I/O-Efficient Construction of Constrained Delaunay Triangulations

Pankaj K. Agarwal[1⋆], Lars Arge[2,1⋆⋆], and 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.

1 Introduction

With the emergence of new terrain mapping technologies such as Laser altimetry (LIDAR), 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 river and road networks, in which case one would like to construct a TIN that is consistent with this data, that is,

⋆ Supported in part by NSF under grants CCR-00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grants W911NF-04-1-0278 and DAAD19-03-1-0352, and by a grant from the U.S.–Israel Binational Science Foundation.

⋆⋆ Supported in part by the US NSF under grants CCR–9984099, EIA–0112849, and INT–0129182, by ARO grant W911NF-04-1-0278, and by an Ole Rømer Scholarship from the Danish National Science Research Council.

⋆⋆⋆ Supported by NSF under grants CCR-02-04118, CCR–9984099, EIA–0112849, and by ARO grant W911NF-04-1-0278.

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* $\mathrm{CDT}(P, S)$ 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. $\mathrm{CDT}(P, \emptyset)$ 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 reserve the term "edges" for the other edges in the triangulation $\mathrm{CDT}(P, S)$.

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

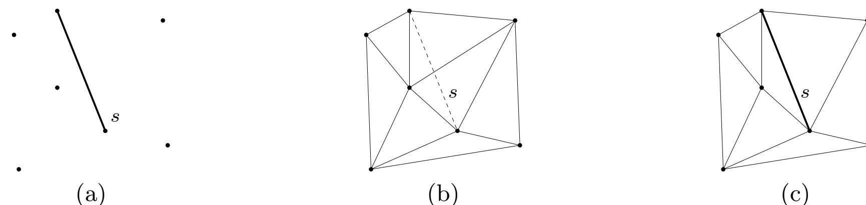

Fig. 1. (a) The point set P of 7 points and segment set S of 1 segment s. (b) $\mathrm{DT}(P) = \mathrm{CDT}(P, \emptyset)$. (c) $\mathrm{CDT}(P, S)$.

are based on different standard paradigms, such as divide-and-conquer and sweep-line. A randomized incremental algorithm with $O(N \log N)$ expected running time was proposed in [12]. By now efficient implementations of many of the developed algorithms are also available. For example, the widely used software package `triangle`, developed by Shewchuk [16], has implementations of all three algorithms mentioned above. Both CGAL [7] and LEDA [14] software libraries also offer Delaunay triangulation implementations.

By modifying some of the algorithms for Delaunay triangulation, $O(N \log N)$ time RAM-model algorithms have been developed for constrained Delaunay triangulations [8, 15]. However, these algorithms are rather complicated and do not perform well in practice. A common practical approach for computing $\mathrm{CDT}(P, S)$, e.g. used by `triangle` [16], is to first compute $\mathrm{DT}(P)$ and then add the segments of S one by one and update the triangulation.

Although I/O-efficient algorithms have been designed for Delaunay triangulations [10, 11, 13], no I/O-efficient algorithm is known for the constrained case.

Our results. By modifying the algorithm of Crauser et al. [10] 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. 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.

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 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 $\mathrm{CDT}(P, S)$ that follows the framework of Crauser et al. [10] for constructing Delaunay triangulations. However, we first introduce the notion of extended Voronoi diagrams, originally proposed by Seidel [15], and define conflict lists and kernels.

Extended Voronoi diagrams. We extend the plane to a more complicated surface as described by Seidel [15]. 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 the circumcircle definition, the constrained Delaunay triangulation can be defined in the same way as standard Delaunay triangulations, i.e., $\mathrm{CDT}(P, S)$ consists of all triangles $\triangle uvw, u, v, w \in P$, whose circumcircles do not enclose any point of P. We define the *extended Voronoi region* of a point $p \in P$ as $\mathrm{EV}(p, S) = \{x \in \Sigma_S \mid d(x, p) \leq d(x, q), \forall q \in P\}$, and the *extended Voronoi diagram* of P (with respect to S) as $\mathrm{EVD}(P, S) = \{\mathrm{EV}(p, S) \mid p \in P\}$. Seidel [15] showed that $\mathrm{CDT}(P, S)$ is the dual of $\mathrm{EVD}(P, S)$, in the sense that

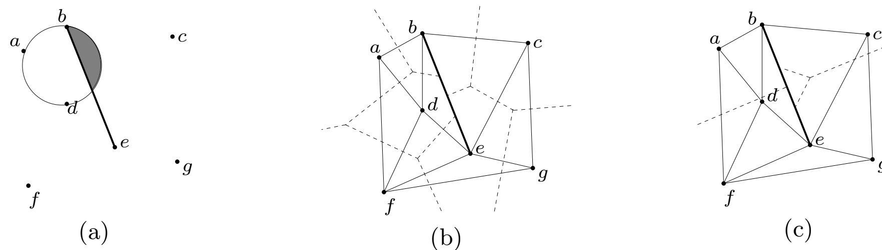

Fig. 2. (a) For the point set of Figure 1(a), a portion of $D(a, b, d; S)$ lies in the primary sheet (unshaded), the other portion lies in the secondary sheet Σ_{be} (shaded). (b) $\mathrm{CDT}(P, S)$ (solid lines) and the portion of $\mathrm{EVD}(P, S)$ (dashed lines) that lies in the primary sheet.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

an edge pq appears in $\mathrm{CDT}(P, S)$ if and only if $\mathrm{EV}(p, S)$ and $\mathrm{EV}(q, S)$ 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. [10] 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* [9] of e, denoted by $P_{|e} \subseteq P$, as the set of points of P that lie in $D(p, q, u; S) \cup D(p, q, v; S)$. If there exists a point $p' \in P_{|e}$, then 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, denoted by $\tau(e)$, which is contained in $\mathrm{EV}(p, S) \cup \mathrm{EV}(q, S)$. A point $x \in \mathrm{EV}(p, S)$ (resp. $x \in \mathrm{EV}(q, S)$) lies in $\tau(e)$ if the ray $\overrightarrow{px}$ (resp. $\overrightarrow{qx}$) intersects the common edge between $\mathrm{EV}(p, S)$ and $\mathrm{EV}(q, S)$. Refer to Figure 3. Note that the kernel of an edge e can be determined with knowing only e and its two adjacent triangles in $\mathrm{CDT}(R, S)$.

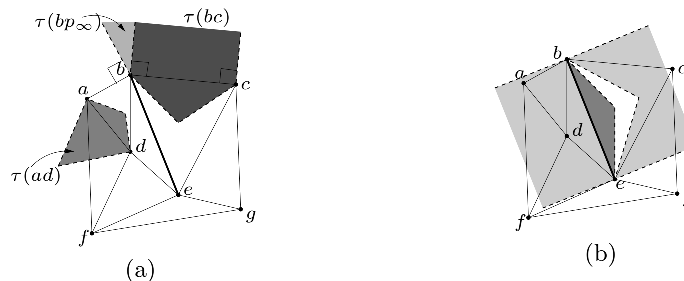

Fig. 3. (a) The kernels of edges ad, bc, and bp_∞. (b) The kernel of the edge be; the darker part lies in the primary sheet, and the lighter part lies in the secondary sheet Σ_{be}.

The following properties of conflict lists and kernels, whose proofs are omitted from this abstract, lead to a recursive algorithm for computing $\mathrm{CDT}(P, S)$.

(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.

(iii) Let $E \subseteq R \subseteq P$. 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) Let $E \subseteq R \subseteq P$. For any $\triangle uvw \in \mathrm{CDT}(P, S)$ with circumcenter ξ, if e is the edge of $\mathrm{CDT}(R, S)$ such that $\xi \in \tau(e)$, then $u, v, w \in P_{|e}$ and $D(u, v, w; I_e) \cap P_{|e} = \emptyset$.

These properties imply that if we have computed $\mathrm{CDT}(R, S)$, we can compute $\mathrm{CDT}(P, S)$ by repeating the following steps 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 $C(u, v, w; I_e)$ lies inside $\tau(e)$. Below we describe how to do this efficiently.

Our algorithm. As mentioned, the overall structure of our algorithm is the same as that of the algorithm of Crauser et al. [10]. 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 $\mathcal{T}_i = \mathrm{CDT}(P_i, S)$, we scan its conflict list and determine $P_{i+1|e}$.
2. We consider each $P_{i+1|e}$ in turn:
   2.1 Let $t_e = \lceil |P_{i+1|e}|/c_1(M/B) \rceil$. We 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 we compute $\mathcal{T}_e = \mathrm{CDT}(Y_e, I_e)$ using an internal memory algorithm. Next for each edge e' of $\mathcal{T}_e$ we determine $(P_{i+1|e})_{|e'}$ by scanning $P_{i+1|e}$ on disk. If for any e we have $|(P_{i+1|e})_{|e'}| > c_1 M/B$, we repeat this step by taking a new sample Y_e.
   2.2 For each edge e' of $\mathcal{T}_e$, we load $(P_{i+1|e})_{|e'}$ into memory and compute $\mathcal{T}_{e'} = \mathrm{CDT}((P_{i+1|e})_{|e'}, I_{e'})$. We report only the triangles of $\mathcal{T}_{e'}$ that have their circumcircles centered inside $\tau(e) \cap \tau(e')$. We then scan $P_{|e}$ to build the conflict lists for these triangles (unless this is the last round). 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 3. 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. [10] that is based on the bounds on the expected size of the conflict lists and their higher moments [9]. However, unlike [10], 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, whose proof is given in the full version of the paper.

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

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

In our algorithm, $P_i \setminus E$ is a p_i-sample of $P \setminus E$, therefore,

$$E\left[\sum_{e \in \mathcal{T}_i} |P_{|e}|^c\right] = O\left(\frac{|E| + |P_i \setminus E|}{p_i^c}\right) = O\left(\frac{|E|}{p_i^c} + \frac{|P \setminus E|}{p_i^{c-1}}\right), \tag{1}$$

Assuming $|E| \leq c_1 M$, we have that $|E| \leq c_1' E[|P_i - E|]$ for all i, 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 is $O\left(\sum_{e \in \mathcal{T}_i} t_e \log t_e \cdot \frac{|P_{|e}|}{B}\right)$ with high probability. Using (1) we can argue that the expected value of this expression is $O(\frac{N}{B})$, with details left in the full version of the paper. Summing this expected cost over all rounds of the algorithm, we obtain the following.

Theorem 1. *The constrained Delaunay triangulation of a set of N points $\mathbb{R}^2$ 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

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 0.1 million 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.
2. Construct $\mathrm{CDT}(P_1, S)$ in memory using the `triangle` package.
3. For each point $p \in P$ in turn we determine the edges that p is in conflict with, generating a pair (e, p) for each such edge $e \in \mathrm{CDT}(P_1, S)$. We then sort all these pairs to construct the conflict list $P_{|e}$ for each edge e. If any conflict list is larger than M, we restart the algorithm and take a new sample.
4. For each edge $e \in \mathrm{CDT}(P_1, S)$ in turn we load its conflict list $P_{|e}$ into memory and construct $\mathrm{CDT}(P_{|e}, I_e)$ using the `triangle` package. Then we report all the triangles whose circumcenters are inside $\tau(e)$.

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

Since Step 3 is the only nontrivial step in the algorithm, we describe it in a little more detail. We first scan through the input points, and find conflicting edges with $\mathrm{CDT}(P_1, S)$ 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; $\triangle uvw$ is in conflict with p if $p \in D(u, v, w; S)$. 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.

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", computing the conflict list for each edge group, and solving the subproblem for each edge group. Nevertheless, some technical subtleties need to be taken care of when implementing this idea, which we explain in details in the full version.

Experimental setup and datasets. We implemented our simplified constrained Delaunay triangulation algorithm in C++ using TPIE [4]. 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.

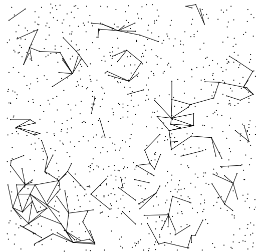

Fig. 4. Sample datasets of 1000 points from uniform distribution with segments.

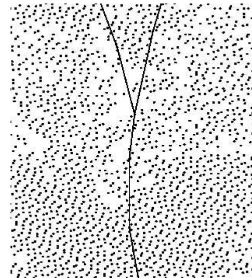

Fig. 5. LIDAR data.

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: uniform, normal, Kuzmin, and line singularity. See [6] for a definition of these distributions. Due to lack of space, we only report results on the uniform distribution in this abstract. Complete experiment results can be found in the full version of the paper.

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 other segment in S and the length of s is smaller than some threshold δ. In our experiments we fixed $\alpha = 0.2$. An example of the segments generated this way are shown in Figure 4.

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 [17]. 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. A portion of the LIDAR data is shown in Figure 5.

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.

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 [16]. Since it is known that pre-sorting the points along some space-filling curve improves the performance of D&C and especially INC greatly with 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 uniform distribution with datasets of sizes varying from 10^6 to 10^7 are shown in Figure 6. Note that the 128MB main memory can only hold the data structure for triangulating roughly 1 million points. The results from the other distributions are similar. In all experiments, INC performs best. Its running time is almost linear in the data size because its data structure is visited in a highly local manner. The running time of our EM algorithm is 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, as soon as the dataset size grows larger, its performance quickly degenerates.

Constrained Delaunay triangulation experiments. Next we compared our EM algorithm with the algorithm (INC) implemented in `triangle` [16], which first constructs a Delaunay triangulation on the input points P (using the INC algorithm discussed above), and then inserts all the segments in S one by one. As before we pre-sorted the points by Hilbert values; we sorted the segments by the Hilbert value of one of their endpoints.

The running times of our first set of experiments on the uniform distribution are shown in Figure 7. 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

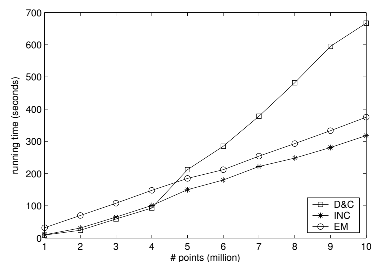

Fig. 6. Delaunay triangulation results on uniform distribution.

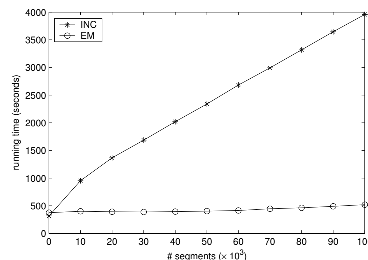

Fig. 7. Constrained Delaunay triangulation results on uniform distribution.

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 a small constrained Delaunay triangulation in memory ($\mathrm{CDT}(P_1, S)$), the INC algorithm incrementally inserts segments in a much larger (and larger than main memory) constrained Delaunay triangulation containing all the points.

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 use about 5MB per 10,000 points, and sample $3K$ points; the system daemons use at least 30MB). 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 degenerate into INC.

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 8. The results show that the running times of both algorithms are relatively unaffected by segment length. 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.

The running times of our experiments with the LIDAR datasets are shown in Figure 9. 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

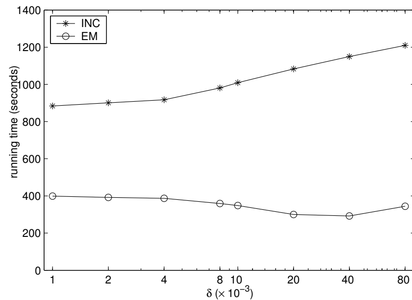

Fig. 8. Constrained Delaunay triangulation results with varying segment lengths.

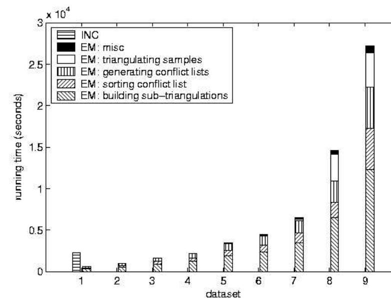

Fig. 9. Constrained Delaunay triangulation results on real datasets.

run INC except on the smallest dataset. In Figure 9 we 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 CDT(P_1, S) starts to be significant. However, since P_1 is still much smaller than the entire dataset, our algorithm is still much faster than building the entire constrained Delaunay triangulation directly.

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. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
10. 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.
11. 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.
12. L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
13. P. Kumar and E. A. Ramos. I/O-efficient construction of voronoi diagrams. Technical report, 2002.
14. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
15. R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. *Computer Science Division*, ??, June 1989. UC Berkeley.
16. 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.
17. *TIGER/Line™ Files, 1997 Technical Documentation*. Washington, DC, September 1998. http://www.census.gov/geo/tiger/TIGER97D.pdf.