THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRA1/06

Efficient Constrained Delaunay Triangulation for Large Spatial Databases

Xinyu WU, David HSU and Anthony K. H. TUNG

January 2006

Technical Report

Foreword

Efficient Constrained Delaunay Triangulation for Large Spatial Databases

Xinyu Wu David Hsu Anthony K.H. Tung

National University of Singapore
{wuxy, dyhsu, atung}@comp.nus.edu.sg

Abstract

Delaunay Triangulation (DT) and its extension *Constrained Delaunay Triangulation* (CDT) are spatial data structures that have wide applications in spatial data processing. Our recent survey shows, however, that there is a surprising lack of algorithms for computing DT/CDT for large spatial databases. In view of this, we propose an efficient algorithm based on the divide and conquer paradigm. It computes DT/CDT on in-memory partitions before merging them into the final result. This is made possible by discovering mathematical property that precisely characterizes the set of triangles that are involved in the merging step. Our extensive experiments show that the new algorithm outperforms another provably good disk-based algorithm by roughly an order of magnitude when computing DT. For CDT, which has no known disk-based algorithm, we show that our algorithm scales up well for large databases with size in the range of gigabytes.

1 Introduction

Delaunay triangulation (DT) is a spatial data structure that has been studied extensively in many areas of computer science. A triangulation of a planar point set S is a partition of a region of the plane into non-overlapping triangles with vertices all in S. A Delaunay triangulation has the additional nice property that it tends to avoid long, skinny triangles, which lead to bad performance in applications (Figure 1). In this work, we develop an efficient algorithm that computes DT and its extension, *constrained Delaunay triangulation*, for data sets that are too large to fit in the memory.

DT is an important tool for spatial data processing:

Spatial data interpolation. In geographical information systems (GIS), a common task is terrain modeling from measurements of the terrain height at sampled points. One way of constructing a terrain surface is to first compute the DT of the sample points and then interpolate the data based on the triangulation [17, 18, 23]. Figure 2 shows a terrain surface constructed this way. The same interpolation method easily extends to other spatial data, such as aerial photographs [9] and readings from sensor networks [15].

Mesh generation. Many physical phenomena in science and engineering are modelled by partial differential equations, *e.g.*, fluid flow or wave propagation. These equations are usually too complex to have closed form solutions, and need numerical methods such as finite element analysis to approximate the solution on a mesh. DT is a preferred method for mesh generation [1]. As an example, in the Quake project [5], finite element analysis is applied to billions of points to simulate the shock wave of earthquakes, and DT is used to generate the meshes needed for simulation.

Proximity search. Voronoi diagram is an efficient data structure for nearest neighbor search. Since the DT of a point set is in fact the dual graph of the corresponding Voronoi diagram [7, 23] and is easier to compute, it is common to compute the DT first and obtain the Voronoi diagram by taking the dual.

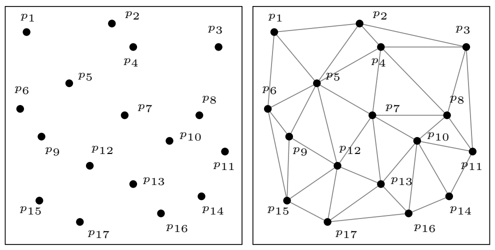


Figure 1: A set of points (left) and its Delaunay triangulation (right).

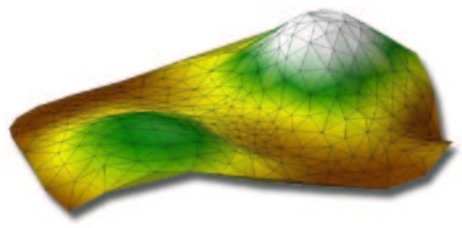


Figure 2: A terrain surface constructed using Delaunay-based spatial interpolation.

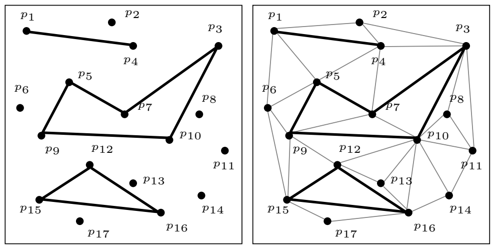


Figure 3: Input data points and constraint edges (left) and the corresponding Delaunay triangulation (right).

The application of DT extends further if we allow in the input data constraint edges that must be preserved in the final triangulation. Intuitively, this extension, called the constrained Delaunay triangulation (CDT), is as close as one can get to the DT, given the constraint edges (Figure 3). Constraint edges occur naturally in many applications. We give three representative examples. In spatial data interpolation, allowing constraint edges helps to incorporate domain knowledge into the triangulation. For example, if the data points represent locations where pedestrian traffic flow is measured, the constraint line segments and polygons may represent obstacles to the pedestrians. It therefore makes sense to interpolate "around" the obstacles rather than through them. Likewise, in mesh generation for finite element analysis, constraint edges mark the boundaries between different mediums, *e.g.*, regions where water cannot flow through. Finally, in spatial searching and data mining, it is often useful to take into consideration obstacles modelled as constraints [26, 30, 31]. One important operation in these applications is to compute, between two given points, the shortest path that does not cut through any obstacles. To do this, one may use the visibility graph method [28], which takes $O(n^2)$ time and space in the worst case, where n is the number of input data points. In comparison, computing the DT/CDT takes only $O(n \lg n)$ time and $O(n)$ space. The DT/CDT can then be used to find an approximate shortest path with guaranteed approximation bounds [21, 20].

The importance of DT and CDT to applications has led to intensive research. Many efficient DT algorithms have been proposed, and they follow three main approaches: divide-and-conquer, incremental construction, and plane sweep [7]. Of the three approaches, the first two are also applicable to CDT, as well. Unfortunately, although many applications of DT and CDT involve massive data sets, most existing algorithms assume that the input data is small enough to fit entirely in the memory, and their performance degrades drastically when this assumption breaks down.

If the input data do not fit into the memory, incremental construction is unlikely to be efficient, because a newly-inserted point may affect the entire triangulation and results in many I/O operations. The only remaining option is then divide-and-conquer. The basic idea is to divide the data into blocks, triangulate the data in each block separately, and then merge the triangulations in all the blocks by "stitching" them together along the block boundaries. The key challenge here is to devise a merging method that is efficient in both computational time and I/O performance, when the triangulation can not fit in the memory completely.

Motivated by the observation that there is limited work on practical algorithms for disk-based DT/CDT despite their importance, our work focuses on scalable computation of CDT, with DT as a special case. We believe this work makes the following contributions:

- We present an efficient disk-based algorithm for CDT using the divide-and-conquer approach (Section 4). We give a precise characterization of the set of triangles involved in merging, leading to an efficient method for merging triangulations in separate blocks. Our algorithm makes use of an in-memory algorithm for triangulation within a block, but the merging method is independent of the specific in-memory algorithm used. In this sense, our approach can convert any in-memory DT/CDT algorithm into a disk-based one.

- We describe in details the implementation of our algorithm (Section 5). One interesting aspect of our implementation is that after computing the triangulation in each block and identifying the triangles involved in merging, we can merge the triangulations using only sorting and standard set operations, and maintain no explicit topological information. These operations are easily implementable in a relational database, enabling our algorithm to be integrated with various spatial data processing techniques that are now commonly found in the industry [3, 4, 2]. Furthermore, since the algorithm uses no floating-point calculation during merging, it is more robust.

- We have performed extensive experiments to test the scalability of our algorithm for both DT and CDT (Section 6). For DT, we compare our algorithm with an existing disk-based algorithm that is provably good, and show that our algorithm is faster by roughly an order of magnitude. For CDT, to our knowledge, there is no implemented disk-based algorithm. We compare the performance of our algorithm with an award-winning in-memory algorithm [24] and show that the performance of our algorithm degrades much more gently when the data size increases.

2 Previous Work

Intensive research has led to several in-memory DT algorithms that are asymptotically optimal. They fall in three main categories: divide-and-conquer, randomized incremental construction, and plane sweep. See [7] for a good survey. Experiments show that of the three approaches, divide-and-conquer is most efficient and robust in practice [24, 25]. Similarly, for CDT, a divide-and-conquer algorithm achieves the asymptotically optimal running time [13], but the algorithm is quite complex and difficult to implement. In practice, incremental construction is the most popular approach for CDT. It proceeds by first constructing the DT and then conforming the triangulation to the constraint edges by inserting them one at a time [6, 16, 27].

All the algorithms mentioned in the above paragraph assume that the input data set is small enough to fit in the memory. When the data set becomes too large, they rely entirely on virtual memory management by the operating system and perform poorly due to a large number of unnecessary I/O operations. Research on efficient disk-based algorithm for DT and CDT has been limited. While there are DT algorithms that are optimal in I/O performance under certain theoretical models [14, 19], we are not aware of implementation of these algorithms and experimental results on their performance in practice. In fact, we have only found one experimental study of a disk-based DT algorithm [22] in the literature. Furthermore, it appears, surprisingly, that there is no earlier work on efficient disk-based CDT algorithms.

Another related line of research is parallel DT/CDT algorithms (see, *e.g.*, [11, 12]), but their emphasis is on parallel efficiency and not I/O efficiency. The divide-and-conquer approach that we use is related to that used in the work of Chen *et al.* on parallel DT computation [12], but our merging method is more efficient, and we handle CDT as well as DT.

3 Preliminaries

Let S be a set of points in the plane. The convex hull of S is the smallest convex set that contains S, and a triangulation of S is a partition of the convex hull into non-overlapping triangles whose vertices are in S (Figure 1). The boundary of a triangulation then clearly coincides with the boundary of the convex hull. In general, a point set admits different triangulations, and we can impose additional conditions to obtain desirable properties and make the triangulation unique. The Delaunay triangulation of S is a triangulation with the additional *empty-circle* property:

Definition The *Delaunay triangulation* of a point set S, denoted by $DT(S)$, is a triangulation such that for every triangle t in the triangulation, the circumcircle $R(t)$ of t contains no points in S in its interior.

One can show that DT tends to avoid long, skinny triangles, resulting in many benefits in practice [10].

DT can be generalized, if the input data contains not only points, but also line segments acting as constraints. A planar straight line graph (PSLG) is a set S of points and a set K of non-intersecting line segments with endpoints in S. The points can be used to model service sites, and the line segments can be linked together to model polygonal obstacles of arbitrary shapes. Given a PSLG (S, K), we say two points p and q in S are *visible* to each other if the straight-line segment between p and q does not intersect with any segment in K. Using this notion of visibility, the constrained Delaunay triangulation of (S, K) is defined as follows:

Definition Given a PSLG (S, K), a triangulation T is a *constrained Delaunay triangulation* of (S, K), denoted by

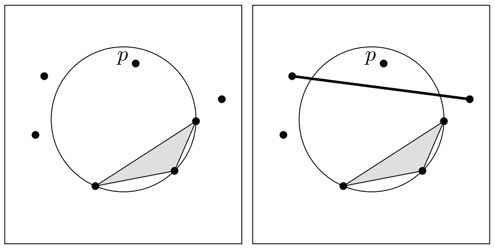


Figure 4: For DT, the shaded triangle is invalid because its circumcircle contains the point p (left). For CDT, the same shaded triangle is valid because p is not visible to the vertices of the triangle (right).

$CDT(S, K)$, if

- every constraint segment $k \in K$ is an edge of some triangle in T, and
- for each triangle $t \in T$, there is no point $p \in S$ such that p is both in the interior of the circumcircle of t and visible to all three vertices of t.

Note that if there is no constraint segment passing through the circumcircle of t, then the second condition above is equivalent to the the empty-circle property for DT, and so it is a natural extension of the empty-circle property when constraint segments are present (Figure 4).

4 Disk-Based Constrained Delaunay Triangulation

4.1 Overview

The input to our algorithm is a PSLG (S, K), which consists of a set S of points in the plane and a set K of non-intersecting constraint segments. We assume that (S, K) is so large that it cannot fit into the main memory, and our problem is to compute $CDT(S, K)$.

Our proposed algorithm initially ignores the constraint segments K and computes $DT(S)$. Then it adds the constraint segments back and updates the triangulation to construct $CDT(S, K)$. To reduce the memory requirement, our algorithm uses a divide-and-conquer approach. Specifically, it goes through four main steps:

1. **Divide:** Partition the input PSLG (S, K) into small blocks so that each fits in the memory;
2. **Conquer:** Use an in-memory DT algorithm to compute the DT for each block;
3. **Merge:** Stitch together Delaunay triangulations from all the blocks and build the complete $DT(S)$;
4. **Conform:** Insert constraint segments block by block and update the triangulation to build $CDT(S, K)$.

Both the merging and conforming steps potentially require updating the entire triangulation, which leads to high I/O cost, because the triangulation is too large to be stored in the memory. Our goal is therefore to design an algorithm

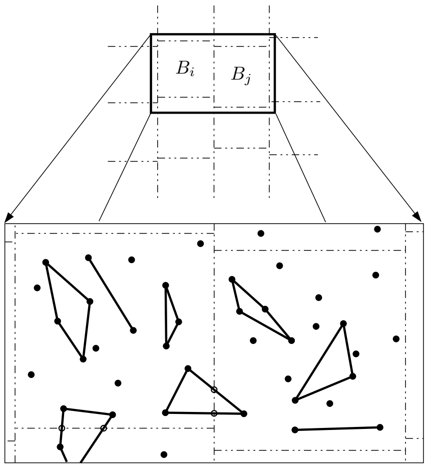


Figure 5: The dividing step: partition the input PSLG into blocks of roughly equal size so that each fits into the memory. In the zoomed-in picture, small circles indicate Steiner points created at the intersections of input segments and block boundaries.

that minimizes the number of unnecessary I/O operations during merging and conforming.

We now give details on the four steps. Section 4.2 describes the first three steps, which compute $DT(S)$. Section 4.3 describes the last step, which enforces the constraints.

4.2 Computing the Delaunay Triangulation

In the dividing step, we partition the rectangular region containing (S, K) into rectangular blocks $B_i, i = 1, 2, \ldots$ so that the number of points and segments in each block is small enough for the data to fit into the memory (Figure 5). As a convention, each block contains the right and top edges, but not the left and bottom edges. We assume that every segment is completely contained within a block. If a segment goes through multiple blocks, we can split it by adding additional points at the intersections of the segments and block boundaries. These additional points are called *Steiner* points by the convention in the literature. See Section 5 for details and alternatives.

The conquering step is straightforward. Let $S_i \subseteq S$ be the subset of points that lie in B_i. We simply invoke an in-memory DT algorithm to construct $DT(S_i)$ for each block. Suppose that t is a triangle in $DT(S_i)$ and $R(t)$ is its circumcircle. If $R(t)$ lies entirely within B_i, then no point in another block can enter $R(t)$ and fail the empty-circle test of t (Figure 6). Thus t remains valid after merging. If $R(t)$ crosses the boundary of B_i, a point in another block may fall inside $R(t)$ and cause t to be invalidated during merging. This fact is summarized in the lemma below:

Lemma 4.1 *Let $S_i \subseteq S$ be the subset of points in block B_i. For a triangle $t \in DT(S_i)$, if the circumcircle of t lies entirely within B_i, t must remain valid after merging; otherwise, t may be invalidated.*

For convenience, we make the following definition:

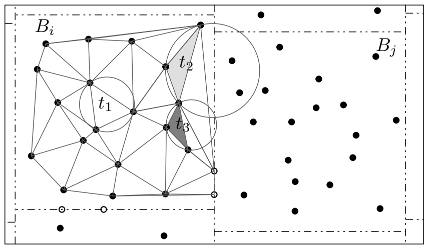


Figure 6: The conquering step: compute DT in each block. The triangle t_1 is safe, and both t_2 and t_3 are unsafe.

Definition Let $S_i \subseteq S$ be the subset of points in block B_i. A triangle $t \in DT(S_i)$ is *safe* if its circumcircle lies within B_i; otherwise, t is *unsafe*.

Distinguishing between safe and unsafe triangles is valuable, because safe triangles are unaffected by merging and can be reported directly in the conquering step. Only the unsafe triangles need to be loaded into the memory in the merging step, thus significantly reducing the memory requirement.

We now move on to the more difficult step, merging. If we merge $DT(S_i)$ with $DT(S_j)$ in an adjacent block. Some unsafe triangles in $DT(S_i)$ may be invalidated, because the points in S_j fail the empty-circle tests for those triangles. In addition, some new triangles must be created to stitch together $DT(S_i)$ and $DT(S_j)$.

First let us consider the triangles that are created during merging. We start with some terminology.

Definition A triangle whose vertices all lie in the same block is called a *non-crossing* triangle; otherwise, it is called a *crossing* triangle.

Suppose that t is a non-crossing triangle in $DT(S)$, the final DT of S. Then t must satisfy the empty-circle test, meaning that no point in S lies within the circumcircle of t. Assuming that t lies within block B_i, we know by the definition of Delaunay triangulation that t is also a triangle in $DT(S_i)$, because $S_i \subseteq S$. So we have the next lemma:

Lemma 4.2 *Let $S_i \subseteq S$ be the subset of points in block B_i. If $t \in DT(S)$ is a non-crossing triangle that lies inside B_i, then $t \in DT(S_i)$. Hence merging $DT(S_1), DT(S_2), \ldots$ cannot create any new non-crossing triangle.*

Lemma 4.2 implies that we only need to focus on crossing triangles. Denote by S' the set of point in S such that every point in S' is either a vertex of an unsafe triangle or on the boundary of $DT(S_i)$, for some block B_i. The set S' is called the *seam*. According to the lemma below, we can obtain all the crossing triangles by computing $DT(S')$ (Figure 7).

Lemma 4.3 *A triangle t is a crossing triangle in $DT(S)$ if and only if t is also a crossing triangle in $DT(S')$.*

Intuitively this lemma holds, because we can obtain $DT(S')$ from $DT(S)$ by deleting all the non-seam points,

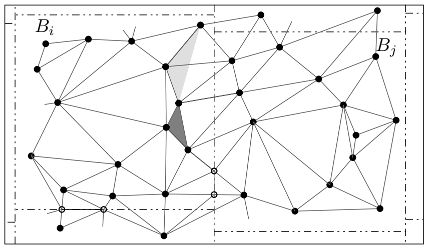


Figure 7: The merging step: compute the DT of the seam. After merging B_i and B_j, t_2 becomes invalid and is deleted, but t_3 remains valid.

$S \backslash S'$, and re-triangulating. All crossing triangles would remain unchanged in this process, because their vertices all lie in S' and are not deleted. See Appendix A for a formal proof.

Next let us identify those unsafe triangles in $DT(S_i)$ that are invalidated during merging. One possibility is to test whether an unsafe triangle t overlaps some crossing triangle in $DT(S')$. However, the overlapping test is difficult because it is unclear which crossing triangles t may overlap. Checking against all crossing triangles is clearly inefficient. Furthermore the overlapping test requires numerical calculation which increases computational cost and decreases robustness. Fortunately the following lemma helps to solve the problem much more easily and efficiently.

Lemma 4.4 *$DT(S')$ contains all the valid unsafe triangles and no invalid unsafe triangles.*

This lemma can be proven in a similar way as Lemma 4.2 (see Appendix A). Now let U denote the set of unsafe triangles for all the blocks. We can sort the triangles in U and $DT(S')$ in lexigraphical order according to the indices of their vertices and perform a set intersection of U and $DT(S')$. The result is exactly the set of valid unsafe triangles that need to be reported.

To summarize, in the dividing step, we partition the input data into blocks $B_i, i = 1, 2, \ldots$. In the conquering step, we compute $DT(S_i)$ for each block B_i. We report all the safe triangles as valid triangles for $DT(S)$ and store the set U of unsafe triangles. In the merging step, we need only U and the seam S'. This is an important reason for the memory space efficiency of our algorithm, as typically U and S' are much smaller than the original input S. After computing $DT(S')$, we report all the crossing triangles in $DT(S')$ as valid triangles in $DT(S)$. We then compute the set intersection of U and $DT(S')$ and report the resulting triangles. The theorem below establishes the correctness of these steps.

Theorem 4.5 *The combination of dividing, conquering, and merging steps computes $DT(S)$ correctly.*

PROOF: $DT(S)$ consists of two types of triangles: non-crossing triangles, each of which is contained entirely within some block B_i, and crossing triangles. According to Lemma 4.3, all the crossing triangles in $DT(S)$ are

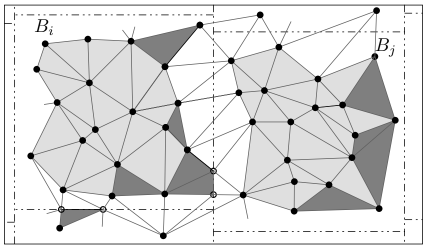


Figure 8: The DT of input data points. There are three types of triangles: triangles in light shade are the safe triangles obtained in the conquering step; triangles in dark shade are the valid unsafe triangles that are preserved during the merging step; the rest are crossing triangles.

obtained in the merging step by computing $DT(S')$. Non-crossing triangles are further divided into safe and unsafe triangles. By Lemma 4.1 and 4.2, all the safe triangles in $DT(S)$ are reported in the conquering step. From Lemma 4.4, we can infer that all the unsafe triangles in $DT(S)$ are computed correctly by taking the set intersection of U and $DT(S')$. Therefore all the triangles in $DT(S)$ are captured correctly. ■

4.3 Inserting Constraint Segments

Now we add the constraints segments back and compute $CDT(S, K)$. To do this efficiently, we need the following result [27]:

Lemma 4.6 *Let $CDT(S, K)$ be the CDT of a point set S and a constraint segment set K, and let $\overline{pq}$ be a new segment such that the endpoints of the segment, p and q, are in S and $\overline{pq}$ does not intersect with any segment in K. To compute $CDT(S, K \bigcup \{\overline{pq}\})$, we only need to re-triangulate the region covered by the triangles overlapping $\overline{pq}$ (Figure 4.3).*

This lemma says that adding an new constraint segment $\overline{pq}$ into an existing CDT only affects those triangles overlapping $\overline{pq}$. This greatly restricts the set of triangles that need to be considered and localizes the updates. Using this result, we can add the segments in blocks and process each block B_i almost independently. Let $K_i \subseteq K$ be the subset of segments in block B_i. Conceptually we compute a series of triangulations $T_0, T_1, T_2, \ldots$, where T_0 is simply $DT(S)$ and T_i for $i \geq 1$ is an updated triangulation after K_i is inserted into T_{i-1}.

We now explain how to process B_i and compute T_i. First we load all triangles in T_{i-1} that lie inside or cross the boundary of B_i. This set of triangles forms a triangulation Q. We insert the segments K_i into Q and compute the CDT using an in-memory CDT algorithm. The result is a new triangulation Q'. By Lemma 4.6, loading Q is sufficient, because all segments in K_i lie B_i according to our assumption and cannot affect any triangles in other blocks. Furthermore, the new triangles in Q' do not affect any triangles in other blocks, either. This entire process can thus be completed in the memory, and in the end, we report the

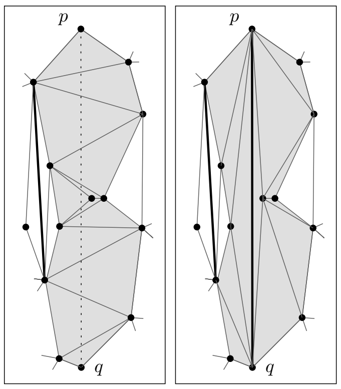


Figure 9: Inserting constraint segment $\overline{pq}$ only requires re-triangulating grey region consisting of triangles intersecting with $\overline{pq}$.

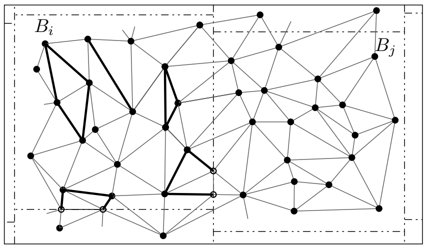


Figure 10: The conforming step: insert constraint segments K_i from B_i and update the triangulation.

triangles in Q' and obtain the updated triangulation T_i. Of course, since the intermediate triangulation T_i resides on the disk, we must be careful to minimize the I/O operations when loading triangles from T_{i-1} and reporting triangles in Q'. These data organization issues are discussed in the next section. Figure 10 illustrates the result of conforming the triangulation to K_i.

The theorem below shows the correctness of our CDT algorithm.

Theorem 4.7 *Our algorithm computes $CDT(S, K)$ correctly.*

PROOF: We use induction to show that T_i is a correct CDT for S and $K = \bigcup_i K_i$. By Theorem 4.5, $T_0 = DT(S)$ is correct. Assume that T_{j-1} is a correct CDT of $(S, \bigcup_{i=1}^{j-1} K_i)$. To process block B_i, we insert the constraint segments K_i to T_{j-1}. Lemma 4.6 ensures that re-triangulating captures all the changes that occur as a result of inserting K_j and T_j is the CDT of $(S, \bigcup_{i=1}^{j} K_i$. It follows that the algorithm computes correctly $CDT(S, K)$ when all $K_i, i = 1, 2, \ldots$ are inserted. ■

Figure 11 shows the final $CDT(S, K)$ after the insertion of all the segments in K.

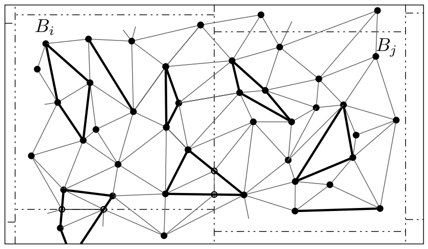


Figure 11: The final CDT of the input PSLG.

Table 1: List of data tables.

Table	Description	Fields
S	Set of data points and Steiner points each indexed by a primary key, i	i, x, y
S_i	Set of data and Steiner points in B_i	i, x, y
S'	Seam	i, x, y
K	Segments set, i_1 and i_2 are primary key in S	i_1, i_2
K_i	Set of segments in B_i	i_1,i_2
U	Set of unsafe triangles represented by the primary key of their vertices	i_1, i_2, i_3

5 Implementation

This section describes the implementation of our algorithm. In our implementation, a point in the plane is represented by its x- and y-coordinates, a segment by two indices to its endpoints, and a triangle by three indices to its three vertices. Our implementation consists of insertion and deletion to multiple tables and they are listed in Table 1 for ease of discussion. We will give more details on these tables as we go along. Our discussion here will come in two parts: (1) divide and conquer (2) merge and conform.

5.1 Divide and Conquer

The input to our disk-based CDT algorithm consists of a set of points S, and a set of constraint segments K. We first describe how the set of points and constraint segments are divided into partitions.

Let N be the number of input points and M be the block size, which is governed by the physical memory space. For simplicity, let us assume $N = r^2 M$ for some integer r. We divide S into r^2 rectangular blocks. First we sort all the points in S according to their x-coordinate values and divide the the point sets vertically into r disjoint columns, each containing rM points. Then the points in each column are re-sorted according to their y-coordinate values and cut horizontally into blocks of size M. If such integer r does not exist, we can do a rounding off to make sure each block does not contain more than M points.

This way of partitioning data with alternating vertical and horizontal cuts is chosen so that the shape of each block is close to square for a uniform distribution of points in a square area. This is preferable because generally the computational cost (time and I/O) of the merging step is closely related to the sum of the circumference lengths of all the blocks, which is minimized when all the blocks are square. As our experiments show, the alternating cut also works well for non-uniform data distributions.

In the previous section, we stated one assumption on the segment set K is that none of the segments overlaps different blocks. This assumption is of course not true in general. Here we give two ways to handle those overlapping segments. One way is to delay the insertion of those overlapping constraint segments and compute the CDT first with the segments that completely lie in some single block, then insert the overlapping constraint segments one by one into the triangulation. Alternatively one can break all the overlapping constraint segments into pieces by creating Steiner points at the intersections of the constraint segments and boundaries of the blocks. The first approach computes the true CDT of the input PSLG. However as we know, each insertion of constraint segment involves locating the segment in the triangulation which is computationally expensive time when the whole CDT does not fit into the memory. The second approach computes the CDT of the input point sets and the Steiner points. We adopted the second approach in our implementation because it enables us to process all the segments in batches in the conforming step, which is much more I/O efficient. A small number of Steiner points are often allowed and sometimes necessary in most applications. For conciseness, we hereby use S for the union of the set of input points and the set of Steiner points, and S_i for the set of points within block B_i. By the convention we adopted in 4.2, S_i includes Steiner points on the right and top edges of B_i.

Having sorted all the points in S, we assign a unique primary key i to each point based on that order. This is important for us to map most of our processing into database operation instead of geometrical computation. Correspondingly, each segment in K will then be represented by the primary keys of those points marking its ends. From here on, we can see S and K as tables. Similarly, we add in the corresponding primary key for each point into S_i for each block.

The conquering step is quite simple. By our way of partitioning the data, input points from the same block B_i are stored sequentially on disk. The conquering step first load the set S_i and compute $DT(S_i)$. Then as described in the previous section, we need to classify the triangles as safe or unsafe in $DT(S_i)$. The status of the triangle is decided by checking whether its circumcircle intersects the boundary of B_i. If it does intersect, the triangle is considered safe, else the triangle will be considered unsafe. All safe triangles are directly reported to the final triangulation $DT(S)$; all the unsafe triangles are added into the list U, and all the vertices which are either incident to some unsafe triangle or lying on the boundary of $DT(S_i)$ for some block B_i are added into the *seam*, S'. Some points can be reported to S' multiple times, so S' so duplicate points should be filtered off from S'.

Again U and S' can be seen as tables with each triangle in U represented by the primary key of its vertices while points in S' are kept in sorted order of the primary key together with the x, y coordinates.

Algorithm 1 Conquer

```
Input:
    boundaries /* the boundaries for all blocks */
    $S_i$ /* the partitioned point set for each block $B_i$*/
Output:
    $DT(S)$ /* the final DT of the point set stored on disk */
    $S'$ /* the set of points that will be needed in merging step */
    $U$ /* the set of unsafe triangles */
 1: $S' = \emptyset$
 2: $U = \emptyset$
 3: for all blocks $B_i$ do
 4:     compute $DT(S_i)$
 5:     for all $t \in DT(S_i)$ do
 6:         if circumcircle $R(t)$ crosses the boundary of $B_i$ then
 7:             add the vertices of $t$ into $S'$
 8:             add $t$ into $U$
 9:         else
10:             report $t$ to the final $DT(S)$
11:         end if
12:     end for
13: end for
14: remove duplicate points in $S'$
```

Algorithm 2 Merge

```
Input:
    $S'$ /* the set of points needed in merging */
    $U$ /* the set of unsafe triangles */
Output:
    $DT(S)$ /* the final DT of the point set stored on disk */
 1: compute $DT(S')$
 2: for all $t \in DT(S')$ do
 3:     if $t$ is a crossing triangle then
 4:         report $t$ to the final $DT(S)$
 5:     end if
 6: end for
 7: for all $t \in DT(S') \cap U$ do
 8:     report $t$ to the final $DT(S)$
 9: end for
```

5.2 Merge and Conform

The merging step of our algorithm computes all the crossing triangles and valid unsafe triangles in $DT(S)$. By Lemma 4.3, we can find all the crossing triangles from DT(S').

Lemma 4.4 states that the set of valid unsafe triangles are also stored in $DT(S')$. Thus we only need to compute $DT(S')$ to find these two sets of triangles, which saves a lot of memory space as S' is usually significantly smaller than S. Note that since S' might not fitted into the main memory, we might have to recursively perform another disk-based DT of S'. We will give more details on this in the discussion section later.

We scan through $DT(S')$ to select the crossing triangles. A triangle is crossing if it overlaps different blocks. The valid unsafe triangles can be expressed as the set intersection $U \cap DT(S')$, U being the set of unsafe triangles obtained in the conquering step. This can be easily computed since both U and $dt(S')$ are represented by the primary key of their vertices [1].

We next look at the conform step. Section 4.3 briefly describes how to process the segments block by block and progressively update the triangulation to obtain

[1]For easy comparison, we stored the vertices of each triangle in a anti-clockwise order starting with the vertices that have the smallest x-coordinate

$CDT(S,K)$. Let $K_i \subseteq K$ be the subset of segments in block B_i. Lemma 4.6 shows that inserting all segments in K_i only affects the triangulation Q formed by triangles lie completely in or cross the boundary of B_i. The result of conforming Q to K_i is Q'. Once Q and K_i are loaded, we can simply call an in-memory CDT subroutine to compute Q'. Here we focus on how to load Q and report Q'. The loading and reporting must be done carefully. Otherwise imagine that we simply report all the triangles in Q' sequentially to the disk. Some of the triangles in Q' overlap other blocks. It will be very difficult to load these triangles when we process the blocks they overlap.

We can classify the triangles in Q and Q' into two groups: triangles totally contained in B_i, and those overlapping other blocks. The triangles totally contained in B_i can be sequentially loaded and reported straight away, as they cannot be affected by segments in other blocks by Lemma 4.6. The triangles overlapping other blocks are managed using a cache mechanism.

After $DT(S)$ is constructed in the conquer step, we duplicate the crossing triangles for each block it overlaps so that for any block B_i, we can sequentially load all crossing triangles in $DT(S)$ that overlap B_i. Thus Step 5 in the conform function can be done with minimal I/O time.

To capture the changes due to the insertion of segments, we maintain two sets C_1 and C_2 of triangles in main memory as caches. C_1 stores newly created triangles overlapping unprocessed blocks, while C_2 stores dirty triangles overlapping unprocessed blocks. Denote the set of triangles in Q that overlap other blocks by A. Both A and A' are initialized to be empty. We first read into A all crossing triangles in $DT(S)$ that overlap B_i. Then we add all triangles overlapping B_i from C_1 into A, and delete all dirty triangles found in C_2 from A. A combined with all triangles in $DT(S)$ that lie entirely in B_i clearly gives us Q.

After we conform Q to K_i to obtain Q', we can report all the triangles that do not overlap any other block to the final $CDT(S,K)$. All the remaining triangles overlap other blocks. Denote them by A'. We append C_1 with the set difference $A' \backslash A$ as all triangles in $A' \backslash A$ are newly created ones. Similarly, we append C_2 with $A \backslash A'$. It is safe to immediately report all triangles in C_1 that do not overlap any unprocessed block, and delete all such triangles from C_2 as they are no longer in use. Alternatively, one can choose lazy evaluation depending on the caches' capacity.

Algorithm 3 Conform

```
Input:
      $DT(S)$ /* the DT of $S$ stored on disk*/
      $K_i$ /* the segments contained in each block */
Output:
      $CDT(S,K)$ /* the final CDT stored on disk */
 1: $C_1 = \emptyset$
 2: $C_2 = \emptyset$
 3: for all blocks $B_i$ do
 4:     $A = \emptyset$
 5:     load interior triangles in $B_i$ from $DT(S)$
 6:     load crossing triangles overlapping $B_i$ from $DT(S)$ into $A$
 7:     add all triangles in $C_1$ overlapping $B_i$ into $A$
 8:     delete all triangles found in $C_2$ from $A$
 9:     combine $A$ with interior triangles to form $Q$
10:     conform $Q$ to $K_i$ to get $Q'$
11:     report all triangles in $Q'$ that lie within $B_i$ to the final
        $CDT(S,K)$
12:     $A'$ = the set of triangles remained in $Q'$
13:     $C_1 = C_1 \cup (A' \backslash A)$
14:     $C_2 = C_2 \cup (A \backslash A')$
15:     report all triangles in $C_1$ that do not overlap any unprocessed
        block to the final $CDT(S,K)$
16:     delete all triangles in $C_2$ that do not overlap any unprocessed
        block
17: end for
```

6 Experimental Evaluation

Our program is implemented in C++. For in-memory DT/CDT, it uses TRIANGLE [24], which is awarded the J. H. Wilkinson Prize for Numerical Software for its efficiency and robustness.

We tested our implementation extensively on both DT and CDT. For DT, we compare our algorithm with both TRIANGLE and a provably good disk-based algorithm, which, as we have mentioned in Section 1, appears to be only one in the literature with implementation and experimental studies. For CDT, since there is no implemented disk-based algorithm, we compare our algorithm with TRIANGLE and test for scalability on large data sets [2].

Our experimental platform is an Intel Pentium 4 PC, which has one 1.4GHz CPU and 512MB memory, and runs RedHat Linux Fedora I. The code is compiled with option `-O`. We use the Linux `time` command to measure the running time and the `vmstat` command to measure the I/O operations. One drawback of `vmstat` is that it only monitors the overall I/O activity of the whole system. So we kept all other system activities at the minimum when performing the experiments to maximize measurement accuracy.

6.1 Delaunay Triangulation

Data Distribution

We ran our program on point sets with three different distributions: Kuzmin, Line Singularity and Uniform (Figure 12). These are standard distributions for evaluating the performance of DT algorithms [11, 12].

Kuzmin distribution. The Kuzmin distribution models the distribution of star clusters in flat galaxy formations. It is a radically symmetric distribution with the distribution function

$$M(r) = 1 - \frac{1}{\sqrt{1+r^2}}, \tag{1}$$

where r is the distance to the center. This distribution converges to the center faster than the normal distribution.

Line Singularity distribution. Line Singularity is an example of distributions that converge to a line. It has a parameter b, which is set to 0.01 in our experiments.

[2]We use synthetic datasets in this paper to ensure that they are sufficiently large and to control the distribution of the data. Performance on real-life geographical datasets is available in [29], but these datasets are not sufficiently large for us to make any conclusion. We exclude them due to lack of space

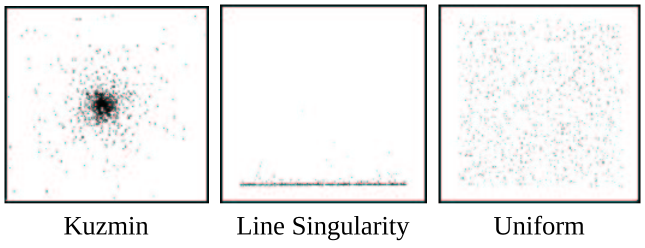


Figure 12: Data distributions for testing DT.

To take a sample (x, y) from the Line Singularity distribution, we pick a uniform random sample (u, v) and apply the formula

$$(x, y) = (\frac{b}{u - bu + b}, v). \tag{2}$$

Uniform distribution. The uniform distribution consists of points picked uniformly at random from the unit square.

Both Kuzmin and Line Singularity are highly skewed distributions, and so standard partition techniques such as bucketing do not work well. For each distribution, we ran several experiments with different data size ranging from 5 to 80 million points. The data size of 12 million points was chosen because TRIANGLE is usually killed by the operating system on data sets of roughly 13 million points. In the experiments, we the block size in our program for data partitioning to be 2 million points.

Results

Figure 13*a* compares the running time of TRIANGLE and our algorithm for all three distributions. We consider both CPU time and I/O time.

First, observe that both algorithms perform almost identically on all three distributions, indicating that they are insensitive to data distributions.

¿From Figure 13*a*, we see that our disk-based algorithm generally outperforms *Triangle* in total running time on data sets of more than 5 million points. As the data size increases, TRIANGLE spends more and more time on I/O. This is not surprising. As an in-memory algorithm, TRIANGLE stores all the data, such as points, triangles, *etc.*, in arrays. As the data size grows, the arrays become too large to fit completely in the memory, and part of the data must be swapped to the disk. Yet TRIANGLE continues to access these large arrays randomly. As a result, the CPU must stall frequently and wait for the data to be loaded from the disk. In contrast, I/O time for our disk-based algorithm is much smaller and grows gently with the data size. This is attributed to the efficient data management by our algorithm. Figure 13*b*, which shows the amount of data throughput between the memory and the disk, further confirms this view. Our algorithm shows a steady linear growth in I/O cost, while TRIANGLE shows a much faster super-linear growth. Furthermore TRIANGLE cannot handle very large data sets: the process was killed by the operating system if the data sets contained more then 13 million points. What Figure 13*b* cannot show is that our algorithm not only generates fewer I/O operations, but also access the disk access sequentially most of the time, resulting lower I/O cost per operation on the average. Overall, our algorithm is faster in in total running time, as a result of effective I/O management, and can process much larger data sets.

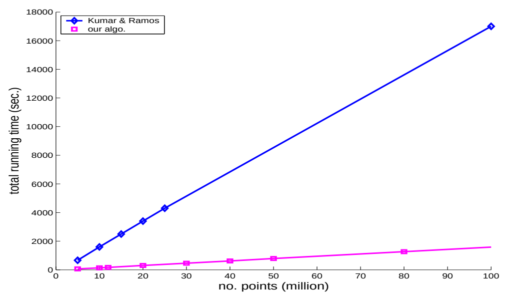


Figure 14: Comparison of our algorithm with a provably-good disk-based DT algorithm.

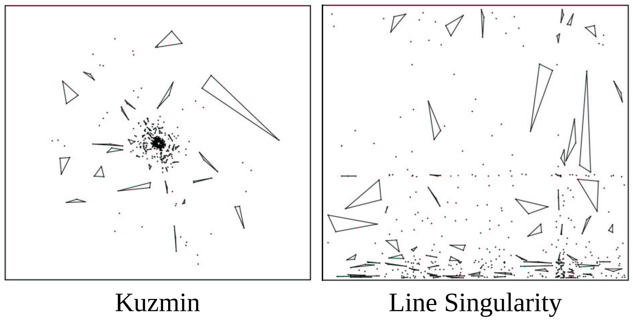


Figure 15: Examples of generated PSLGs using different distributions.

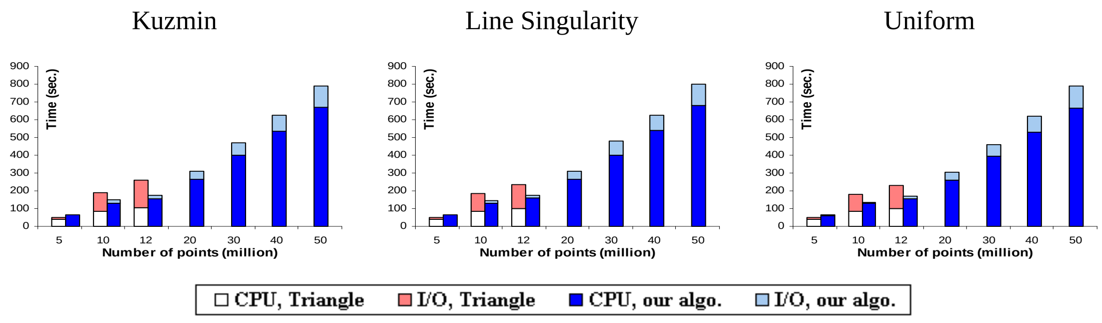
We also compared our algorithm with another disk-based DT algorithm by Kumar and Ramos [22]. Kumar and Ramos' algorithm is provably efficient. In their experiments, they used a dual-processor Athlon MP 1800 system with 1GB memory. Despite the slight disadvantage of our hardware system, our algorithm demonstrated roughly an order of magnitude speedup in total running time Figure 14. The reason, we believe, is that our data partitioning and merging methods are more effective and avoid processing the same data multiple times.

6.2 Constrained Delaunay Triangulation

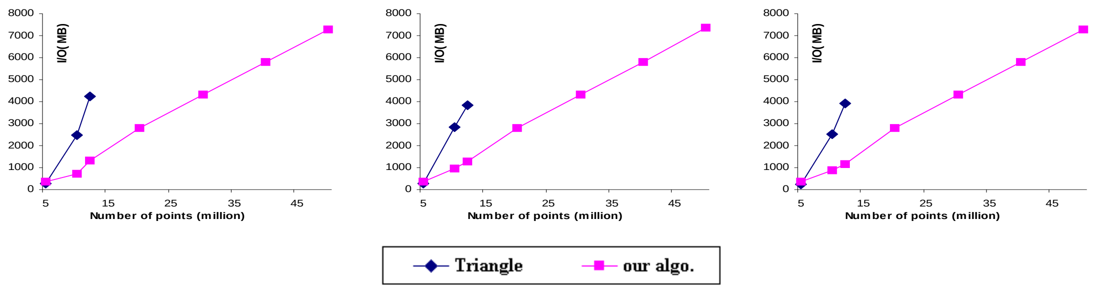
Data Distribution

The point sets for the input PSLGs are again generated with Kuzmin, Line Singularity, and Uniform distributions. There are two parameters for data generation: the total number of points N and the ratio the ratio of the number of constraints segments versus the number of points $0 \leq \alpha \leq 1$, which is used to control the density of segments. Below we describe how the data sets are generated for each distribution:

Kuzmin PSLG We first randomly generate $\sqrt{N/3}$ values for radius r using the distribution function $M(r)$ of the Kuzmin distribution. Then we generate $\sqrt{N/3}$



(a) Running time.



(b) I/O cost.

Figure 13: Running time and I/O cost comparison of DT algorithms on three data distributions.

values for angle θ from $[0, 2\pi)$. Each combination of (a, r) represents a point in the polar coordinate system. Together these combinations form a spiderweb with $N/3$ cells. In each cell, we randomly sample three points, which are then connected with constraint segments to form a triangle with probability α. See Figure 15 for an example.

Line Singularity PSLG (Figure 15 right) We first generate a *Uniform* PSLG with the same parameters N and α, and then map each point (u, v) in the PSLG to (x, y) using (2).

Uniform PSLG We uniformly and randomly partition the unit square into a grid of $N/3$ cells. In each cell, we sample three points and decide with probability α whether to create constraint segments to connect them.

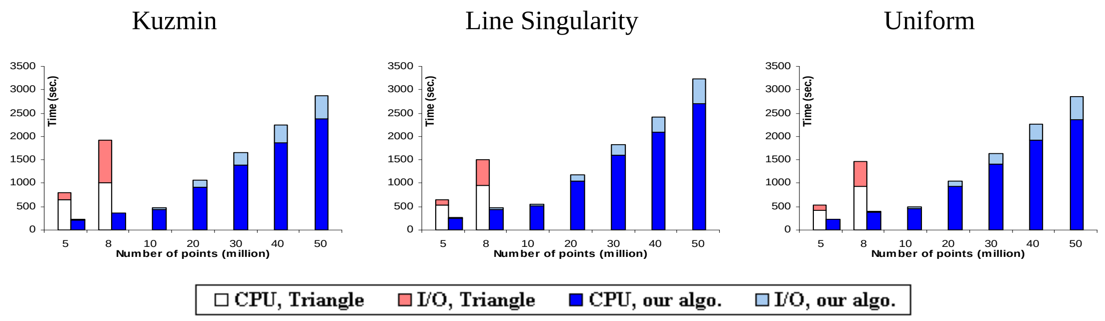
Results

For CDT, we compare with TRIANGLE only, since there are no practical disk-based algorithms (see Section 2). Our experiments consist of two parts. In the first part, we fix the segments to points ratio α, and vary the number of points N. In the second part, we fix N and vary α.

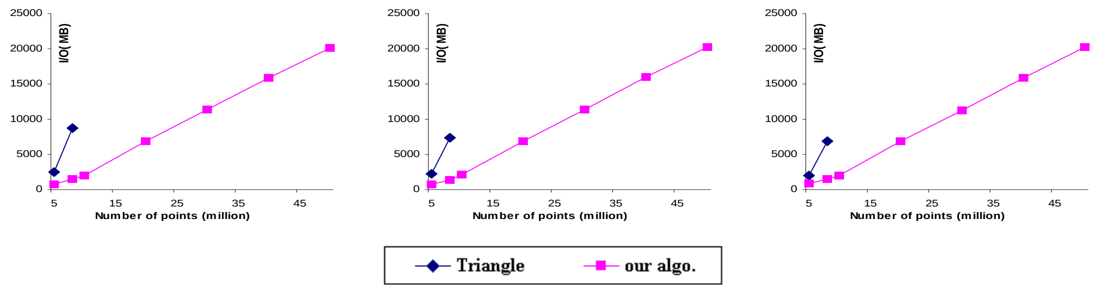
In the first part, we set $\alpha = 50\%$, and ran data set with 5 to 50 million points for all three distributions. The data sets with 8 million points were chosen because TRIANGLE got killed by the OS on the data set with 9M points and $\alpha = 50\%$. The charts (Figure 16) are organized in the same way as for DT.

As Figure 16 shows, the performance of both algorithms is very similar for all three distributions, which means that both are insensitive to data distributions for CDT as well. The performance comparison yields similar conclusion as that for DT, only that the advantage of the disk-based algorithm becomes even more obvious. TRIANGLE builds the DT first and constructs the CDT by inserting the segments one by one. Each insertion requires searching the triangulation and finding the location to insert the segment. When the triangulation cannot be stored in the memory completely, the search incurs significant I/O cost, which explains the dramatic increase in running time and I/O cost. Our disk-based CDT program processes the segments in batches. For each batch of segments, only a much smaller triangulation of the corresponding block needs to be searched. As a result, the search can be done entirely in the memory, which greatly reduces the running time and I/O cost.

Next, we fix the number of points N at 8 million and vary the segments to points ratio α from 10% to 90%. The performance of both algorithms is very similar for all three distributions. For brevity, only the chart on Kuzmin distribution is presented here. As Figure 17 illustrates, both TRIANGLE and our algorithm demonstrate linear growth in running time and I/O cost with respect to α, but the rate of growth for our algorithm is much smaller. Although TRIANGLE processes segments one by one while our algorithm does it in batches, both algorithms are incremental construction in nature. Since the size of triangulation is not affected by the number of segments, one would expect that the average cost to insert a segment into the triangulation remains relatively constant as the density of constraint segments increases. This explains the linear growth in computational cost.

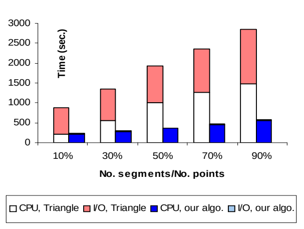


(a) Running time.

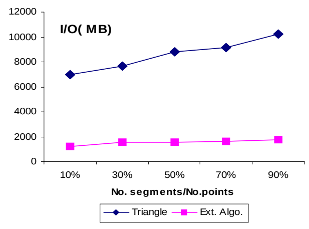


(b) I/O cost.

Figure 16: Running time and I/O cost comparison of CDT algorithms on three data distributions.



(a) Running time.



(b) I/O cost.

Figure 17: Comparison between TRIANGLE and our algorithm on Kuzmin PSLGs with different segments/points ratios.

7 Discussion

Currently the merging step of our algorithm computes the DT of the seam, $DT(S')$, in the memory. This has worked well in all of our experiments, despite the large input data size. Typically the seam size is less than 0.6% that of the original input data. The largest seam encountered has only 281934 points, well within the memory capacity. Nevertheless, as the data size grows, the seam will eventually fail to fit in the memory. In this case, we propose to apply our algorithm recursively to S'. For truly massive data sets, we can apply the recursion multiple times and obtain the final triangulation, as long as each recursive step reduces the seam size by a significant fraction. The recursive extension of our algorithm works well, except for some pathological cases, *e.g.*, all the points lying on a parabolic curve. Such a pathological case would fail all disk-based algorithms based on divide-and-conquer, unless all the data fit in the memory. However, one simple way for breaking such pathological cases in practice is to insert a few randomly sampled points into the input data as a preprocessing step.

8 Conclusion

This paper presents an efficient disk-based algorithm for CDT on large spatial databases. We have tested the algorithm extensively for both DT and CDT. Experimental results show that for DT, our algorithm outperforms a provably good disk-based algorithm by roughly an order of magnitude. For CDT, which has no previously implemented disk-based algorithms, we show that our algorithm scales up well for large databases. In the future, we plan to look at how our algorithm can be used and extended as a pre-processing step in applications such as spatial proximity search, location based services, and spatial data interpolation.

References

[1] Archimedes. http://www-2.cs.cmu.edu/ quake/archimedes.html.

[2] Enterprise mapping deployments: Mapinfo spatialware white paper.

[3] Ibm informix spatial datablade module user's guide, version 8.20 (g251-1289-00).

[4] Oracle spatial and locator(white paper).

[5] The quake project. http://www-2.cs.cmu.edu/ quake/quake.html.

[6] M. V. Anglada. An improved incremental algorithm for constructing restricted delaunay triangulations. *Computers & Graphics*, 21:215–223, 1997.

[7] F. Aurenhammer. Voronoi diagrams-a survey of a fundamental geometric data structure. *ACM Computing Surveys*, pages 345–405, 1991.

[8] F. Aurenhammer. Voronoi diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier, 2000.

[9] T. Barclay, D. R. Slutz, and J. Gray. Terraserver: A spatial data warehouse. In *SIGMOD Conference*, pages 307–318, 2000.

[10] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. 1:23–90, 1992.

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

[12] M.-B. Chen, T.-R. Chuang, and J.-J. Wu. A parallel divide-and-conquer scheme for delaunay triangulation. In *9th International Conference on Parallel and Distributed Systems*, pages 571–576, 2002.

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

[14] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. Technical report, Max-Planck-Institut für Informatik, 1998.

[15] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[16] L. D. Floriani and E. Puppo. An on-line algorithm for constrained delaunay triangulation. *Computer Vision, Graphics and Image Processing*, 54:290–300, 1992.

[17] C. M. Gold. A review of potential applications of voronoi methods in geomatics. In *Proceedings of Canadian Conference on GIS*, pages 1647–1656, 1994.

[18] C. M. Gold. Review: Spatial tesselations - concepts and applications of voronoi diagrams. *International Journal of Geographical Information System*, 8:237–238, 1994.

[19] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. 1993.

[20] M. I. Karavelas and L. J. Guibas. Static and kinetic geometric spanners with applications. *Symposium on Discrete Algorithms*, 2001.

[21] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete and Computational Geometry*, pages 13–28, 1992.

[22] P. Kumar and E. A. Ramos. I/o-efficient construction of voronoi diagrams. 2002. http://www.ams.sunysb.edu/ piyush/ramos/.

[23] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations-Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons, 1992.

[24] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *First Workshop on Applied Computational Geometry*, pages 124–133, 1996.

[25] P. Su and R. L. S. Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 61–70, 1995.

[26] A. K. H. Tung, J. Hou, and J. Han. Spatial clustering in the presence of obstacles. In *Proceedings of the 17th International Conference on Data Engineering*, pages 359–367, 2001.

[27] C. A. Wang. Efficiently updating constrained delaunay triangulations. *Nordic Journal of Computing*, 33:238–252, 1993.

[28] E. Welzl. Constructing the visibility graph for n line segments in $o(n^2)$ time. *Information Processing Letter*, 20:167–171, 1985.

[29] X. Wu, D. Hsu, and A. K. H. Tung. Efficient constrained delaunay triangulation on large spatial databases. Technical report, National University of Singapore, 2005.

[30] C. Xia, D. Hsu, and A. K. H. Tung. A fast filter for obstructed nearest neighbor queries. In *21st British National Conference on Databases*, pages 203–215, 2004.

[31] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Spatial queries in the presence of obstacles. In *International Conference on Extending Database Technology*, pages 366–384, 2004.

A Proofs

LEMMA 4.3. PROOF: First we show that if $t \in DT(S)$, then $t \in DT(S')$. $DT(S')$ can be obtained by deleting all the points in $S \backslash S'$ from $DT(S)$ and re-triangulating. Deleting a point p from a DT only affects those triangles incident to p; a triangle t not incident to p remains unchanged, because the empty-circle property for t is unaffected by deletion of points. For any point $p \in S \backslash S'$, p cannot be incident to any crossing triangle; otherwise, p would have already been included in S'. Therefore all the crossing triangles in $DT(S)$ remain after the deletion of points in $S \backslash S'$. It then follows that for any crossing triangle $t \in DT(S)$, $t \in DT(S')$. To prove the other direction, simply observe that adding a point back only creates those triangles that are deleted. ■

LEMMA 4.4. PROOF: First we show that $DT(S_i)$ contains no invalid unsafe triangles. If t is an invalid unsafe triangle from some block, it must intersect a crossing triangle in $DT(S)$. Since $DT(S')$ and $DT(S)$ have exactly the same set of crossing triangles by Lemma 4.3, t intersects some crossing triangle in $DT(S')$. This is impossible, because $DT(S')$ is a well-formed triangulation. Hence $DT(S_i)$ contains no invalid unsafe triangles.

Next we show that $DT(S')$ contains all the valid unsafe triangles. All such triangles must be present in $DT(S)$, as they are valid. Now we apply the same point deletion argument in the proof of Lemma 4.3. We obtain $DT(S')$ from $DT(S)$ by deleting all the points in $S \backslash S'$. Since unsafe triangles are unaffected by the deletion of these points, all the valid unsafe triangles remain in $DT(S')$. ■