I/O-EFFICIENT ALGORITHM FOR CONSTRAINED DELAUNAY TRIANGULATION WITH APPLICATIONS TO PROXIMITY SEARCH

XINYU WU

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgement

Although only one name appears on the cover, this thesis would not be possible without support of various people who accompanied me during last two and a half years. I take this opportunity to express my thanks to all of them.

First and foremost, I would like to express my sincere gratitude to Dr. David Hsu and Dr. Anthony Tung for their guidance, encouragement, and friendship throughout my time as master candidature. As my supervisors, they have constantly motivated me to explore new knowledge and reminded me to remain focusing on achieving my main goal as well. Dr. Tung initiated the idea of using the constrained Delaunay triangulation to facilitate obstructed proximity search and introduced me to this exciting research direction. During my study, I enjoyed numerous memorable conversations with Dr. Hsu. Without his insightful observations and comments, this thesis would never have been completed. But more importantly, I want to thank my supervisors for teaching me the values of persistence, discipline and priority. These lessons will benefit me for the rest of my life.

I am grateful to Huang Weihua, Henry Chia, Yang Rui, Yao Zhen, Cui Bin,

and all other friends and colleagues in the TA office and Database group for their friendship and willing to help in various ways. Working with them has certainly been a wonderful experience. Further, I want to thank the university for providing me with world-class facilities and resources.

My special thanks go to my beloved family in China for being supportive to every decision I made.

Finally, my wife Liu Li helped me with most of the real-life data sets used in the experiments. But that is the least thing I want to thank her for. I will have to devote my whole life to repaying her unconditional love, understanding, and support.

CONTENTS

LIST OF FIGURES

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, however, shows that there is a surprising lack of I/O-efficient algorithms for computing DT/CDT on large spatial databases. In view of this, we propose an external-memory algorithm for computing CDT on spatial databases with DT being computed as a special instances.

Our proposal is based on the divide and conquer paradigm which compute DT/CDT of in-memory partitions before merging them into the final result. This is made possible by discovering mathematical properties that precisely characterize the set of triangles that are involved in the merging step. Extensive experiments show that our algorithm outperforms another provably good external-memory algorithm by roughly an order of magnitude when computing DT. For CDT, which has no known external-memory algorithm, we show experimentally that our algorithm scale up well for large databases with size in the range of gigabytes.

Obstructed proximity search has recently attracted much attention from the spatial database community due to its wide applications. One main difficulty for

processing obstructed proximity search queries lies in how to prune irrelevant data effectively to limit the search space. The performance of the existing pruning strategies is unsatisfactory for many applications. We propose a novel solution based on the spanner graph property of the CDT to address this key weakness. In particular, we show how our pruning strategy can be used to process the obstructed k-nearest-neighbors and range queries.

CHAPTER 1

Introduction

In this thesis we present an I/O-efficient algorithm for construction of large-scale constrained Delaunay triangulations. We also propose effective methods based on the constrained Delaunay triangulation for processing obstructed proximity search queries in spatial database systems.

1.1 Motivation

Delaunay triangulation (DT) is a geometric 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 1.1). In this work, we develop an efficient algorithm that computes DT and its extension, *constrained Delaunay triangulation*, for data sets that are too large

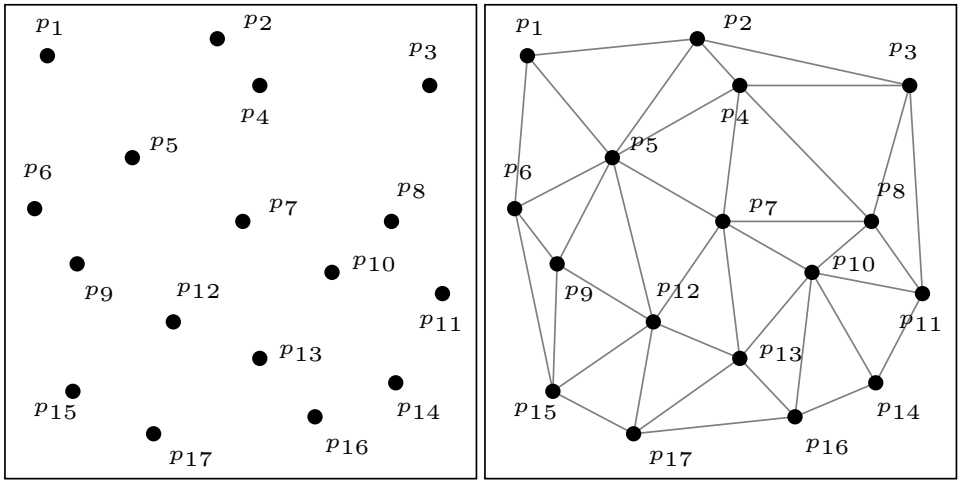


Figure 1.1: A set of points (left) and its Delaunay triangulation (right).

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 modelling from measurements of the terrain height at sampled points. One way for constructing a terrain surface is to first compute the DT of the sample points and then interpolate the data based on the triangulation [22, 23, 37]. Figure 1.2 shows a terrain surface constructed this way. The same interpolation method easily extends to other spatial data, such as readings from a sensor network.

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, finite element analysis is applied to billions of points to simulate the shock wave of earthquakes, and DT is used to generate

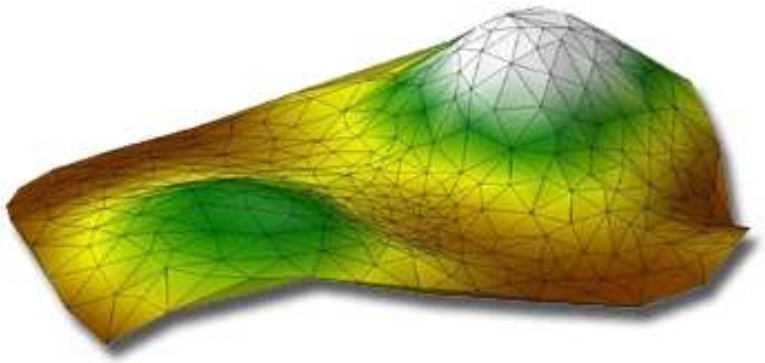


Figure 1.2: A terrain surface constructed using Delaunay-based spatial interpolation.

the meshes needed for simulation [3].

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, 37] and is easier to compute, it is common to compute the DT first and obtain the Voronoi diagram by taking the dual.

The application of DT extends further if we allow in the input data constraint edges that must be present in the final triagulation. Intuitively, this extension, called the constrained Delaunay triangulation (CDT), is as close as one can get to the DT, given the constraint edges (Figure 1.3). Constraint edges occur naturally in many applications. We give two 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,

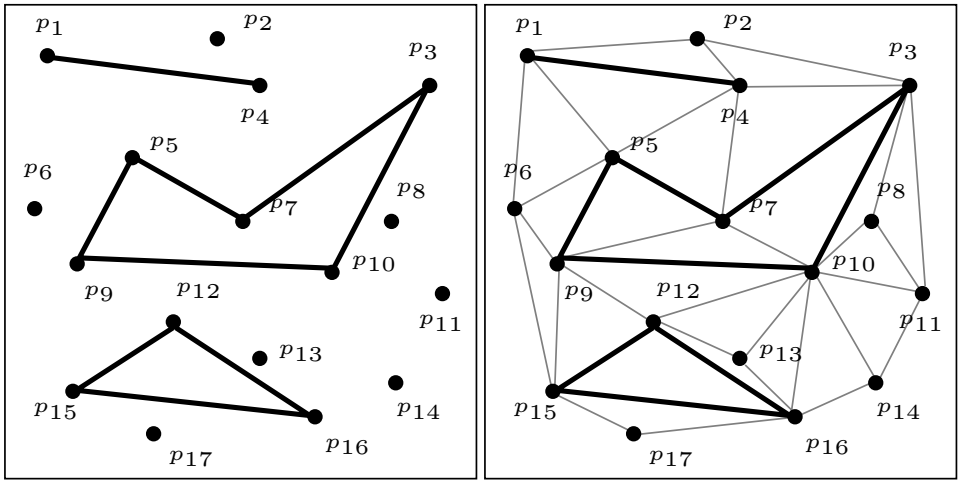


Figure 1.3: Input data points and constraint edges (left) and the corresponding Delaunay triangulation (right).

e.g., regions where water cannot flow through.

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, 8]. 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 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 whole triangulation can not fit in the memory completely.

One of our motivations for designing large-scale CDT algorithm is to facilitate obstructed proximity search. Despite the presence of obstacles in many applications, most traditional spatial proximity search queries, such as k-nearest-neighbors and range queries, measure the distance using simple metric, *e.g.*, the L_1 distance or Euclidean distance. The advantage of adopting these simple metrics is the computational efficiency. However, many real-life scenarios cannot be modelled accurately by these simple metrics due to the blocking of obstacles. For example, a nearest gas station under the Euclidean metric may not mean so much to a car driver if it is across the river. Obstructed proximity search queries addresses this inaccuracy by measuring, between two points, the length of the shortest obstacle-avoiding path. In the literature, this length is often called the geodesic distance, and the shortest obstacle-avoiding path the shortest geodesic path. The obstructed proximity search queries have wide applications in geographical information systems, facility location planning, and virtual environment walk-through. In addition, they can also serve as a useful tool for spatial data mining algorithms such as clustering and classification [41].

Because of its importance, obstructed proximity search queries have recently attracted a lot of attention from the spatial database community [44, 45]. The basic operation of all obstructed proximity search is to compute the shortest geodesic path. This can be done by constructing and searching the so-called visibility graph. Unfortunately the visibility graph has super-quadratic complexity in both time and space and therefore cannot be pre-materialized. One way to circumvent this is to prune irrelevant data and build local visibility graph online. However, the existing pruning strategies are often not effective enough and result in great computational waste in computing local visibility graph. The need to design better pruning strat-

egy is becoming more and more apparent.

1.2 Objectives and Contributions

Motivated by the observation that there is limited work on practical algorithms for external-memory DT and CDT despite their importance, the first objective of this thesis is to design a scalable method for the construction of CDT, with DT as a special case. We believe that our work makes the following contributions:

- We present an efficient external-memory algorithm for CDT using the divide-and-conquer approach (Section 3.3). 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 internal-memory algorithm for triangulation within a block, but the merging method is independent of the specific internal-memory algorithm used. In this sense, we can convert any internal-memory DT/CDT algorithm into an external-memory one, using our approach.

- We describe in details the implementation of our algorithm (Section 3.4). 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. They require no floating-point calculation, thus improving the robustness of the algorithm.

- We have performed extensive experiments to test the scalability of our algorithm for both DT and CDT (Section 3.5). For DT, we compare our algorithm

with an existing external-memory 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 external-memory algorithm. We compare the performance of our algorithm with an award-winning internal-memory algorithm [39] and show that the performance of our algorithm degrades much more gently when the data size increases.

The second objective of this thesis is to improve the efficiency of processing obstructed proximity search queries. The main problem of such queries is how to prune irrelevant data effectively to limit the size of the local visibility graph. The existing pruning strategy is not powerful enough for many applications. We present a more effective solution based on the spanner graph property of the CDT (Section 2.3). Our contribution towards the second objective are the following:

- We have conducted extensive experiments on real-life data set to examine the true stretch factor of the CDT as spanner graph of the visibility graph (Section 4.2). Our experiment lends support to the general belief that the CDT indeed approximates the visibility graph significantly better than the theoretically proven bound.

- We introduce a provably good pruning strategy based on CDT for processing obstructed proximity search queries. In particular, we apply our strategy successfully to k-nearest-neighbors and range queries (Section 4.3).

1.3 Outline

The remaining of the thesis is organized as follows: Chapter 2 is a literature review of the previous work in DT/CDT construction algorithms and the obstructed

proximity search problem; In Chapter 3, we present our external-memory CDT algorithm in detail and provide extensive experimental evaluation of its performance. In Chapter 4, we first examine the stretch factor of CDT as the spanner graph through experiments on real-life data sets, and then propose a new pruning strategy for processing obstructed proximity search queries. Chapter 5 concludes our work with a summary of the main results and suggests directions for future research.

CHAPTER 2

Previous Work

Due to its importance for applications, DT has received much attention. Intensive research has led to many efficient algorithms, using various approaches. In this chapter, we review some of the current main memory, external-memory and parallel algorithms for computing DT and CDT. Also found in this chapter is a brief survey of the proximity search problem in the presence of obstacles.

2.1 Main Memory DT/CDT Algorithms

Efficient main memory algorithms for computing DT have been discovered for a long time. Three types of commonly used algorithms are divide-and-conquer algorithms, plane sweep algorithms and incremental algorithms. The divide-and-conquer approach recursively divides the input data into roughly equal parts, computes the triangulation for each part, and then merge the resulting triangulations. The plane sweep approach sorts the data according to their x-coordinates and processes the

data from left to right in the sorted order [21]. The randomized incremental construction processes the input data vertices one by one and updates the triangulation when a data vertices is inserted [31]. See [8] for a good survey. Many of these algorithms achieve the $O(n \log n)$ running time, which is optimal asymptotically. n is the number of input vertices.

Experiments show that of the three approaches, divide-and-conquer is the most efficient and robust one in practice [40]. Although the external-memory algorithm we propose follows a different design principle of minimizing disk I/O, it is also based on the divide-and-conquer paradigm and therefore share certain common characteristics with the main memory divide-and-conquer approach. We discuss the main memory divide-and-conquer approach in some depth here.

Shamos and Hoey [38] found a divide-and-conquer algorithm for computing Voronoi diagram, based on which DT can be easily built as it is the dual graph to Voronoi diagram. Lee and Schachter [34] first gave a divide-and-conquer algorithm directly constructing DT. Nevertheless, their original algorithm and proof are rather intricate, and Guibas and Stolfi [25] introduced an ideal data structure to fill out many tricky details. The original algorithm partitions the data into vertical strips. Dwyer [18] provided a simple yet effective optimization by alternating vertical and horizontal cuts to partition the data into cells of size $O(\log n)$ and merging DT of cells first into vertical strips and stitching strips into the whole triangulation. The optimized algorithm achieves better asymptotic performance on some distributions of vertices and runs faster in practice as well. Inspired by Dwyer's idea, our external-memory algorithm also partitions the data with alternating cuts, though the cell size is determined by other factors.

The central step of the divide-and-conquer algorithm is to merge two half triangulations, here denoted by L and R, into the whole triangulation. Firstly, the

lower common tangent e_1 of L and R is found. e_1 must be in DT, as we can always construct an empty circle pertaining to cord e_1 by starting with any such circle and growing it away from the triangulation. e_1 is the first edge crossing the separating line between L and R. Inductively suppose that e_i is the i-th cross edge and all the cross edges below e_i are correctly constructed. If e_i is the upper common tangent of L and R, then the merging step is finished. Otherwise we can imagine growing an empty cycle pertaining to cord e_i upwards until it touches the first vertex v, referring to Figure 2.1. It can be shown that v must be connected to the end of e_i that lies on the same side of v. The algorithm then creates a new cross edge e_{i+1} connecting v with the other end of e_i. All the original edges in triangulations of L and R that cross e_{i+1} are deleted. The merging step works from bottom up until the upper common tangent is met. As one might expect, the algorithm has to store some connectivity information like pointers from an edge to its incident edges [25] or from a triangle to its incident triangles [39] so that the updates can be efficiently performed.

Lee and Lin [33] first investigated the CDT and proposed an $O(n^2)$ algorithm for its construction. Later, Chew [13] described a divide-and-conquer algorithm that reduced the time bound to asymptotically optimal $O(n \log n)$, n being the number of vertices. The algorithm is however very demanding to implement. The most popular and probably the easiest to implement algorithm for constructing constrained CDT is the incremental algorithm [4, 20, 42]. An incremental CDT algorithm first computes DT of the input point set. Then the segments are inserted in to the triangulation. Each insertion of the segment may affect a certain region in the triangulation. Specifically, the region comprises all the triangles that cross the segment. As the segment must be included in the CDT, all the edges crossing the segment are removed. The affected region is hence cut by the segment

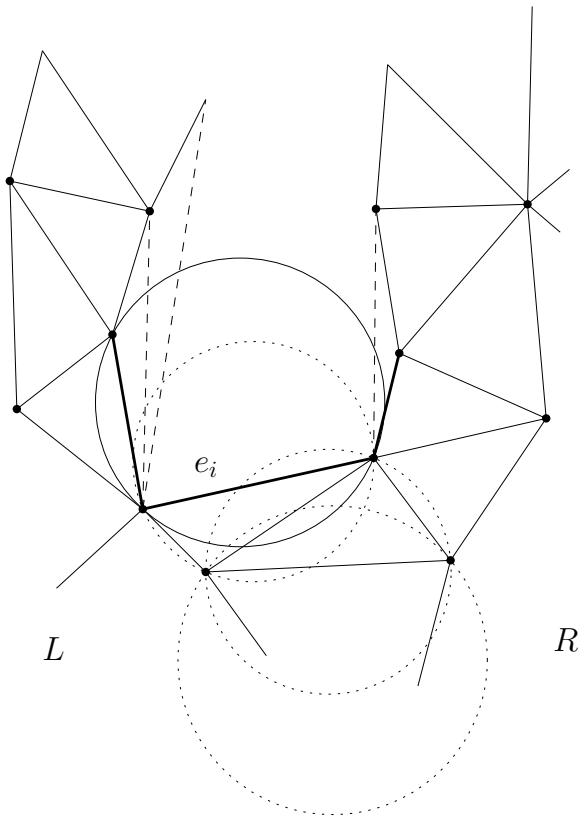


Figure 2.1: The rising bubble.

into two sub-regions. It can be shown that only these two sub-regions need to be re-triangulated to conform the triangulation to the segment. The complexity of an insertion includes two parts. The first part is to locate the affected region. Theoretically, one can build a $O(n)$ index structure to answer location queries in $O(\log n)$ time. However, this does not usually work well in practice due to preprocessing and storage requirements. One practical solution is the *jump-and-walk* algorithm proposed by Mücke *et al.* [36]. The second step is to re-triangulate the affected region. Wang [42] discovered a difficult algorithm that runs in asymptotically optimal $O(k)$ time, k being the number of vertices of the affected region. k is normally a small number unless the segment is very long, and a simple $O(k^2)$ algorithm [20] is usually adopted in practice.

2.2 DT/CDT Algorithms in Other Computational Models

The algorithms listed above all assume a sequential random access model of computation and do not consider the efficiency with respect to disk access. When the data is too massive to fit into the memory, they completely rely on the virtual memory of the OS and perform poorly due to huge amount of I/O operations. The situation is even worse for constrained DT. As in the conventional incremental algorithm, each insertion of the segment involves a location query which is very expensive when the triangulation is stored on disk. In this section, we survey the external-memory algorithms for constructing DT.

Another class of DT algorithms that caught our attention are parallel algorithms. We discuss parallel algorithms because they share similar design principles with the external-memory algorithm and many techniques used in parallel algorithms can be easily extended to external-memory algorithm or vice versa.

External-Memory Algorithms

The memory of a modern computer system is typically organized into a hierarchy. From top to bottom, we have CPU registers, L1 cache, L2 cache, main memory, and disc. Each level is faster, smaller, and more expensive per byte than the next level. For large-scale information-processing applications, the I/O communication between fast main memory and slower external storage devices such as disks and CD-ROMs often forms the bottle-neck of the overall execution. In this context, a theoretical simplified memory hierarchy was proposed to analyze the program performance [24]. In this model, there are only two kinds of memory: the very fast main memory and the very slow disk. A disk is divided into contiguous blocks.

The size of each block is B; The size of the problem instance is N; and the size of the memory is M. For the purpose of analyzing external-memory algorithm, M is assumed to be smaller than N. All the I/O-efficient DT algorithms that we know are designed based on this model. However, before we survey these algorithms we need to stress two limitations of this model. Firstly, the model assumes a unit cost for accessing any block of data in disk and does not consider the fact that reading contiguous blocks is typically much cheaper than random reads. Secondly, the I/O analysis done under this model often focuses too much on asymptotical bound in terms of M and N and neglects the hidden constant factor. Thus an asymptotically optimal algorithms may not yield good practical performance.

In [24], Goodrich *et al.*. introduced several I/O-efficient algorithms for solving large scale geometric problems. They described an algorithm for solving the 3-d convex hull problem with an I/O bound of $O((N/B)\log_{M/B}(N/B))$. By well-known reductions [9], the algorithm can also be used to solve DT problem with the same I/O performance, which is asymptotically optimal. However, the algorithm is "esoteric" as they described. Crauser *et al.*. developed a new paradigm based on *gradation* for optimal geometric computation using external-memory and achieved the same optimal I/O bound for DT construction [16]. Both algorithms presented in [24] and [9] are *cache-aware* in the sense that they need to know the parameters M and B in advance. Subsequently, Piyush and Ramos [30] studied the *cache-oblivious* version of DT construction, where the algorithm only assumes an optimal replacement strategy to decide which block is to be evicted from internal memory instead of the actual values of M and B. Moreover, they implemented a simplified version of their algorithm and reported the running time of their program. That is the only experimental study of an external-memory DT algorithm that we have found in the literature. All the above algorithms are based on random

sampling. For a concrete example, we summarize the algorithm Piyush and Ramos implemented in [30] below.

The algorithm adopts a divide-and-conquer approach. Given the input of n vertices, it first draws a random sample of the vertices that is small enough to fit into the memory and computes DT of the sample using any efficient main memory algorithm. For convenience, the sample actually includes 4 points in infinity so that the triangulation covers the whole space. Then the algorithm computes the conflict list of each triangle in DT of the sampled vertices. The conflict list of a triangle is the set of all vertices that invalidates the triangle, that is, the set of all vertices that lie within the circumcircle of the triangle. For each pair of triangles in the sample that share a common edge, connect the two common vertices together with the circumcenters of the two triangles to form a diamond. See Figure 2.2. It is easy to see that all such diamonds form a partition of the space, therefore any triangle in the final triangulation of the whole vertices set must has its circumcenter in one of those diamonds, ignoring the case where the circumcenter lies on the boundary between diamonds for brevity. So in the conquering step, the algorithm finds all the triangles circumcentered in each diamond. To do this, the algorithm loads all the vertices in the union of the conflict lists of the two triangles that define the diamond, calls a main memory algorithm to compute DT of these vertices, and scan from the triangulation for triangles circumcentered in the diamond. It can be shown that these triangles are precisely those in the overall triangulation whose circumcenters lie in the diamond.

Note that in the conquering step, one cannot be theoretically certain that the vertices from the union of conflict lists fit into the memory. At best, one can argue this is the case with high probability. As experiments demonstrate, it is good enough for practical purposes. There are two sources of inefficiency in the

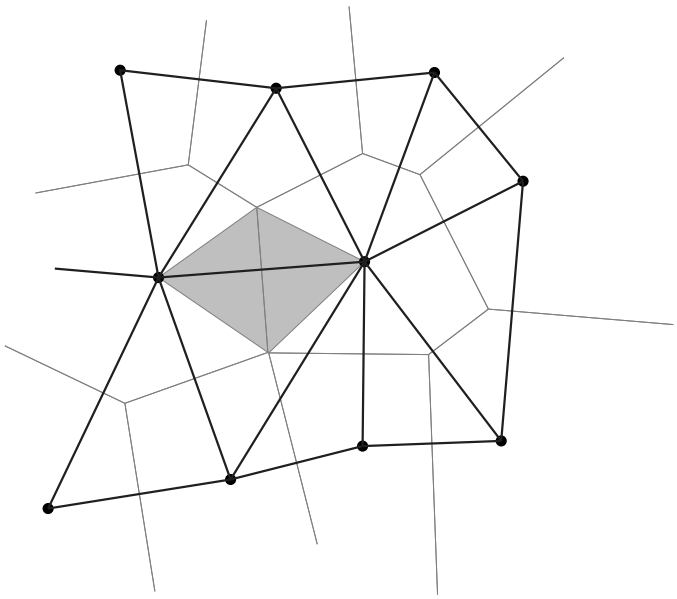


Figure 2.2: a diamond shape.

algorithm, though. One is the computation of the conflict set of a triangle. The algorithm does this in an inverse way. For each point, it finds the triangles conflicts with this point. Then the conflict lists are produced by sorting. Still, this requires doing point location for every input vertex. The other inefficiency lies in the computation of triangles circumcentered in a diamond. The area of a diamond is usually greatly smaller than the area of the union of the circumdiscs of the two triangles that define the diamond. Therefore, it is wasteful to load all the vertices in the union of the conflict lists and triangulate all of them only to find triangles circumcentered in the diamond. Moreover, a vertex conflicts with multiple triangles in DT of the sample; each edge in these triangles corresponds to a diamond; and the vertex needs to be loaded once for each such diamond, which is a big waste in both time and space.

We are not aware of any external-memory algorithm for constructing constrained DT in the literature.

Parallel Algorithms

Another class of DT algorithms that caught our attention are parallel algorithms which use several processors working simultaneously to solve large scale problems. We discuss parallel algorithms here because they share similar certain design principles with external algorithms. For example, one of the main objectives in designing parallel algorithms is to minimize inter-processor communication, which naturally corresponds to minimizing disk access in the single processor model. However, I/O efficiency and parallel efficiency are not equivalent. For example, parallel algorithms need to address the inter-processor synchronization problem, while external-memory algorithms cannot simultaneously load everything into the memory to partition the data.

Most parallel DT algorithms work by decomposing the problem domain into sub-domains of roughly equal size, distributing sub-domains to different processors, and merging the sub-solutions into the overall triangulation. Unsurprisingly, the major difficulty in parallelizing DT algorithm lies in the merging phase. And most research has been centered around improving the efficiency. Many parallel DT algorithms such as [15] use special techniques like bucketing to achieve good performance on uniformly distributed data set. We do not discuss them here as their performances degrade significantly when the data distribution is non-uniform. Of those algorithms that are insensitive to data distribution, Blelloch *et al.*. [10] proposed the "marriage before conquest" strategy which pre-computes the inter-processor region boundary to separate the computation of the interior region of the processors. For every boundary, the algorithm needs to project the point set twice, first onto a 3D paraboloid and then to a plane perpendicular to x- and y-coordinates, and compute the lower 2D convex hull of the projection of the point set on the plane. They showed that the points whose projected images lie on the

convex hull precisely define the boundary. An external-memory algorithm using this strategy may not be efficient for the need to compute convex hull of point sets that do not fit into the memory. Chew *et al.* [14] introduced an incremental insertion parallel algorithm that can compute the CDT, but their focus of using constraint is to minimize inter-processor communication. The divide-and-conquer approach that we adopt is related to that used in the work of Chen *et al.* on parallel DT computation [11], but our merging method is more efficient, and we handle CDT as well as DT.

2.3 Obstructed Proximity Search Problem

Spatial proximity search such as k-nearest-neighbors (kNN) and range queries in the presence of obstacle has recently emerged as a new research frontier due to its broad applications in spatial database systems. The first part of this section gives a background knowledge of the current techniques for the construction of geodesic shortest path which is the basic operation for all obstructed proximity search. In the second part, we review some of the existing work on processing obstructed queries in spatial database systems. Specifically, we focus on the kNN and range queries.

Geodesic Shortest Path Algorithms

We assume the obstacles are modelled as polygons and consider both exact and approximation algorithms for computing geodesic shortest path.

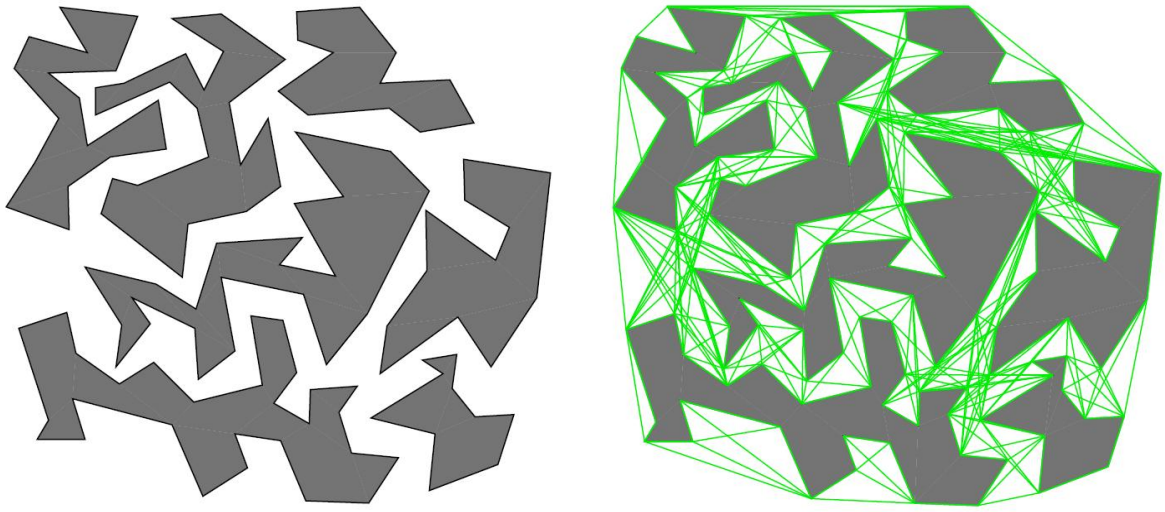


Figure 2.3: A set of polygonal obstacles (left) and the visibility graph (right).

Exact Algorithms

There have been two fundamentally different approaches for computing the exact geodesic shortest path—the *visibility graph search* and *continuous Dijkstra method*.

Given a set O of polygonal obstacles and a set of sites S, the visibility graph G contains all the vertices in O and S as its nodes. Two nodes n_i and n_j are connected if and only if they are mutually visible, *i.e.*, the line segment intersecting n_i and n_j does not intersect the interior of any obstacle.

Using simple local optimality argument, one can easily show that the geodesic shortest path must lie on the visibility graph. Also note that any path on the visibility graph must be obstacle-avoiding by definition of the visibility graph. Therefore the shortest path between two vertices on the visibility graph is exactly the geodesic shortest path. Thus we can construct the visibility graph first and use Dijkstra's algorithm to compute the geodesic shortest path. The naive algorithm to construct visibility construction runs in $O(n^3)$ by simply checking for every pair of points whether the line segment connecting them intersects any obstacle edge, n being the total number of input vertices. Lee [32] gave an $O(n^2 \log n)$ algorithm which

was based on a radial sweep about each vertex. The time complexity comes from the use of n independent radial sortings of the vertices. Later the time complexity was improved to asymptotically optimal $O(n^2)$ by Welzl [43] and Asano *et al.* [6]. The fatal shortcoming the approach of computing geodesic distance by visibility graph search is that the visibility graph can have as many as $\Omega(n^2)$ number of edges. The space requirement makes constructing the whole visibility graph impractical for any reasonably large data set.

The continuous Dijkstra method achieves the asymptotically optimal running time of $O(n \log n)$ and has the same space complexity [27]. It computes geodesic shortest path by simulating the "wavefront" propagation out from a source point. At any given time the wavefront maintains a set of curve pieces just like the ripple generated by throwing a stone into the water. The algorithm is very sophisticated and mainly for theoretical interest.

Approximation Algorithms

There have been several asymptotically efficient methods to approximate the geodesic shortest path [35, 5]. However the derivation of their asymptotical bound often requires sophisticated analysis. The algorithms are complicated to implement and have big constant factors. Here we concentrate on one type of simple algorithms that use geodesic $t-$spanners to compute the approximate geodesic distance. A $t-$spanner is a graph G that contains all the input vertices such that for every pair of input vertices, there is a path on G whose length is at most t-times their true distance. Note that the true distance can be according to any predetermined metric, *e.g.*, Euclidean, network, or geodesic. So when we say a $t-$spanner, we must specify the underlying metric.

The first geometric spanner result was given by Chew. In [12], he demonstrated

that the DT can be constructed according to L_1 metric is a spanner graph that approximates the Euclidean distance between any pair of points with stretch factor $t = \sqrt{10} \approx 3.16$. Dobkin, Friedman and Supowit [17] showed the length of the shortest path between two vertices on DT approximates their Euclidean distance with a stretch factor of $(1 + \sqrt{5})\pi/2 \approx 5.08$. Later the bound was improved to $2\pi/(3\cos(\pi/6)) \approx 2.42$ by Keil and Gutwin [29]. Karavelas and Guibas [28] generalized the proof in [17] to prove the same stretch factor for the CDT as a spanner graph for the visibility graph. That is, the length of the shortest path between two vertices on the CDT is at maximum 5.08 times their geodesic distance. The true stretch factor of both DT and CDT are generally believed to be much smaller than the theoretically proven bound. The worst-case lower bound of the stretch factor for DT and CDT is $\pi/2$, which is also due to Chew. In Chapter 4, we present extensive experimental results on real life data which indeed lends support to the general belief that the stretch factor is very small.

Obstructed Proximity Search Queries

Conventional spatial databases usually store the objects in R-tree [26]. Efficient Euclidean proximity search are supported by utilizing the lower bound and upper bound properties of R-tree. Recently, there has been some efforts to integrate geodesic shortest path algorithms into the spatial database systems to handle obstructed proximity search queries [44, 45]. The existing obstructed query processing methods use the visibility graph to compute the exact geodesic distance. The visibility graph of the whole data set cannot be pre-materialized due to its extreme size. These methods try to circumvent this difficulty by online constructing the local visibility graph of only the obstacles and sites that are relevant to the queries. To do this, they need a lower bound to geodesic distance to prune the obstacles and

sites. Invariably, the Euclidean distance is chosen as the lower bound. The simple argument is that the geodesic distance is at least as long as the Euclidean distance. Below we focus on the processing of two obstructed spatial queries—obstructed $k-$nearest-neighbors (k-ONN) and range query.

Obstructed Range Query

Given a query point p, a set of sites S, a set of obstacles O, and a range r, the obstructed range query returns all the sites within geodesic distance r to p. Zhang *et al.* described a simple algorithm in [45] to process the obstructed range query. The algorithm first performs a Euclidean range query to collect all the obstacles and sites that intersect the disc centered at p with radius r. By the lower bound property of the Euclidean distance, any site outside the disc cannot be within the geodesic range; and no obstacle outside the disc can affect the range query result. Obviously, not all the sites intersecting the disc fall into the geodesic range due to the blocking of obstacles. The algorithm then constructs a local visibility graph of only the selected obstacles and sites and employs the Dijkstra's algorithm on the visibility graph to find the sites within the geodesic range.

Obstructed $k-$Nearest-Neighbors Query

The k-ONN query returns the k nearest sites to the query point p in geodesic distance. The k-ONN query is harder than range query because of the lack of lower bound. The range r of the range query is a natural lower bound. Xia *et al.* and Zhang *et al.* gave two incremental algorithms for processing k-ONN queries. The two algorithms are similar in nature. Each algorithm successively look at the sites according to their Euclidean distance in ascending order. The termination condition for both is when the k-th nearest neighbor the algorithm has found so far

has geodesic distance shorter than the Euclidean distance of the next site to look at. Both algorithms incrementally retrieve obstacles that block the provisional geodesic shortest paths and recompute the visibility graph, but their retrieval strategies are different.

Zhang *et al.*'s algorithm grows a disc centered at p outwards, the new radius being set to be the provisional geodesic distance to the kth nearest neighbor in the last iteration. Then the algorithm loads new obstacles and sites that intersect the larger disc. It terminates when the provisional geodesic shortest paths to the k nearest neighbors remain the same in two subsequent iterations. Xia *et al.*'s algorithm has two levels of iterations. In each outer iteration, the algorithm works the same way as Zhang *et al.*'s algorithm to load new sites and obstacles. But instead of computing the geodesic distances by directly constructing the visibility graph of everything as the first algorithm does, it uses an incremental refinement algorithm to do the work. In each inner iteration, it adds the obstacles that intersect the provisional shortest paths into a list of obstacles it maintains. This can be done in main memory. Then the algorithm only constructs the visibility graph of the obstacles in the list and all the retrieved sites, and re-compute the shortest geodesic paths. The inner loop repeats until no new obstacle intersects any of the shortest provisional paths to the current k nearest neighbors. The two algorithms are incomparable in strengths. While the first algorithm is likely to construct visibility graphs that contain irrelevant obstacles, the second algorithm may generate too many inner cycles in the refinement process.

There three major drawbacks of these methods. The first drawback is that the Euclidean distance does not approximate the geodesic distance well in general. The lower bound is often too loose, and causes these methods to compute very large visibility graphs consisting mostly of irrelevant data. Secondly, in order to

prune irrelevant obstacles and sites, these methods need to invoke Euclidean range query, which is costly. This is especially bad for incremental algorithms which require performing the Euclidean range query repetitively. The last problem of these methods is that they do not offer a tradeoff between optimality of the query result and the computational cost. Due to the quadratic complexity of the visibility graph, sometimes it is simply infeasible to compute the exact geodesic path due to computational resource limitation. Instead of having the execution of a query for exact result terminated by the OS, one may wish to have a quick and reasonably good result. In Chapter 4, we propose new methods for processing obstructed proximity search query based on the CDT that address these three weaknesses.

CHAPTER 3

External-Memory Constrained Delaunay Triangulation

3.1 Introduction

Despite the importance of DT/CDT for various spatial database applications, our recent survey show that there is a surprising lack of I/O-efficient algorithms for computing large-scale DT/CDT. In this chapter, we propose a novel external-memory DT/CDT construction algorithm based on the divide and conquer paradigm. The algorithm makes clever use of several key properties of DT/CDT to achieve high efficiency. In particular it does not need the connectivity information in the merging step and avoids expensive geometric computation as much as possible.

The chapter is organized as follows: Section 3.2 Section 3.3 establish the theoretical foundation and gives an outline of the algorithm. Section 3.4 describes the implementation of the algorithm in detail. In Section 3.5, we report our extensive experimental study of our algorithm. Section 3.6 ends the whole chapter with a brief discussion of the general assumptions of the algorithm.

3.2 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 1.1). 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, $DT(S)$, is a triangulation with the additional property 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 [9].

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 line segment between p and q does not intersect with any segment of K. Using this notion of visibility, the *constrained Delaunay triangulation* of (S, K), denoted by $CDT(S, K)$, is defined as follows:

Definition Given a PSLG (S, K), a triangulation T of S is a constrained Delaunay triangulation of (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.

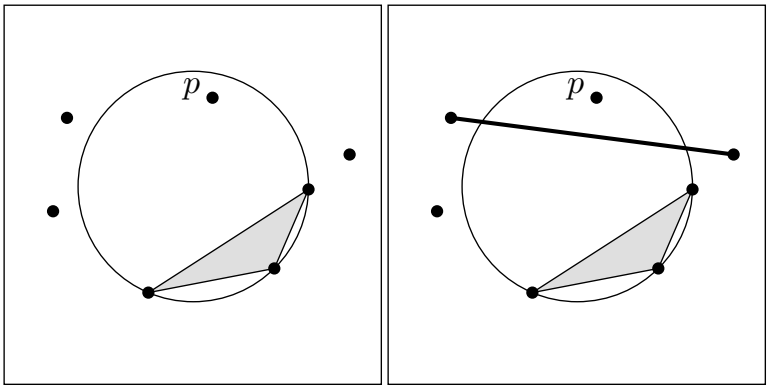


Figure 3.1: the triangle $\triangle pqr$ fails the in-circle test in the unconstrained case because s lies in the interior of its circumcircle. In the constrained case, $\triangle pqr$ survives the test as s is not visible to the its vertices.

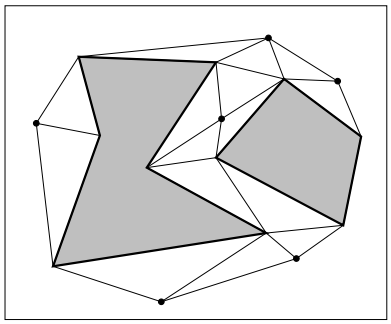


Figure 3.2: Example of CDT of the open space. Triangles inside the holes are deleted.

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 3.1).

In some applications, we are interested in the CDT of the open space (Figure 3.2. Specifically, when the input data contains polygonal holes whose interiors are of no interest to us, we sometimes want to remove the triangles inside these holes from the CDT. This is beneficial for certain simulations that involve impenetrable regions. We are going to see one such application in Chapter 4.

3.3 Disk Based Method

3.3.1 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 internal-memory DT algorithm to compute the DT for each block;

3. **Merge:** Stitch together DTs 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 that minimizes the number of unnecessary I/O operations during merging and conforming.

We now give details on the four steps. Section 3.3.2 describes the first three steps, which compute $DT(S)$. Section 3.3.3 describes the last step, which enforces

the constraints. Lastly, Section 3.3.4 shows how to compute the CDT of the open space by removing triangles in polygonal holes.

3.3.2 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 3.3). 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 3.4 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 internal-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 3.4). 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 3.3.1 *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:

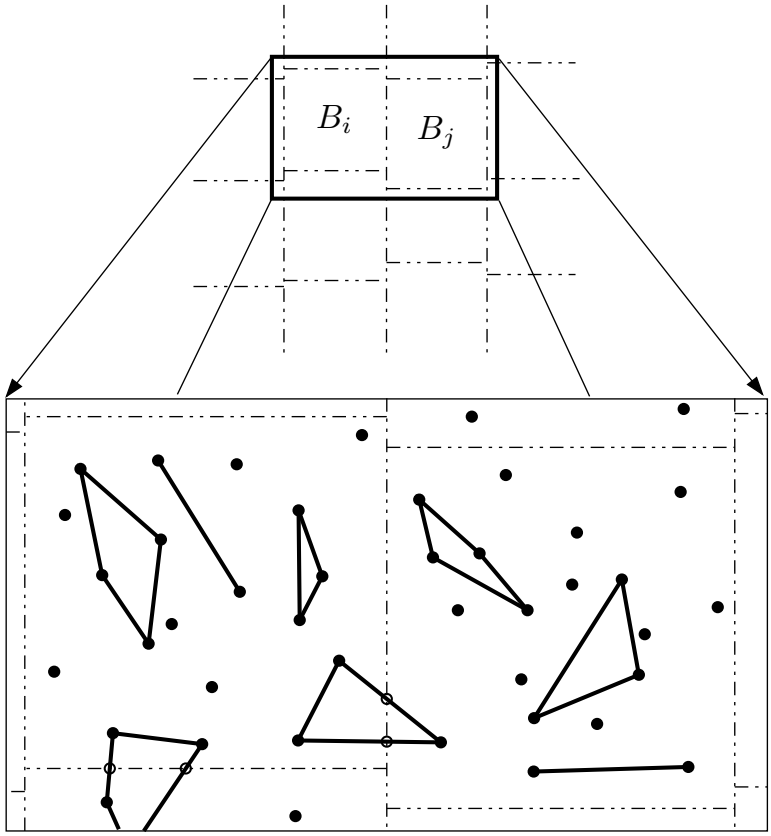


Figure 3.3: The dividing step: partition the input PSLG into blocks of roughly equal size so that each fits into the memory. In the zoomed-in picture, small circles indicate Steiner points created at the intersections of input segments and block boundaries.

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,

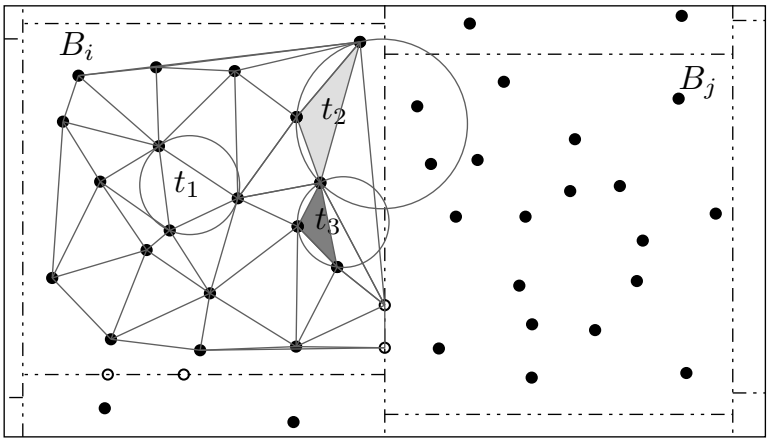


Figure 3.4: The conquering step: compute DT in each block. The triangle t_1 is safe, and both t_2 and t_3 are unsafe.

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 the DT that t is also a triangle in $DT(S_i)$, because $S_i \subseteq S$. So we have the next lemma:

Lemma 3.3.2 *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 3.3.2 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 3.5).

Lemma 3.3.3 *A triangle t is a crossing triangle in $DT(S)$ if and only if t is also a crossing triangle in $DT(S')$.*

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. ■

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

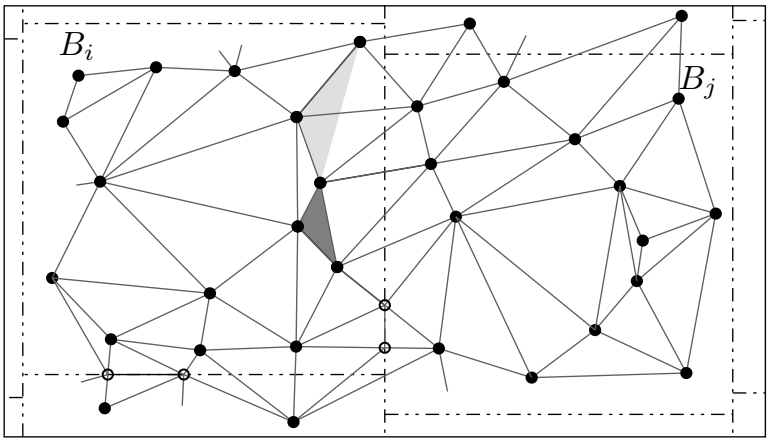


Figure 3.5: The merging step: compute the DT of the seam. After merging B_i and B_j, t_2 becomes invalid and is deleted, but t_3 remains valid.

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 3.3.3, 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 3.3.3. 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 $DT(S')$. ■

Now let U denote the set of unsafe triangles for all the blocks. We can sort the triangles in U and $DT(S')$ in lexicographical 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 3.3.5 *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 3.3.3, all the crossing triangles in $DT(S)$ are obtained in the merging step by computing $DT(S')$. Non-crossing triangles are further divided into safe and unsafe triangles. By Lemma 3.3.1 and 3.3.2, all the safe triangles in $DT(S)$ are reported in the conquering step. From Lemma 3.3.4, 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. ■

3.3.3 Inserting Constraint Segments

Now we add the constraints segments back and compute $CDT(S, K)$. To do this efficiently, we need the following result [42]:

Lemma 3.3.6 *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,*

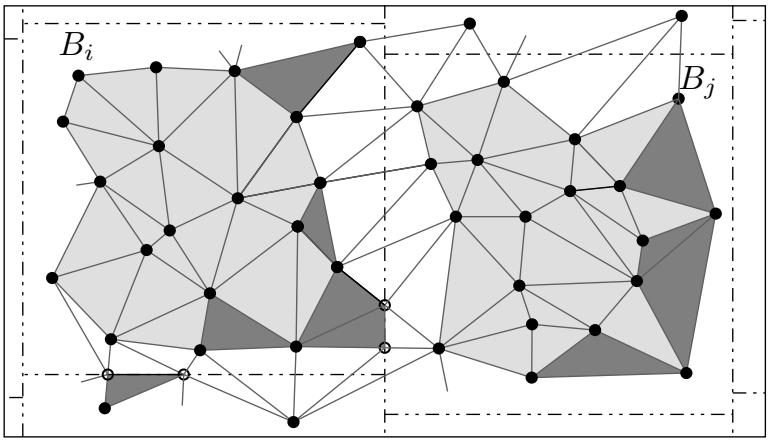


Figure 3.6: The DT of input data points. There are three types of triangles: triangles in light shade are the safe triangles obtained in the conquering step; triangles in dark shade are the valid unsafe triangles that are preserved during the merging step; the rest are crossing triangles.

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 3.7).

This lemma says that adding an new constraint segment $\overline{pq}$ into an existing CDT only affects those triangles overlapping $\overline{pq}$ Figure 3.7. 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 internal-memory CDT algorithm. The result is a new triangulation Q'. By Lemma 3.3.6, loading Q is sufficient, because all segments in K_i lie B_i according

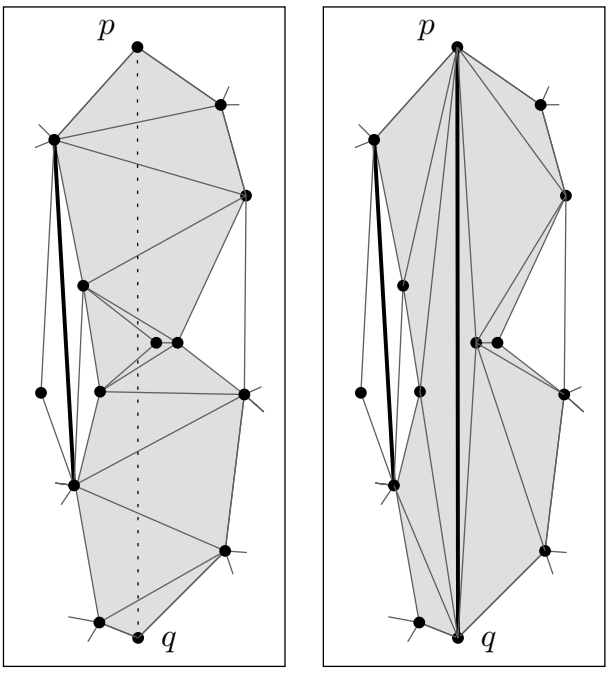


Figure 3.7: Inserting constraint segment $\overline{pq}$ only requires re-triangulating grey region consisting of triangles intersecting with $\overline{pq}$.

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 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 3.8 illustrates the result of conforming the triangulation to K_i.

The theorem below shows the correctness of our CDT algorithm.

Theorem 3.3.7 *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 3.3.5, $T_0 = DT(S)$ 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}.

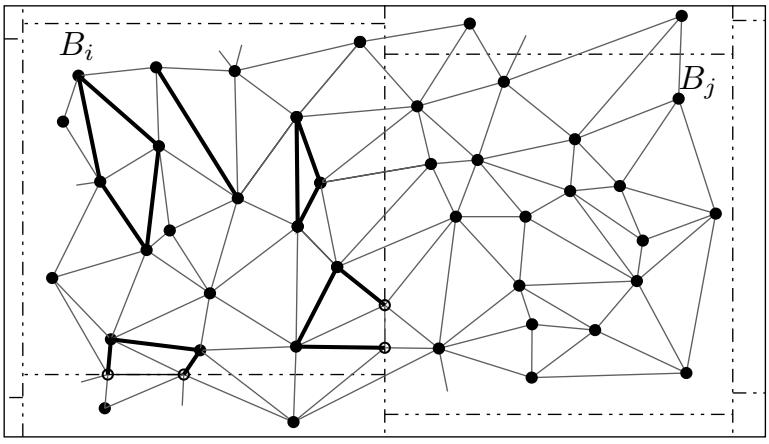


Figure 3.8: The conforming step: insert constraint segments K_i from B_i and update the triangulation.

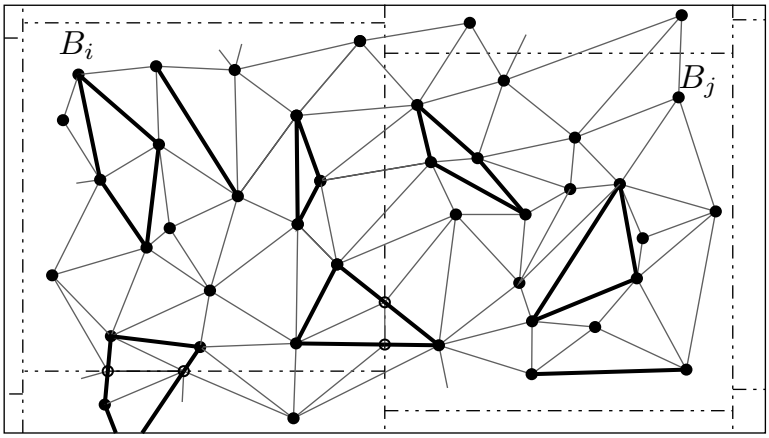


Figure 3.9: The final CDT of the input PSLG.

Lemma 3.3.6 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 3.10 shows the final $CDT(S, K)$ after the insertion of all the segments in K.

3.3.4 Removing Triangles in Polygonal Holes

Some applications require to compute the CDT of the open space. In this case, we need to remove the triangles inside the polygonal holes. This task is simple for main memory algorithms. The conventional method [39] empties each polygonal hole by locating one triangle inside it, and then expand from the source triangle by a breadth-first search to remove the triangles until the progress is blocked by the boundary of the hole. The problem become much subtler when the triangulation does not fit into the memory, as in this case, the breadth-first search can lead to heavy I/O operations. In view of the difficulty of removing triangles from the overall CDT, we propose a method that deletes triangles as we process each individual block. Specifically, we register for each block the polygonal holes it contains in the dividing step. In the conforming step, we remove the triangles inside the polygonal holes of a block right after we insert all the constraints of this block. As we have seen in 3.3.3, the triangulation involved in the conforming each block can be stored in main memory. Hence, we can apply any main memory technique for the removal of triangles.

Apparently, there may be holes that overlap different blocks. In this case, we introduce artificial boundaries and additional points to split the hole into components each of which fits squarely into some block.

3.4 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

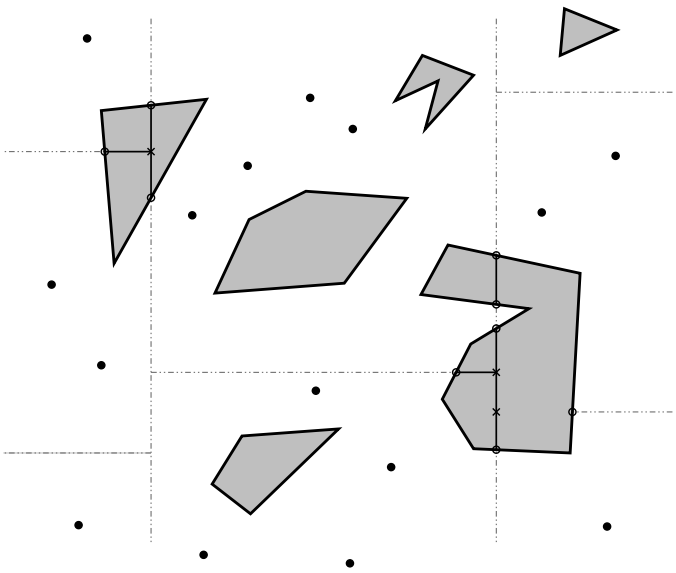


Figure 3.10: The final CDT of the input PSLG.

Table 3.1: 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

listed in Table 3.1 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.

3.4.1 Divide and Conquer

The input to our external-memory 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 $N = r^2M$ 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 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 3.3.2, 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'.

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' = ∅
2: U = ∅
3: for all blocks B_i do
4:    compute DT(S_i)
5:    for all t ∈ 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'
```

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.

3.4.2 Merge and Conform

The merging step of our algorithm computes all the crossing triangles and valid unsafe triangles in $DT(S)$. By Lemma 3.3.3, we can find all the crossing triangles from DT(S').

Lemma 3.3.4 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

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 ∈ 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 ∈ DT(S') ∩ U do
8:    report t to the final DT(S)
9: end for
```

perform another external-memory 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 3.3.3 briefly describes how to process the segments block by block and progressively update the triangulation to obtain $CDT(S, K)$. Let $K_i \subseteq K$ be the subset of segments in block B_i. Lemma 3.3.6 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 internal 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

[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

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 3.3.6. 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.

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**

Alternatively, one can choose lazy evaluation depending on the caches' capacity.

3.5 Experimental Evaluation

Our program is implemented in C++. For internal-memory DT/CDT, it uses Triangle [39], 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 external-memory algorithm, which, as we have mentioned in Section 2.2, appears to be only one in the literature with implementation and experimental studies. For CDT, since

there is no implemented external-memory algorithm, we compare our algorithm with TRIANGLE and test for scalability on large data sets.

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.

3.5.1 Delaunay Triangulation

Data Distribution

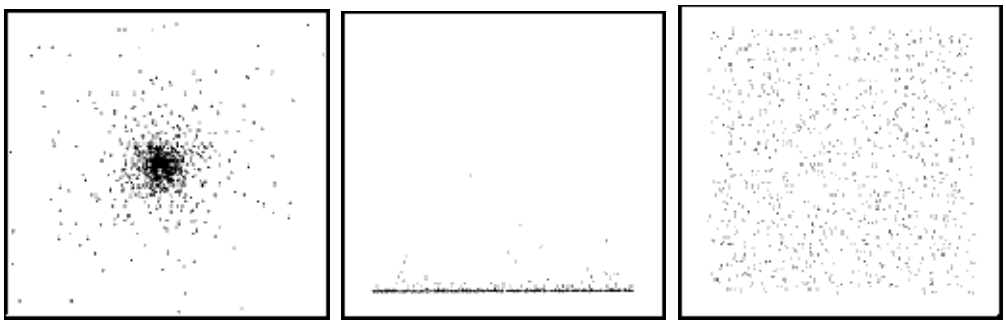
We ran our program on point sets with three different distributions: Kuzmin, Line Singularity and Uniform (Figure 3.11). These are standard distributions for evaluating the performance of DT algorithms [10, 11].

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

$$M(r) = 1 - \frac{1}{\sqrt{1+r^2}}, \tag{3.1}$$

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



Kuzmin Line Singularity Uniform

Figure 3.11: Data distributions for testing DT.

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

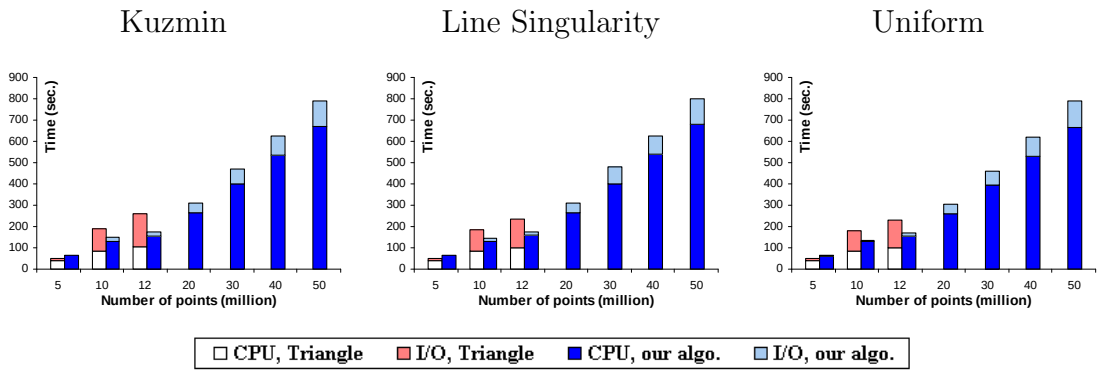
$$(x, y) = (\frac{b}{u - bu + b}, v). \tag{3.2}$$

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

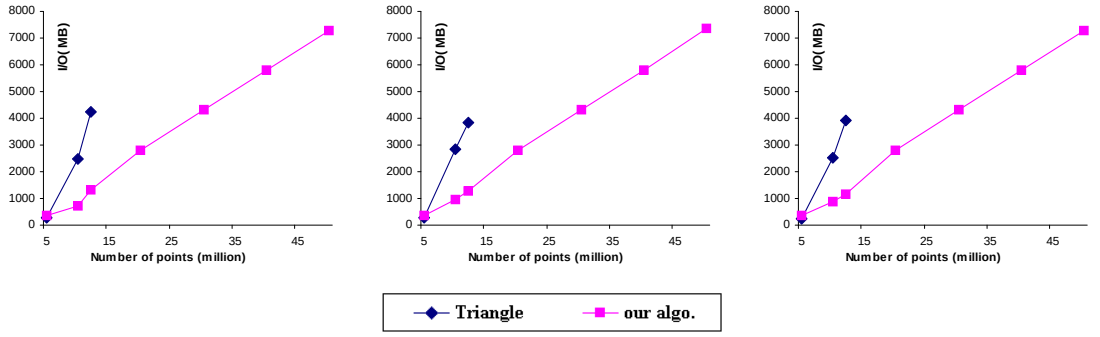
Both Kuzmin and Line Singularity are highly skewed distributions, and so standard partition techniques such as bucketing do not work well. For each distribution, we ran several experiments with different data size ranging from 5 to 80 million points. The data size of 12 million points was chosen because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

Results

Figure 3.12*a* compares the running time of TRIANGLE and our algorithm for all three distributions. We consider both CPU time and I/O time.



(a) Running time.



(b) I/O cost.

Figure 3.12: Running time and I/O cost comparison of DT algorithms on three data distributions.

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

From Figure 3.12a, we see that our external-memory 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 internal-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 external-memory algorithm is

much smaller and grows gently with the data size. This is attributed to the efficient data management by our algorithm. Figure 3.12*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 3.12*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 total running time, as a result of effective I/O management, and can process much larger data sets.

We also compared our algorithm with another external-memory DT algorithm by Kumar and Ramos [30]. Kumar and Ramos' algorithm is provably efficient. Unfortunately, we cannot obtain their running code, so we use their experimental result [30] for comparison. All the data sets they used are generated with uniform distribution except one real data set with 79 million points. We do not have the real data set, so all comparison are performed on uniformly distributed data. 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 3.13. The reason, we believe, is that our data partitioning and merging methods are more effective and avoid processing the same data multiple times.

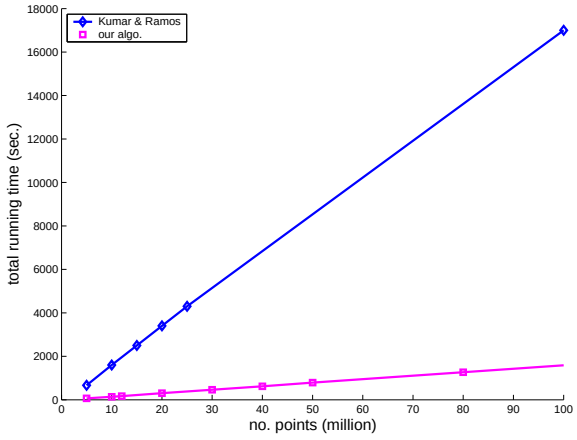
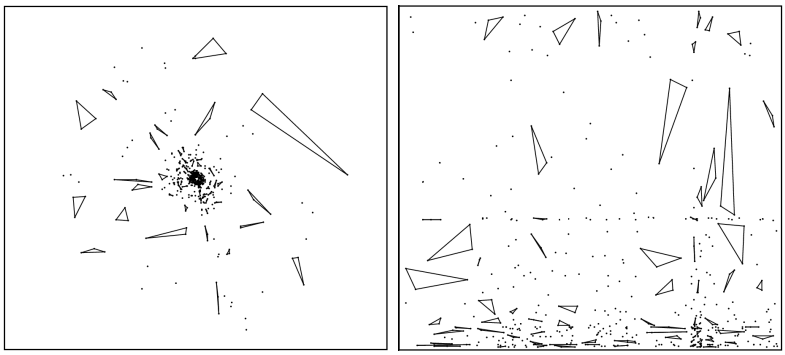


Figure 3.13: Comparison of our algorithm with a provably-good external-memory DT algorithm.



Kuzmin Line Singularity

Figure 3.14: Examples of generated PSLGs using different distributions.

3.5.2 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 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 $\sqrt{N/3}$ 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 3.14 for an example.

Line Singularity PSLG (Figure 3.14 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 (3.2).

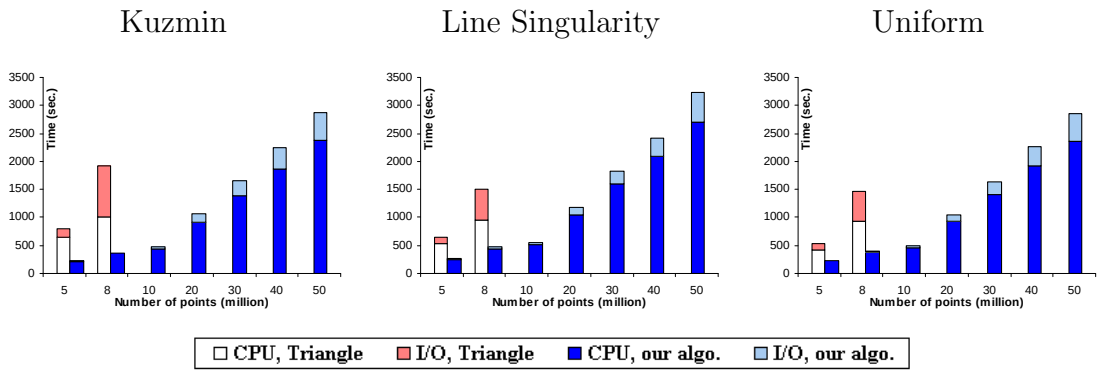
Uniform PSLG We uniformly and randomly partition the unit square into a grid of $N/3$ cells. In each cell, we sample three points and decide with probability α whether to create constraint segments to connect them.

Results

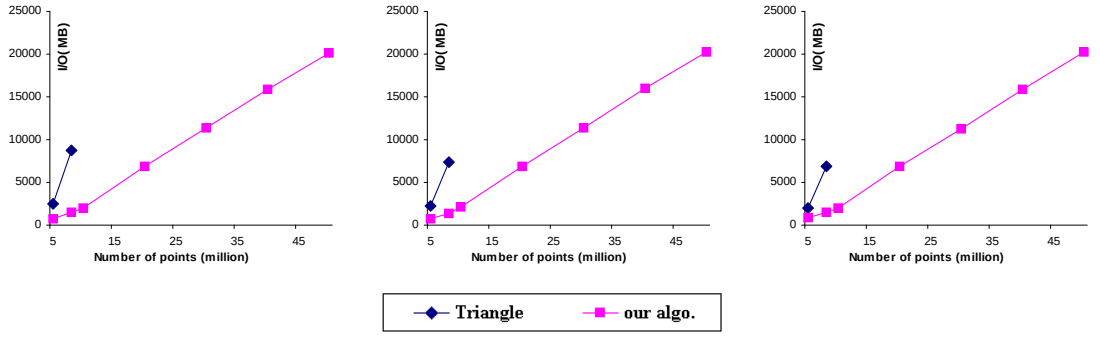
For CDT, we compare with Triangle only, since there are no practical external-memory algorithms (see Section 2.2). Our experiments consist of two parts. In the first part, we fix the segments to points ratio α, and vary the number of points N. In the second part, we fix N and vary α.

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 3.15) are organized in the same way as for DT.

As Figure 3.15 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 external-memory algorithm becomes even



(a) Running time.

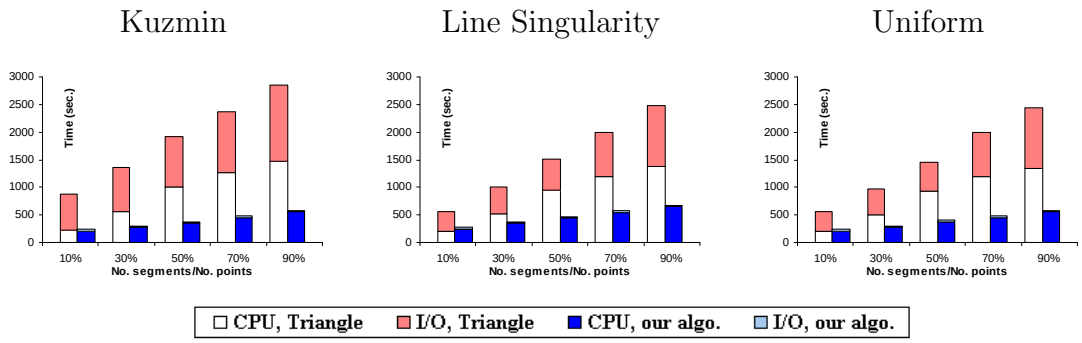


(b) I/O cost.

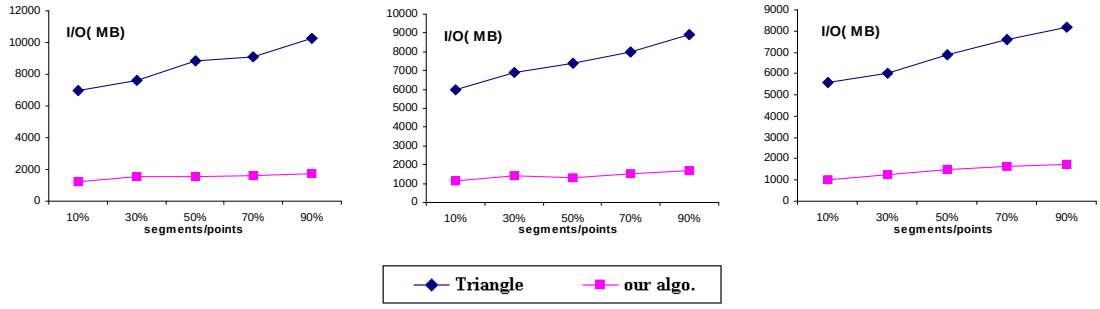
Figure 3.15: Running time and I/O cost comparison of CDT algorithms on three data distributions.

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 external-memory 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



(a) Running time.



(b) I/O cost.

Figure 3.16: Comparison between TRIANGLE and our algorithm on Kuzmin PSLGs with different segments/points ratios.

three distributions. As Figure 3.16 illustrates, for all three data distributions, 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.

3.6 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 external-memory 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.

CHAPTER 4

Obstructed Proximity Search

4.1 Introduction

One of the motivations for designing and implementing our CDT algorithm was to facilitate proximity search in the presence of obstacles. The basic operation of obstructed proximity search is to compute the geodesic shortest path between two given points. As we discussed in Section 2.3, there are two main approaches for computing the exact Geodesic shortest path—the *visibility graph search* and *continuous Dijkstra method*. The former approach constructs the visibility graph of the input vertices and runs Dijkstra's algorithm on the visibility graph to compute the shortest path. Unfortunately, the visibility graph can have $\Omega(n^2)$ edges, where n is the number of vertices in the input PSLG. The space requirement makes this approach impractical for any reasonably large data set. The continuous Dijkstra method achieves asymptotically optimal $O(n \log n)$ bound in running time and space. But the algorithm is too complex to implement and has a big constant factor.

In many applications, we are satisfied with an obstacle-avoiding path that is not necessarily the shortest, but reasonably short. This is usually accomplished by

using spanner graphs of the visibility graph. In this chapter, we focus on using the CDT as the spanner graph to solve the approximate geodesic shortest path problem. For convenience, let us call the length of the shortest path on the CDT between two vertices their CDT distance. Karavelas and Guibas [28] generalized the proof in [17] to prove that the CDT distance approximates the geodesic distance with a stretch factor of 5.08. However, the true stretch factors of both DT and CDT as spanner graphs are generally believed to be much smaller. The best-known lower bound for the worst-case stretch factor is $\pi/2$.

Computing the geodesic path is one of the basic operations for obstructed proximity search. Existing methods for processing obstructed proximity search queries construct local visibility graph to compute exact shortest geodesic path. They rely on the Euclidean distance as the lower bound to prune the irrelevant obstacles and sites. These methods suffer from that the Euclidean distance does not approximate geodesic distance well in general. Moreover they do not offer tradeoff between the optimality of the result and the computational cost. In this chapter, we propose new query processing methods based on the spanner property of CDT to overcome these problems.

This chapter is organized in the following way: in Section 4.2, we present experiments on real data sets to show that in practice the stretch factor of the CDT as spanner graph for the visibility graph is much better than the proven theoretical bound. In Section 4.3, we introduce a more efficient pruning strategy based on the CDT and describe methods that use the CDT as the preprocessing step to answer the approximate and exact obstructed k-nearest-neighbors and range queries.

Figure 4.1: Indonesian Archipelago

4.2 Experimental Evaluation

There is a large gap between the best-known worst-case stretch factor, $\pi/2$ of the CDT as the spanner graph of the visibility graph and the existing theoretical bound, 5.08. To show that the length of the shortest path between two vertices on CDT indeed approximates their geodesic distance very well, we conducted extensive experiments to compare the approximate shortest path and exact shortest path on real data sets. The data sets we use are taken from the map of the Indonesian archipelago, referring to Figure 4.1. There are over 14 thousand islands of Indonesia, ranging from a tiny speck on the map to the island of Sumatera which is approximately the size of California. Each island is represented by a simple polygon. In total, the map consists of 78000 polygon vertices, which makes it very expensive to construct the visibility graph of the whole archipelago. We chose three very different groups of islands as our data sets to ensure that our study of the approximation ratio is comprehensive. Each data set contains around 2500 polygonal vertices. The first data set is a dense distribution of tens of small islands (Figure 4.2); the second one is a very sparse distribution of tiny islands (Figure 4.3); and the last one is a dense distribution of medium-sized islands (Figure 4.4).

To compute the exact shortest path, we implemented Lee's $O(n^2 \log n)$ algorithm [32] using C++. We did not implement the asymptotically optimal $O(n^2)$ algorithm such as the ones in [6, 19, 43] because they are very complex and does not yield better performance in practice. We also used the GeoWin package of LEDA [2] for displaying the results.

As Figures 4.2*a*, 4.2*b*, 4.3*a*, 4.3*b*, 4.4*a*, and 4.4*b* suggest, the size of the visibility graph is significantly larger than that of the CDT. The entire open space is almost covered by the edges of the visibility graph. After the visibility graph and CDT are constructed, we can pick any vertex v, and run the Dijkstra's algorithm to compute the so-called single source shortest path (SSSP) tree, which represents the shortest paths from v to all other vertices. Figures 4.2*c*, 4.2*d*, 4.3*c*, 4.3*d*, 4.4*c*, and 4.4*d* are examples of SSSP trees on the visibility graph and CDT.

For each data set, we randomly pick 2000 pairs of vertices, and compute the exact and approximate shortest geodesic paths between every pair on the visibility graph and CDT respectively. The comparison of the exact and approximate geodesic distances are summarized in Figure 4.5.

From Figure 4.5, we see that the approximation ratio indeed never exceeds the conjectured worst-case bound $\pi/2$. Moreover, it seems that the CDT approximates particularly well at the narrow channels between obstacles. The approximation ratio for densely distributed data sets such as 1 and 3 is generally better than that for sparse distribution like 2. The reason is difficult to ascertain with certainty, but one can make educated inferences. A geodesic path consists of a series of line segment links. The ends of each link are *visible* to each other. The overall approximation ratio for the whole path depends very much on how well these links are approximated. If a link passes through a narrow channel, it is likely to cut through many thin triangles in the CDT whose long edges form small angles

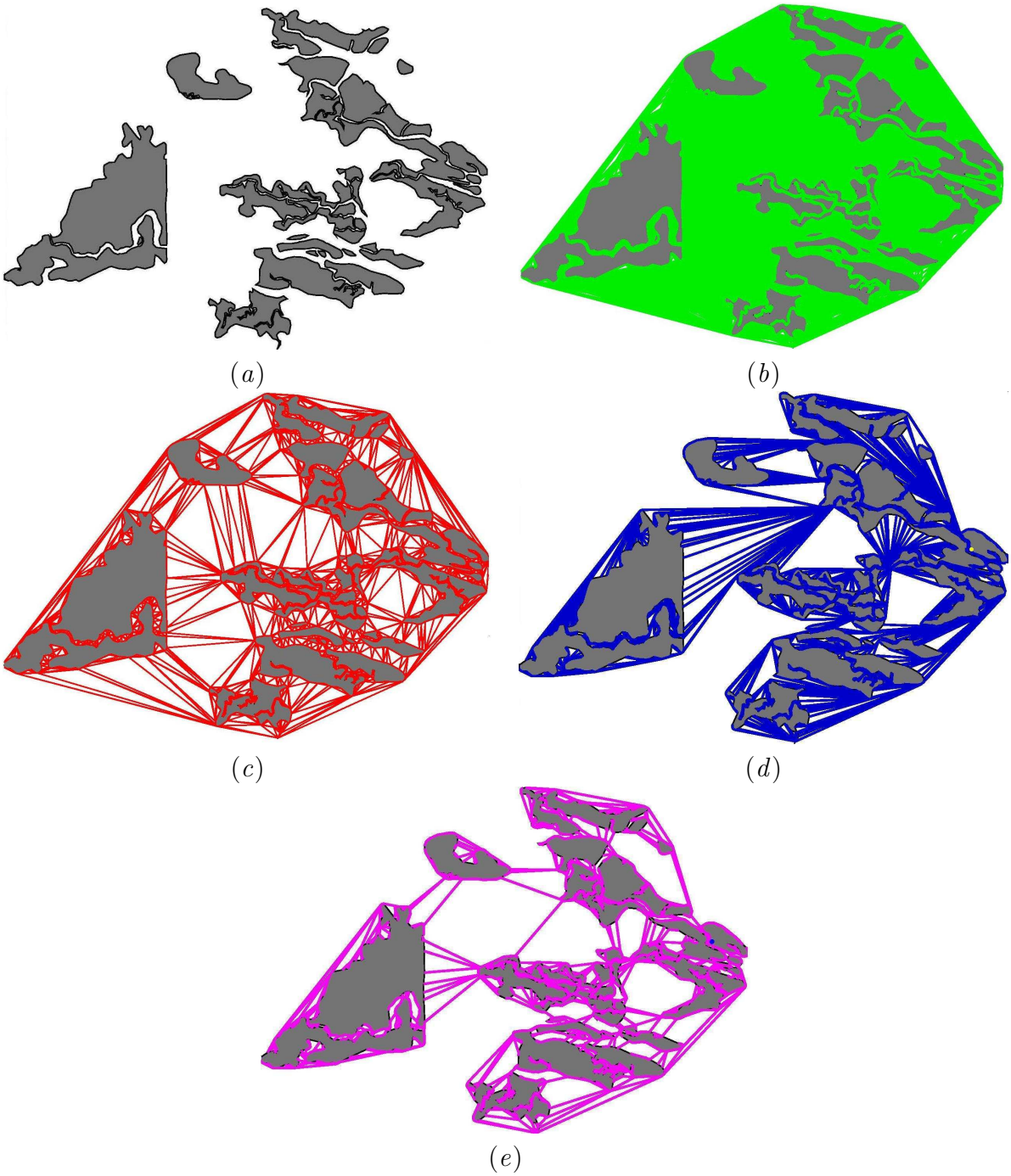
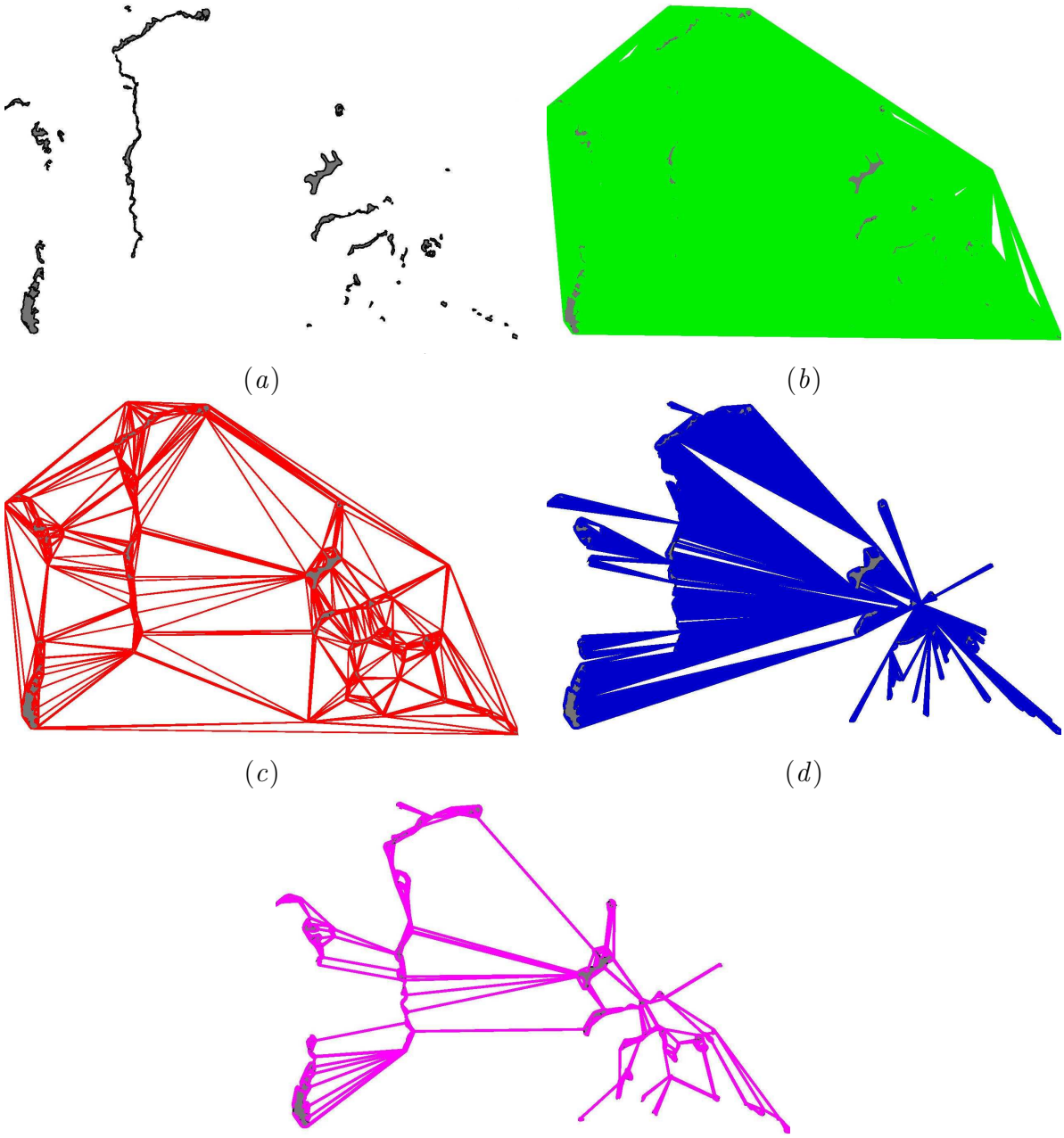
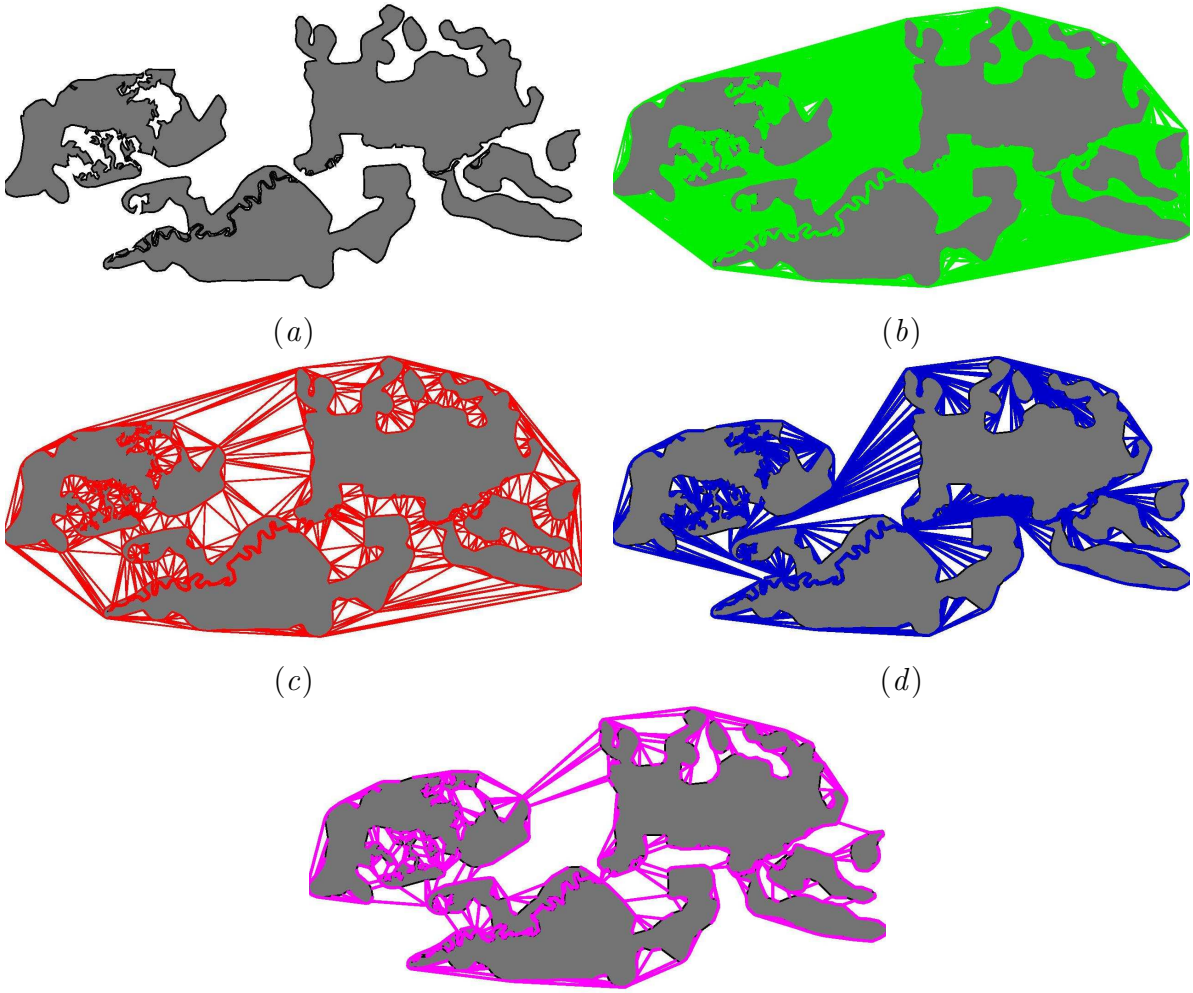


Figure 4.2: Data Set 1: (a) a group of islands; (b) The visibility graph; (c) The CDT of the open space; (d) An SSSP tree rooted at an input vertex based on the visibility graph; and (e) the SSSP tree rooted at the same vertex based on the CDT.



(a) (b) (c) (d) (e)

Figure 4.3: Data Set 2.



(a) (b) (c) (d) (e)

Figure 4.4: Data Set 3.

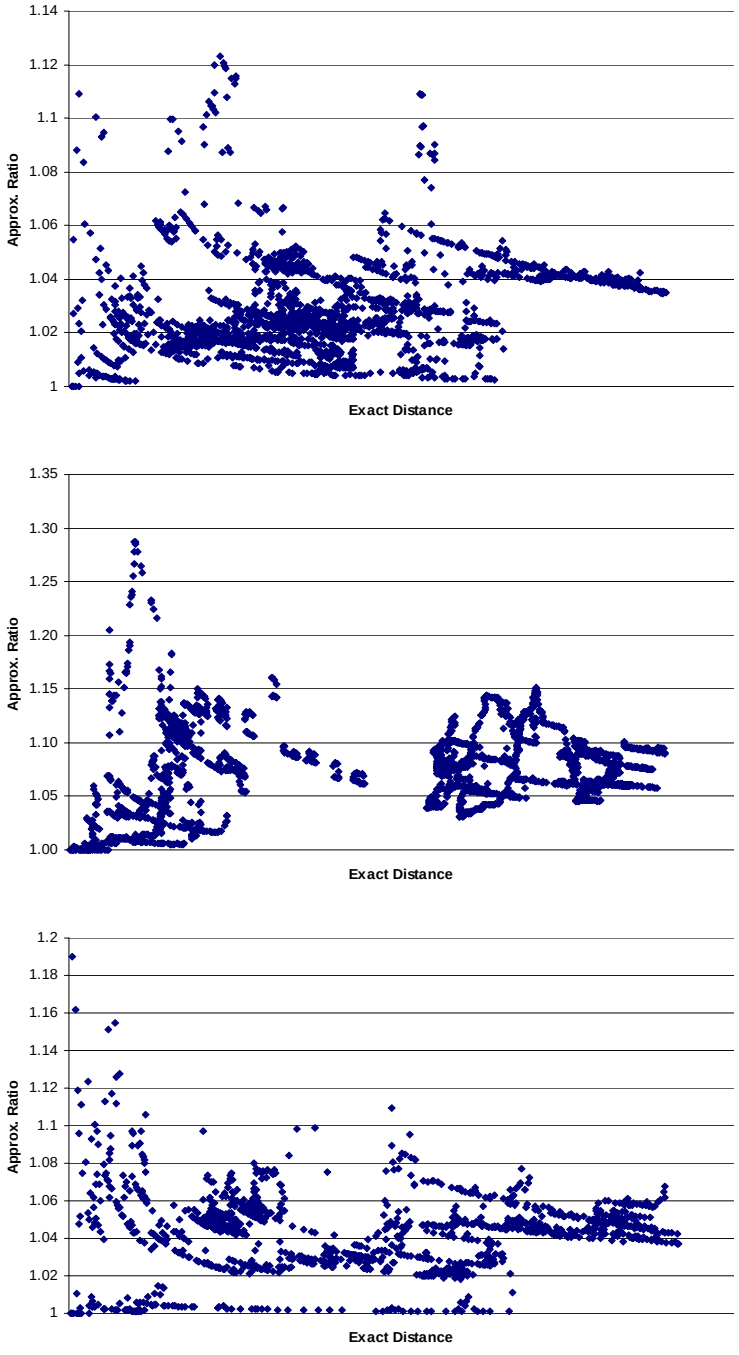


Figure 4.5: The approximation ratio for the three data sets

with the straight-line link. As a result, travelling along these edges offers good approximation to the link. In contrast, when a link passes through an open area, the edges of those triangles it intersects often form much larger angle with the link. A path along these edges does not approximate the link as well.

4.3 Obstructed Proximity Search Queries

Processing obstructed proximity search queries involves computing the geodesic distance between vertices by constructing and searching the visibility graph. Due to the quadratic complexity of the visibility graph, the global visibility graph cannot be pre-materialized except for very small data sets. One practical approach to circumvent this difficulty is to build a local visibility graph of only the relevant data online. There are two things we should keep in mind about this approach. First, the approach does not always work. There are certainly situations where even the local visibility graph of only the relevant data is still too large. In this case, there is not much we can do. It is simply almost impossible to compute the exact geodesic distance. Second, it is very unlikely that we can determine precisely what data are relevant beforehand. The best we can do is therefore to prune all the data that we can be absolutely certain that are irrelevant. Thus the effectiveness of this approach can only be improved by developing better pruning strategy.

The pruning strategies of existing work [44, 45] are all based on using the Euclidean distance as the lower bound. The argument is that the Euclidean distance between two points must be smaller than their geodesic distance. Thus if we are looking for objects that are within r in geodesic distance to some query point p, it is safe to prune anything whose Euclidean distance to p is larger than r. While these pruning strategies are simple, they are often insufficient and inefficient. As

we stated earlier, the main weaknesses of these methods are the following.

- The Euclidean distance does not approximate geodesic distance well in general. This often results in very large visibility graph consisting mostly of irrelevant data.
- In order to filter out irrelevant data, these methods need to perform costly Euclidean range query. This is especially bad for incremental algorithms, *e.g.*, k-ONN in [44, 45].
- They do not offer tradeoff between the optimality of the solution and the computational cost.

In this section, we propose CDT-based pruning strategy to address the above three weaknesses. Our strategy is inspired by the spanner property of CDT. In contrast to the visibility graph which has quadratic space complexity, the CDT is a planar graph and only takes $O(n)$ space and therefore can be fully pre-materialized. We demonstrate our pruning strategy by describing methods that process k-ONN and obstructed range queries efficiently.

4.3.1 Obstructed Range Query

Given a query point p and a range r, the obstructed range query returns all the sites that are within geodesic distance r to p. Computing the geodesic distance between two vertices requires computing the visibility graph which is expensive. Thus we need to exclude as many as possible irrelevant sites and obstacles in the construction of the visibility graph.

The main advantage of the CDT-based pruning strategy for processing obstructed range query is that the CDT distance offers a constant-bounded approximation to the geodesic distance. This is in contrast to using Euclidean distance

as lower bound where the approximation can be arbitrarily bad. As a result, the CDT-based pruning strategy can greatly reduce the size of the local visibility graph used to compute exact geodesic distance.

Our CDT-based range query processing method consists of the following steps:

1. search the CDT to report sites that are within l_1 in CDT distance to p and obstacles each of which has at least one vertex that is within l_2 in CDT distance to p;

2. construct the visibility graph based on the reported sites and obstacles;

3. search the visibility graph to find all the sites that are within geodesic range.

Steps 2 and 3 require no explanations. Step 1 prunes irrelevant sites and obstacles. Below we show how to set the CDT search range l_1 and l_2 and prove our pruning strategy is correct.

First we treat the simple case where the query point coincides with some vertex of the CDT. In this case, we set $l_1 = rt$, where t is the stretch factor of CDT as spanner graph, $\pi/2 \leq t < 5.08$. It is safe to use l_1 to prune the sites because any site s that is with in geodesic range r to p cannot be more than rt in CDT distance away from p. How to set l_2 to prune obstacles? One may be tempted to think that if all the vertices of an obstacle have CDT distances longer than rt, the obstacle can be pruned. Unfortunately, this is not the case. Refer to Figure 4.6 for an illustration. In this example suppose that s is a site that is within CDT range rt to the query point p, and Obstacle o has all its vertices outside the rt CDT distance range. The lower path is the true shortest geodesic path from p to s, but without the presence of o, we have a shorter upper path, which actually cuts through the obstacle o.

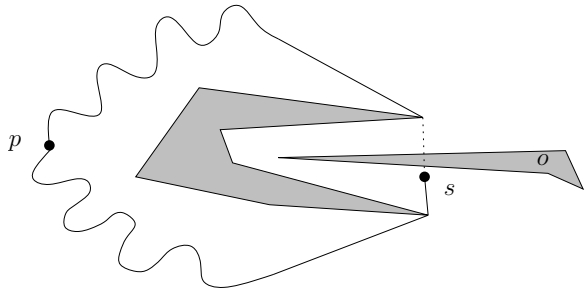


Figure 4.6: Obstacle o having all its vertices out of rt CDT distance range still affects the geodesic path.

The reason for this counter-example is that the geodesic distance from p to an obstacle can be shorter than the geodesic distance to any of its vertices. By the geodesic distance from p to an obstacle, we treat the obstacle as a point set, and take the minimum of the geodesic distances from p to all points of the point set. One may attempt to fix this problem by introducing additional edges in order to also approximate the distance to obstacles. But that greatly complicates the underlying CDT structure and its construction, and can lead to large increase in space requirement. Below we show that we can still prune obstacles based on CDT alone by properly setting the value of l_2.

Lemma 4.3.1 *Let A and B be two distinct paths that connect two points p and q on the plane. For any segment $\overline{x_1x_2}$ that is completely caught in between A and B (the segment can touch A or B), the length of the segment is no longer then half the total length of A and B (Figure 4.7).*

PROOF: Extend both ends of the segment to meet A or B at x_1' and x_2'. The straight-line segment $\overline{x_1'x_2'}$ is no longer than half the total lengths of paths A and B because it is the shortest among all paths between them. ■

Theorem 4.3.2 *Let s be a site that is within $l_1 = rt$ in CDT distance to p. Suppose all the vertices of an obstacle o have CDT distances to p larger than $rt(t+3)/2$.*

(1) *If the true geodesic distance between p and s is no larger than r, removing o CANNOT affect the shortest geodesic path from p to s; and* ***(2)*** *If the geodesic distance between p and s is larger than r, removing o can NOT reduce the geodesic distance from p to s to less than r.*

PROOF: **(1)**: In this case, the geodesic distance between s and p is no larger than r. Assume to the contrary that the removal of Obstacle o as described in the theorem can indeed create a new path from p to s that is no longer than the true geodesic shortest path. This new path must intersect the deleted obstacle. Consider the first segment of the obstacle that intersects this new path from p to s. The segment must have one end that is caught in between the new path and the true shortest geodesic path. Denote the intersection and the segment end by x and e. Refer to Figure 4.8. According to the assumptions, both the shortest geodesic path and the new path between p and s are shorter than r. Hence the segment $\overline{xe}$ must be shorter than r by Lemma 4.3.1. The geodesic distance between p and x is also less than r. Therefore the geodesic distance between p and e must be less than $2r$. By the spanner property of the CDT, the CDT distance between p and e must be smaller than $2rt < rt(t+3)/2$. But this contradicts to the assumption on the CDT distance from p to the obstacle's vertices.

(2): Still consider Figure 4.8. Now suppose the geodesic distance between s and

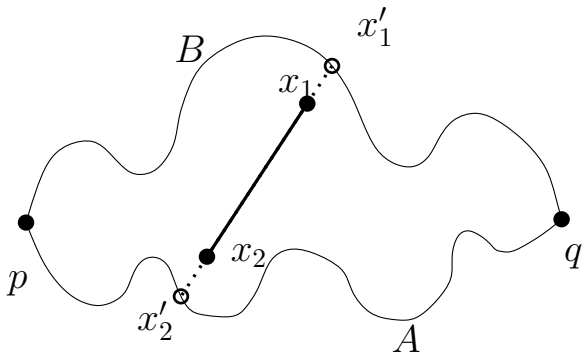


Figure 4.7: $\overline{x_1x_2}$ is shorter than half the total length of paths A and B

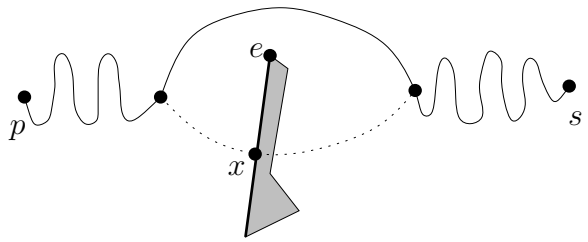


Figure 4.8: The shortest geodesic path (solid) and a shorter path that cuts through the removed obstacle (dotted)

p is larger than r. Assume to the contrary that the removal of the obstacle creates a new path between s and p that is shorter than r. The geodesic distance from p to x must be shorter than r. Since s survives the pruning, the geodesic shortest path from p to s is no longer than rt. Thus the total length of the shortest geodesic path and the new path between p and q is shorter than $rt + r$. By Lemma 4.3.1, the segment $\overline{xe}$ is shorter than $(rt+r)/2$. Combining the inequalities, the geodesic distance between p and e is less than $r(t+3)/2$. Hence the CDT distance from p to e must be less than $rt(t+3)/2$. However, this is a contradiction to the assumption on the CDT distances of the vertices of the obstacle to p. ■

By Theorem 4.3.2, it is safe to set l_2 to be $rt(t+3)/2$. The idea behind Theorem 4.3.2 is that if removing an obstacle can shorten the geodesic distance between a query point and a site which are already geodesically close, the obstacle must have one vertex that is geodesically close to the query point.

Next we generalize the above solution to handle arbitrary query point in the open space. When the query point p does not coincide with any vertex of the CDT, the spanner property of CDT does not apply to p directly. To overcome this problem, we locate the triangle in the CDT that contains p. Let v be the vertex of the triangle that is the closest to p; and denote their Euclidean distance by d. Then we search the CDT as in Step 1 from v with the geodesic range parameter r

being replaced by $r' = r + d$, and setting $l_1 = r't$, and $l_2 = r't(t+3)/2$ accordingly. It is not difficult to verify that all the candidate sites are captured because for any site s that is less than or equal to r in geodesic distance to p, the geodesic distance between s and v is no greater than $r + d$. To ensure the correctness of pruning obstacles, we need the following theorem which is akin to Theorem 4.3.2.

Theorem 4.3.3 *Let p, v, r, r', and d be as described above. Let s be a site that is within $l_1 = r't$ in CDT distance to v. Suppose all the vertices of an obstacle o have CDT distances to v larger than $r't(t+3)/2$. **(1)** If the true geodesic distance between p and s is less than or equal to r, removing o can Not affect the shortest geodesic path from p to s; and **(2)** if the true geodesic distance between p and s is larger than r, removing o can NOT reduce the geodesic distance from p to s to less than r otherwise.*

PROOF: We still refer to Figure 4.8 for both cases.

(1): In this case, both the shortest geodesic path and the new path between p and s are no longer than r. By Lemma 4.3.1, the segment $\overline{xe}$ is shorter than r. The geodesic distance between p and x is also smaller than r. Hence the geodesic distance between p and e must be shorter than $2r$. By triangle inequality, the geodesic distance between v and e must be shorter than $2r + d$. According to the spanner graph property, the CDT distance between v and e must be smaller than $(2r + d)t \leq 2r't < r't(t + 3)/2$, which is a contradiction to the assumption on the CDT distance of the vertices of the removed obstacle to v.

(2): Since s is not pruned, the geodesic distance between s and v is less than $r't = (r + d)t$. By triangle inequality, the geodesic distance between s and p must be less than $(r + d)t + d$. That the new path from p to s is shorter than r implies that the segment $\overline{xe}$ is shorter than $((r+d)t+d+r)/2$. We also know the geodesic distance between p and x is smaller than r. So the geodesic distance between e and

p is smaller than $(r+d)(t+1)/2+r$. Again by triangle inequality, the geodesic distance between e and v must be smaller than $(r+d)(t+3)/2 = r'(t+3)/2$. Therefore, the CDT distance between e and v can be no larger than $r't(t+3)/2$. Again, a contradiction has been reached. ■

4.3.2 Obstructed k-Nearest-Neighbors Query

Given a query point p and a number k, the obstructed k-ONN query returns the k nearest sites in geodesic distance to p. We process the k-ONN query in a similar way as we handle the range query. The k-ONN query is slightly harder than the range query in the sense that we do not have a predetermined geodesic bound with which we can prune irrelevant sites and obstacles. However, such a bound is not difficult to obtain. First let us assume the query point coincides with one of the vertices of the CDT. In this case, we search CDT from p to find the k nearest neighbors to p in CDT distance. Let s be the k-th nearest neighbor; and denote by r the CDT distance between p and s. We use r as the geodesic range, set l_1 and l_2 according to r, and follow exactly Step 1 of processing range query to prune obstacles and sites. Then we construct the visibility graph of all the sites and obstacles that survived the pruning and search for the k nearest neighbors in the visibility graph. All the true k geodesic nearest neighbors will be captured in range because their geodesic distances to p can be not larger than r.

When the query point does not coincide any CDT vertex, we locate the vertex v as described above in the range query subsection. Denote by d the Euclidean distance between p and v. We search the CDT to find the k nearest neighbors to v in CDT distance. Let s be the k-th nearest neighbor to v; and denote their CDT distance by r_0. Setting $r = r_0 + d$, by triangle inequality, all of these k sites

are within r in geodesic distance from p. Then we exclude irrelevant sites and obstacles using the pruning step of processing range query with arbitrary query point. Finally we build the visibility graph and search for the geodesic k nearest neighbors, the same way as for the case where p coincides some CDT vertex.

In addition to providing constant-bounded approximation to the visibility graph, our CDT-based method for processing k-ONN also has the advantage that instead of invoking range queries to incrementally discover the local visibility graph, it performs the graph search on CDT to directly compute the local visibility graph within which the k nearest neighbors can be found. Recall that the k-ONN algorithms in [44, 45] are all incremental. They need to perform Euclidean range query, and re-construct the visibility graph every iteration they enlarge the search space.

Finally processing obstructed proximity search queries based on the CDT can offer a tradeoff between the optimality of the result and the computational cost. This is valuable because often we do not want and optimal solution that takes hours to compute, but a quick and reasonably good solution. Moreover, sometimes the shortest geodesic path is even impossible to compute due to the quadratic complexity of the visibility graph. In such cases, we propose to use CDT distance as an intermediate result and let the user decide whether he wants to continue for the optimal solution. For example, imagine that some user issues a query to find the nearest hospital in an obstructed domain. The spatial database system can first return the nearest hospital H in CDT distance and tell the user, "this is a hospital that is at most 20 miles from you. The true nearest hospital will not be much closer than this one. Do you want to find it?" Based on the CDT distance to H, the system can also report to the user the estimated time required to construct the visibility graph and find the true nearest hospital. Such a spatial database system will be of great values in many applications.

CHAPTER 5

Conclusion

DT and CDT are fundamental geometric data structures that have broad applications. However there has been limited work devoted to practical external-memory algorithms for DT/CDT. An I/O-efficient algorithm has to partition the data and exploit the locality of the DT and CDT. The major challenge is therefore how to efficiently merge sub-solutions of the partitions into the whole triangulation.

One of our motivations for designing large-scale CDT algorithm is to facilitate obstructed proximity search. Obstructed proximity search in database systems has recently emerged as a new research frontier. Its main difficulty lies in how to prune irrelevant data to limit the search space. Existing pruning strategies all use Euclidean distance as the lower bound, and are insufficient for many applications.

5.1 Summary of Main Results

We have presented an efficient external-memory algorithm for DT/CDT. The highlight of our algorithm is a merging step which is completely combinatorial and avoids all heavy geometric computations. This is made possible by a precise characterization of the set of triangles involved in merging. We have tested the algorithm extensively for both DT and CDT. Experimental results show that for DT, our algorithm outperforms a provably good external-memory algorithm by roughly an order of magnitude. For CDT, which has no previously implemented external-memory algorithms, we show experimentally that our algorithms scales up well for large databases.

We also demonstrated an interesting application of the CDT to processing obstructed proximity search such as k-ONN and range queries in spatial databases. Our method is based on the spanner graph property of the CDT. There is a large gap between the proven worst-case lower and upper bound for the stretch factor. So we performed extensive experiments on real-life data sets to show that the CDT indeed approximates the visibility graph very well in practice. We introduced a new pruning strategy for obstructed proximity search and demonstrated how it can be applied successively in solving the k-ONN and range queries.

5.2 Future Work

In the future, our work can be extended in the following ways:

- Many of the main theorems and lemmas in Section 3.3 can be generalized to DT/CDT with dimensions higher than two as well. It is intriguing to design and implement algorithms based on our method for DT/CDT of 3D or higher dimensions and study their performance in practice.

- Currently our algorithm does not support online updates of the triangulation, *e.g.*, insertion and deletion of points or segments. Efficiently updating large-scale triangulation online is a very challenging problem and has expansive applications in areas like data interpolation and proximity search.

- A growing new trend in spatial query processing is to handle continuously moving points. It is interesting to investigate how to utilize the spanner graph property of CDT to process spatial queries of moving points in obstructed domains.

BIBLIOGRAPHY

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

[2] Leda—library for efficient data types and algorithms.

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

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

[5] S. R. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. H. M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *European Symposium on Algorithms*, pages 514–528, 1996.

[6] T. Asano, T. Asano, L. J. Guibus, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1:49–63, 1986.

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

[10] 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.

[11] 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.

[12] L. P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the 2nd Symposium on Computational Geometry*, pages 169–177, 1986.

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

[14] L. P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained delaunay meshing. In *Proceedings of the 1st Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, 1997.

[15] P. Cignoni, C. Montani, and R. Scopigno. Dewall: A fast divide and conquer delaunay triangulation algorithm in e^d. *Computer-Aided Design*, 30:333–341, 1998.

[16] 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.

[17] D. P. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987.

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

[19] H. Edelsbrunner. *Algorithms in Combinatorial Geometry. EATCS Monographs on Theorectical Computer Science*, volume 10. Springer-Verlag, 1987.

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

[21] S. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

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

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

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

[25] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivision and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985.

[26] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–57, 1984.

[27] J. Hershberger and S. Suri. An optimal algortihm for euclidean shortest paths in the plane. 1995.

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

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

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

[31] C. L. Lawson. Software for c^1 surface interpolation. pages 161–194, 1977.

[32] D. T. Lee. Proximity and reachability in the plane. Technical report, Department of Electrical Engneering, University of Illinois, Urbana, 1978.

[33] D. T. Lee and B. J. Lin. Generalized delaunay triangulation for planar graphs. *Discrete and Computational Geometry*, 1:201–207, 1986.

[34] D. T. Lee and B. J. Schachter. Two algorithms for constructing the delaunay triangulation. *International Journal of Computer and Information Sciences*, 9:219–242, 1980.

[35] J. S. B. Mitchell. Geometric shortest paths and network optimization, 1998.

[36] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *Pro-*

ceedings of the 11th Annual Symposium on Computational Geometry, pages 274–283, 1996.

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

[38] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

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

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.