s, y \in \Sigma_t$, and the segment xy crosses s and t but no other segment. For $x, y \in \Sigma_S$, we define the distance $d(x, y)$ between x and y to be the length of the segment connecting them if they are visible, and $d(x, y) = \infty$ otherwise.

For $p, q, r \in \Sigma_S$, if there is a point $y \in \Sigma_S$ so that $d(p, y) = d(q, y) = d(r, y)$, then we define the *circumcircle* $C(p, q, r; S) = \{x \in \Sigma_S \mid d(x, y) = d(p, y)\}$. Otherwise $C(p, q, r; S)$ is undefined. Note that portions of $C(p, q, r; S)$ may lie on different sheets of Σ_S. We define $D(p, q, r; S)$ to be the open disk bounded by $C(p, q, r; S)$, i.e., $D(p, q, r; S) = \{x \in \Sigma_S \mid d(x, y) < d(p, y)\}$. Refer to Figure 2(a). Using this circumcircle definition, the constrained Delaunay triangulation can be defined in the same way as standard Delaunay triangulations, i.e., $\text{CDT}(P, S)$ consists of all
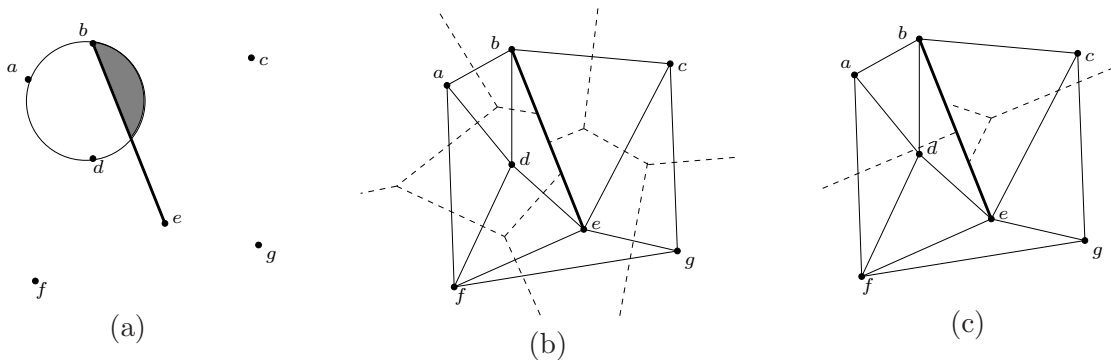

Figure 2: (a) For the point set of Figure 1(a), a portion of $D(a, b, d; S)$ lies in the primary sheet (unshaded), the other portion lies in the secondary sheet Σ_{be} (shaded). (b) $\text{CDT}(P, S)$ (solid lines) and the portion of $\text{EVD}(P, S)$ (dashed lines) that lies in the primary sheet. (c) The portion of $\text{EVD}(P, S)$ (dashed lines) that lies in the secondary sheet Σ_{be}.

4

triangles $\triangle uvw, u, v, w \in P$, whose circumcircles do not enclose any point of P [21]. We define the *extended Voronoi region* of a point $p \in P$ as $\mathrm{EV}(p; P, S) = \{x \in \Sigma_S \mid d(x, p) \leq d(x, q), \forall q \in P\}$, and the *extended Voronoi diagram* of P (with respect to S) as $\mathrm{EVD}(P, S) = \{\mathrm{EV}(p; P, S) \mid p \in P\}$. Seidel [21] showed that $\mathrm{CDT}(P, S)$ is the dual of $\mathrm{EVD}(P, S)$, in the sense that an edge pq appears in $\mathrm{CDT}(P, S)$ if and only if $\mathrm{EV}(p; P, S)$ and $\mathrm{EV}(q; P, S)$ share an edge. Refer to Figure 2(b) and 2(c). This duality relation will be useful in extending the algorithm by Crauser et al. [11] to computing $\mathrm{CDT}(P, S)$.

Conflict lists and kernels. Let $R \subseteq P$ be a subset of points such that $E \subseteq R$. Let $e = pq$ be an edge of $\mathrm{CDT}(R, S)$, and let $\triangle pqu$ and $\triangle pqv$ be the two triangles adjacent to e. (Since $p_\infty \in R$, each edge is adjacent to two triangles.) We define the *conflict list* [10] of e, denoted by $P_{|e} \subseteq P$, as the set of points of P that lie in $D(p, q, u; S) \cup D(p, q, v; S)$, plus p, q, u and v. This definition implies that for any $p' \in P \setminus \{p, q, u, v\}$, $p' \in P_{|e}$ if and only if at least one of $\triangle pqu$ and $\triangle pqv$ does not appear in $\mathrm{CDT}(R \cup \{p'\}, S)$.

One basic step in our algorithm will be to compute a triangulation of each $P_{|e}$ and then merge the results together to form $\mathrm{CDT}(P, S)$. Let $I_e = \{e\}$ if $e \in S$, and $\emptyset$ otherwise. Then the triangulation we will compute for $P_{|e}$ is $\mathrm{CDT}(P_{|e}, I_e)$. In order to identify the triangles of $\mathrm{CDT}(P_{|e}, I_e)$ that appear in $\mathrm{CDT}(P, S)$, we define the notion of the *kernel* of e (with respect to R and S), denoted by $\tau(e)$, which is contained in $\mathrm{EV}(p; R, S) \cup \mathrm{EV}(q; R, S)$. A point $x \in \mathrm{EV}(p; R, S)$ (resp. $x \in \mathrm{EV}(q; R, S)$) lies in $\tau(e)$ if the ray $\overrightarrow{px}$ (resp. $\overrightarrow{qx}$) leaves $\mathrm{EV}(p; R, S)$ through the common edge of $\mathrm{EV}(p; R, S)$ and $\mathrm{EV}(q; R, S)$. Figure 3 gives some examples, including some special cases such as that one of the endpoints of e is p_∞, and that e itself is a segment of S. Note that $\tau(e)$ depends and only depends on e's two adjacent triangles in $\mathrm{CDT}(R, S)$, since it just consists of two triangles formed by p, q, and the common edge of $\mathrm{EV}(p; R, S)$ and $\mathrm{EV}(q; R, S)$, whose endpoints are the circumcenters of e's adjacent triangles by the duality of $\mathrm{CDT}(R, S)$ and $\mathrm{EVD}(R, S)$.
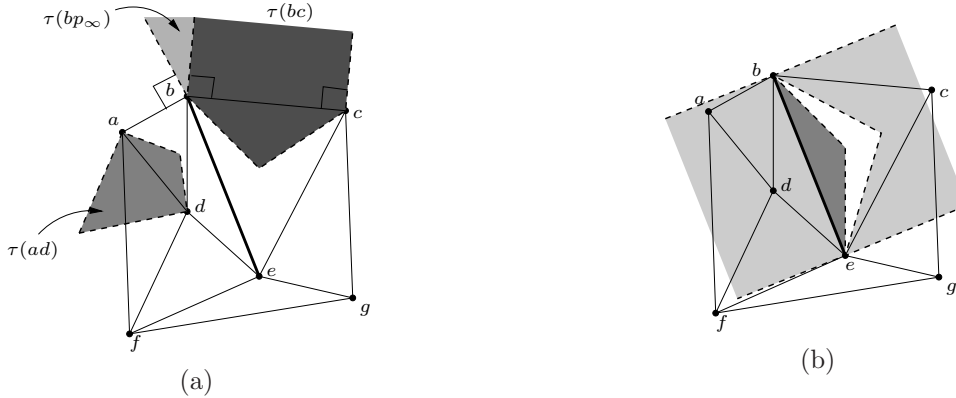

Figure 3: (a) The kernels of edges ad, bc, and bp_∞. (b) The kernel of the edge be; the darker part lies in the primary sheet, and the lighter part lies in the secondary sheet Σ_{be}.

Lemma 1 *Let $E \subseteq R \subseteq P$. The conflict lists and kernels of the edges in $\mathrm{CDT}(R, S)$ have the following properties:*

   (i) The interiors of $\tau(e), e \in \mathrm{CDT}(R, S)$ are pairwise disjoint.

   (ii) $\{\tau(e) \mid e \in \mathrm{CDT}(R, S)\}$ covers the points of Σ_S that do not lie in an outer region.

5

(iii) For any edge $e \in \mathrm{CDT}(R, S)$ and for any $u, v, w \in P_{|e}$ such that $C(u, v, w; S)$ is defined with ξ being the center, if $\xi \in \tau(e)$ and $D(u, v, w; I_e) \cap P_{|e} = \emptyset$, then $D(u, v, w; S) \cap P = \emptyset$.

(iv) For any $\triangle uvw \in \mathrm{CDT}(P, S)$ with circumcenter ξ, there exists an edge $e \in \mathrm{CDT}(R, S)$ such that $\xi \in \tau(e)$, $u, v, w \in P_{|e}$, and $D(u, v, w; I_e) \cap P_{|e} = \emptyset$.

Proof:

(i) Since any point Σ_S belongs to only one Voronoi region, it belongs to at most kernel.

(ii) For any $x \in \Sigma_S$, let $x \in \mathrm{EV}(p; R, S)$ for some $p \in R$. If x does not lie in an outer region, then the ray $\overrightarrow{px}$ will eventually enter the extended Voronoi region of some $q \neq p$, since $p_\infty \in R$. By the duality of $\mathrm{CDT}(R, S)$ and $\mathrm{EVD}(R, S)$, pq is an edge of $\mathrm{CDT}(R, S)$ and $x \in \tau(pq)$.

(iii) We first show that $D(u, v, w; S)$ does not contain any point of $P_{|e}$. We know that $D(u, v, w; I_e)$, the circum-disk defined on the surface Σ_{I_e} consisting of the primary sheet and just one secondary Σ_e if $e \in S$, does not contain any point of $P_{|e}$. Since $D(u, v, w; S) \cap \Sigma$, the portion of $D(u, v, w; S)$ lying in the primary sheet, is contained in $D(u, v, w; I_e) \cap \Sigma$, and all points of P are on the primary sheet Σ, $D(u, v, w; S)$ does not contain any point of $P_{|e}$, either.

Next we show that $D(u, v, w; S)$ does not contain any point of $P \setminus P_{|e}$. Suppose otherwise it contains such a point p, then p is closer to ξ than any point in $P_{|e}$, in particular the endpoints of e. This means that if we add p to R, the shape of $\tau(e)$ will be changed since ξ will not belong to the Voronoi region of either endpoint of e. Since $\tau(e)$ depends only on the two adjacent triangles of e, this also implies that the insertion of p will destroy at least one of these two triangles. Thus p is in conflict with e, which contradicts with the earlier assumption that $p \in P \setminus P_{|e}$.

(iv) First, from Property (ii), we know that there must exist an edge $e \in \mathrm{CDT}(R, S)$ such that $\xi \in \tau(e)$ because ξ cannot lie in any outer region. Second, all of u, v and w must be in $P_{|e}$, since they are either the endpoints of e, or they are closer to ξ than the endpoints of e, in which case the insertion of any of them will change $\tau(e)$. By the same reason as above, we have $u, v, w \in P_{|e}$.

Now we show that $D(u, v, w; I_{|e})$ does not contain any point of $P_{|e}$. Consider the surface Σ_{I_e}. $D(u, v, w; I_e)$ must be fully contained in the union of the two circum-disks (also defined on Σ_{I_e}) of e's adjacent triangles, because otherwise we could add a point p' to P that is closer to ξ than the endpoints of e, and is not in conflict with e, which are contradictory. Since $\triangle uvw$ is a valid triangle of $\mathrm{CDT}(P, S)$, $D(u, v, w; S)$ does not contain any point of $P_{|e}$. Let p be a point of P that lies in $D(u, v, w; I_{|e}) \setminus D(u, v, w; S)$, then there exists a segment $s \in S$ that cuts $D(u, v, w; I_{|e})$ and separates p from u, v, w, so s must also cut at least one of the circum-disks of e's adjacent triangles. Since u, v, w are visible from e, hence on the same side of s as e, it follows that s separates p from e and p is not in conflict with e. $\qquad\square$

These properties imply that if we have computed $\mathrm{CDT}(R, S)$, we can compute $\mathrm{CDT}(P, S)$ by repeating the following step for each $e \in \mathrm{CDT}(R, S)$: Compute $\mathrm{CDT}(P_{|e}, I_e)$ and report a triangle $\triangle uvw \in \mathrm{CDT}(P_{|e}, I_e)$ if the center of $C(u, v, w; S)$ lies inside $\tau(e)$. Property (iii) guarantees that the reported triangles are all valid triangles of $\mathrm{CDT}(P, S)$; Property (ii) and (iv) ensure that no

triangle of $\mathrm{CDT}(P, S)$ is missing; and Property (i) makes sure that no duplicated triangles are reported.

There is one more technical subtlety when we check if the center of $C(u, v, w; S)$ lies inside $\tau(e)$: Although both $C(u, v, w; S)$ and $\tau(e)$ are defined on Σ_S, it suffices to do the check on the simpler surface Σ_{I_e}, since if adding the other secondary sheets associated with $S \setminus I_e$ moves the center of $C(u, v, w; S)$ from the primary sheet to a secondary sheet, the corresponding region of $\tau(e)$ will also be moved to the same sheet.

In our algorithm, we will carry out the above basic step recursively and in an I/O-efficient manner.

Our algorithm. As mentioned, the overall structure of our algorithm is the same as that of the algorithm of Crauser et al. [11]. We call a subset $R \subseteq P$ a p-sample if R is obtained by choosing each point of P with probability p. We choose a sequence of subsets of P, called a *gradation*:

$$P_1 \subseteq P_2 \subseteq \cdots \subseteq P_l = P,$$

where $E \subseteq P_1$ and $P_i \setminus E$ is a (B/M)-sample of $P_{i+1} \setminus E$. P_1 is small enough so that $\mathrm{CDT}(P_1, S)$ can be computed in main memory.

Initially, our algorithm constructs $\mathrm{CDT}(P_1, S)$ using an internal memory algorithm. Then we scan P and for each point $p \in P \setminus P_1$ determine the edges of $\mathrm{CDT}(P_1, S)$ that it is in conflict with; for each such edge e, we generate an (e, p) pair. In the end we sort these pairs to create the conflict lists for all the edges of $\mathrm{CDT}(P_1, S)$.

Next, we proceed in $l - 1$ rounds. In the i-th round, we are given $\mathrm{CDT}(P_i, S)$ and the conflict lists for all the edges of $\mathrm{CDT}(P_i, S)$, and construct $\mathrm{CDT}(P_{i+1}, S)$ and the conflict lists for the edges of $\mathrm{CDT}(P_{i+1}, S)$ (the conflict lists need not be generated for the last round). This is accomplished by the following steps.

1. For each edge e of $\mathcal{T}_i = \mathrm{CDT}(P_i, S)$, scan its conflict list and determine $P_{i+1|e}$.

2. Consider each $P_{i+1|e}$ in turn:

   2.1 Let $t_e = \lceil |P_{i+1|e}|/c_1(M/B) \rceil$. First take a $1/(c_2 t_e \log t_e)$-sample Y_e of $P_{i+1|e}$; we add the four vertices of the two adjacent triangles of e if they are not chosen in the sample. Then compute $\mathcal{T}_e = \mathrm{CDT}(Y_e, I_e)$ using an internal memory algorithm. Next for each edge e' of $\mathcal{T}_e$ determine $(P_{i+1|e})_{|e'}$ by scanning $P_{i+1|e}$ on disk. If for any e we have $|(P_{i+1|e})_{|e'}| > c_1 M/B$, repeat this step by taking a new sample Y_e.

   2.2 For each edge e' of $\mathcal{T}_e$, load $(P_{i+1|e})_{|e'}$ into memory and compute $\mathcal{T}_{e'} = \mathrm{CDT}((P_{i+1|e})_{|e'}, I_{e'})$. Report only the triangles of $\mathcal{T}_{e'}$ that have their circumcircles centered inside $\tau(e) \cap \tau(e')$. If this is not the last round, scan $P_{|e}$ to build the conflict lists for these triangles ($\triangle uvw$ is in conflict with p if $p \in D(u, v, w; S)$). We do so by allocating one main memory block for each of the $O(\frac{M}{B})$ triangles and writing points to the relevant block as they are processed; when a block is full it is written to disk.

3. After all edges of $\mathrm{CDT}(P_i, S)$ have been processed, $\mathcal{T}_{i+1} = \mathrm{CDT}(P_{i+1}, S)$ is simply all the triangles reported in Step 2.2. The conflict list for an edge of $\mathrm{CDT}(P_{i+1}, S)$ is simply the union of the conflict lists of its two adjacent triangles.

Analysis of I/O. We wish to follow the analysis of Crauser et al. [11] that is based on the bounds on the expected size of the conflict lists and their higher moments [10]. However, unlike [11], P_i is not a completely random sample of P_{i+1} in our case, which makes the analysis more complicated. Nevertheless, we can prove similar bounds on the expected size of conflict lists. The following lemma summarizes the main technical result.

Lemma 2 *Let R be a p-sample of $P \setminus E$. For any constant integer $c \geq 1$,*

$$\mathbf{E}\left[\sum_{e \in \mathrm{CDT}(R \cup E, S)} |P_{|e}|^c \right] = O\left(\frac{|P| - |E|}{p^{c-1}} + \frac{|E|}{p^c} \right). \tag{1}$$

Proof: Our proof is based on the probabilistic technique of Clarkson and Shor [10]. We define a *cell* to be a 4-tuple $\sigma = (p, q, u, v)$ for any 4 points $p, q, u, v \in P$. Define σ's *conflict list* $P_{|\sigma}$ to be the set of points of P that lie inside $D(p, q, u; S) \cup D(p, q, v; S)$, and let $l(\sigma) = |P_{|\sigma}|$. For any $R \subseteq P \setminus E$, let $\Pi(R)$ be the set of cells whose defining four points are in $R \cup E$. We say that a cell $\sigma \in \Pi(R)$ is at level c if $R \cup E$ contains exactly c points of $P_{|\sigma}$. A cell at level 0 is also called an *active* cell of $\Pi(R)$. Let $\Pi^c(R)$ be the set of cells of $\Pi(R)$ at level c.

Consider the active cells $\Pi^0(R)$ and $\mathrm{CDT}(R \cup E, S)$. Since each edge of $\mathrm{CDT}(R \cup E, S)$ corresponds to four cells in $\Pi^0(R)$, and the conflict list size of the edge is equal to conflict list size of each of the cells plus 4, it suffices to prove the following, for a $(1/p)$-sample R of $P \setminus E$.

$$\mathbf{E}\left[\sum_{\sigma \in \Pi^0(R)} \binom{l(\sigma)}{c} \right] = O\left(\frac{|P| - |E|}{p^{c-1}} + \frac{|E|}{p^c} \right) \tag{2}$$

We first show that the LHS of (2) is $O(\mathbf{E}[|\Pi^c(R)|/p^c])$. A cell σ appears in $\Pi^c(R)$ if and only if all of its 4 defining points and exactly c points of $P_{|\sigma}$ are contained in $R \cup E$. Let $d_0(\sigma)$ and $l_0(\sigma)$ respectively be the number of defining and conflicting points of σ in E. Then,

$$\mathbf{Pr}[\sigma \in \Pi^c(R)] = \binom{l(\sigma) - l_0(\sigma)}{c - l_0(\sigma)} p^{4 - d_0(\sigma) + c - l_0(\sigma)} (1 - p)^{l(\sigma) - c}.$$

Since

$$\mathbf{E}[|\Pi^c(R)|] = \sum_{\sigma} \mathbf{Pr}[\sigma \in \Pi^c(R)],$$

we have

$$\mathbf{E}[|\Pi^c(R)|] = \sum_{\sigma} \binom{l(\sigma) - l_0(\sigma)}{c - l_0(\sigma)} p^{4 - d_0(\sigma) + c - l_0(\sigma)} (1 - p)^{l(\sigma) - c}. \tag{3}$$

The LHS of (2) is

$$\sum_{\sigma} \binom{l(\sigma)}{c} \mathbf{Pr}[\sigma \in \Pi^0(R)] = \sum_{\sigma} \binom{l(\sigma)}{c} \binom{l(\sigma) - l_0(\sigma)}{-l_0(\sigma)} p^{4 - d_0(\sigma) - l_0(\sigma)} (1 - p)^{l(\sigma)},$$

which is bounded by $O(\mathbf{E}[|\Pi^c(R)|/p^c])$ by comparing with (3) and noting that

$$\binom{l(\sigma)}{c} \binom{l(\sigma) - l_0(\sigma)}{-l_0(\sigma)} \leq \binom{l(\sigma) - l_0(\sigma)}{c - l_0(\sigma)}.$$

8

Next, we take a 1/2-sample R' of R and consider $\Pi^0(R')$. A cell $\sigma \in \Pi^c(R)$ appears in $\Pi^0(R')$ if and only if all of its $4 - d_0(\sigma)$ defining objects in R and none of its $c - l_0(\sigma)$ conflicting points in R are taken into R'. So

$$\mathbf{Pr}[\sigma \in \Pi^0(R')] = \frac{1}{2^{4-d_0(\sigma)+c-l_0(\sigma)}}.$$

If follows that

$$\mathbf{E}[|\Pi^0(R')|] = \sum_{\sigma \in \Pi(R)} \mathbf{Pr}[\sigma \in \Pi^0(R')] \geq \sum_{\sigma \in \Pi^c(R)} \mathbf{Pr}[\sigma \in \Pi^0(R')] \geq \frac{|\Pi^c(R)|}{2^{4+c}}.$$

Therefore, the LHS of (2) is bounded by

$$
\begin{aligned}
O\left(\frac{\mathbf{E}[|\Pi^c(R)|]}{p^c}\right) &= O\left(\frac{\mathbf{E}[|\Pi^0(R')|]}{p^c}\right) = O\left(\frac{\mathbf{E}[|R' \cup E|]}{p^c}\right) = O\left(\frac{\mathbf{E}[|R \cup E|]}{p^c}\right) \\
&= O\left(\frac{|P| - |E|}{p^{c-1}} + \frac{|E|}{p^c}\right),
\end{aligned}
$$

as desired. $\qquad\square$

Remark 1 In our proof, the use of the random sampling framework of [10] crucially depends on the fact that all the segments and there endpoints are included in the constrained Delaunay triangulation of the sample, which allows us to use 4 points to define a cell. Otherwise the size of a cell's defining set becomes unbounded and the framework of [10] is not applicable.

In our algorithm, $P_i \setminus E$ is a p_i-sample of $P \setminus E$. If $|E| \leq c_1 M$ for some constant c_1 small enough, we have $|E| \leq c_2 \mathbf{E}[|P_i - E|]$ for some constant c_2, which means that "on average" at least a constant fraction of the samples in P_i are random. In this case (1) becomes $O(N/p_i^{c-1})$. Setting $c = 1$ yields that the expected total size of the conflict lists is linear.

Since the conflict list size is expected linear, the initialization step of our algorithm takes expected $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. In each round, Step 1 takes $O(\frac{N}{B})$ I/Os, and since Step 2.1 is repeated only a constant number of times with high probability, the total cost of Step 2 and 3 is

$$O\left(\sum_{e \in \mathcal{T}_i} t_e \log t_e \cdot \frac{|P_{|e|}|}{B}\right) \tag{4}$$

for the last round and

$$O\left(|\mathcal{T}_{i+1}| + \sum_{e \in \mathcal{T}_i} t_e \log t_e \cdot \frac{|P_{|e|}|}{B}\right)$$

for the other rounds with high probability.

Since $\mathbf{E}[|\mathcal{T}_{i+1}|] = O(\frac{N}{B})$ for all but the last round, we only need to argue that the expected value of (4) is also $O(\frac{N}{B})$. Following [11], for any integer constant c, we have

$$
\begin{aligned}
\mathbf{E}\left[\sum_{e \in \mathcal{T}_i} t_e^c \frac{|P_{|e|}|}{B}\right] &= \frac{1}{B} \cdot O\left(\frac{1}{(M/B)^c} \cdot \mathbf{E}\left[\sum_{e \in \mathcal{T}_i} |P_{i+1|e|}|^c \cdot |P_{|e|}|\right]\right) \\
&= \frac{1}{B} \cdot O\left(\frac{p_{i+1}^c}{(M/B)^c} \cdot \mathbf{E}\left[\sum_{e \in \mathcal{T}_i} \cdot |P_{|e|}|^{c+1}\right]\right) \\
&= \frac{1}{B} \cdot O\left(\frac{p_{i+1}^c}{(M/B)^c} \cdot \frac{N}{p_i^c}\right) = O\left(\frac{N}{B}\right).
\end{aligned}
$$

Since the number of rounds of the algorithm is $O(\log_{M/B} \frac{N}{B})$ with high probability, summing this expected cost over all the rounds, we obtain the following.

Theorem 1 *The constrained Delaunay triangulation of a set of N points in $\mathbb{R}^2$ and a set of segments S can be computed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ expected I/Os, provided that $|S| \leq c_0 M$, where c_0 is a constant.*

3 Experiments

In this section we describe experiments with our constrained Delaunay triangulation algorithm using both synthetic and real-life data. We implemented and experimented with a simplified version of the theoretical algorithm described in Section 2; we describe the simplification and implementation details in Section 3.1. In Section 3.2 we describe our experimental setup, including the datasets we used, and in Sections 3.3, 3.4 and 3.5 we report our experimental results.

3.1 Simplified algorithm and implementation details

We implemented and experimented with a simplified version of the theoretical algorithm described in Section 2. The main observation behind our simplification is that one round of the multi-round theoretical algorithm is enough to handle most real-world datasets. Even if we only have 128MB of main memory, which is more than the amount of memory needed to triangulate 10^5 points, about $(10^5)^2 = 10^{10}$ points can be processed with just one round. This naturally leads to the following simple and practical algorithm:

1. Compute a random sample P_1 of P of size $c \cdot \max\{K, \sqrt{N}\}$ that includes all endpoints of segments in S, where c is a constant. We set $c = 3$ in all our experiments.

2. Construct $\text{CDT}(P_1, S)$ in memory using an internal memory algorithm.

3. For each point $p \in P$ in turn determine the edges that p is in conflict with, generating a pair (e, p) for each such edge $e \in \text{CDT}(P_1, S)$. Then sort all these pairs to construct the conflict list $P_{|e}$ for each edge e. If any conflict list is larger than M, restart the algorithm by taking a new sample.

4. For each edge $e \in \text{CDT}(P_1, S)$ in turn load its conflict list $P_{|e}$ into memory and construct $\text{CDT}(P_{|e}, I_e)$ using an internal memory algorithm. Then report all the triangles whose circumcenters are inside $\tau(e)$.

Note that since we compute $\text{CDT}(P_1, S)$ in Step 2, we require that both K and $\sqrt{N}$ are smaller than the memory size.

Implementation. Each of the four steps in the algorithm is relatively easy to implement both I/O and internal memory efficiently. Step 1 can be implemented by scanning through the input points P, and to implement Step 2 we simply use the `triangle` package [22].

In Step 3, we first scan through the input points, and find conflicting edges with $\text{CDT}(P_1, S)$ kept in internal memory. To find the edges in conflict with a point p (internal memory) efficiently, it is sufficient to find all triangles in conflict with p. Since all triangles in conflict with p are connected, we simply first locate the triangle containing p and then perform a BFS search to find

all triangles that are in conflict with p. Rather than using a complicated (internal memory) point location structure to find the triangle of $\mathrm{CDT}(P_1, S)$ containing p, we pre-sort all points according to the Hilbert space-filling curve, which has high spatial locality, and use a simple point-location algorithm while processing the points in Hilbert order: To locate a point p, we start from the triangle γ where the previous point was located and "walk towards" p by traversing all triangles intersected by the line segment from the centroid of γ to p. Since the locations of consecutive points are likely to be very close (due to the Hilbert ordering), we in practice perform each point location query in constant time. At the end of Step 3 we sort the list of edge-point pairs.

Finally, in Step 4, we scan this list and use the `triangle` package to compute the constrained Delaunay triangulation of each conflict list.

Reducing conflict list size. In practice, the efficiency of our simplified algorithm mainly depends on the total size of the conflict lists. The theoretical analysis in Section 2 shows that the expected total size is linear and in practice the constant is roughly 9. We reduce the total conflict size and thus improve the overall efficiency of the algorithm by combining several adjacent edges into a single "edge group". More precisely, we put several edges into an edge group, and a point is in conflict with the edge group if it conflicts with any edge in the group. We then compute the conflict list and solve the subproblem for each edge group. The idea is that if the conflict lists of the individual edges in the same edge group contain many of the same points, then the total size of the conflict lists of the edge groups can be much smaller than the combined size of the individual conflict lists. However, if we merge many edges into an edge group, we need to do more work when we report valid triangles in Step 4, as we need to report a triangle if its circumcenter lies in the kernel of one of the edges in the group. So we need to choose an appropriate edge group size.

We use the following heuristic to merge edges into edge groups. Initially all edges are unmarked. To form an edge group G, we first randomly choose a "seed" triangle from $\mathrm{CDT}(P_1, S)$ with all three edges unmarked. If no such a triangle exists, we choose one with two unmarked edges (or one if such a triangle still does not exists). Then we mark all it unmarked edges and add them to G. Next from this seed, we grow the edge group in a BFS manner, favoring neighboring triangles with most unmarked edges. Once a triangle is visited, all of its unmarked edges are marked and added to G. We continue this process to form G until we have visited ρ triangles, or there are no neighboring triangles with at least one unmarked edge. If there are still unmarked edges left, we form another edge group by repeating the process.

As discussed there is a tradeoff between the total conflict list size and the cost to check for valid triangles in Step 4. The larger ρ is, the smaller the conflict list becomes, but it is also more costly to check for valid triangles. In order to determine an appropriate value of ρ, we performed experiments with uniformly randomly generated datasets with different values of ρ. We found that $\rho = 10$ gave the best improvement: The constant factor in the conflict list size is reduced to around 3 and the total running time is reduced by 50%. In fact, the improvements with ρ ranging from 5 to 20 do not differ much. We used $\rho = 10$ in all of our experiments.

There is a technical subtlety here when dealing with edge groups instead of individual edges. In Step 4, we just build $\mathrm{CDT}(P_{|e}, I_e)$ if only a single edge e is considered. However, for an edge group G, it is not enough to build $\mathrm{CDT}(\cup_{e \in G} P_{|e}, \cup_{e \in G} I_e)$, since points from the conflict lists of different edges may "interfere" with each other, and generate wrong triangles. Such an example is shown in Figure 4(a). To ensure correctness, we need to include all the *effective* segments of the triangles adjacent to the edges of G during the computation of the sub-triangulation of G. An effective segment of an triangle $\triangle uvw$ is a segment of S that intersects the circumcircle of $\triangle uvw$ and is not

separated from the triangle by any other segment. Some examples are shown in Figure 4(b). We determine the effective segments of each triangle of $\text{CDT}(P_1, S)$ in an additional step before Step 4: We first build an R-tree on the bounding boxes of all circumcircles, and then query the R-tree with the segments of S in turn. Each triangle always keeps its effective segments seen so far, and discard segments as soon as they become not effective. In the end, each triangle of $\text{CDT}(P_1, S)$ obtains a list of all its effective segments.

(a)   (b)
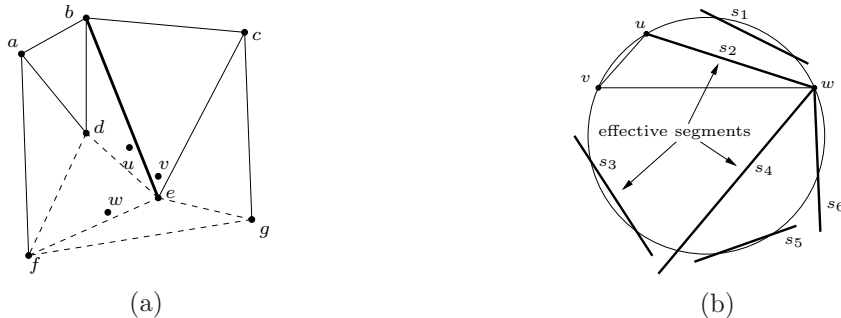
Figure 4: (a) Suppose the sample is $P_1 = \{a, \ldots, g\}$, and set of segment is $S = \{be\}$. Let u, v, w be three points in $P \setminus P_1$, and $u, w \in P_{|de}, v \in P_{|eg}$. Consider the edge group G consisting of the dashed edges. We also need to include the segment be when building the constrained Delaunay triangulation of G, otherwise a wrong triangle $\triangle uvw$ would be reported. (b) Segments s_2, s_3, s_4 are the effective segments of $\triangle uvw$, while the other segments are not.

3.2   Experimental setup and datasets

We implemented our simplified constrained Delaunay triangulation algorithm in C++ using TPIE [4]. TPIE is a library that provides support for implementing I/O-efficient algorithms and data structures. We used `double` to store the coordinates of each point. For experimentation, we used a 2.4GHz Intel XEON machine with hyperthreading, running Linux with kernel 2.4.5-smp, and a local disk system consisting of four 10000RPM 72GB SCSI disks in RAID-0 configuration. The machine had 1GB main memory, but we restricted it to use only 128MB of memory in order to obtain a large data size to memory size ratio. All input, output and temporary files were stored on the local disk system.

Synthetic datasets.   We experimented with both synthetic and real-life data. For the synthetic data, we used four different distributions that have been used to evaluate the performance of Delaunay triangulation algorithms [6]: uniform, normal, the Kuzmin, and line singularity.

- *Uniform distribution*: Random points in a unit square.

- *Normal distribution*: Points (x, y) where x, y are independent samples from the normal distribution.

- *The Kuzmin distribution*: Points form a radically symmetric distribution with the accumulative probability function $M(r) = 1 - 1/\sqrt{1 + r^2}$ where r is the distance to the center. The Kuzmin distribution converges to the center at a much faster rate than the normal distribution. It is used to model the distribution of star clusters in flat galaxy formations.
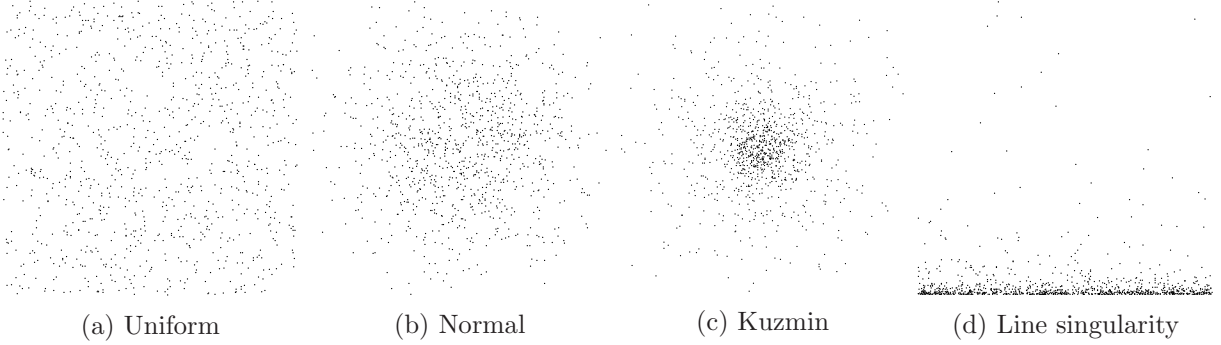
(a) Uniform   (b) Normal   (c) Kuzmin   (d) Line singularity

Figure 5: Sample datasets of 1000 points for the four distributions.
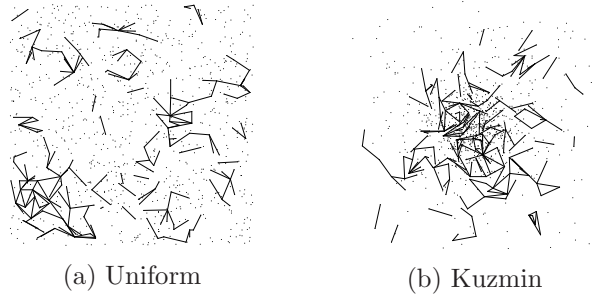

(a) Uniform   (b) Kuzmin

Figure 6: Sample datasets of 1000 points with segments.
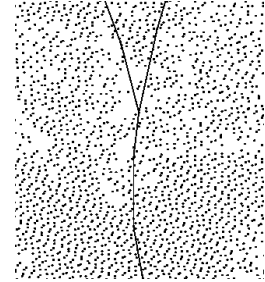

Figure 7: LIDAR data.

- *Line Singularity distribution*: Points form a distribution that converges to a line instead of a point. Let u and v be two independent uniform variables in $[0, 1]$. A point (x, y) is obtained by the transformation $(x, y) = (b/(u - bu + b), v)$. We set $b = 0.01$.

After generating a point set P from one of the distributions, we generate the segment set S as follows: To obtain a segment $s \in S$, we first choose one endpoint uniformly at random from P. With some probability α we choose the other endpoint uniformly at random from P; with probability $1 - \alpha$ we choose it uniformly at random from the endpoints of the segments already in S. We add s to S if it does not intersect any existing segment in S and its length is smaller than some threshold δ. In our experiments we fixed $\alpha = 0.2$. Some examples of the segments generated this way are shown in Figure 6.

Real data. Our real-life datasets consist of LIDAR data for the Neuse River Basin of North Carolina [1]. This data consist of points $p = (x, y, z)$ in $\mathbb{R}^3$ and to obtain a point set P in $\mathbb{R}^2$ we simply used the x and y coordinates. We broke the data into a number of "tiles" geographically, and concatenated different subsets of the tiles together to create 9 datasets of increasing sizes. For the segments S, we used road data segments obtained from the TIGER/Line data [24]. For each point dataset P, we used all the segments from the TIGER/Line data that completely fall into the bounding box of P. The numbers of points and segments of the datasets are listed in Table 1; the last dataset covers the entire Neuse River Basin and has half of billion points. Compared with the synthetic data, the LIDAR data points are more regularly distributed, but with a few low density
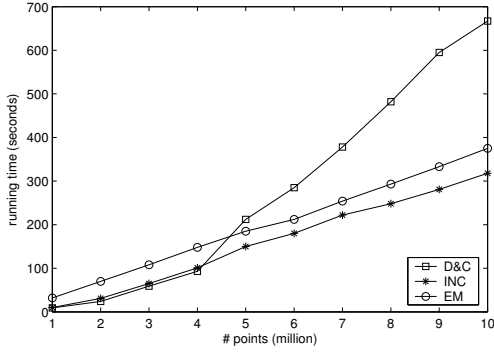
13

Figure 8: Delaunay triangulation results on uniform distribution.
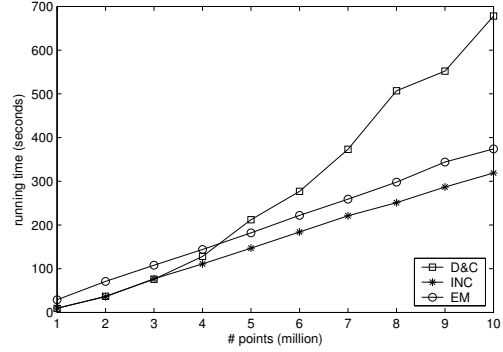

Figure 9: Delaunay triangulation results on normal distribution.

regions (e.g. lakes); the segments are better structured and tend to form long chains. A small portion of the LIDAR data is shown in Figure 7.

Dataset	1	2	3	4	5	6	7	8	9
# points (million)	16.8	27.7	44.5	58.5	90.8	116.2	163.1	257.1	503.7
# segments (thousand)	19.5	27.8	55.7	44.9	50.5	77.3	137.3	627.1	755.0
Input file size (MB)	336	554	890	1176	1816	2324	3262	5142	10074

Table 1: The number of points and segments in each dataset of the Neuse River Basin.

3.3 Delaunay triangulation experiments

We first investigate the performance of our algorithm when $S = \emptyset$, that is, when we are computing standard Delaunay triangulations. We compared our external memory algorithm (EM) with the (internal memory) divide-and-conquer (D&C) and incremental (INC) algorithm as implemented in the `triangle` package [22]. Since it is known that pre-sorting the points along some space-filling curve improves the performance of D&C and especially INC greatly on modern memory hierarchies [3], we sorted the points along the Hilbert curve in all our experiments. If the points are not sorted, D&C starts thrashing and takes more than 10 hours to complete on a dataset of only 5 million points; INC starts thrashing on an even smaller dataset of 2 million points. The time used to perform the Hilbert curve sort is not included in the computation times reported below.

The experimental results of our experiments on the four distributions with datasets of sizes varying from 10^6 to 10^7 are shown in Figure 8, 9, 10 and 11. Note that the 128MB main memory can only hold the data structure for triangulating roughly 1 million points. In all experiments, INC performs best. Its running time is almost linear in the data size, which can be explained by the fact that the `triangle` package starts by checking if a point to be inserted lies in any of the neighboring triangles of the last inserted point; this way the more complicated point location is almost always avoided when the points are inserted in Hilbert order, and each point is almost always handled in constant time (since the number of structural changes needed when inserting a point is also almost always constant). The number of I/Os (page faults) is also low since INC visits its data structure
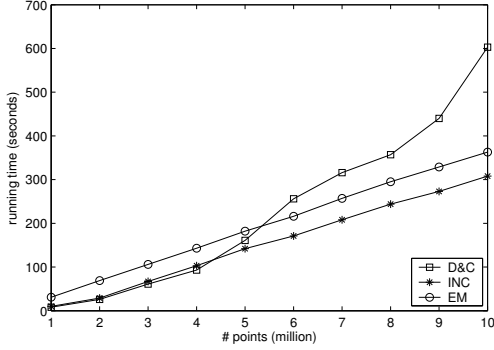
14

Figure 10: Delaunay triangulation results on the Kuzmin distribution.
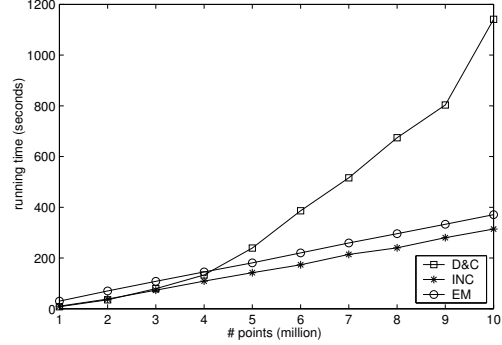

Figure 11: Delaunay triangulation results on line singularity distribution.

in a highly local manner. The running time of our EM algorithm is also linear in the input size, which can be explained by the fact that the $O(\log_{M/B} \frac{N}{B})$ factor in the number of I/Os it performs is basically a constant for the range of N's we tested. Still the algorithm performs around 20% worse than INC because of the overhead in the conflict lists. Although the D&C algorithm is faster than the two algorithms as long as the dataset fits in main memory, which agrees with previous experimental studies on Delaunay triangulation algorithms [22, 23], when the dataset size grows larger than the available main memory, its performance quickly degenerates. Although the Hilbert ordering of points improves its performance (since the points visited in a merge step are likely to be placed consecutively in memory), the binary nature of the algorithm results in an $O(\frac{N}{B} \log_2 \frac{N}{B})$ I/O behavior.

3.4   Constrained Delaunay triangulation experiments

Next we compared our EM algorithm with the algorithm (INC) implemented in `triangle` [22], which first constructs a Delaunay triangulation on the input points P, and then inserts all the segments in S one by one. The `triangle` package offers several options for constructing Delaunay triangulations, we used the incremental algorithm since it is shown to be fastest in the previous section. When inserting a segment $s \in S$, INC first destroys all the triangles s intersects, and then retriangulates the two polygons on either side of s. To retriangulate a polygon of size t it uses a simple worst-case $O(t^2)$ algorithm, which runs in linear time in most common cases. There exist theoretically optimal linear-time simple polygon constrained Delaunay triangulation algorithms [9, 17], but they are too complicated to implement and unlikely to be practical. As before we pre-sorted the points by Hilbert values; we sorted the segments by the Hilbert value of one of their endpoints.

Experiments with synthetic data.   The running times of our experiments on the four distributions are shown in Figure 12, 13, 14 and 15. We fixed the number of points to be 10^7 and generated up to 10^5 segments, each of length at most $\delta = 0.003$. The range of the number of segments are chosen to resemble the segment-to-point ratios of the real-life LIDAR datasets, as well as larger ratios. The experimental results show that our EM algorithm performs significantly better than INC. The main reason for this is probably that while our algorithm incrementally inserts points in

15

a small triangulation in memory ($\mathrm{CDT}(P_1, S)$), the INC algorithm incrementally inserts segments in a much larger (and larger than main memory) triangulation containing all the points. The other reason is that the `triangle` package is not fully optimized for constrained Delaunay triangulations. See Section 3.5 for more details.

The performance of the EM algorithm starts to (very slowly) degenerate at around 60,000 segments. This can be explained by the fact that the memory usage of the algorithm almost only depends on the sample size $|P_1|$; at $K = 60,000$ the sample is about the size of the main memory (we consume about 5MB memory per 10,000 points, and sample $3K$ points; the system daemons use roughly 30MB memory). Although in theory our algorithm only works when the sample fits in internal memory, we see that thrashing does not happen when this assumption is violated. Instead the performance of the algorithm degrades quite gracefully because the algorithm has a very local memory access pattern. Note that as the number of segments approaches N, our algorithm will eventually degenerate into INC.

Both algorithms are relatively insensitive to the data distribution. However, as the number of segments grows, the running time of the EM algorithm degrades faster on line singularity distribution than on the other distributions. This is because the points are much denser in this dataset,
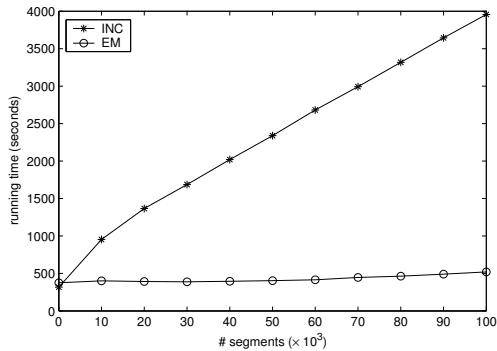

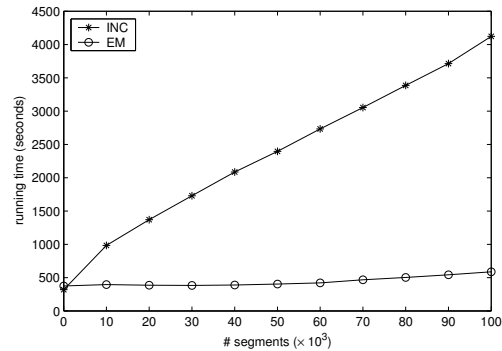Figure 12: Constrained Delaunay triangulation results on uniform distribution.

Figure 13: Constrained Delaunay triangulation results on normal distribution.
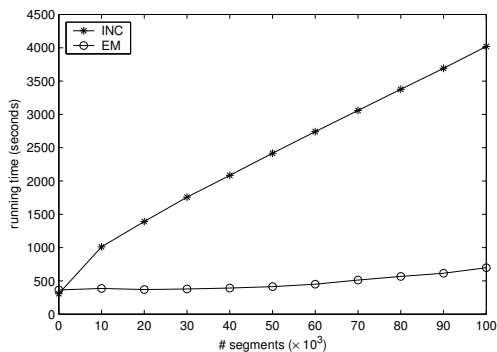

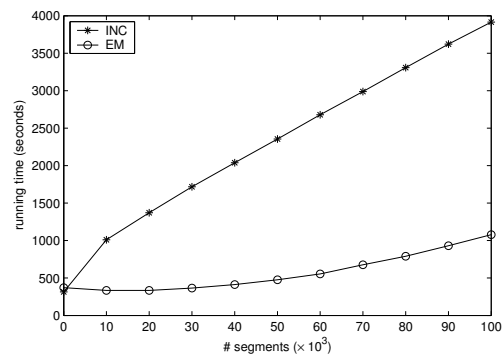Figure 14: Constrained Delaunay triangulation results on the Kuzmin distribution.

Figure 15: Constrained Delaunay triangulation results on line singularity distribution.

16

meaning that segments are relatively longer. The EM algorithm spends more time on the R-tree construction since one segment may intersect a lot of circumcircles.

Next we investigated how the segment length affects performance. Using 10^7 points from the uniform distribution, we generated 10,000 segments with varying δ from 0.001 to 0.1 using only segments of length between $\delta/2$ and δ. The results of the experiments with these datasets are given in Figure 16. As expected, the running time of INC increases as the segments get longer, since more triangles are destroyed and created when a longer segment is inserted in the triangulation, but the increase is not substantial. Maybe somewhat counter-intuitively, the running time of EM decreases as the segments get longer. This is probably because while longer segments increase the time to triangulate the sample, they also reduce the conflict list size somewhat.
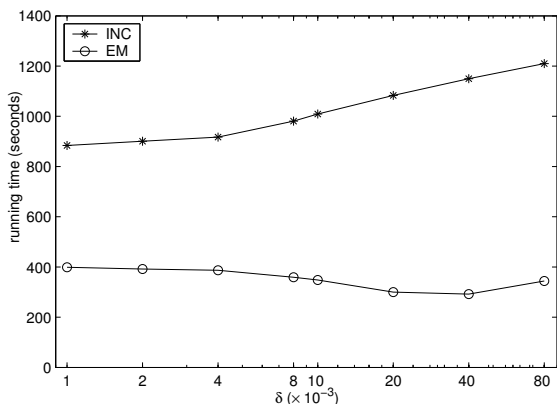

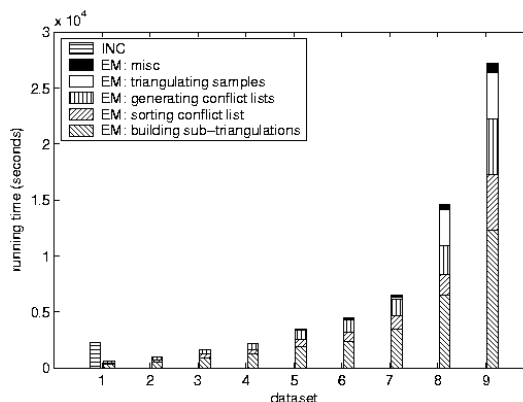Figure 16: Constrained Delaunay triangulation results with varying segment lengths.

Figure 17: Constrained Delaunay triangulation results on real datasets.

Experiments with real data. The running times of our experiments with the LIDAR datasets are shown in Figure 17. Note that the smallest LIDAR dataset is larger than the largest of our synthetic dataset, thus, due to insufficient address space on a 32-bit machine (there is a 4GB limit on the address space for each process), we were unable to run the `triangle` program except on the smallest dataset. In Figure 17 we also show a breakdown of the running time of the EM algorithm into different phases: triangulating the samples, generating the conflict lists, sorting the conflict lists, and building the sub-triangulations. Except on the last two datasets, the total running time is dominated by the last three phases, which essentially depends on the number of points. On the last two datasets, the number of segments is much larger than in the other datasets, and the time spent on building $\mathrm{CDT}(P_1, S)$ starts to be significant. However, since P_1 is still much smaller than the entire dataset, our algorithm should still be much faster than building the entire constrained Delaunay triangulation directly.

3.5 Optimizing the `triangle` program

As mentioned in Section 1, the problem of building large-scale constrained Delaunay triangulations efficiently has not been seriously considered before. Although it is one of the most popular Delaunay triangulation softwares publicly available, through our experiments we found that `triangle` still has some efficiency issues when constrained Delaunay triangulations are considered. In this section

we describe how we have further optimized portions of `triangle`'s code that deals with inserting segments in a triangulation, and give additional experimental results with the improved code.

Improvements to the `triangle` program. The following improvements apply to `triangle` version 1.5 (released June 4, 2004, available at `http://www-2.cs.cmu.edu/~quake/triangle.html`).

- `locate()`: This function is used in the point location routine. The code contains a bug when taking samples from the triangulation, which sometimes causes the program to generate segmentation faults when inserting segment to a triangulation, or using the incremental algorithm to build a Delaunay triangulation. We corrected the bug.

- `flip()`: This function is used in building the initial Delaunay triangulation with the incremental algorithm, and in retriangulating the two polygons after a segment has been inserted. When the program inserts segments, it maintains a mapping from vertices to triangles, so that the point location procedure can be avoided when trying to locate the endpoints of the segments. However, when an edge is flipped in this function, this mapping is not restored correctly, causing some of the subsequent segment insertions to fail to use the mapping to locate the endpoints directly, and the expensive point location procedure has to be invoked. We added code to restore the mapping after each flip, if the flip is caused by inserting a segment.

- `getvertex()`: This function is called to retrieve an endpoint of a segment from the list of existing vertices. The current code may take $O(N)$ time, and we improved it to constant time.

Additional experiment results with the improved code. With the improved code, we have rerun all the experiments of Section 3.4.

The results on the synthetic datasets are shown in Figure 18–22. From Figure 18–21, we see that these improvements have drastically reduced the running times of `triangle` when the number of segment is large. These curves in fact more reasonably demonstrate the behavior of the INC algorithm: When there are not so many segments, each insertion may cause a page fault; but when there are a lot of segments and they are inserted in the Hilbert order, the location of a segment is very likely to be close to that of the previous one, thus the likelihood of a page fault is small. This roughly explains why the INC curves start to level off as the number of segments grows. We do not see major differences in Figure 22 for the varying-segment-length experiment.

On the smallest LIDAR dataset, the running time of `triangle` decreases from 2272 seconds to 1252 seconds, while that of the EM algorithm is 607 seconds. We were still unable to run `triangle` on any of the other larger datasets.

In comparison with the improved `triangle` program, our EM algorithm is still generally a factor of two faster, when the number of segments is smaller than the memory size. However, when there are more segments and the point distribution is highly skewed, one needs to be careful in choosing which triangulator to use.
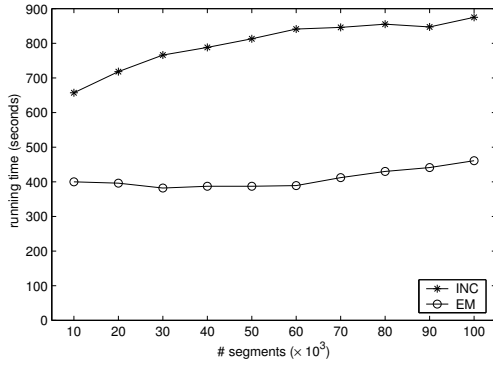
Acknowledgement

Figure 18: Constrained Delaunay triangulation results on uniform distribution (with improved code).
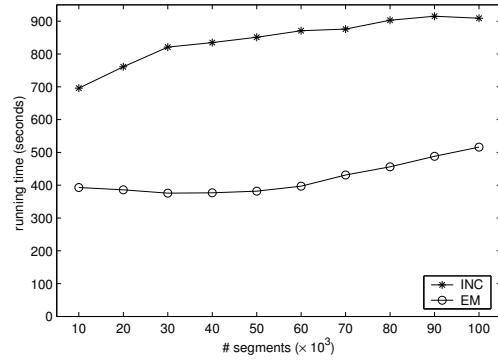

Figure 19: Constrained Delaunay triangulation results on normal distribution (with improved code).
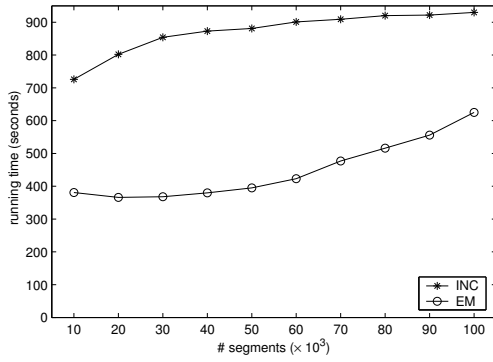

Figure 20: Constrained Delaunay triangulation results on the Kuzmin distribution (with improved code).
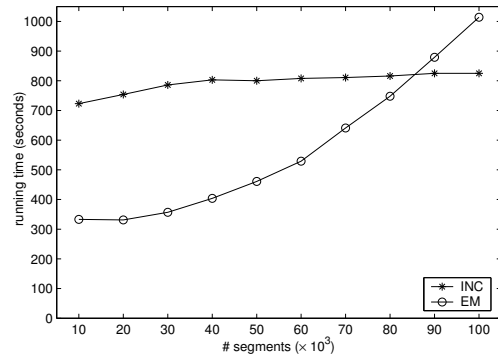

Figure 21: Constrained Delaunay triangulation results on line singularity distribution (with improved code).
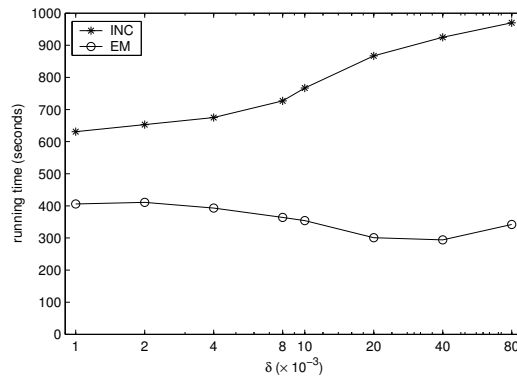

Figure 22: Constrained Delaunay triangulation results with varying segment lengths (with improved code).

References

[1] North Carolina Flood Mapping Program. http://www.ncfloodmaps.com.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *Proc. 19th Annu. ACM Sympos. Comput. Geom.*, pages 221–219, 2003.

[4] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.

[5] F. Aurenhammer and R. Klein. Voronoi diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[6] G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3):243–269, 1999.

[7] *The CGAL Reference Manual*, 1999. Release 2.0.

[8] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[9] F. Chin and C. A. Wang. Finding the constrained Delaunay triangulation and constrained Voronoi diagram of a simple polygon in linear time. *SIAM J. Comput.*, 28:471–486, 1998.

[10] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[11] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[12] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[13] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[14] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[15] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[16] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Computational Geometry*, 2:127–151, 1987.

[17] R. Klein and A. Lingas. A linear-time randomized algorithm for the bounded Voronoi diagram of a simple polygon. *Internat. J. Comput. Geom. Appl.*, 6:263–278, 1996.

[18] P. Kumar and E. A. Ramos. I/O-efficient construction of voronoi diagrams. Technical report, 2002.

[19] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.

[20] K. Mulmuley. *Computational Geometry. An introduction through randomized algorithms.* Prentice-Hall, 1994.

[21] R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. *Computer Science Division*, ??, June 1989. UC Berkeley.

[22] J. R. Shewchuk. Triangle: engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry.* Association for Computing Machinery, May 1996.

[23] P. Su and R. Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Comput. Geom. Theory Appl.*, 7:361–386, 1997.

[24] *TIGER/Line*[TM] *Files, 1997 Technical Documentation.* Washington, DC, September 1998. http://www.census.gov/geo/tiger/TIGER97D.pdf.

A   Randomized Incremental Construction of Constrained Delaunay Triangulations

Randomized incremental construction (RIC) is a powerful paradigm for designing many geometric algorithms. Typically such algorithms are conceptually simple, easy to implement, and are capable of doing the construction in an online fashion. While optimal constrained Delaunay triangulations algorithms following other paradigms are known, as mentioned in Section 1, there have been no provably optimal incremental algorithms for constrained Delaunay triangulations to date, although such an algorithm is often the choice of practitioners (e.g. Shewchuk's Triangle software [22]). The lack of an RIC algorithm also poses a major difficulty for us to develop an external memory constrained Delaunay triangulation algorithm. Therefore, it will be interesting to analyze an RIC algorithm for constructing constrained Delaunay triangulations.

Like in the Delaunay triangulation case, the first step is to bound the expected number of structural changes when we add the segments into a Delaunay triangulation in a random order. In this section, we report some preliminary results on this combinatorial problem, with the hope that they may shed some lights on future investigations of RIC algorithms for constrained Delaunay triangulations.

We first show that the expected number of structural changes is $\Omega(N \log K)$ by giving such an example. This is to be contract with the well known $\Theta(N)$ bound on such changes when a standard Delaunay triangulation is constructed. Then we give an $O(N \log^2 K)$ upper bound.

Lower bound.   Our lower bound example consists of $N + 3K = \Theta(N)$ points and K segments. We put K points $p_1, \ldots, p_K$ on the top of a circle, and N points $q_1, \ldots, q_N$ on the bottom of the circle (see Figure 23(a)). Since we are interested in the lower bound, we will only focus on the edges connecting some p_i to some q_j, ignoring all other edges. We call these edges interesting edges. To

make sure the Delaunay triangulation is always uniquely defined, we move p_1 downward by a small distance ϵ_1, such that the edges shown in Figure 23(a) are all Delaunay edges. Next, we move p_2 downward by a distance of ϵ_2, which is chosen small enough such that it does not affect the edges going out from p_1, but if p_1 did not exist, then all the q_j's would be connected to p_2. Similarly we move p_i downward by a distance of ϵ_i such that all the q_j's are connected to p_i if and only if all of $p_1, \ldots, p_{i-1}$ do not exist. Finally we place the K segments, along with the remaining $2K$ points serving as their endpoints, as follows. For each p_i, we put a segment s_i that hides it from the circle, with both endpoints of s_i being outside the circle (see Figure 23(b)). This is always possible if we choose the ϵ_i's small enough. It is also clear that these $2K$ points do not affect the interesting edges at all.

We have finished the description of our example, now let us compute the expected number of changes among the interesting edges when we add the K segments in a random order. It suffices to only count the number of edges that are destroyed during the process. Let T_i be the number of edges destroyed by the insertion of s_i. By our construction, T_i is N if $s_1, \ldots, s_{i-1}$ are inserted before s_i, and 0 otherwise. So, we have $\mathbf{E}[T_i] = N/i$. By linearity of expectation, the expected total number of edges destroyed is $\mathbf{E}[\sum_{i=1}^{K} T_i] = \Theta(N \log K)$.
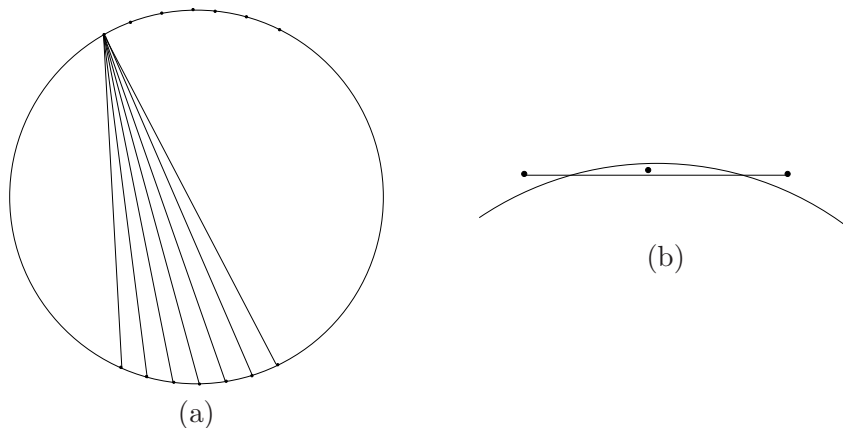

(a)

(b)

Figure 23: The lower bound example.

Upper bound. Let s_i be the i-th segment inserted, and let $S_i = \{s_1, \ldots, s_i\}$, for $i = 1, \ldots, K$. Let T_i be the number of edges of $\mathrm{CDT}(P, S_{i-1})$ intersected (in the interior) by s_i, thus, the insertion of s_i destroys $T_i + 1$ triangles and creates $T_i + 1$ new ones, and the total number of structural changes is $K + \sum_{i=1}^{K} T_i$.

For any segment e in the plane, define S_e as the subset of segments in S that intersect e. Consider the range space $(S, \{S_e : e \text{ is a segment in } \mathbb{R}^2\})$, which has constant VC dimension. The result by Haussler and Welzl [16] on ε-nets implies that a random sample of S of size $O((1/\varepsilon) \log K)$ is an ε-net of the range space with probability at least $1 - 1/K^{O(1)}$. This means that when $i = \Theta((1/\varepsilon) \log K)$, with this probability, no edge in $\mathrm{CDT}(P, S_{i-1})$ intersects more than εK segments in S, since no edge intersects any of the sampled segments. So the expected size of S_e is $O(\varepsilon K) = O(K/i \cdot \log K)$ for each edge e in $\mathrm{CDT}(P, S_{i-1})$. When $i < \log K$, the $O(K/i \cdot \log K)$ bound trivially holds.

Since s_i is randomly chosen from $S \setminus S_{i-1}$, we have

$$\mathbf{E}[T_i] = O\left(\frac{\sum_{e \in \mathrm{CDT}(P, S_{i-1})} |S_e|}{K - i}\right) = O\left(\frac{NK \log K}{i(K - i)}\right).$$

Then the expected total number of structural changes is

$$\mathbf{E}\left[K + \sum_{i=1}^{K} T_i\right] = O\left(\sum_{i=1}^{K} \frac{NK \log K}{i(K - i)}\right) = O(N \log^2 K).$$