Simple, Faster Kinetic Data Structures

by

Zahed Rahmati
B.Sc., University of Isfahan, 2007
M.Sc., Sharif University of Technology, 2010

A Dissertation Submitted in Partial Fullfilment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Simple, Faster Kinetic Data Structures

by

Zahed Rahmati
B.Sc., University of Isfahan, 2007
M.Sc., Sharif University of Technology, 2010

Supervisory Committee

ABSTRACT

Proximity problems and point set embeddability problems are fundamental and well-studied in computational geometry and graph drawing. Examples of such problems that are of particular interest to us in this dissertation include: finding the closest pair among a set P of points, finding the k-nearest neighbors to each point $p \in P$, answering reverse k-nearest neighbor queries, computing the Yao graph, the Semi-Yao graph and the Euclidean minimum spanning tree of P, and mapping the vertices of a planar graph to a set P of points without inducing edge crossings.

In this dissertation, we consider so-called kinetic versions of these problems, that is, the points are allowed to move continuously along known trajectories, which are subject to change. We design a set of data structures and a mechanism to efficiently update the data structures. These updates occur at critical, discrete times. Also, a query may arrive at any time. We want to answer queries quickly without solving problems from scratch, so we maintain solutions continuously.

We present new techniques for giving kinetic solutions with better performance for some of these problems, and we provide the first kinetic results for others. In particular, we provide:

- a simple kinetic data structure (KDS) to maintain all the nearest neighbors and the closest pair. Our *deterministic* kinetic approach for maintenance of all the nearest neighbors improves the previous *randomized* kinetic algorithm by Agarwal, Kaplan, and Sharir.
- an exact KDS for maintenance of the Euclidean minimum spanning tree, which improves the previous KDS by Rahmati and Zarei.
- the first KDS's for maintenance of the *Yao graph* and the *Semi-Yao graph.*
- the first KDS to consider maintaining plane graphs on moving points.
- the first KDS for maintenance of all the k-nearest neighbors, for any $k \geq 1$.
- the first KDS to answer the reverse k-nearest neighbor queries, for any $k \geq 1$ in any fixed dimension, on a set of moving points.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First, I would like to thank my mother, father, brothers, and sisters, who have over the years loved, supported, and encouraged me not to give up on my dreams.

Second, I wish to thank my co-supervisors, Valerie King and Sue Whitesides, for their support and advice during these past three years. They have done a great job at challenging me to do my best, introducing me to the research world, and sponsoring my travels to several conferences in and outside Canada.

Next, I want to thank Mohammad Ali Abam and Alireza Zarei for their discussions and comments on results in Chapters 4 and 5.

I also want to thank the University of Victoria for two fellowships in the two academic years 2011 and 2012, the British Columbia Ministry of Advanced Education, Innovation and Technology for a fellowship in the academic year 2013, and the Natural Sciences and Engineering Research Council (NSERC) for supporting my research through my co-supervisors.

Finally, I wish to thank my external examiner, Timothy M. Chan, for the helpful suggestions, and the thoughtful questions and comments at the final oral examination.

Chapter 1

Introduction

The geometric world around us is in motion, full of moving objects, *e.g.*, robots, flying machines, ships, buses, and mobile devices such as cell phone. Motion is typically a continuous change in the position of an object with respect to time. In many applications, objects move with predictable *trajectories* in the short term, but may be subject to unpredictable changes in trajectory in the long term.

Modeling real-world phenomena by computer raises different types of (data structure) problems. Two types of such problems are *static problems* and *dynamic problems*. For static problems, we are given the input objects once and for all, and for dynamic problems, we can have insertions and deletions to the input objects. Motion has been considered in the context of theoretical computer science over the last few decades, and adds a new challenging category, namely *kinetic problems*, to data structure problems.

For kinetic problems, the input objects move along trajectories given by functions of time. Kinetic problems deal with both discrete and continuous aspects: moving objects with continuous trajectories can have collisions at discrete moments, and *attributes* (properties such as the closest pair) of moving objects change at discrete times. Some common examples of kinetic problems include *"Is there a data structure to answer a query X at any time, for a set of moving objects, with efficient time complexity?"* and *"Is there a data structure to solve a problem Y efficiently, at any time, without recomputing from scratch?"*.

Consideration of kinetic problems arises in many areas, *e.g.*, mobile communication, air-traffic control, power consumption control, database systems, and geographic information systems. Tracking attributes of moving objects in computational geometry has been studied extensively over the past 15 years [3, 6, 10, 11, 12, 17, 57, 79, 81];

several PhD dissertations [1, 15, 82] have been written that focus on maintaining (tracking) attributes of moving objects.

The efficiency of data structures for static and dynamic problems is commonly evaluated in terms of time and space, *i.e.*, construction time, space of the data structure, query response time, and update time of insertions and deletions. It is important to understand what is meant by efficiency when we design data structures for solving kinetic problems. In addition to efficiency in preprocessing time and space of the data structure, we might want to consider the efficiency in (*i*) the *time* to obtain a solution to the kinetic problem at any given moment during the motion, (*ii*) the *time* to make changes to the data structure repeatedly as the points move along trajectories, and (*iii*) the *time* to update the data structure when an object changes its trajectory. To obtain efficiency in such times, in 1997, Basch, Guibas and Hershberger [17] introduced a theoretical framework, which is called the *kinetic data structure (KDS) framework*.

We describe the concepts and terminology for the KDS framework in Section 1.1 as basic background for our work. Then in Section 1.2, we formally review the statements of the various proximity problems we consider. Next in Section 1.3, we present the techniques, the improvements, and the concrete results we obtain in this dissertation.

1.1 KDS Framework

Basch, Guibas and Hershberger [17] first introduced the *kinetic data structure (KDS) framework* to track the attributes (*e.g.*, the closest pair) of a set of moving points. This framework has been used extensively to model motion [15, 52, 1, 82]. In the KDS framework, we assume each point in d-dimensional space has a trajectory given by d known polynomial functions of bounded degree s, where each function gives one of the d coordinates of the point; however, the point can change its trajectory at any moment which is not necessarily known in advance.

Using the KDS framework one can design and analyse a set of data structures and algorithms, namely a *kinetic data structure* (KDS), for maintenance of attributes of moving points. A KDS includes a set of *certificates* that together attest that the attribute of interest holds over time. A certificate is a boolean function of time, and it may have a failure time t. A certificate is valid until time t. The failure time t of a certificate obtains by solving a polynomial equation of constant degree. We assume that there is a model of computation which solves any polynomial equation exactly in constant time. When the failure time of a certificate is equal to the current time, we say an *event* occurs. To track the next failure time after the current time we define a *priority queue* of the failure times of the certificates over time. The certificate with highest priority in the priority queue is the next certificate after the current time that will become invalid. When an event occurs, we invoke an update mechanism to replace the certificates that become invalid with new valid ones, and then we apply the necessary changes to the kinetic data structure. Similarly, if a trajectory changes at some moment, we invoke update mechanisms.

The performance of a KDS is measured in terms of four criteria.

1. *Responsiveness:* This is one of the most important KDS performance criteria, namely the processing time to handle an event. The KDS is called *responsive* if the response time of the update mechanism for an event is $O(\log^c n)$, where n the number of points and c is a constant.

2. *Compactness:* This is the total number of certificates stored in the KDS at any fixed time. The number of certificates is not necessarily the same as the amount of space used by the KDS, but the space of a KDS bounds the compactness of the KDS. If the compactness is $O(n \log^c n)$, the KDS is called *compact*.

3. *Locality:* This is the number of certificates associated with any particular point at any fixed time. If it is $O(\log^c n)$, the KDS is called *local*. Locality is an important criterion. Satisfaction of this criterion ensures that when a point changes its trajectory, no point participates in too many certificates, and therefore, only a small number of changes are needed in the KDS.

4. *Efficiency:* The efficiency of a KDS concerns the number of events in the KDS over time. In the KDS framework, to count the number of events over time we make the assumption that the trajectories of the points are polynomial functions of maximum degree bounded by some constant s.

   We identify two types of events to analyse the efficiency of a KDS. Some events do not necessarily change the attribute of interest (also called the *desired attribute*) and may only change some internal data structures. Such events are called *internal events*. Those events that change the attribute of interest are called *external events*. The efficiency of a KDS is the ratio of the worst-case number of internal events in the KDS to the worst-case number of external events. If the efficiency is $O(\log^c n)$, the KDS is called *efficient*. The efficiency of a KDS can be viewed as measuring the fraction of events that are due to overhead.

1.2 Problem Statement

Let P be a set of n points in arbitrary but fixed dimension d. Here we describe some of the proximity problems and the point set embedding problem that we consider in this dissertation.

Proximity Problems. Finding the nearest neighbor in P to a query point is called the *nearest neighbor* problem. The *all nearest neighbors* problem is to find the nearest point in P to each point $p \in P$. The (directed) graph constructed by connecting each point $p \in P$ to its nearest neighbor $p_1 \in P$ with a (directed) edge $\overrightarrow{pp_1}$ is called the *nearest neighbor graph* (NNG); Figure 1.1 depicts the NNG of a set of points in the plane. The *closest pair* problem is to find a pair of points whose separation distance is minimum; the endpoints of the edge(s) with minimum length in the NNG constitute the closest pair(s). Given any $\epsilon > 0$, the *all* $(1+\epsilon)$*-nearest neighbors* problem is to find some $\hat{q} \in P$ for each point $p \in P$, such that the Euclidean distance $|p\hat{q}|$ between p and $\hat{q}$ is within a factor of $1+\epsilon$ of the Euclidean distance between p and its nearest neighbor p_1.

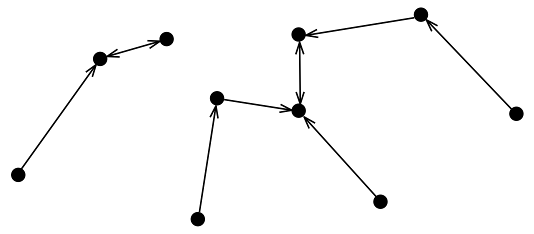


Figure 1.1: The nearest neighbor graph of a set of 10 points.

Generalizations of the above problems are defined as follows. The *k-nearest neighbor* (kNN) problem is to find the k-nearest neighbors to a query point among the points in P. The *all k-nearest neighbors* problem is to find the k-nearest neighbors in P to each point in P. The graph that is constructed by connecting each point to its k-nearest neighbors is called the *k-nearest neighbor graph* (k-NNG).

The *reverse k-nearest neighbor* (RkNN) problem is a popular variant of the kNN problem that asks for the influence of a query point on a point set P. Given an integer k, $1 \leq k < n$, and a query point $q \notin P$, the RkNN problem is to find the set RkNN(q) of all p in P for which q is one of k-nearest neighbors of p. Thus RkNN$(q) = \{p \in P : |pq| \leq |pp_k|\}$, where $|.|$ denotes Euclidean distance, and p_k is the k^{th} nearest neighbor of p among the points in P. Unlike the kNN problem, the exact number of reverse k-nearest neighbors of a query point is not known in advance, but as we prove in Chapter 6 the number is upper-bounded by $O(k)$.

For a point set P, there exists a complete graph $G(V, E)$, where $V = P$ and E is the set of edges in the graph, such that the weight of each edge is the distance between its two endpoints in the L_p metric. An *L_p-minimum spanning tree* (L_p-MST) of G is a connected subgraph of G such that the sum of the edge weights in the L_p metric is minimum possible; Figure 1.2 depicts the Euclidean minimum spanning tree (EMST or L_2-MST, for short) of a set of points in the plane.

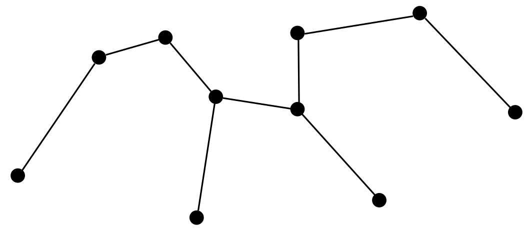


Figure 1.2: The Euclidean minimum spanning tree (EMST) of 10 points.

The *Delaunay triangulation* of a point set P based on a convex shape is the maximal set of edges such that no two edges intersect except at common endpoints, and such that the endpoints of each edge lie on the boundary of an empty scaled translate of the convex shape. Figure 1.3 depicts the Delaunay triangulation based on a circle in the plane.

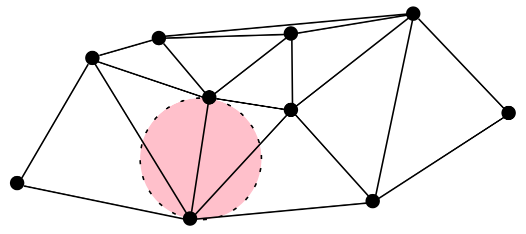


Figure 1.3: The Delaunay triangulation of 10 points based on a circle.

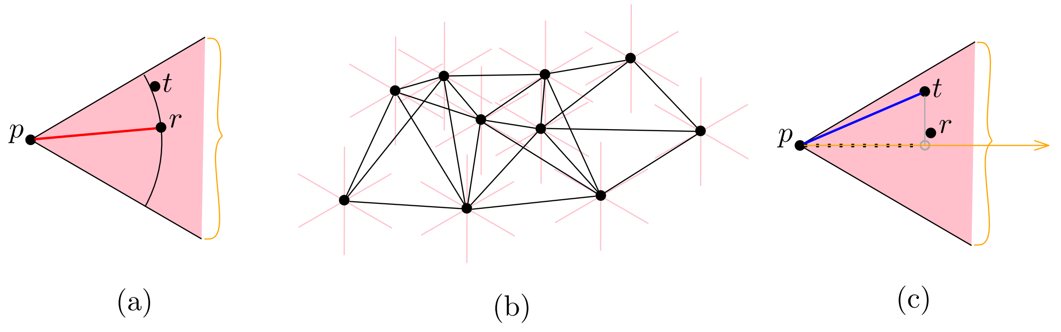


Figure 1.4: (a) Point r is the nearest point to p in the indicated cone of p, and is associated with p in the case of the Yao graph where $\theta = \pi/3$. (b) The Yao graph of 10 points in the plane, where space around each point is partitioned into six cones. (c) Point t is the point associated with p in the case of the Semi-Yao graph where $\theta = \pi/3$.

The *Yao graph* and the *Semi-Yao graph*[1] of P with respect to θ are two geometric graphs with vertex set P and edges defined in the following way. At each point $p \in P$, space is partitioned into a set of polyhedral cones of opening angle θ with apex at p. Then for each cone the apex p is connected to a particular point inside the cone. In the Yao graph, the particular point is the point in the cone with the minimum Euclidean distance to p (see Figures 1.4(a) and 1.4(b)). In the Semi-Yao graph, we define an axis for each cone, a ray emanating from the apex of the cone (in Figure 1.4(c), the cone axis is the bisector of the cone); the particular point is the point in the cone with the minimum length projection on the axis of the cone.

Such problems (*e.g.*, the closest pair problem) that deal with attributes arising from the distances between points are known as *proximity problems*. Tracking and maintenance of proximity attributes on moving points in order to solve a proximity problem is called a *kinetic proximity problem*. From now on, when we say *maintenance of an attribute* over time, we mean having the exact value of the attribute over time; precisely, at any moment t within the time interval of consideration (but not at critical times), the problem solution (*e.g.*, the closest pair) is available to be the output.

In the kinetic setting, for each point $p \in P$ in $\mathbb{R}^d$, we denote the d coordinates of the trajectory of p by d polynomial functions of maximum degree bounded by some constant s; the trajectory can be changed to a new one at any time.

The goal is to provide a kinetic data structures to maintain the attributes of the

[1]This graph is called the theta-graph in [60], but we prefer to call it the Semi-Yao graph instead of the theta-graph, because of its close relationship to the Yao graph.

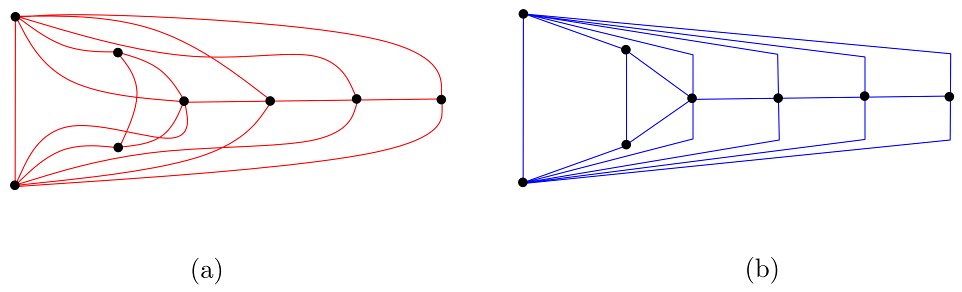


Figure 1.5: (a) A planar graph G. (b) A planar embedding of G.

moving points so that certain queries about these points can be answered efficiently.

Point Set Embedding Problem. A graph $G(V, E)$ is typically represented as a set V of vertices and a set E of edges such that each edge in E is a curve with endpoints at two vertices in V. A *straight-line drawing* of G is a drawing in which each edge of G is mapped to a curve that is a line segment. A k*-bend drawing* is a drawing of G such that each edge is mapped to a chain (curve) of at most $k+1$ line segments; this is also known as a *polyline drawing* with at most k bends per edge; Figure 1.5(b) depicts a 1-bend drawing. A *planar graph* is a graph that can be embedded in the plane such that no two edges intersect except at common vertices; see Figure 1.5(a). A *plane graph* is a planar embedding of a planar graph, such that each vertex of the planar graph maps to a distinct point in the plane and each edge maps to a curve so that no two edges intersect except at common vertices; Figure 1.5(b) depicts a plane graph, a planar embedding of the graph in Figure 1.5(a). A plane graph partitions the plane into a set of *internal faces* and an *outer face* (infinite face); the circular order of the edges incident to each vertex in the plane graph is fixed.

Given a plane graph G on n vertices and a point set P, the problem of *point set embedding without mapping* is to *draw* G on P such that each vertex is mapped to a point of P and the curves representing the edges intersect only at common vertices. Figure 1.6 depicts an embedding of a plane graph on a set of points.

The *kinetic point set embedding problem without mapping*, with at most k bends per edge, is to maintain without mapping a k-bend drawing of an embedding of G on a set P of n moving points in the plane. In the kinetic setting, as mentioned before, the trajectory of each point $p_i(t) = (x_i(t), y_i(t))$ is defined by two known polynomial functions of constant maximum degree s. As the points move, the drawing of an embedding changes over time and may develop crossings or change the embedding.

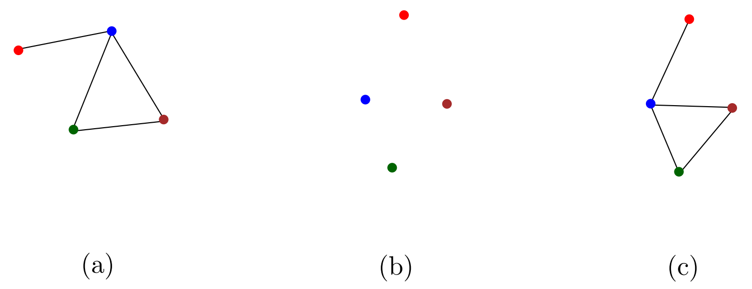


Figure 1.6: (a) A plane graph G with four vertices. (b) A set P of four points. (c) A straight-line embedding of G on P.

It is easy to imagine a moving drawing, where vertices move and edges are drawn as straight segments between them. The goal is to design a kinetic data structure to repair the embedding by remapping the vertices to the points of P.

Our motivation for kinetic graph drawing is to initiate an investigation of kinetic versions of graph drawing problems. However, the specific kinetic point set embedding problem described above does not arise naturally in a kinetic setting, as far as we are aware. Problems such as morphing problems and visualizing social networks may give rise to more natural kinetic graph drawing problems.

1.3 Dissertation Contributions

Here we present our contributions in three subsections. Sections 1.3.1 and 1.3.2 provide the contributions for giving kinetic solutions to some of the proximity problems in $\mathbb{R}^2$ and $\mathbb{R}^d$, respectively. Section 1.3.3 provides our contribution for considering the point set embedding problem for a set of moving points in the plane.

1.3.1 Proximity Problems in $\mathbb{R}^2$

For a set of moving points in the plane, we provide a simple method that underlies all the results we obtain for maintenance of all the nearest neighbors, the closest pair, the Euclidean minimum spanning tree (EMST or L_2-MST), the Yao graph, and the Semi-Yao graph. We present our techniques, concrete results, and improvements as follows.

The heart of our approach is to define, compute, and kinetically maintain supergraphs for the Yao graph and the Semi-Yao graph. Then we take advantage of the fact that these graphs are themselves supergraphs of the EMST and the nearest

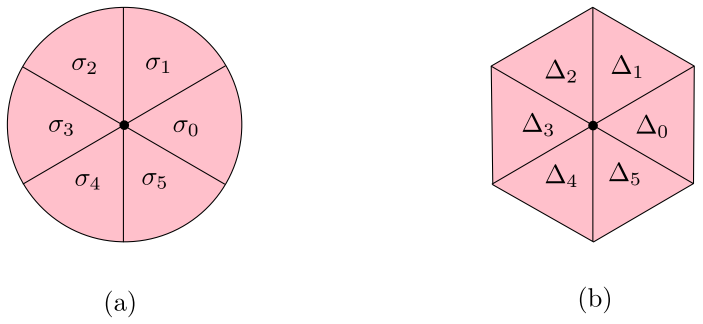


Figure 1.7: (a) Partition of a unit disk into six pieces of pie. (b) Partition of a hexagon into six equilateral triangles.

neighbor graph, respectively.

We define a supergraph for the Yao graph as follows. We partition a unit disk into six "pieces of pie" $\sigma_0, \sigma_2, ..., \sigma_5$ with equal angles such that all the σ_l, $l = 0, ..., 5$, share a point at the center of the disk; see Figure 1.7(a). Each piece of pie σ_l is a convex shape. For each σ_l we construct a triangulation of a set P of points as follows. Using the fact that a Delaunay triangulation of P can be defined based on any convex shape [35, 43], we define a Delaunay triangulation DT_l based on each piece of pie σ_l. We prove that the union of all these Delaunay triangulations DT_l, $l = 0, ..., 5$, which we call the *Pie Delaunay graph*, is a supergraph of the Yao graph. The Yao graph is guaranteed to contain the EMST, where the plane is partitioned into at least six equal wedges at each point of P. Thus the Pie Delaunay graph contains the EMST.

We provide a similar approach to obtain a supergraph for the nearest neighbor graph. We partition a hexagon into six equilateral triangles $\Delta_0, \Delta_2, ..., \Delta_5$ (see Figure 1.7(b)), and for each equilateral triangle Δ_l we define a Delaunay triangulation DT_l. The union of all of these Delaunay triangulations DT_l, $l = 0, ..., 5$, which we call the *Equilateral Delaunay graph*, is a supergraph of the Semi-Yao graph. We prove that the Semi-Yao graph is a supergraph of the nearest neighbor graph, which implies that the Equilateral Delaunay graph is a supergraph of the nearest neighbor graph.

In the case that the Delaunay triangulation DT_l is based on a piece of pie, the triangulation can easily be maintained over time. This leads us to have a kinetic data structure for the union of the DT_l's, *i.e.*, the Pie Delaunay graph. Then we show how to use the Pie Delaunay graph over time to give kinetic data structures for maintenance of the Yao graph and the EMST. Similarly, in the case that each DT_l arises from an equilateral triangle, we are lead to a kinetic data structure for the Equilateral Delaunay graph. Using the kinetic Equilateral Delaunay graph, we give kinetic data structures for maintenance of the Semi-Yao graph, all the nearest neighbors, and the closest pair.

The following items describe our results and improvements.

- We give the first KDS for maintenance of a well-studied sparse graph, the Yao graph, and provide a new exact KDS for maintenance of the EMST. Our KDS's for maintenance of the Yao graph and the EMST use $O(n)$ space, take $O(n \log n)$ preprocessing time, and process $O(n^3 \beta_{2s+2}^2(n) \log n)$ events, each in amortized time $O(\log n)$. Here, $\beta_s(n)$ is an extremely slow-growing function. In terms of the KDS performance criteria, which are formally defined in Section 1.1, our KDS's are *responsive* in an amortized sense, *compact*, and *local* on average.

  We improve the previous EMST KDS by Rahmati and Zarei [79] by a near-linear factor in the number of events.

- We give simple KDS's for maintenance of all the nearest neighbors and the closest pair, and give the first KDS for maintenance of the Semi-Yao graph. Our KDS's use $O(n)$ space and $O(n \log n)$ preprocessing time. The all nearest neighbors KDS and the closest pair KDS process $O(n^2 \beta_{2s+2}^2(n) \log n)$ events, and the Semi-Yao graph KDS processes $O(n^2 \beta_{2s+2}(n))$ events; each event can be handled in amortized time $O(\log n)$. Our KDS's are *responsive* in an amortized sense, *compact*, *local* on average, and *efficient*.

  The *certificates* of our KDS for maintenance of the closest pair are simpler than the certificates of the previous kinetic algorithms by Basch, Guibas, and Hershberger [16], Basch, Guibas, and Zhang [18], and Agarwal, Kaplan, and Sharir [10].

  Our *deterministic* algorithm for maintenance of all the nearest neighbors in $\mathbb{R}^2$ is simpler and more efficient than the *randomized* kinetic algorithm by Agarwal, Kaplan, and Sharir [10] in the following ways. While their kinetic algorithm and ours need a priority queue containing all certificates of the KDS, our priority queue uses linear space, but their priority queue uses $O(n \log^2 n)$ space. Furthermore, our KDS uses a graph data structure for the Equilateral Delaunay graph and a tournament trees for each point, but their KDS uses a 2-*dimensional range tree* implemented by randomized search trees (treaps), a constant number of sorted lists, and in fact it maintains $O(\log^2 n)$ tournament trees for each point. In particular,

- we perform one-dimensional range searching, as opposed to the two-dimensional range searching of their work;
- the sparse graph representation allows us to obtain a linear space KDS, which improves the space complexity $O(n \log^2 n)$ of their KDS. Their KDS in fact maintains a supergraph of the nearest neighbor graph with $O(n \log^2 n)$ candidate edges;
- in our kinetic algorithm, the number of changes to the Equilateral Delaunay graph when the points are moving is $O(n^2\beta_{2s+2}(n))$; this leads us to have total processing time $O(n^2\beta^2_{2s+2}(n)\log^2 n)$, which is an improvement of the total expected processing time $O(n^2\beta^2_{2s+2}(n)\log^4 n)$ of their randomized algorithm;
- each point in our KDS participates in $O(1)$ certificates on average, but in their KDS each point participates in $O(\log^2 n)$ certificates on average.

1.3.2 Proximity Problems in $\mathbb{R}^d$

We provide a KDS for maintenance of all the nearest neighbors on moving points in any fixed dimension d, and also give the first solution to the kinetic RkNN problem, for any $k \geq 1$.

Our technique for maintenance of all the nearest neighbors in $\mathbb{R}^d$ is similar to our technique in $\mathbb{R}^2$: In both cases we maintain a supergraph, the Semi-Yao graph, for the nearest neighbor graph. For the $\mathbb{R}^d$ case, we use a constant number of d-*dimensional range trees* to maintain the Semi-Yao graph, whereas for the $\mathbb{R}^2$ case, we use a constant number of planar graphs, the Equilateral Delaunay triangulations.

Our Semi-Yao graph KDS uses $O(n \log^d n)$ space and processes $O(n^2)$ events with total processing time $O(n^2\beta_{s+2}(n)\log^{d+1} n)$, and it is *compact*, *efficient*, *responsive* in an amortized sense, and *local*. Our all nearest neighbors KDS uses $O(n \log^d n)$ space and processes $O(n^2\beta^2_{2s+2}(n)\log n))$ events with total processing time $O(n^2\beta_{2s+2}(n)\log^{d+1} n)$. It is *compact*, *efficient*, *responsive* in an amortized sense, and *local* on average. We improve the previous randomized result by Agarwal, Kaplan, and Sharir [10] by a factor of $\log^d n$ in the number of events and by a $\log n$ factor in the total cost. Note that for maintaining all the nearest neighbors, neither our KDS nor the KDS by Agarwal *et al.* is local in general, and furthermore, each event in our KDS and in their KDS is handled in polylogarithmic time in an amortized sense. To satisfy the locality criterion and to get a worst-case processing time for handling events, we provide a KDS for maintenance of all the $(1+\epsilon)$-nearest neighbors in $\mathbb{R}^d$. This KDS uses $O(n \log^d n)$ space. It handles $O(n^2 \log^d n)$ events, each in worst-case time $O(\log^d n \log \log n)$, and it is *compact*, *efficient*, *responsive*, and *local*.

We answer RkNN queries as follows. For a query point $q \notin P$, we partition d-dimensional space into a set of polyhedral cones around q, and then among the points of P in each cone, we examine the k points having shortest projections on the cone axis. We obtain $O(k)$ candidate points for q such that q might be one of their k-nearest neighbors. If we maintain the k^{th} nearest neighbor p_k of each point $p \in P$, we can easily check whether a candidate point p is one of the reverse k-nearest neighbors of q; this can be done by checking whether $|pq| \leq |pp_k|$.

For answering RkNN queries on a set of n continuously moving points in a fixed dimension d, our KDS uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time. In the preprocessing step, we introduce a new, simple method for reporting all the k-nearest neighbors for all the points $p \in P$ in order of increasing distance from p. It uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time. For $k = \Omega(\log^{d-1} n)$, both our result and the best previous result by Dickerson and Eppstein [42] have the same complexity for reporting all the k-nearest neighbors, but in our view, our method is simpler in practice.

Given a query point at any time t, our KDS finds $O(k)$ candidate points. To check whether a candidate point is a reverse neighbor to the query point at time t, our kinetic approach maintains all the k-nearest neighbors over time. This is the *first* KDS for maintenance of all the k-nearest neighbors, for any $k \geq 1$. Our all k-nearest neighbors KDS processes $O(\phi(s,n) * n^2)$ events, each in amortized time $O(\log n)$. Here, $\phi(s,n)$ is the complexity of the k-level of a set of n partially-defined polynomial functions, such that each pair of them intersects at most s times. At any time t, an RkNN query can be answered in time $O(\log^d n + k)$. Note that if an event occurs at the same time t, we first spend $O(\log n)$ amortized time to update all the k-nearest neighbors, and then we answer the query.

Table 1.1 summarizes all the (previous and new) results for the kinetic proximity problems. In this table, “Dim.”, “Num.”, “Proc.”, and “Ch.” stand for “Dimension”, “Number”, “Processing”, and “Chapter”, respectively. Here, $\beta(n)$ is an extremely slow-growing function.

Kinetic problem	Dim.	Space	Num. of events	Proc. time	/event(s)	Local
Closest pair in [16]	2	$O(n)$	$O(n^2\beta(n)\log n)$	$O(\log^2 n)$	/event	Yes
Closest pair in [18]	$O(1)$	$O(n\log^{d-1} n)$	$O(n^2\beta(n)\log n)$	$O(\log^d n)$	/event	Yes
Closest pair in [10]	$O(1)$	$O(n\log^{d-1} n)$	$O(n^2\beta(n)\log n)$	$O(\log^d n)$	/event	Yes
Closest pair in Ch. 3	2	$O(n)$	$O(n^2\beta^2(n)\log n)$	$O(n^2\beta^2(n)\log^2 n)$	/events	No
All NNs in [10]	$O(1)$	$O(n\log^d n)$	$O(n^2\beta(n)\log^{d+1} n)$	$O(n^2\beta(n)\log^{d+2} n)$	/events	No
All NNs in Ch. 3	2	$O(n)$	$O(n^2\beta^2(n)\log n)$	$O(n^2\beta^2(n)\log^2 n)$	/events	No
All NNs in Ch. 5	$O(1)$	$O(n\log^d n)$	$O(n^2\beta^2(n)\log n)$	$O(n^2\beta(n)\log^{d+1} n)$	/events	No
All $(1+\epsilon)$-NNs in Ch. 5	$O(1)$	$O(n\log^d n)$	$O(n^2\log^d n)$	$O(\log^d n\log\log n)$	/event	Yes
All kNNs in Ch. 6	$O(1)$	$O(n\log^{d+1} n+kn)$	$O(\phi(s,n)*n^2)$	$O(\phi(s,n)*n^2\log n)$	/events	No
$(1+\epsilon)$-EMST in [18]	$O(1)$	$O((1/\epsilon)^{(d-1)/2}n\log^{d-1} n)$	$O((1/\epsilon)^{d-1}n^3)$	$O(\log^d n)$	/event	Yes
EMST in [79]	2	$O(n)$	$O(n^4)$	$O(\log^2 n)$	/event	No
EMST in Ch 4	2	$O(n)$	$O(n^3\beta^2(n)\log n)$	$O(n^3\beta^2(n)\log^2 n)$	/events	No
Yao graph in Ch. 4	2	$O(n)$	$O(n^3\beta^2(n)\log n)$	$O(n^3\beta^2(n)\log^2 n)$	/events	No
Semi-Yao graph in Ch. 3	2	$O(n)$	$O(n^2\beta(n))$	$O(n^2\beta(n)\log n)$	/events	No
Semi-Yao graph in Ch. 5	$O(1)$	$O(n\log^d n)$	$O(n^2)$	$O(n^2\beta(n)\log^{d+1} n)$	/events	Yes

Table 1.1: The previous results and our results for kinetic proximity problems.

1.3.3 Point Set Embedding Problem

We investigate the problem of point set embedding of a plane graph on a set of points as follows. We maintain a 3-bend drawing "without mapping" of a given plane graph G with n vertices on a set P of n moving points.

Denote by $p_1, ..., p_n$ the points of P sorted in non-decreasing order by their x-coordinates. We find a Hamiltonian cycle $(v_1, ..., v_n, v_1)$ of the plane graph G that has an edge v_1v_n on the outer face, and then we map the Hamiltonian path $(v_1, ..., v_n)$ to the chain $(p_1, ..., p_n)$. Note that if in a plane graph there is no Hamiltonian cycle, by adding some dummy points and dummy edges to the plane graph, we can find a Hamiltonian cycle. For each edge v_iv_j of G, we draw a polygonal curve of line segments such that the slopes of the line segments are defined based on the difference between the subscripts i and j and the maximum slope of the edges $p_1p_2, p_2p_3, ..., p_{n-1}p_n$. This assignment prevents intersections between edges during the motion except at times when two points change the ordering of their x-coordinates.

Our KDS uses $O(n)$ space and $O(n \log n)$ preprocessing time, and it processes $O(n^2\beta_{2s+2}(n) \log n)$ events, each in worst-case time $O(\log^2 n)$. In terms of the four standard KDS performance criteria, our KDS is *efficient*, *responsive*, *local*, and *compact*. At any time t, the drawing of an edge v_iv_j of G can be obtained in an efficient time; if an event occurs at time t, it takes $O(\log^2 n)$ to generate the slopes for the drawing of v_iv_j; otherwise, it takes $O(1)$ time.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews the necessary background for our work and reviews the most important related work.

Chapters 3 - 6 consider the kinetic proximity problems: Chapters 3 and 4 provide simple KDS's for maintenance of the closest pair, all the nearest neighbors, and the EMST in $\mathbb{R}^2$. Chapters 5 and 6 give KDS's for maintenance of all the k-nearest neighbors and answering the reverse k-nearest neighbor queries for a set of moving points in $\mathbb{R}^d$.

Chapter 7 considers the kinetic point set embedding problem for plane graphs.

The last chapter contains discussions and some open problems.

Chapter 2

Background and Related Work

In this chapter, we describe the necessary background for our work, and give an overview of previous results. In Sections 2.1 and 2.2, we briefly introduce two KDS's, *kinetic tournament trees* and *kinetic range trees*, and review the theorems that we use throughout this dissertation. In Section 2.3, we discuss the best current results to the problems we consider.

2.1 Kinetic Tournament Trees

Here we introduce a basic KDS, which is called a *kinetic tournament tree* [17]. We use this KDS for maintenance of an attribute of interest for a set of moving points, and then we analyse the KDS based on the four standard KDS performance criteria as defined in Section 1.1.

Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n moving points in the plane, where the y-coordinate $y_i(t)$ of each point p_i is a polynomial function of at most constant degree s. The attribute of interest that we want to maintain is the *lowest point* with respect to the y-axis among the set P of moving points. In Figure 2.1, the lower envelope tracks the lowest point of four moving points over time; the lowest point changes at the breakpoints on the lower envelope at times $t_1, ..., t_4$.

To maintain the lowest point over time we can use a kinetic tournament tree. A kinetic tournament tree is a balanced binary tree T such that the points are stored at the leaves of the tree T in an arbitrary order, and each internal node v of the tree maintains the lowest point between its two children. In more detail, denote by T_v the subtree rooted at internal node v and denote by $P(v)$ the set of points stored at

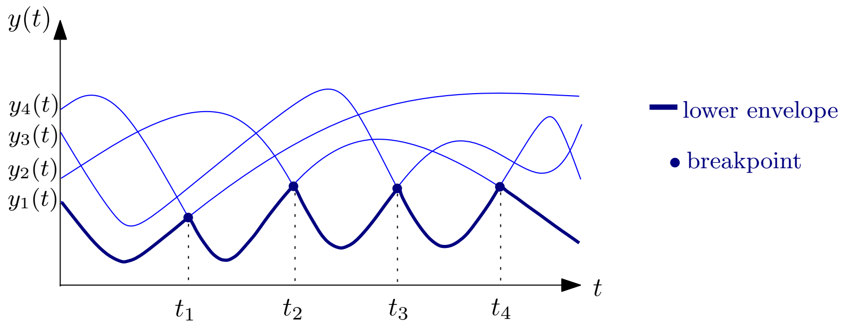


Figure 2.1: Tracking the lowest point among a set of four moving points.

the leaves of T_v. The point stored at v in the tournament tree is the lowest point among all the points in $P(v)$; this point is called the *winner* of the subtree T_v. For each internal node v of the tournament tree, we define a *certificate* to attest whether the left-winner (winner of the left subtree) or the right-winner (winner of the right subtree) is the winner for v. The failure time of the certificate corresponding to v is the time when the winner at v changes. The set of all certificates is stored in a *priority queue*, with the failure times as the keys, to track the next time after the current time that a certificate will become invalid.

When the certificate corresponding to an internal node v fails, it may imply the need to change some winners on the path from the parent of v to the root. In some cases the winner of a node v' on the path does not change, but the failure time corresponding to the certificate of the node v' may change. Therefore, we must update the failure times of the certificates of the nodes on the path from the parent of v to the root, and then we must replace the invalid certificates with new valid ones in the priority queue; this takes $O(\log^2 n)$ time, which implies that the KDS is *responsive*. Since the size of the priority queue is linear, the KDS is *compact*. Each point participates in $O(\log n)$ certificates, which implies that the KDS is *local*.

We need the following theorem to obtain the efficiency of the KDS. It follows from Theorem 2.1 that the number of external events, which in fact is the number of changes to the root of the tournament tree, is at most $\lambda_s(n)$. Here, $\lambda_s(n) = n\beta_s(n)$ is the maximum length of Davenport-Schinzel sequences of order s on n symbols, and $\beta_s(n)$ is an extremely slow-growing function. In particular, the following states the sharp bounds on $\lambda_s(n)$.

$$\lambda_s(n) = \begin{cases} n, & \text{for } s = 1; \\ 2n - 1, & \text{for } s = 2; \\ 2n\alpha(n) + O(n), & \text{for } s = 3; \\ \Theta(n2^{\alpha(n)}), & \text{for } s = 4; \\ \Theta(n\alpha(n)2^{\alpha(n)}), & \text{for } s = 5; \\ n2^{(1+o(1))\alpha^t(n)/t!}, & \text{for } s \geq 6; \end{cases}$$

here $t = \lfloor (s-2)/2 \rfloor$ and $\alpha(n)$ denotes the inverse Ackermann function [72].

The number of internal events for all the internal nodes is $\sum_v \lambda_s(|P(v)|) = O(\lambda_s(n) \log n)$. Thus the ratio of the number of internal events to the number of external events is $O(\log n)$. This implies that the KDS is *efficient*.

Theorem 2.1. [85] *The number of breakpoints on the lower envelope of n totally-defined, continuous, univariate functions, such that each pair of them intersects at most s times, is at most $\lambda_s(n)$.*

Note that Theorem 2.1 holds for totally-defined functions; there exists a similar result for partially-defined functions:

Theorem 2.2. [85] *The number of breakpoints on the lower envelope of n partially-defined, continuous, univariate functions, such that each pair of them intersects at most s times, is at most $\lambda_{s+2}(n)$.*

Dynamic and Kinetic Tournament Trees. Now consider a *dynamic* version of the above example, where insertions and deletions into the point set P are allowed, such that the y-coordinates of newly inserted points are polynomials of maximum degree bounded by s.

It is convenient for our purpose to make the kinetic tournament tree dynamic, to support *point insertions and deletions*; the dynamic version of the kinetic tournament tree is called a *dynamic and kinetic tournament tree* [10]. The dynamic and kinetic tournament tree can be implemented using any dynamic balanced search tree. Agarwal *et al.* [10] used a *weight-balanced (BB(α)) tree* [68, 70] to obtain their results for the number of the events. The total number of events in a kinetic tournament tree is $O(\sum_v \lambda_{s+2}(|P(v)|)) = O(\beta_{s+2}(n) \sum_v |P(v)|)$. Therefore, for a sequence of m insertions and deletions, we only need to count the number of increases in $\sum_v |P(v)|$.

Each insertion in a weight-balanced tree increases by one the size of all the subtrees rooted at nodes v on the corresponding search path. Since the length of each search path is $O(\log n)$, the number of increases in $\sum_v |P(v)|$ by a sequence of m insertions is $O(m \log n)$. Furthermore, each rotation around an edge (v_i, v_j) in a weight-balanced tree causes one of $|P(v_i)|$ and $|P(v_j)|$ to increase and the other one to decrease. Though the cost of a rotation in a weight-balanced tree is a function of $|P(v_i)|$ and $|P(v_j)|$, the total cost for a sequence of m operations is $O(m \log n)$ [68, 70]. Hence rotations can contribute $O(m \log n)$ to $\sum_v |P(v)|$. Therefore, for any sequence of m insertions and deletions, the tournament tree implemented by a weight-balanced tree generates $O(m\beta_{s+2}(n) \log n)$ events.

The total cost to handle all the events is bounded by the processing time of each event in the priority queue, which is $O(\log n)$, times the total number of events. Thus it is $O(m\beta_{s+2}(n) \log^2 n)$. Note that each insertion or deletion in a weight-balanced tree takes time $O(\log n)$ [68, 70], and it may change $O(\log n)$ certificates along the corresponding search path. Thus replacing the invalid certificates with new valid ones in the priority queue takes time $O(\log^2 n)$, which implies that an update can be handled in worst-case time $O(\log^2 n)$.

The following theorem summarizes the results above.

Theorem 2.3. `(Theorem 3.1. of [10])` *Assume one is given a sequence of m insertions and deletions into a kinetic tournament tree whose maximum size at any time is n (assuming $m \geq n$). The tournament implemented by a weight-balanced tree tree generates $O(m\beta_{s+2}(n) \log n)$ events for a total cost of $O(m\beta_{s+2}(n) \log^2 n)$. Each update can be handled in time $O(\log^2 n)$. A kinetic tournament tree on n elements can be constructed in $O(n)$ time.*

The Complexity of k-level. Consider a set of n moving points, where the y-coordinate $y_i(t)$ of each point p_i is a polynomial function of at most constant degree s. The *k-level* of these polynomial functions is a set of points $q \in \mathbb{R}^2$ such that each point q lies on a polynomial function, and such that it is above exactly $k-1$ other polynomial functions; Fugure 2.2 depicts the 3-level and breakpoints on the 3-level of four polynomials. The k-level tracks the k^{th} lowest point with respect to y-axis.

Theorems 2.1 and 2.2 give the complexity of the 1-level (*i.e.*, the number of breakpoints on the lower envelope) for a set of polynomial functions. The following theorem gives the current bounds on the complexity of the k-level.

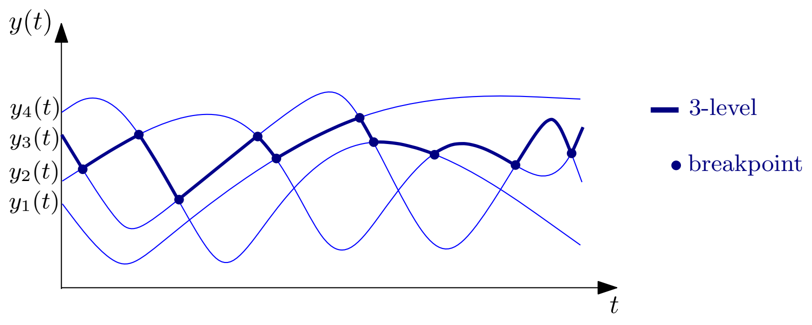


Figure 2.2: The 3-level of a set of four moving points.

Theorem 2.4. [29, 30] *The complexity of the k-level of a set of n partially-defined polynomial functions, such that each pair of them intersects at most s times, is as follows.*

$$\phi(s,n) = \begin{cases} O(n^{3/2}\log n), & \textit{for } s=2; \\ O(n^{5/3}\textit{poly}\log n), & \textit{for } s=3; \\ O(n^{31/18}\textit{poly}\log n), & \textit{for } s=4; \\ O(n^{161/90-\delta}), & \textit{for } s=5, \textit{ for some constant } \delta>0; \\ O(n^{2-1/2s-\delta_s}), & \textit{for odd } s, \textit{ for some constant } \delta_s>0; \\ O(n^{2-1/2(s-1)-\delta_s}), & \textit{for even } s, \textit{ for some constant } \delta_s>0. \end{cases}$$

In general, a bound $f(n)$ of $\phi(s,n)$ can be converted to the k-sensitive bound $O(f(k)(n/k)\beta(n/k))$ [5]. The complexity of the $(\leq k)$-level is $O(kn\beta(n/k))$ [84]. Here, $n\beta(n)$ is the complexity of the lower envelope.

2.2 Kinetic Range Trees

Denote by $u_1, u_2, ...,$ and u_d the d coordinate axes in $\mathbb{R}^d$. The *range query problem* is to process a set P of stationary points in $\mathbb{R}^d$ to report the points of P inside a rectangular query range $R := [x_1 : x'_1] \times [x_2 : x'_2] \times ... \times [x_d : x'_d]$. The rectangular range queries can efficiently be answered using a *d-dimensional range tree data structure* [21]. A d-dimensional range tree is a *multi-level data structure* which is described as follows. The first-level tree is a balanced binary search tree $\mathcal{T}$ built on the u_1-coordinate of the points in P, where the points are stored at the leaves of $\mathcal{T}$. For each internal node or leaf node v of a tree at level i, the points in $P(v)$ (*i.e.*, the set of points stored

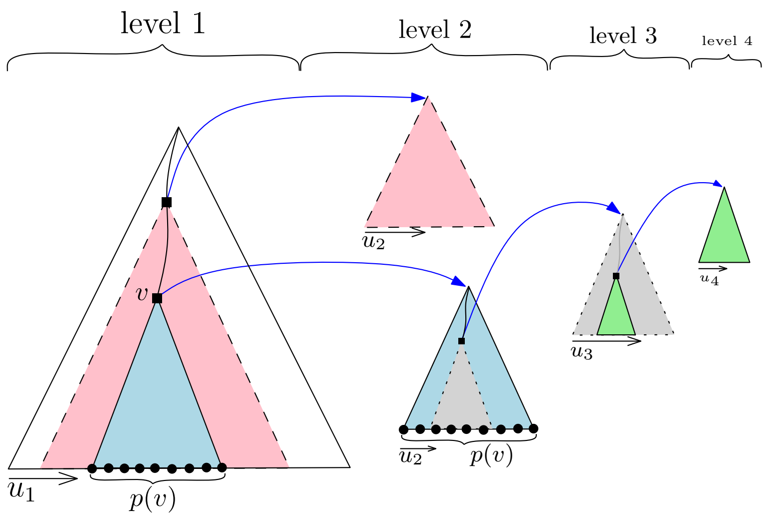


Figure 2.3: Some parts of a 4-dimensional range tree.

at the leaves of the subtree rooted at v) are stored in a balanced binary search tree $\mathcal{T}_v$ at level $i+1$ according to their u_{i+1}-coordinates. Figure 2.3 depicts some parts of a 4-dimensional range tree, where the points at level i are stored at the leaves of the i-level tree(s) according to their u_i-coordinates. A rectangular range query can be answered by finding $O(\log^d n)$ subtrees at level d of the range tree, and then reporting the points of the subtrees. The following gives the complexity for answering rectangular range queries.

Theorem 2.5. [21] *Let P be a set of n points in d-dimensional space, where $d \geq 2$. A range tree for P uses $O(n \log^{d-1} n)$ storage and it can be constructed in $O(n \log^{d-1} n)$ time. One can report the points in P that lie in a rectangular query range in $O(\log^{d-1} n + k)$ time, where k is the number of reported points* [1].

Let $p = (x_1, x_2, ..., x_d)$ be a point in $\mathbb{R}^d$. Let $C(p)$ be the cone bounded by d half-spaces $u_1 - x_1 \geq 0, u_2 - x_2 \geq 0, ..., u_{d-1} - x_{d-1} \geq 0$, and $u_d - x_d \geq 0$; Figure 2.4 depicts a cone $C(p)$ in $\mathbb{R}^2$. For reporting the points of P inside a cone $C(p)$, one can use Theorem 2.5 with the rectangular query range $R := [x_1 : \infty] \times [x_2 : \infty] \times ... \times [x_d : \infty]$. Thus the points in $P \cap C(p)$ can be reported in $O(\log^d n + k)$ time, where $k = |P \cap C(p)|$ is the cardinality of the set $P \cap C(p)$.

[1]For a set of stationary points, there are lots of improvements for answering rectangular range queries (*e.g.*, see [32]).

Range Trees on Moving Points. Given a set P of moving points, where the trajectory of each point is a polynomial function of constant maximum degree s, the *kinetic range query problem* is to process the moving points in P such that the points of P inside a query range $C(p)$ can efficiently be reported at any given time t. By designing a KDS for maintenance of a range tree, one can report the points in $P \cap C(p)$ for a query range $C(p)$ at time t.

The range tree remains unchanged as long as the order of the points in each of the coordinates $u_1, ..., u_{d-1}$, and u_d remains unchanged. Therefore, for applying necessary changes to the range tree over time, we maintain sorted lists $L(u_1), ..., L(u_{d-1})$, and $L(u_d)$ of the points in each of the coordinates $u_1, ..., u_{d-1}$, and u_d, respectively. For each two consecutive points in each sorted list $L(u_i)$ we define a certificate that certifies the order of the two points in the u_i-coordinate. To track the closest time to the current time that an event (*i.e.*, a change to the range tree) occurs we put the failure times of all the certificates in a priority queue; the element with the highest priority in the queue gives the closest time. When an event occurs the update mechanism is invoked to repair the range tree data structure.

Basch *et al.* [18] and Agarwal *et al.* [10] use dynamic balanced trees to implement a range tree for a set of moving points. Using rebalancing operations, they handle events to maintain a range tree. In particular, in their approaches, when an event between two points p and q occurs, we must delete p and q and reinsert them into the range tree. Thus the range tree can be maintained over time using a dynamic range tree. One of the approaches to update the range trees is to carry out local and global rebuilding after a few operations, which gives $O(\log^d n)$ amortized time per operation [69]. Another approach, which uses merge and split operations, gives worst-case time $O(\log^d n)$ per operation [96].

Abam and de Berg [2] introduced a variant of the range trees, a *rank-based range tree* (RBRT), which avoids rebalancing the range tree and gives a polylogarithmic worst-case processing time when an event occurs. Similar to that of a regular range tree, the points at level i of the RBRT are sorted at the leaves in ascending order according to their u_i-coordinates. The skeleton of an RBRT is independent of the position of the points in $\mathbb{R}^d$ and depends on the ranks of the points in each of the u_i-coordinates. The *rank* of a point in a tree at level i of the RBRT is its position in the sorted list of all the points ordered by their u_i-coordinates. Any tree at any level of the RBRT is a balanced binary tree, and no matter how many points are in the tree, it is a tree on n ranks. Here we shall give a detailed description of an RBRT.

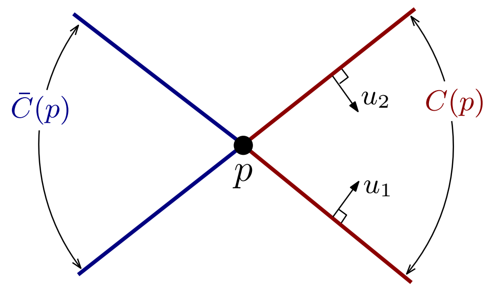


Figure 2.4: A cone $C(p)$ and its reflection $\bar{C}(p)$ associated to $p = (x_1, x_2)$.

Let v be an internal node or a leaf node at level d of the RBRT. Denote by $R(v)$ the set of points at the leaves of the subtree rooted at v. Corresponding to v we define another set $B(v)$. Let $p \in P$ be a point stored at a leaf node of a tree at level d of the RBRT. Denote by $\mathcal{P}_p$ the path from the parent of p to the root. A point p belongs to $B(v)$ if v is the right child of some node $\bar{v} \in \mathcal{P}_p$. The set of all the pairs $(B(v), R(v))$, for all the nodes v at level d of the RBRT, is called a *cone separated pair decomposition* (CSPD) for P; denote this set by $\Psi = \{(B_1, R_1), ..., (B_m, R_m)\}$.

Let $\bar{C}_l(p) = -C_l(p)$ be the reflection of $C_l(p)$ through p, which is intuitively formed by following the lines through p in the half-spaces of $C_l(p)$; see Figure 2.4. In particular, the cone $\bar{C}(p)$ is the intersection of the half-spaces $u_1 - x_1 \leq 0, u_2 - x_2 \leq 0, ..., u_{d-1} - x_{d-1} \leq 0$, and $u_d - x_d \leq 0$.

The following gives the complexity of a (kinetic) RBRT.

Theorem 2.6. [2] *A d-dimensional rank-based range tree (RBRT) uses $O(n \log^d n)$ storage and can be constructed in $O(n \log^d n)$ time. It can be described as a set of pairs $\Psi = \{(B_1, R_1), ..., (B_m, R_m)\}$ with the following properties.*

- *Each pair $(B_j, R_j) \in \Psi$ is generated from an internal node or a leaf node of a tree at level d of the RBRT.*
- *For any two points p and q in P where $q \in C(p)$, there is a unique pair $(B_j, R_j) \in \Psi$ such that $p \in B_j$ and $q \in R_j$.*
- *For any pair $(B_j, R_j) \in \Psi$, if $p \in B_j$ and $q \in R_j$, then $q \in C(p)$ and $p \in \bar{C}(q)$.*
- *Each point $p \in P$ is in $O(\log^d n)$ pairs of (B_j, R_j) which implies that the number of elements of all the pairs (R_j, B_j) is $O(n \log^d n)$.*
- *For any point $p \in P$, all the sets B_j (resp. R_j) where $p \in B_j$ (resp. $p \in R_j$) can be found in time $O(\log^d n)$.*

- *The set $P \cap C(p)$ (resp. $P \cap \bar{C}(p)$) of points is the union of $O(\log^d n)$ sets R_j (resp. B_j), where the subscript j is such that $p \in B_j$ (resp. $p \in R_j$).*

For a set of n moving points, where the trajectories are polynomial functions of constant degree, the RBRT can be maintained by processing $O(n^2)$ events, each in worst-case time $O(\log^d n)$.

2.3 Previous Results

Here we review the best current results to each of the problems for two scenarios: a set of stationary points, and a set of moving points.

2.3.1 All Nearest Neighbors and Closest Pair

Stationary setting. The nearest neighbor problem, which was also called the *post office problem* by Donald Knuth in 1973 [61], is a fundamental, well-studied proximity problem in computational geometry. The closest pair problem, a variant of the nearest neighbor problem, was efficiently solved in 1975 by Shamos and Hoey [83]. They gave an $O(n \log n)$-time algorithm to compute the closest pair of a set of n stationary points in the plane. For a set of n points in $\mathbb{R}^d$, Bentley and Shamos [20] solved the problem with the same complexity. There is also a linear-time randomized algorithm to find the closest pair whereas the known deterministic algorithms take $O(n \log n)$ time [74].

Vaidya [91] in 1989 gave an $O(n \log n)$-time algorithm to solve the all nearest neighbors problem in $\mathbb{R}^d$. A few years later the all k-nearest neighbors problem was solved, for any $k \geq 1$. In time $O(kn \log n)$ [42] one can report all the k-nearest neighbors, for a point set in any fixed dimension, where the neighbors are reported in order of increasing distance from each point; reporting the unordered set takes time $O(n \log n + kn)$ [27, 38, 42].

Kinetic setting. Basch, Guibas, and Hershberger [16] provided a KDS for maintenance of the closest pair for a set of moving points in $\mathbb{R}^2$. Their KDS uses linear space and processes $O(n^2 \beta_{2s+2}(n) \log n)$ events, each in worst-case time $O(\log^2 n)$. Their KDS is efficient, responsive, compact, and local.

Basch, Guibas, and Zhang [18] used d-dimensional range trees to maintain the closest pair. For a fixed dimension d, their KDS uses $O(n \log^{d-1} n)$ space and processes $O(n^2\beta_{2s+2}(n)\log n)$ events, each in worst-case time $O(\log^d n)$. Their KDS is responsive, efficient, compact, and local.

Agarwal, Kaplan, and Sharir [10] gave KDS's for both maintenance of the closest pair and all the nearest neighbors in $\mathbb{R}^d$. Agarwal *et al.* claimed that their closest pair KDS simplifies the certificates used by Basch, Guibas, and Hershberger [16]; but Agarwal *et al.* independently presented a KDS for the closest pair with the same approach as that of [18] by Basch, Guibas, and Zhang. The closest pair KDS by Agarwal *et al.* uses $O(n\log^{d-1} n)$ space and processes $O(n^2\beta_{2s+2}(n)\log n)$ events, each in amortized time $O(\log^d n)$. Their closest pair KDS is efficient, responsive (in an amortized sense), local, and compact.

Agarwal *et al.* gave the first efficient KDS to maintain all the nearest neighbors in $\mathbb{R}^d$. To obtain the efficiency of their KDS, they implemented multidimensional range trees by using randomized search trees (treaps). Their randomized kinetic approach uses $O(n\log^d n)$ space and processes $O(n^2\beta^2_{2s+2}(n)\log^{d+1} n)$ events; the expected time to process all events is $O(n^2\beta^2_{2s+2}(n)\log^{d+2} n)$. On average, their KDS is local, meaning that each point in their KDS participates in $O(\log^d n)$ certificates. Their all nearest neighbors KDS is efficient, responsive (in an amortized sense), compact, but in general is not local.

To the best of our knowledge there is no KDS for maintenance of all the k-nearest neighbors, for any $k > 1$, even for a set of points in $\mathbb{R}^2$.

2.3.2 Euclidean Minimum Spanning Tree

Stationary setting. The EMST problem is an old, well-studied proximity problem in computational geometry and has been extensively studied in the literature. Given a supergraph for the EMST of m edges, one can compute the EMST in time $O(m\log n)$ using the Prim's algorithm [73], or in time $O(m + n\log n)$ using the Kruskal's algorithm [63]. Shamos and Hoey [83] used the Delaunay triangulation, a supergraph of the EMST, to solve the problem. Since a Delaunay triangulation of a point set P in the plane has $O(n)$ edges, and can be constructed in time $O(n\log n)$ [21], the EMST in $\mathbb{R}^2$ can be obtained in time $O(n\log n)$. Yao [99] proved that the EMST can be found in time $O(n^{1.8}\log^{1.8} n)$ for a set of points in $\mathbb{R}^3$, and in time $O(n^{2-1/2^{d+1}}\log^{1-1/2^{d+1}} n)$ for a set of points in $\mathbb{R}^d$, where $d \geq 3$. These bounds were later improved by Agarwal *et al.* [7]. Their method, which is randomized, yields running times $O(n^{4/3}\log^{4/3} n)$ for $d = 3$, and $O(n^{2-2/(\lceil d/2+1\rceil)+\sigma})$ for $d \geq 4$ and any

$\sigma > 0$. It is still an open problem whether the EMST in $\mathbb{R}^d$ can be computed in time close to the lower bound $\Omega(n \log n)$ [41]; event for $d = 3$, it is conjectured that there is no $o(n^{4/3})$-time algorithm to compute the EMST [45, 40].

For any given $\epsilon > 0$, Vaidya [92] gave an $O((1/\epsilon)^d n \log n)$-time algorithm to compute a $(1+\epsilon)$-EMST, whose total weight is within a factor of $1+\epsilon$ of the total weight of an exact EMST. This bound was later improved to $O^*(n \log n + (1/\epsilon)^{d/2} n)$ by Callahan and Kosaraju [26]; here, notation O^* is used to hide factors $1/\epsilon^c$, where c is a small constant not more than 1 or 2. Recently, Arya and Chan [31] achieved better bounds $O^*((1/\epsilon)^{d/3} n \log n)$ and $O^*(n + (1/\epsilon)^{d/3} n / \log^{2/3} n)$ to construct a $(1+\epsilon)$-EMST.

Kinetic setting. Fu and Lee [47] proposed the first kinetic algorithm for maintenance of an EMST on a set of n moving points in the plane. Their algorithm uses $O(n^4 \log n)$ preprocessing time and $O(\tilde{m})$ space, where $\tilde{m}$ is the maximum possible number of changes in the EMST from time $t = 0$ to $t = \infty$. At any given time, the algorithm constructs the EMST in linear time.

Agarwal *et al.* [8] proposed a sophisticated algorithm for a restricted kinetic version of the MST over a graph where the distance between each pair of points in the graph is defined by a linear function of time. The processing time for each combinatorial change in the MST is $O(n^{2/3} \log^{4/3} n)$; the bound reduces to $O(n^{1/2} \log^{3/2} n)$ for planar graphs. Their data structure does not explicitly bound the number of changes, but a bound of $O(n^4)$ is easily seen.

For any $\epsilon > 0$, Basch, Guibas, and Zhang [18] presented a KDS for a $(1+\epsilon)$-EMST. For a set of points in $\mathbb{R}^d$, their KDS uses $O(\epsilon^{-(d-1)/2} n \log^{d-1} n)$ space and processes $O(\epsilon^{-(d-1)} n^3)$ events, each in $O(\log^d n)$ time. They state that their structure can be used to maintain the MST in the L_1 and L_∞ metrics.

Rahmati and Zarei [79] improved the previous result by Fu and Lee [47]; they presented an exact kinetic algorithm for maintenance of the EMST on a set of n moving points in $\mathbb{R}^2$. In $O(n \log n)$ preprocessing time and $O(n)$ space, they build a KDS that processes $O(n^4)$ events, each in $O(\log^2 n)$ time. Their KDS uses the method of Guibas *et al.* [53] to track changes to the Delaunay triangulation, which is a supergraph of the EMST [71]. Whenever two edges of the Delaunay triangulation swap their length order, their kinetic algorithm makes the required changes to the EMST. In fact, the number of changes in their algorithm is within a linear factor of the number of changes to the Delaunay triangulation [53]. Rubin proved that the number of discrete changes to the Delaunay triangulation is $O(n^{2+\epsilon})$, for any $\epsilon > 0$, under the assumption that any four points can be co-circular at most twice [81], or at most three times where each point moves along a straight line at unit speed [80]. Under these assumptions, the kinetic algorithm of Rahmati and Zarei processes $O(n^{3+\epsilon})$ events, and hence within a linear factor of changes to the Delaunay triangulation.

The kinetic approach by Rahmati and Zarei [79] can maintain the minimum spanning tree of a plane graph whose edge weights are polynomial functions of constant maximum degree; the processing time of each event is $O(\log^2 n)$.

2.3.3 Reverse k-Nearest Neighbor Queries

Stationary setting. The reverse k-nearest neighbor problem was first posed by Korn and Muthukrishnan [62] in the database community, and then considered extensively in this community due to its many applications, *e.g.*, decision support systems, profile-based marketing, traffic networks, business location planning, clustering and outlier detection, and molecular biology [64, 65, 87, 88, 89, 100, 101].

In computational geometry community, there exist two data structures [66, 34] that give solutions to the RkNN problem. Both of these solutions answer RkNN queries for a set P of stationary points and both only work for $k = 1$. Maheshwari *et al.* [66] gave a data structure to solve the R1NN problem in $\mathbb{R}^2$. Their data structure creates an arrangement of largest empty circles centered at the points of P and answers R1NN queries by point location in the arrangement. Their data structure uses $O(n)$ space and $O(n \log n)$ preprocessing time, and an R1NN query can be answered in time $O(\log n)$. Cheong *et al.* [34] considered the R1NN problem in fixed dimension $\mathbb{R}^d$, where $d = O(1)$. Their method, which uses a compressed quadtree, partitions space into cells such that each cell contains a small number of candidate points. To answer an R1NN query, their solution finds a cell that contains the query point and then checks all the points in the cell. Their data structure uses $O(n)$ space and $O(n \log n)$ preprocessing time, and can answer an R1NN query in $O(\log n)$ time. It seems that the approach by Cheong *et al.* can be extended to answer RkNN queries with preprocessing time $O(kn \log n)$, space $O(kn)$, and query time $O(\log n + k)$.

Kinetic setting. The reverse k-nearest neighbor queries for a set of continuously moving objects has attracted the attention of the database community [94, 19, 56, 97, 98, 56, 44]. To the best of our knowledge, in computational geometry, there is no KDS to answer the reverse k-nearest neighbor queries.

2.3.4 Point Set Embeddability for Plane Graphs

Stationary setting. Cabello [25] proved that deciding whether a graph G can be embedded by straight-line edges without mapping onto a given set P of points is NP-complete, even when G is 2-connected. Gritzmann *et al.* [51] showed that the class of planar graphs such that all vertices are on the outer face (*outerplanar* graphs) is the largest class of graphs that can be embedded with straight-line edges without mapping onto any point set in general position (no three or more points collinear). There are algorithms for special cases in which the graph G is a tree [24, 55] or an outerplanar graph [23, 51].

For k-bend drawing without mapping, Kaufmann and Wiese [59] gave a 1-bend drawing algorithm for 4-connected plane graphs and a 2-bend drawing algorithm for general plane graphs; for general graphs, their algorithm takes time $O(n \log n)$ (resp. $O(n^2)$) to draw a point set embedding with at most three (resp. two) bends per edge. In particular, their algorithm draws a 3-bend drawing in $O(n \log n)$ time and then spends $O(n^2)$ time, using rotation, to transfer the 3-bend drawing to a 2-bend drawing. In addition, they proved that deciding whether there is a mapping such that each edge has at most one bend is NP-complete. Giacomo *et al.* [50] presented an $O(n \log n)$-time algorithm which improves the previous $O(n^2)$-time algorithm by Kaufmann and Wiese [59] and guarantees that no rotation is needed to obtain a 2-bend drawing.

Kinetic setting. To our knowledge there are no previous results for point set embedding of a plane graph on a set of points moving along predictable trajectories.

Chapter 3

Kinetic All Nearest Neighbors and Closest Pair in the Plane

In this chapter we present KDS's for maintenance of all the nearest neighbors and the closest pair, for points moving with known trajectories given by bounded degree polynomial functions. In Section 3.1, we first introduce two new supergraphs of the nearest neighbor graph, the Semi-Yao graph and the *Equilateral Delaunay graph* (EDG), and then we show that these graphs are in fact the same. We provide the first KDS to maintain the Semi-Yao graph in Section 3.2. Finally, in Sections 3.3 and 3.4, we use the Semi-Yao graph KDS to give simple, *deterministic* KDS's for maintenance of all the nearest neighbors and the closest pair.

Our KDS for maintenance of all the nearest neighbors improves the previous *randomized* KDS by Agarwal, Kaplan, and Sharir [10].

The results of this chapter were published as a paper in the Proceedings of the 29th ACM Symposium on Computational Geometry (SoCG 2013) [76].

3.1 New Method for Computing All Nearest Neighbors and Closest Pair

Partition the plane into six *wedges* (cones) $W_0, ..., W_5$, each of angle $\pi/3$ with common apex at the origin o. For $0 \leq l \leq 5$, let W_l span the angular range $[(2l-1)\pi/6, (2l+1)\pi/6)$. Denote by x_l the unit vector in the direction of the bisector ray of W_l. Let $W_l(p_i)$ denote the translate of wedge W_l that moves the apex to point p_i, and let $\mathcal{V}_l(p_i)$ denote the intersection of P with wedge $W_l(p_i)$: $\mathcal{V}_l(p_i) = P \cap W_l(p_i)$. Denote

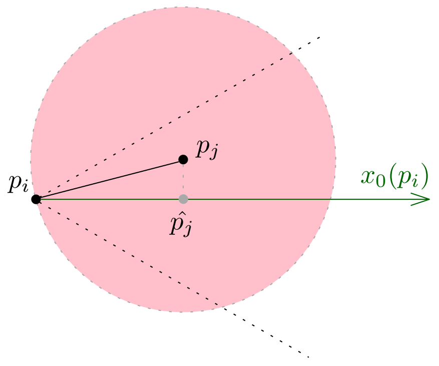


Figure 3.1: Projection of the point p_j to the bisector $b_0(p_i)$ of the wedge $W_0(p_i)$.

by $x_l(p_i)$ the unit vector emanating from p_i in the direction of the bisector ray of $W_l(p_i)$; see Figure 3.1. Observe that, in Figure 3.1, since p_i is the closest point to p_j, there are no other points of P in the interior of the disc centered at p_j with radius $d(p_i, p_j)$, where $d(p_i, p_j)$ is the distance between points p_i and p_j.

The following straightforward lemma is key for obtaining our KDS's for the *all nearest neighbors* and the *closest pair* problems. Consider $p_j \in P$, and let p_i denote the point of P closest to p_j and distinct from p_j. Let $W_l(p_i)$ denote the wedge of p_i that contains p_j, and denote by $\hat{p}_j$ the projection of p_j to the bisector $x_l(p_i)$; see Figure 3.1.

Lemma 3.1. (Lemma 2.1. of [10]) *Point p_j has the minimum length projection to $x_l(p_i)$, where the minimum is taken over $\mathcal{V}_l(p_i)$. That is,*

$$|\hat{p}_j p_i| = \min\{|\hat{p}_k p_i| : \ p_k \in \mathcal{V}_l(p_i)\} \tag{3.1}$$

Proof. We prove the lemma by contradiction: Assume there is a point $p_r \in \mathcal{V}_l(p_i)$ whose x_l-coordinate is less than the x_l-coordinate of p_j; see Figure 3.2. Consider the triangle $p_i p_j p_r$, which is inscribed in an equilateral triangle. Since p_i is the closest point to p_j, $|p_j p_i| < |p_j p_r|$, which implies that the angle $\angle p_j p_i p_r > \angle p_j p_r p_i$. This is a contradiction, because $\angle p_j p_i p_r \leq \pi/3$ and $\angle p_j p_r p_i > \pi/3$. □

Thus Lemma 3.1 gives a necessary condition for p_i to be the nearest neighbor to p_j. We now use this lemma to define a supergraph of the nearest neighbor graph of P. To find the nearest neighbor for each point $p_j \in P$, we seek a set of candidate points $\mathcal{C}(p_j) = \{p_i | \ p_i$ *and* p_j *satisfy Equation* (3.1)$\}$. From now on, when we say p_j has the minimum x_l-coordinate inside the wedge $W_l(p_i)$, we mean that p_j and p_i

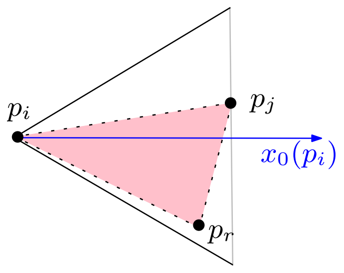


Figure 3.2: Point p_i is the closest point to p_j; among the points in $\mathcal{V}_l(p_i)$, p_j has the minimum length projection on the bisector $x_0(p_i)$.

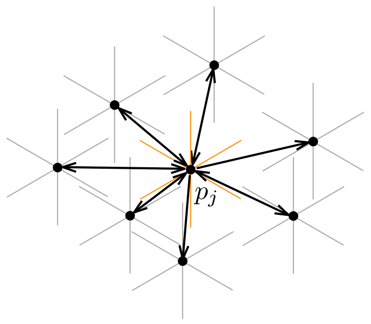


Figure 3.3: In-edges and out-edges of p_j.

satisfy Equation (3.1).

Consider a Semi-Yao graph which is constructed as follows. Connect each point $p_i \in P$ to a point $p_j \in \mathcal{V}_l(p_i)$ with a directed edge $\overrightarrow{p_jp_i}$ from p_j to p_i whenever p_j is the point with the minimum x_l-coordinate, among all the points in $\mathcal{V}_l(p_i)$. The edge $\overrightarrow{p_jp_i}$ is called an *in-edge* for p_i and it is called an *out-edge* for p_j. Each point in the Semi-Yao graph has at most six in-edges and has a set of out-edges; Figure 3.3 depicts the in-edges and the out-edges of the point p_j. Denote by $S_{out}(p_j)$ the end points of the out-edges of p_j. From the above discussion, it is easy to see the following observation, which gives Lemma 3.2 below.

Observation 3.1. $\mathcal{C}(p_j) = S_{out}(p_j)$.

Lemma 3.2. *The Semi-Yao graph of a point set P is a supergraph of the nearest neighbor graph of P.*

From now on, when we say a convex set, *e.g.*, a triangle, is *empty*, we mean it has no point of P in its interior.

From Lemma 3.1, we can get the following straightforward observation, which makes a connection to the Delaunay triangulations of the point set P.

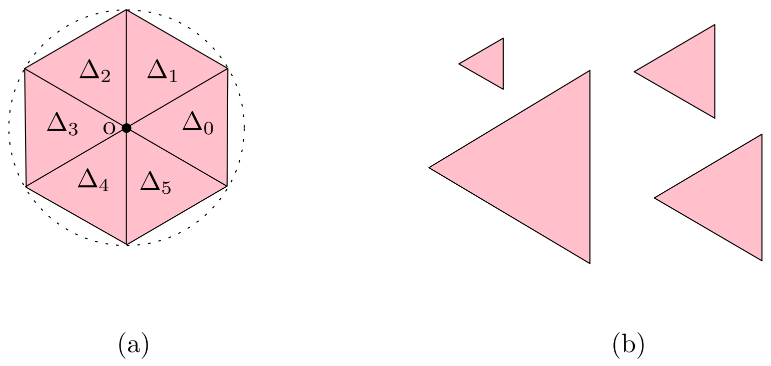


Figure 3.4: (a) Partitioning the unit regular hexagon into six equilateral triangles. (b) Some 0-tri's.

Observation 3.2. *If p_j has the minimum x_l-coordinate inside the wedge $W_l(p_i)$, then p_i and p_j touch the boundary of an empty equilateral triangle; p_i touches a vertex and p_j touches an edge of the triangle.*

A *unit regular hexagon* is a regular hexagon whose edges have unit length; let $\hexagon$ be the unit regular hexagon with center at the origin o and vertices at $(\sqrt{3}/2, 1/2)$, $(0, 1)$, $(-\sqrt{3}/2, 1/2)$, $(-\sqrt{3}/2, -1/2)$, $(0, -1)$, and $(\sqrt{3}/2, -1/2)$; see Figure 3.4(a). Partition $\hexagon$ into six equilateral triangles $\triangle_l$, $l = 0, 1, .., 5$ and call any translated and scaled copy of $\triangle_l$ an *l-tri*; see Figure 3.4(b).

A Delaunay graph can be defined based on any convex shape, *e.g.*, a square, a diamond, any triangle, or a piece of pie [3, 4, 43]. Chew and Drysdale [35, 43] gave divide-and-conquer algorithms to compute the Delaunay triangulation based on a convex shape [1]. The following summarizes the construction time of the Delaunay triangulation in their algorithms.

Theorem 3.1. [35, 43] *The Delaunay triangulation of a set of n points based on a convex shape can be constructed in $O(n \log n)$ time.*

Here we call the Delaunay triangulation constructed based on an equilateral triangle an *Equilateral Delaunay triangulation* (EDT).

There is a nice connection between the Semi-Yao graph and the Equilateral Delaunay triangulations. In general, the Semi-Yao graph is the union of two Equilateral Delaunay triangulations [22]. Next, we describe this connection in a different, and in our view simpler, way than [22].

[1] Other standard approaches, such as the randomized incremental construction [21], would work to obtain a Delaunay triangulation based on a convex shape.

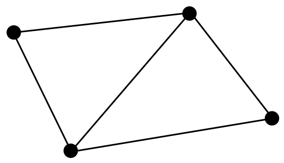


Figure 3.5: The Delaunay triangulation based on the 0-tri.

Denote by EDT_l the Equilateral Delaunay triangulation based on the l-tri. The edge p_ip_j is an edge of EDT_l if and only if there is an empty l-tri such that p_i and p_j are on the boundary of the l-tri; Figure 3.5 depicts EDT_0 for a set of four points. Let $\mathcal{E}(G)$ be the set of edges of graph G; the set of vertices of G is P. Since $\triangle_0$, $\triangle_2$, and $\triangle_4$ are translates of one another, and similarly for $\triangle_1$, $\triangle_3$, and $\triangle_5$, we have that $\mathcal{E}(EDT_0) = \mathcal{E}(EDT_2) = \mathcal{E}(EDT_4)$ and $\mathcal{E}(EDT_1) = \mathcal{E}(EDT_3) = \mathcal{E}(EDT_5)$. Thus there are two different types of l-tri's. We define the *Equilateral Delaunay graph* (EDG) to be the union of EDT_0 and EDT_1, *i.e.*, $p_ip_j \in \mathcal{E}(EDG)$ if and only if $p_ip_j \in \mathcal{E}(EDT_0)$ or $p_ip_j \in \mathcal{E}(EDT_1)$.

Corollary 3.1. *The Equilateral Delaunay graph (EDG) of a set of n points can be constructed in $O(n \log n)$ time.*

Proof. Since each $\triangle_l$, $0 \leq l \leq 5$, is a convex shape, one can construct the corresponding Equilateral Delaunay triangulation EDT_l in $O(n \log n)$ time (by Theorem 3.1). □

Let $p_ip_j \in \mathcal{E}(EDT_l)$. By definition there exists an empty l-tri such that p_i and p_j are on its boundary. By scaling down the l-tri, one of the l-tri vertices will be placed at p_i or p_j; see Figures 3.6(b) and 3.6(c).

Observation 3.3. *If there is an empty l-tri such that p_i and p_j are on its boundary, then there is an empty l-tri with the same property such that either p_i or p_j is a vertex of the l-tri.*

The next lemma proves that the (undirected) Semi-Yao graph and the Equilateral Delaunay graph are equal to each other.

Lemma 3.3. *Edge $p_ip_j \in \mathcal{E}(SYG)$ if and only if $p_ip_j \in \mathcal{E}(EDG)$.*

Proof. Let p_ip_j be an edge of the Semi-Yao graph such that p_j has the minimum x_l-coordinate inside some wedge $W_l(p_i)$; see Figure 3.6(a). The bounded area created by the wedge $W_l(p_i)$ and the line through p_j perpendicular to $x_l(p_i)$ is an l-tri. Therefore,

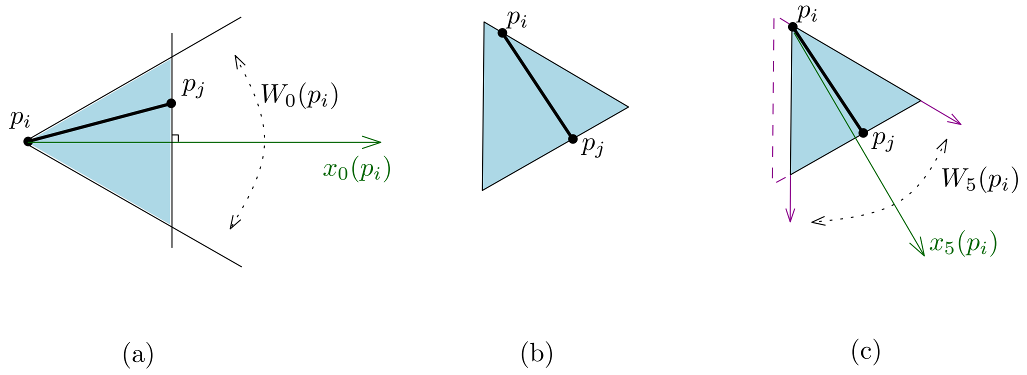


Figure 3.6: (a) The point p_j has the minimum x_0-coordinate inside the wedge $W_0(p_i)$. (b) The 1-tri corresponding to the edge p_ip_j in EDT_1 does not contain any other points of P. (c) The point p_j is inside the wedge $W_5(p_i)$ and has the minimum x_5-coordinate.

for the edge p_ip_j, there exists an empty l-tri such that p_i and p_j are on its boundary. This implies that p_ip_j is an edge of EDT_l.

Let $p_ip_j \in \mathcal{E}(EDT_l)$. By the definition of EDT_l, there exists an empty l-tri such that p_i and p_j are on its boundary; see Figure 3.6(b). By Observation 3.3, we can get a new rescaled l-tri such that p_i and p_j are on its boundary and such that one of the l-tri vertices is p_i or p_j (see Figure 3.6(c)); without loss of generality assume it is p_i. Point p_j is inside the wedge $W_k(p_i)$, where $k \in \{l, (l+2) \bmod 6, (l+4) \bmod 6\}$. Point p_j has the minimum b_k-coordinate inside the wedge $W_k(p_i)$; otherwise, there would be a point of P inside the rescaled l-tri, which means that $p_ip_j \notin \mathcal{E}(EDT_l)$, a contradiction. Therefore, $p_ip_j \in \mathcal{E}(SYG)$. □

Now we can give the following, which is deriving known results in a new way.

Theorem 3.2. *The all nearest neighbors and the closest pair problems in $\mathbb{R}^2$ can be solved in $O(n \log n)$ time.*

Proof. From Corollary 3.1 and Lemma 3.3, the Semi-Yao graph can be constructed in $O(n \log n)$ time. Since the number of edges in the Semi-Yao graph is at most $6n$, by traversing the Semi-Yao graph edges incident to each point, we can find all the nearest neighbors and the closest pair in linear time. □

3.2 Kinetic Equilateral Delaunay Graph

Since $\mathcal{E}(EDT_0) = \mathcal{E}(EDT_2) = \mathcal{E}(EDT_4)$ and $\mathcal{E}(EDT_1) = \mathcal{E}(EDT_3) = \mathcal{E}(EDT_5)$, to maintain the EDG, which is the union of EDT_0 and EDT_1, we only need to have kinetic data structures for EDT_0 and EDT_1. We describe how to maintain EDT_0; EDT_1 is handled similarly.

The Delaunay triangulation EDT_0 is locally stable as long as the points are in general position. Note that we assume the set of points P is in general position with respect to a 0-tri; this means that no four or more points are on the boundary of any scaled, translated 0-tri. When the points are moving, at a moment t this assumption may fail. In fact for moving points, we make a further assumption: no four points are on the boundary of the 0-tri throughout any positive interval of time. This ensures that the points are in general position over time except at some discrete moments. The number of these discrete moments over time is in the order of the number of changes to EDT_0, because the failure of the general position assumption is a necessary condition for changing the topological structure of EDT_0. When a point moves, EDT_0 can change only in the graph neighborhood of the point, and so the correctness of EDT_0 over time is asserted by a set of certificates. Our approach for maintenance of EDT_0 is a known approach also used in [3, 4, 9, 11] for maintenance of Delaunay triangulations based on convex shapes.

Figure 3.7(a) depicts the EDT_0 of a set P of points. Each edge on the boundary of the infinite face of EDT_0, *e.g.*, p_ip_j, is called a *hull* edge; the other edges, *e.g.*, $p_{i'}p_{j'}$, are called *interior* edges. Corresponding to these two types of edges, we define two types of certificates, *NotInWedge* and *NotInTri*, respectively. Below, we first consider the interior edges and then the exterior edges.

NotInTri certificates. Each interior edge $p_{i'}p_{j'} \in EDT_0$ is incident to two triangles $p_{i'}p_{j'}p_{r'}$ and $p_{i'}p_{j'}p_r$; see Figure 3.7(a). For the triangle $p_{i'}p_{j'}p_{r'}$ (resp. $p_{i'}p_{j'}p_r$), there exists an empty 0-tri, denoted by $\Delta^0_{r'}$ (resp. Δ^0_r), such that $p_{i'}$, $p_{j'}$ and $p_{r'}$ (resp. p_r) are on the boundary of $\Delta^0_{r'}$ (resp. Δ^0_r). For $p_{i'}p_{j'}$, we define a *NotInTri* certificate certifying that p_r (resp. $p_{r'}$) is outside $\Delta^0_{r'}$ (resp. Δ^0_r). For sufficiently short time intervals, p_r and $p_{r'}$ are the only points that can change the validity of edge $p_{i'}p_{j'}$ (see [3, 4, 9, 11]). Let t be the time when the four points $p_{i'}$, $p_{j'}$, $p_{r'}$, and p_r are on the boundary of a 0-tri; at time t^-, p_r (resp. $p_{r'}$) is outside $\Delta^0_{r'}$ (resp. Δ^0_r). When p_r (resp. $p_{r'}$) moves inside $\Delta^0_{r'}$ (resp. Δ^0_r), at time t^+, this certificate fails and there

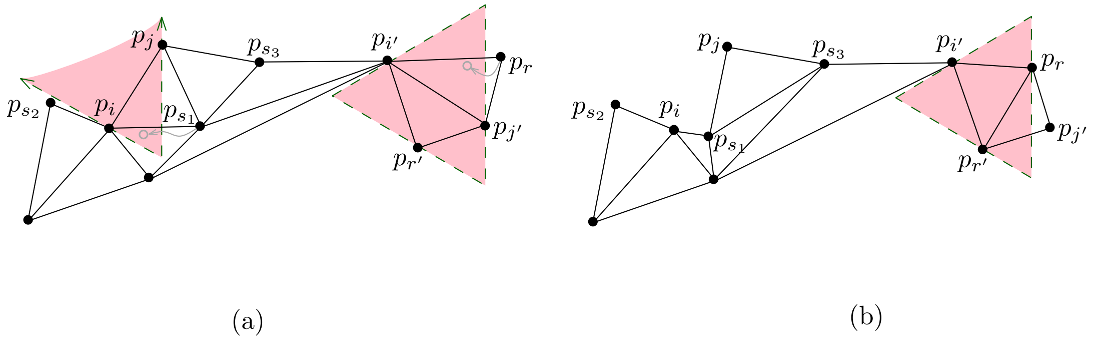


Figure 3.7: (a) The NotInTri certificate corresponding to the edge $p_{i'}p_{j'}$ certifies that p_r is outside the 0-tri of $p_{i'}$, $p_{j'}$, and $p_{r'}$. The NotInWedge certificate of the edge p_ip_j certifies that p_{s_1}, p_{s_2}, and p_{s_3} are outside the corresponding k-wedge. (b) The changes to EDT_0 after p_r moves inside the 0-tri passing through $p_{i'}$, $p_{j'}$, and $p_{r'}$ and after p_{s_1} moves inside the k-wedge of p_ip_j.

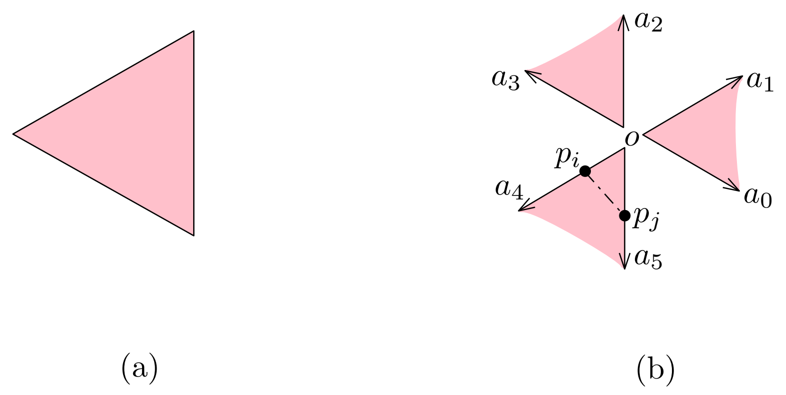


Figure 3.8: (a) A 0-tri. (b) The k-wedges created by the 0-tri; edge p_ip_j divides the 4-wedge $\overleftrightarrow{a_4oa_5}$ into the bounded area $\overline{op_ip_j}$ and the unbounded area $\overleftrightarrow{a_4p_ip_ja_5}$.

is no empty 0-tri such that $p_{i'}$ and $p_{j'}$ are on its boundary. Thus at time t, we have to delete the edge $p_{i'}p_{j'}$ and add the new edge $p_{r'}p_r$, because at time t^+ there exists an empty 0-tri for $p_rp_{r'}$; see Figure 3.7(b). Also, we must define new certificates corresponding to the newly created triangles.

NotInWedge certificates. By removing one of the 0-tri edges and extending the other two edges to infinity, three types of wedges are created (see Figure 3.8); call these wedges *k-wedges*, for $k = \{0, 2, 4\}$, and denote them by $\overleftrightarrow{a_koa_{k+1}}$; the two sides $\overrightarrow{oa_k}$ and $\overrightarrow{oa_{k+1}}$ of the boundary of the k-wedge are parallel to the two corresponding sides of the wedge W_k.

For a hull edge p_ip_j, there exists an empty k-wedge such that p_i and p_j are on the

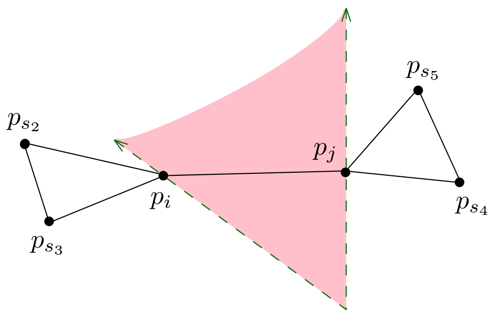


Figure 3.9: A hull edge is incident to at most four other hull edges.

boundary. Each hull edge is incident to at most one triangle $p_ip_jp_{s_1}$ (see Figure 3.7(a)), and adjacent to at most four other hull edges $p_ip_{s_2}, p_ip_{s_3}, p_jp_{s_4}$ and $p_jp_{s_5}$ on the boundary cycle of the infinite face (see Figure 3.9); the point p_{s_1} can be one of the points p_{s_2} to p_{s_5}. If p_ip_j is adjacent to four other hull edges, this edge cannot be incident to a triangle, and if it is incident to a triangle, it cannot be adjacent to more than two other hull edges.

Note that the only points that can change the validity of the edge p_ip_j over a sufficiently short time interval are the points p_{s_i}, $1 \leq i \leq 5$. Therefore, we define at most four *NotInWedge* certificates for the hull edge p_ip_j, certifying that the points p_{s_i}, $1 \leq i \leq 5$, are outside the k-wedge; see Figure 3.7(a). Next we describe how to update the edges of EDT_0 when a NotInWedge certificate fails.

Let t be the time when three points p_i, p_j, and p_{s_i} are on the boundary of the k-wedge; at time t^-, p_{s_i} is outside the k-wedge. As shown in Figure 3.8(b), a hull edge p_ip_j divides its corresponding k-wedge $\overleftrightarrow{a_koa_{k+1}}$ into a bounded area $\overline{op_ip_j}$ and an unbounded area $\overleftrightarrow{a_kp_ip_ja_{k+1}}$.

If p_{s_i} moves inside the bounded area $\overline{op_ip_j}$ at time t^+, the NotInWedge certificate of p_ip_j fails, and we must delete p_ip_j from the hull edges at time t and replace it with two edges incident to p_{s_i}. In Figure 3.7(a), if p_{s_1} moves inside the bounded area $\overline{op_ip_j}$, then we replace the hull edge p_ip_j with two edges $p_ip_{s_1}, p_{s_1}p_j$; in particular, the chain $[..., p_{s_2}p_i, p_ip_j, p_jp_{s_3}, ...]$ of hull edges changes to $[..., p_{s_2}p_i, p_ip_{s_1}, p_{s_1}p_j, p_jp_{s_3}, ...]$ when p_{s_1} moves inside the k-wedge (see Figure 3.7(b)). When this event occurs the previous interior edges $p_ip_{s_1}$ and $p_{s_1}p_j$ become hull edges, and we must replace the previous certificates of these edges with new valid ones.

If p_{s_i} moves inside the unbounded area $\overleftrightarrow{a_kp_ip_ja_{k+1}}$, without loss of generality let p_{s_i} be incident to p_i, we replace the hull edges $p_{s_i}p_i$ and p_ip_j with $p_{s_i}p_j$. Then the previous hull edge p_ip_j either is an edge of EDT_0, in which case we must define a

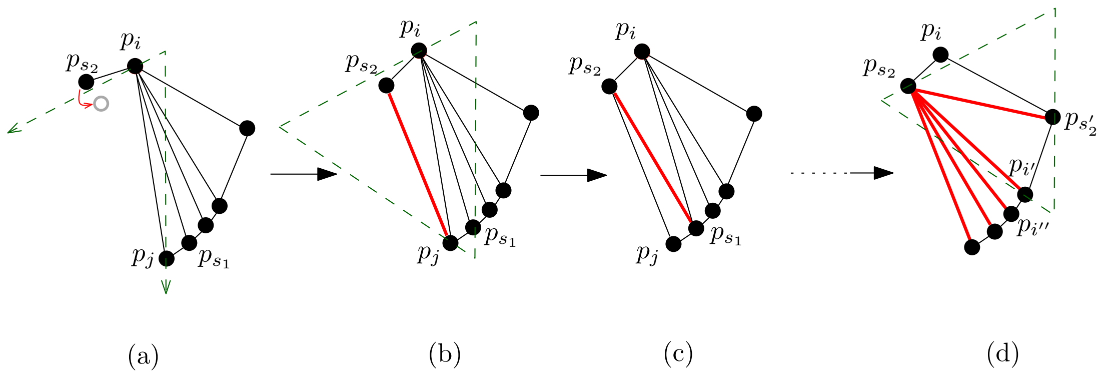


Figure 3.10: The consecutive changes to EDT_0 when p_{s_2} moves inside the k-wedge of p_ip_j.

valid certificate for it, or it is not, in which case we must delete it from EDT_0 and add a new edge; In Figure 3.10(a) we delete p_ip_j and add the new edge $p_{s_i}p_{s_1}$, where p_ip_j is incident to a triangle $p_ip_jp_{s_1}$ (see Figure 3.10(c)).

Consecutive Changes to EDT_0. In some cases, when a certificate fails, we must apply a *sequence* of changes to EDT_0. These kinds of changes occur at incident triangles, and as we will see, they can be handled consecutively.

When a NotInWedge certificate fails, we apply a sequence of edge insertions and edge deletions to EDT_0. In Figure 3.10(a), when p_{s_2} moves inside the k-wedge of p_ip_j, we replace chain $p_{s_2}p_i, p_ip_j$ of hull edges with $p_{s_2}p_j$ (see Figure 3.10(b)), and then we apply a sequence of changes; the previous hull edge p_ip_j is no longer an edge in $\mathcal{E}(EDT_0)$, because now the interior of its corresponding 0-tri contains the point p_{s_2}, and so we replace it with the edge $p_{s_1}p_{s_2}$ (see Figure 3.10(c)). Finally, by checking the 0-tri's of other incident triangles, we can obtain a set of valid edges for EDT_0 (see Figure 3.10(d)).

A similar scenario could happen when a NotInTri certificate fails. In Figure 3.10(d), if p_i moves inside the 0-tri of p_{s_2}, $p_{s'_2}$, and $p_{i'}$, we must apply a sequence of changes to EDT_0 that is the reverse of what we did above when the NotInWedge certificate failed. First we replace $p_{s_2}p_{s'_2}$ with $p_ip_{i'}$. Then we must replace $p_{s_2}p_{i'}$ with $p_ip_{i''}$, because p_i is inside the 0-tri of p_{s_2}, $p_{i'}$, and $p_{i''}$. By checking the 0-tri's of other incident triangles we can obtain a valid set of edges for EDT_0; see Figure 3.10, read from (d) to (a). Therefore, after any change to EDT_0 we must check the validity of the incident triangles, which can be done easily.

Theorem 3.3 below enumerates the changes to the Equilateral Delaunay graph when the points move, and gives the total processing time for all these events

Theorem 3.3. *For a set of n moving points, when they move according to polynomial functions of at most constant degree s, there exists a KDS to maintain the Equilateral Delaunay graph (*i.e., *the Semi-Yao graph) that uses linear space and processes $O(n^2\beta_{s+2}(n))$ events, each in amortized time $O(\log n)$.*

Proof. For each edge in the EDG, there exists a constant number of certificates; this implies that the size of the KDS is linear.

From Lemma 3.3, the Equilateral Delaunay graph changes if and only if the Semi-Yao graph changes. Fix a point p_i in the Yao graph and one of its wedges $W_l(p_i)$. Since the trajectory of each point $p_i(t) = (x_i(t), y_i(t))$ is defined by two polynomial functions of at most constant degree s, each point can insert into $\mathcal{V}_l(p_i)$ at most s times. The x_l-coordinates of the points inserted into $\mathcal{V}_l(p_i)$ create at most sn partial functions of at most constant degree s. From Theorem 2.2, the minimum value of these sn partial functions changes at most $\lambda_{s+2}(sn)$ times, which is equal to the number of all changes for the point with minimum x_l-coordinate among the points in $\mathcal{V}_l(p_i)$. Since s is a constant, we have that $\lambda_{s+2}(sn) = O(\lambda_{s+2}(n))$. Thus the number of all changes for all points is $O(n\lambda_{s+2}(n)) = O(n^2\beta_{s+2}(n))$.

The number of all the certificates over time is in the order of the number of the changes to EDT_0. When a change to EDT_0 occurs, we update the EDT_0 and replace the invalid certificate(s) in the priority queue with new valid one(s). The time to make a constant number of deletions/insertions into the priority queue is $O(\log n)$. Therefore, the total time to process all events is $O(n^2\beta_{s+2}(n)\log n)$. □

3.3 Kinetic All Nearest Neighbors

The Equilateral Delaunay graph (Semi-Yao graph) is a supergraph of the nearest neighbor graph (by Lemma 3.2). Therefore, in order to maintain the nearest neighbor to each point p_i, we need to track the edge with the minimum length among the edges incident to p_i in the Equilateral Delaunay graph.

Let $Inc(p_i)$ be the set all edges incident to p_i in the Equilateral Delaunay graph, and let n_i be the cardinality of the set $Inc(p_i)$. Using a dynamic and kinetic tournament tree (see Section 2.1), we can maintain the edge with the minimum length among the edges in $Inc(p_i)$. For each $Inc(p_i)$, $i = 1, 2, ..., n$, we construct a dynamic and kinetic tournament tree $\mathcal{T}_i$ whose elements are the edges in $Inc(p_i)$; the root of $\mathcal{T}_i$ maintains the edge with minimum length among all edges in $Inc(p_i)$.

Corollary 3.2. *Given a sequence of m_i insertions and deletions into $\mathcal{T}_i$. The tournament tree $\mathcal{T}_i$ generates $O(m_i\beta_{2s+2}(n_i)\log n_i)$ events, for a total cost of $O(m_i\beta_{2s+2}(n_i)\log^2 n_i)$. The tournament tree $\mathcal{T}_i$ of n_i elements can be constructed in $O(n_i)$ time.*

Proof. From Theorem 2.3 and the fact that the lengths of any two edges in $Inc(p_i)$ can become equal at most $2s$ times, the proof obtains. □

Now we can prove the following.

Lemma 3.4. *All the dynamic and kinetic tournament trees $\mathcal{T}_i$'s can be constructed in $O(n)$ time. These dynamic and kinetic tournament trees generate $O(n^2\beta^2_{2s+2}(n)\log n)$ events, for a total cost of $O(n^2\beta^2_{2s+2}(n)\log^2 n)$.*

Proof. By Corollary 3.2, all the dynamic and kinetic tournament trees $\mathcal{T}_i$, $i = 1, ..., n$, generate at most $O(\sum_{i=1}^{i=n} m_i\beta_{2s+2}(n_i)\log n_i) = O(\beta_{2s+2}(n)\log n\sum_{i=1}^{i=n} m_i)$ events. Since each edge is incident to two points, inserting (resp. deleting) an edge p_ip_j into the Equilateral Delaunay graph causes two insertions (resp. deletions) into the tournament trees $\mathcal{T}_i$ and $\mathcal{T}_j$. By Theorem 3.3, the number of all insertions/deletions into the tournament trees is $\sum_{i=1}^{i=n} m_i = O(n^2\beta_{s+2}(n)) = O(n^2\beta_{2s+2}(n))$. Hence the number of all events is $O(n^2\beta^2_{2s+2}(n)\log n)$, and the total cost is $O(n^2\beta^2_{2s+2}(n)\log^2 n)$. □

Now we can prove the following theorem, which gives the results about our kinetic data structure for the all nearest neighbors problem.

Theorem 3.4. *Our kinetic data structure for maintenance of all the nearest neighbors uses linear space and $O(n\log n)$ preprocessing time. It handles $O(n^2\beta^2_{2s+2}(n)\log n)$ events with total processing time $O(n^2\beta^2_{2s+2}(n)\log^2 n)$. It is compact, efficient, responsive in an amortized sense, and local on average.*

Proof. Since $\sum_i n_i = n$, the total size of all the tournament trees $\mathcal{T}_i$, $i = 1, ..., n$, is $O(n)$. The number of all edges in the EDG is $O(n)$. For each edge in the EDG, we define a constant number of certificates. Furthermore, the number of all certificates corresponding to the internal nodes of all $\mathcal{T}_i$ is linear. Thus the KDS is compact.

The ratio of the number of internal events $O(n^2\beta^2_{2s+2}(n)\log n)$ to the number of external events $O(n^2\beta_{2s})$ is polylogarithmic, which implies that the KDS is efficient.

By Lemma 3.4, the ratio of the total processing time to the number of internal events is polylogarithmic, and so the KDS is responsive in an amortized sense.

Since the number of all certificates is $O(n)$, each point participates in a constant number of certificates on average, which implies that the KDS is local on average. □

3.4 Kinetic Closest Pair

The edge p_ip_j with minimum length in the nearest neighbor graph gives the closest pair (p_i, p_j). Since the Equilateral Delaunay graph is a supergraph of the nearest neighbor graph, to maintain the closest pair (p_i, p_j) we need to maintain the edge with minimum length in the Equilateral Delaunay graph.

By constructing a dynamic and kinetic tournament tree, whose elements are the edges of the Equilateral Delaunay graph, we can maintain the closest pair (p_i, p_j) over time; the edge at the root of the dynamic and kinetic tournament tree gives the closest pair. The insertions and deletions into the dynamic and kinetic tournament tree occur when a change to the Equilateral Delaunay graph occurs. Therefore, we can obtain the same results for maintenance of the closest pair over time as we obtained for maintenance of all the nearest neighbors in Theorem 3.4; the analysis is similar.

Theorem 3.5. *Our kinetic data structure for maintenance of the closest pair uses linear space and $O(n \log n)$ preprocessing time. It handles $O(n^2\beta^2_{2s+2}(n)\log n)$ events with total processing time $O(n^2\beta^2_{2s+2}(n)\log^2 n)$, and it is compact, efficient, responsive in an amortized sense, and local on average.*

Chapter 4

Kinetic Euclidean Minimum Spanning Tree in the Plane

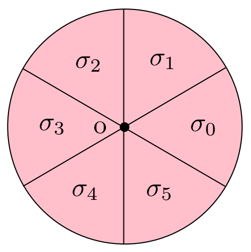
In this chapter, we provide a KDS for maintenance of the EMST, when the points move according to bounded degree polynomial functions. In Section 4.1, we introduce a new supergraph of the Yao graph, namely the *Pie Delaunay graph* (PDG), which in fact gives a new supergraph of the EMST, and give a new method for computing the Yao graph and the EMST; our approach is similar to that of idea of computing all the nearest neighbors and the closest pair in Chapter 3. We show how to maintain the PDG over time in Section 4.2. Using the kinetic version of the PDG, we provide the first KDS for maintenance of the Yao graph in Section 4.3. Finally, in Section 4.4, we use the Yao graph KDS to give a KDS for maintenance of the EMST.

Our KDS for maintenance of the EMST improves the previous KDS by Rahmati and Zarei [79] by a near-linear factor in the number of events.

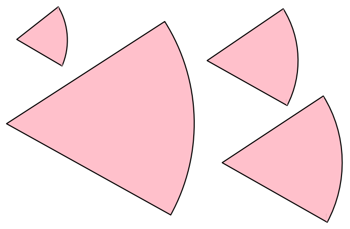
The results of this chapter were published as a paper in the Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2012) [4].

4.1 New Method for Computing the Yao Graph and the EMST

Consider a partition of a unit disk into six *pieces of pie* $\sigma_0, ..., \sigma_5$, each of angle $\pi/3$ with common apex at the origin o. For $0 \leq l \leq 5$, let σ_l span the angular range $[(2l-1)\pi/6, (2l+1)\pi/6)$, and call any translated and scaled copy of σ_l an *l-pie*; Figure 4.1 depicts a unit disc and some 0-pie's.



(a)



(b)

Figure 4.1: (a) Partitioning the unit disk into six pieces of pie. (b) Some 0-pie's.

We define a Delaunay triangulation, which we call a *Pie Delaunay triangulation*, of the set P of n points, based on the convex shape σ_l. Denote by PDT_l the Pie Delaunay triangulation based on the l-pie. For two points p_i and p_j in P, the edge p_ip_j is an edge of PDT_l if and only if there is an empty l-pie such that p_i and p_j are on its boundary. We define the *Pie Delaunay graph* (PDG) to be the union of all PDT_l for $i = 0, ..., 5$; *i.e.*, p_ip_j is a PDG edge if and only if it is an edge in PDT_l, where $0 \leq l \leq 5$.

Lemma 4.1. *The Pie Delaunay graph (PDG) of a set of n points can be constructed in $O(n \log n)$ time.*

Proof. Each σ_l, $0 \leq l \leq 5$, is a convex shape. Therefore, by Theorem 3.1, the corresponding Delaunay triangulation PDT_l can be constructed in $O(n \log n)$ time. $\square$

For each point $p_i \in P$, partition the plane into six wedges $W_0(p), ..., W_5(p)$ of angle $\pi/3$ where p_i is the common apex of the wedges. For $0 \leq l \leq 5$, let $W_l(p_i)$ span the angular range $[(2l-1)\pi/6, (2l+1)\pi/6)$ around p_i. Consider a *Yao graph* which is constructed by connecting the point p_i to its nearest points inside the wedges $W_l(p)$ for all $i = 0, ..., 5$. Denote the Yao graph of a set of n points by YG, the set of its edges by $\mathcal{E}(YG)$, and the set of Pie Delaunay graph edges by $\mathcal{E}(PDG)$. The following lemma shows that the Pie Delaunay graph is a supergraph of the Yao graph (YG).

Lemma 4.2. $\mathcal{E}(YG) \subseteq \mathcal{E}(PDG)$.

Proof. Let edge $p_ip_j \in \mathcal{E}(YG)$ and let p_j to be the nearest point to p_i inside the wedge $W_l(p_i)$; see Figure 4.2. The two sides of the wedge $W_l(p_i)$ are parallel to the

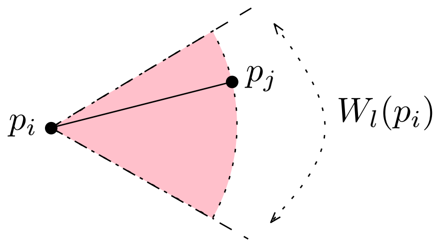


Figure 4.2: Nearest point to p_i inside the wedge $W_l(p_i)$.

two corresponding sides of σ_l, so there is an empty l-pie such that p_i and p_j lie on its boundary. Therefore, p_ip_j is an edge of PDT_l and hence $p_ip_j \in \mathcal{E}(PDG)$. □

Now we can state and prove the main result of this section.

Theorem 4.1. *The Yao graph and the EMST can be constructed in $O(n \log n)$ time.*

Proof. The Pie Delaunay graph is the union of six Pie Delaunay triangulations, which implies that it has a linear number of edges. By Lemma 4.2, the Pie Delaunay graph is a supergraph of the Yao graph. Thus by tracing over the edges incident to each point p_i, we can find the edge with minimum length inside each wedge $W_l(p_i)$, for $l = 0, ..., 5$; this gives the Yao graph. Since the Pie Delaunay graph can be constructed in time $O(n \log n)$ (by Lemma 4.1), the Yao graph can be constructed in time $O(n \log n)$.

The Yao graph is a supergraph of the EMST [99]. Thus the minimum spanning tree of the Yao graph is equal to the EMST. Since the cardinality of the set of edges in the Yao graph is at most $6n$, the EMST can be constructed using the Prim algorithm [73] or the Kruskal algorithm [63] in time $O(n \log n)$. □

4.2 Kinetic Pie Delaunay graph

Our KDS for maintenance of the Pie Delaunay graph is similar to the KDS for maintenance of the Equilateral Delaunay graph in Section 3.2. The Pie Delaunay graph (PDG) is the union of all PDT_l, for $l = 0, .., 5$: $\mathcal{E}(PDG) = \bigcup_l \mathcal{E}(PDT_l)$. Here, we only describe a KDS for PDT_0; the other PDT_l, for $l = 1, .., 5$, are handled similarly.

Similar to Section 3.2, we call each edge that is not on the boundary of the infinite face of PDT_0 an *interior edge* and the other edges on the boundary of the infinite face *hull edges*, and corresponding to them we define two kinds of certificates, *NotInPie* and *NotInCone*, respectively.

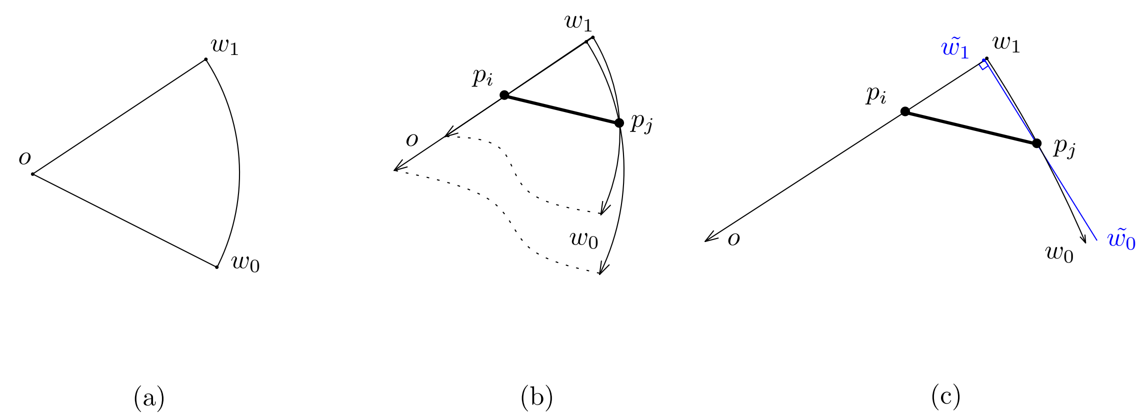


Figure 4.3: (a) A 0-pie. (b) Two k-cones corresponding to the hull edge p_ip_j. (c) The k-cone approaches a right-angled wedge as o goes to infinity.

NotInPie certificates. By definition, an interior edge $p_ip_j \in \mathcal{E}(PDT_0)$ is incident to two triangles of PDT_0 that together form a *quadrilateral*. Let $p_{r'}$ and p_r be the two other vertices of the quadrilateral. For the edge p_ip_j, we define a *NotInPie* certificate which certifies that point p_r (resp. $p_{r'}$) is outside the 0-pie passing through $p_{i'}$, $p_{j'}$, and $p_{r'}$ (resp. p_r). When the certificate fails, we replace p_ip_j by $p_rp_{r'}$. In general, when the certificates corresponding to an interior edge fails, we perform such an edge swap.

NotInCone certificates. Let o, w_0, and w_1 be vertices of a 0-pie (see Figure 4.3(a)). Two of the edges on the boundary of the 0-pie are line segments and one of them is an arc; denote the line segments by $\overline{ow_0}$ and $\overline{ow_1}$ and the arc by $\widehat{w_0w_1}$. By removing one of them and extending the line segment(s) to infinity, a cone can be created. We call these cones *pie-cones*. By definition, the edge p_ip_j is a hull edge of PDT_0 if and only if there exists an empty pie-cone such that p_i and p_j are on its boundary.

Consider the pie-cone ow_1w_0 corresponding to the edge p_ip_j where one of the endpoints p_i lies on the half-line $\overrightarrow{w_0o}$ and the other point p_j lies on the half-arc $\overrightarrow{w_0w_1}$ (see Figure 4.3(b)). Let $\overrightarrow{\tilde{w_1}\tilde{w_0}}$ be the half-line perpendicular to $\overrightarrow{w_1o}$ through p_j. For such a pie-cone we assume that the line segment $\overrightarrow{w_1o}$ goes to infinity. This means that w_1 (resp. w_0) tends to $\tilde{w_1}$ (resp. $\tilde{w_0}$) and the pie-cone approaches a right-angled wedge; see Figure 4.3(c).

Each hull edge p_ip_j is adjacent to at most four other hull edges, denoted by $p_ip_{s_2}$, $p_ip_{s_3}$, $p_jp_{s_4}$, $p_jp_{s_5}$, and incident to at most one triangle. Let p_{s_1} be the third vertex of this triangle if it exists; p_{s_1} can be one of the s_i where $2 \le i \le 5$. If p_ip_j is adjacent to at most four other edges, then it cannot be incident to a triangle. In particular, at any time, the number of points p_{s_i} is at most four. Therefore, for the pie-cone passing through p_i and p_j, we define at most four *NotInCone* certificates certifying that the p_{s_i} are outside of the pie-cone. Note that in the case that a pie-cone approaches a right-angled wedge (see Figure 4.3(c)), the certificate of the hull edge p_ip_j fails when a point either crosses the half-line $\overrightarrow{w_1 o}$, or reaches the line-segment $\overline{\tilde{w}_1 p_j}$, or crosses the half-line $\overrightarrow{p_j \tilde{w}_0}$.

The changes that can occur to PDT_0 are similar to the changes to EDT_0 and can easily be handled; see the paragraph "Consecutive Changes to EDT$_0$" in Section 3.2 for more details.

Next we state a theorem that enumerates the number of the combinatorial changes to the Pie Delaunay graph.

Theorem 4.2. *The number of all changes (edge insertions and edge deletions) to the Pie Delaunay graph of a set of n moving points with trajectories given by polynomial functions of at most constant degree s is $O(n^3\beta_{2s+2}(n))$.*

Proof. Consider PDT_0. The number of hull-edge changes to PDT_0 is $O(n^3)$ as three points are involved in any hull change. Since $n^3 = O(n^3\beta_{2s+2}(n))$, we focus on the number of changes to the triangles of PDT_0.

For each edge p_ip_j of a triangle in PDT_0, four different cases are possible as shown in Figure 4.4. It is easy to see that, for any triangle Δ in the PDT_0, case (a) of Figure 4.4 may happen to one of its edges. We charge any change to Δ to this edge. Therefore, we consider the number of combinatorial changes to PDT_0 for an arbitrary edge p_ip_j that satisfies case (a) of Figure 4.4. The analysis of other cases is similar.

Consider the corresponding 0-pie of the edge p_ip_j; see Figure 4.4(a). The two edges of this 0-pie are line segments $\overline{ow_0}$ and $\overline{ow_1}$ and one of them is an arc $\widehat{w_0w_1}$. The edge p_ip_j partitions this 0-pie into two convex areas op_ip_j and $p_ip_jw_0w_1$. Let $C_{w_0w_1}$ be the cone whose sides are created by removing the arc $\widehat{w_0w_1}$ of the 0-pie and extending the two line segments to infinity; the wedge $C_{w_0w_1}$ is the area between two half-lines $\overrightarrow{ow_0}$ and $\overrightarrow{ow_1}$. Let $\mathcal{V}(C_{w_0w_1})$ be the set of all points inside the wedge $C_{w_0w_1}$. The trajectory of the apex o of the cone $C_{w_0w_1}$ is a polynomial function of constant maximum degree s.

A change for triangle $p_ip_jp_r$ corresponding to p_ip_j occurs in two cases:

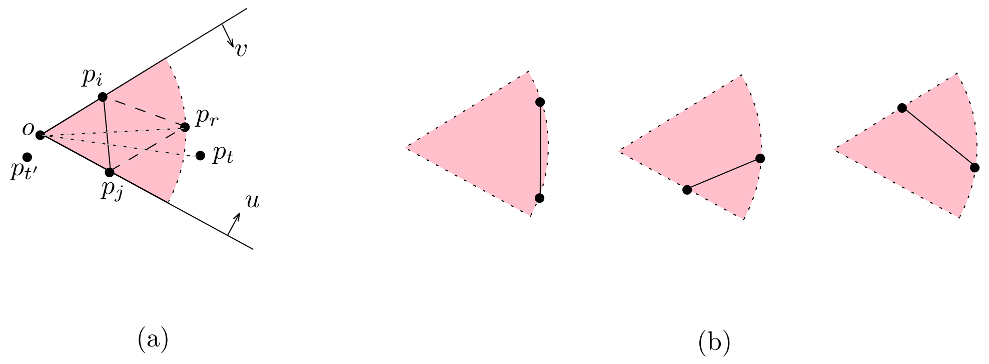


Figure 4.4: Combinatorial changes for an arbitrary edge p_ip_j.

(I) When a point such as $p_{t'}$ passes through the segment op_i or the segment op_j and enters inside the area op_ip_j; see Figure 4.4(a).

Map each point $p_i = (x_i(t), y_i(t))$ to a point $p'_i = (u_i(t), v_i(t))$ in a new parametric plane where $u_i(t) = x_i(t) + \sqrt{3}y_i(t)$ and $v_i(t) = x_i(t) - \sqrt{3}y_i(t)$. Passing the point $p_{t'}$ through the segment op_i or the segment op_j means that the point $p_{t'}$ exchanges its u-coordinate or its v-coordinate with the u-coordinate or v-coordinate of p'_i or p'_j. We call these changes *swap-changes*. Observe that the total number of swap-changes for all cases is bounded by the number of all swaps between points in their ordering with respect to the u-axis and v-axis. The number of the all u-swaps and v-swaps between points is at most $O(n^2)$.

(II) When a point moves inside the area $p_ip_jw_0w_1$. In particular, for some $p_t \in \mathcal{V}(C_{w_0w_1})$, the length of the edge op_t becomes smaller than the length of the edge op_r.

Note that since the degree of each function describing each point's motion is at most s, each point of P except p_i and p_j, can move inside the cone $C_{w_0w_1}$ at most s times. Summing over all points in P there are $O(sn)$ insertions into $\mathcal{V}(C_{w_0w_1})$. The distance of these points from the apex o, in the L_2 metric, creates $O(sn)$ partial functions, and each pair of these functions intersects at most $2s$ times. Therefore, the number of the combinatorial changes corresponding to an arbitrary edge p_ip_j equals $\lambda_{2s+2}(sn)$, which is equal to the number of the breakpoints in the lower envelope of sn partial functions of at most degree $2s$ (see Theorem 2.2). Since the maximum degree s is a constant, $\lambda_{2s+2}(sn) = O(\lambda_{2s+2}(n))$. The number of all possible edges is $O(n^2)$. Thus the number of the combinatorial changes corresponding to all edges is $O(n^2\lambda_{2s+2}(n))$.

Hence the number of changes to the Pie Delaunay graph is $O(n^3\beta_{2s+2}(n))$. □

Now we can get the main results of this section.

Theorem 4.3. *For a set of n points in the plane with trajectories given by polynomial functions of at most constant degree s, there exists a KDS for maintenance of the Pie Delaunay graph that uses linear space, $O(n \log n)$ preprocessing time, and that processes $O(n^3\beta_{2s+2}(n))$ events, each in amortized time $O(\log n)$.*

Proof. After any change to the Pie Delaunay graph, we replace a constant number of (invalid) certificates from the priority queue with new valid ones, which takes $O(\log n)$ time. Since the total number of events is $O(n^3\beta_{2s+2}(n))$ (by Theorem 4.2), the total processing time is $O(n^3\beta_{2s+2}(n)\log n)$. From Lemma 4.1, together with the fact that $|\mathcal{E}(PDG)| = O(n)$, our KDS uses $O(n)$ space and $O(n \log n)$ preprocessing time. □

4.3 Kinetic Yao Graph

To maintain the Yao graph, for each point $p_i \in P$, we must maintain the nearest points to p_i inside the wedges $W_l(p_i)$, where $0 \le l \le 5$. Since the Yao graph is a subgraph of the Pie Delaunay graph (by Lemma 4.2), to maintain the nearest points inside the wedges of p_i, we only need to track the edges of the Pie Delaunay graph incident to p_i with minimum length inside the wedges $W_l(p_i)$ for all $l = 0, ..., 5$.

Let $Inc_l(p_i)$ be the set all edges of the Pie Delaunay graph incident to p_i inside the wedge $W_l(p_i)$. We create a dynamic and kinetic tournament tree $\mathcal{T}_{l,i}$ whose elements are the edges in $Inc_l(p_i)$. The root of $\mathcal{T}_{l,i}$ maintains the winner, the edge with minimum length among all edges in $Inc_l(p_i)$.

Theorem 4.4. *The KDS for maintenance of the Yao graph uses $O(n)$ space, $O(n \log n)$ preprocessing time, and processes $O(n^3\beta^2_{2s+2}\log n)$ (internal) events with total processing time $O(n^3\beta^2_{2s+2}\log^2 n)$. It is compact, responsive in an amortized sense, and local on average, but it is not efficient.*

Proof. Given the KDS of the Pie Delaunay graph and making an analysis similar to that of Corollary 3.4 and Theorem 3.4, the proof obtains. □

For *linearly* moving points in the plane, Katoh *et al.* [58] showed that the number of changes to the Yao graph is $O(n^2\beta_4(n))$. In the following theorem we bound the number of combinatorial changes to the Yao graph of a set of moving points

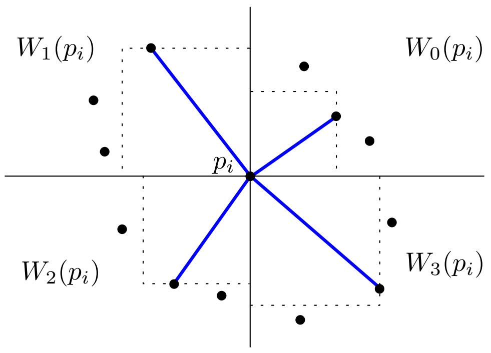


Figure 4.5: The nearest point to p_i inside each wedge around p_i in the Yao graph in the L_∞ metric.

whose trajectories are given by polynomial functions of at most constant degree s. For maintenance of the Yao graph, our KDS processes $O(n^3\beta_{2s+2}^2 \log n)$ events, but the following proves that the number of exact changes to the Yao graph is nearly quadratic, which explains why our KDS is not efficient.

Lemma 4.3. *The number of all changes to the Yao graph, when the points move with polynomial trajectories of at most constant degree s, is $O(n^2\beta_{2s+2}(n))$.*

Proof. Consider the point $p_i \in P$ and one of its wedges $W_l(p_i)$. Each of other points in P can be moved inside the wedge $W_l(p_i)$ at most s times. Thus there exist $O(sn)$ insertions into the wedge $W_l(p_i)$. The distance of these points from p_i creates $O(sn)$ partial functions; each pair of these functions intersects at most $2s$ times. By Theorem 2.2, the lower envelope of these functions, which in fact gives the edge with minimum length, changes at most $\lambda_{2s+2}(sn) = O(\lambda_{2s+2}(n))$ times.

Hence, the number of all changes to the Yao graph of a set of n moving points is $O(n\lambda_{2s+2}(n))$. □

Remark 4.1. Using an argument similar to the KDS we obtained for the Yao graph in the L_2 metric, a KDS for Yao graph in the L_∞ metric can be obtained; Figure 4.5 depicts the nearest points to p_i inside the wedges $W_l(p_i)$, $l = 0, ..., 3$, in the Yao graph in the L_∞ metric.

Denote by □ the *unit square* with corners at $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$ in a Cartesian coordinate system, and call any translated and scaled copy of □ an *SQR*. The edge p_ip_j is an edge of the Delaunay triangulation based on an SQR in the L_∞

metric [1] if and only if there is an empty SQR such that p_i and p_j are on its boundary, *i.e.*, the interior of SQR contains no point of P.

Abam *et al.* [3] showed how to maintain a Delaunay triangulation based on a diamond. They claimed that a Delaunay triangulation based on a diamond can be maintained by processing at most $O(n\lambda_{s+2}(n))$ events, each in worst-case time $O(\log n)$. A similar scenario to that of the example in Section 3.2 can happen to a KDS for maintenance of a Delaunay triangulation based on a diamond; in particular, we might need to apply a sequence of changes to the Delaunay triangulation, when an event occurs. Thus a correction to the claim by Abam and de Berg is that each event can be handled in amortized time $O(\log n)$.

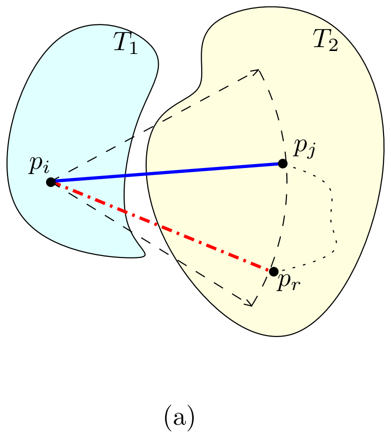
Each SQR is a diamond, so the approach by Abam and de Berg applies. The Delaunay triangulation based on an SQR can be maintained kinetically by processing at most $O(n\lambda_{s+2}(n))$ events, each in amortized time $O(\log n)$. The Delaunay triangulation based on an SQR is a supergraph for the Yao graph in the L_∞ metric. Therefore, we can have a KDS for the Yao graph in the L_∞ metric that uses $O(n)$ space, $O(n \log n)$ preprocessing time, and that processes $O(n^2\beta_{s+2}^2(n)\log n)$ events, each in amortized time $O(\log n)$.

4.4 Kinetic EMST

Our kinetic approach for maintaining the EMST is based on the fact that the EMST is a subgraph of the Yao graph, where the number of the wedges around each point in the Yao graph is greater than or equal to six [99].

The edges of the Yao graph are stored at the roots of the dynamic and kinetic tournament trees $\mathcal{T}_{l,i}$, for each point $p_i \in P$ and $l = 1, ..., 6n$ (see Section 4.3). Let L be a list of the Yao graph edges, sorted with respect to their Euclidean lengths. A change to the EMST may occur when two edges in L change their ordering. For each pair of consecutive edges in L, we define a certificate certifying the respective sorted order of the edges. Whenever the ordering of two edges in this list is changed, we apply the required changes to the EMST KDS. Therefore, to update the EMST when the points are moving, we must track the changes to L. There exist two types of changes to L:

[1]The Delaunay triangulation in the L_1 metric can be constructed/maintained analogously, by rotating all points 45 degrees around the origin and constructing/maintaining the Delaunay triangulation in the L_∞ metric.



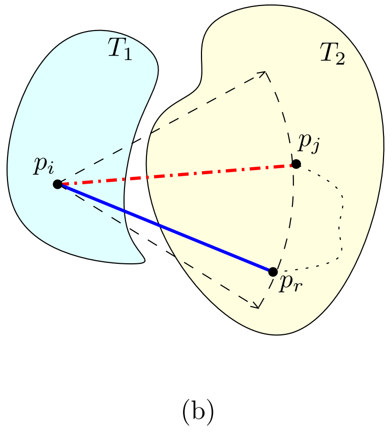


Figure 4.6: The edge connecting two subtrees $T_1(P_1, E_1)$ and $T_2(P_2, E_2)$: (a) At time t^-, $|p_ip_r| > |p_ip_j| > |p_jp_r|$ and the edge connecting T_1 and T_2 is p_ip_j. (b) At time t^+, $|p_ip_j| > |p_ip_r| > |p_jp_r|$ and the edge connecting T_1 and T_2 is p_ip_r.

(I) edge insertion and edge deletion from L, and

(II) a change in the order of two consecutive edges in L

The following discusses how to handle these two types of events.

Handling Case (I). As soon as an edge is deleted from L a new one is inserted. Both the deleted edge and the inserted edge are in the same dynamic and kinetic tournament tree, and both of them have a common endpoint; see Figure 4.6. Call the deleted edge and the inserted edge p_ip_j and p_ip_r, respectively. Denote by $\mathcal{T}_{i,l}$ the dynamic and kinetic tournament tree that contains p_ip_j and p_ip_r. The deleted edge p_ip_j can be one of the EMST edges at time t^- and if so, we have to find a new edge to repair the EMST at time t^+. The following lemma proves that this new edge is p_ip_r.

Lemma 4.4. *Let p_ip_j be the winner of the dynamic and kinetic tournament tree $\mathcal{T}_{i,l}$. Suppose $p_ip_j \in \mathcal{E}(EMST)$ at time t^- and let p_ip_r be the winner of $\mathcal{T}_{i,l}$ at time t^+. Then (i) at time t^-, $p_ip_r \notin \mathcal{E}(EMST)$, and (ii) at time t^+, $p_ip_r \in \mathcal{E}(EMST)$ and $p_ip_j \notin \mathcal{E}(EMST)$.*

Proof. Deleting an edge p_ip_j from EMST creates two subtrees $T_1(P_1, E_1)$ and $T_2(P_2, E_2)$. Let $p_i \in P_1$ and $p_j \in P_2$; see Figure 4.6. At time t^-, since $p_ip_j \in \mathcal{E}(EMST)$, $|p_ip_r| > |p_ip_j| > |p_jp_r|$, and $\angle p_jp_ip_r \leq \pi/3$, we have that $p_r \in P_2$. This can be concluded by contradiction. Thus (i) at time t^-, $p_ip_r \notin \mathcal{E}(EMST)$.

The proof that $p_ip_j \notin \mathcal{E}(EMST)$ at time t^+ is analogous to the proof for (i). Thus, at time t^+, the EMST is the union of two trees T_1 and T_2 and the edge p_ip_r. □

Handling Case (II). Let $path(e)$ be the simple path in the EMST between the endpoints of edge e and let $|e|$ be the Euclidean length of e. A change in the sorted list L corresponds to a pair of edges e and e' in $\mathcal{E}(YG)$ such that at time t^-, $|e| < |e'|$, and at time t^+, $|e| > |e'|$. Thus at time t, e may be replaced by e' in the EMST. It is easy to see the following.

Observation 4.1. *The EMST changes if and only if at time t^-, for some e and e', $|e| < |e'|$, $e \in \mathcal{E}(EMST)$, $e' \notin \mathcal{E}(EMST)$, $e \in path(e')$, and at time t^+, $|e| > |e'|$.*

Such events can be detected and maintained in $O(\log n)$ time per operation using the *link-cut tree data structure* of Sleator and Tarjan [86].

Given a KDS for maintenance of the Yao graph, the following bounds the number of events for maintaining the EMST.

Lemma 4.5. *Given a Yao graph KDS for a set of n points moving with polynomial trajectories of constant maximum degree s, there exists a KDS for maintenance of the EMST that processes $O(n^3\beta_{2s+2}(n))$ events.*

Proof. The set of Yao graph edges is a superset of the set of the EMST edges, and any change in the order of consecutive edges in the sorted list L of the Yao graph edges may change the EMST. More precisely, any edge insertion/deletion in the Yao graph implies an insertion/deletion into L, and each insertion into L may cause $O(n)$ changes to the order of the edges in L. Each change in the order may cause a change in the EMST. From Lemma 4.3, the number of all insertions and deletions into the sorted list L is $O(n^2\beta_{2s+2}(n))$. Therefore, given a KDS for the Yao graph, the number of events that our KDS processes is $O(n^3\beta_{2s+2}(n))$. □

Now we obtain the following.

Theorem 4.5. *The KDS for maintenance of the EMST uses linear space and requires $O(n \log n)$ preprocessing time. The KDS processes $O(n^3\beta^2_{2s+2}(n)\log n)$ events, each in amortized time $O(\log n)$. The KDS is compact, responsive in an amortized sense, and local on average, but it is not efficient.*

Proof. The KDS for maintenance of the EMST uses the Pie Delaunay graph KDS and the Yao graph KDS. Therefore, by combining the results of Theorems 4.3 and

4.4, and Lemma 4.5, and since handling Cases (I) and (II) can be done in $O(\log n)$ time, the proof obtains. $\square$

Chapter 5

Kinetic All Nearest Neighbors in Higher Dimensions

In this chapter, we provide a KDS for maintenance of all the nearest neighbors in any fixed dimension d. Section 5.1 describes the construction of the Semi-Yao graph, a supergraph of the nearest neighbor graph in $\mathbb{R}^d$, and gives a solution to the all nearest neighbors problem. In Section 5.2, we show how the Semi-Yao graph can be maintained kinetically, when the points are moving along trajectories of bounded degree polynomials. Next, in Section 5.3, we use the kinetic Semi-Yao graph to give a KDS for maintenance of all the nearest neighbors. Finally, in Section 5.4, we show how to maintain all the $(1+\epsilon)$-nearest neighbors with better performance than our KDS in Section 5.3 for the exact nearest neighbors.

The results of this chapter were published as a paper in the Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG 2014) [75].

5.1 Computing the Semi-Yao Graph and All Nearest Neighbors in $\mathbb{R}^d$

Here we describe the construction of the Semi-Yao graph and construction of all the nearest neighbors, which will aid in understanding how our kinetic approach works.

Let $\overrightarrow{v}$ be a unit vector in $\mathbb{R}^d$ with apex at the origin o, and let θ be a constant. We define an *infinite right circular cone* K with respect to $\overrightarrow{v}$ and θ to be the set of points $x \in \mathbb{R}^d$ such that the angle between $\overrightarrow{ox}$ and $\overrightarrow{v}$ is at most $\theta/2$; Figure 5.1(a) depicts an infinite right circular cone in $\mathbb{R}^3$. We define a *polyhedral cone* of opening

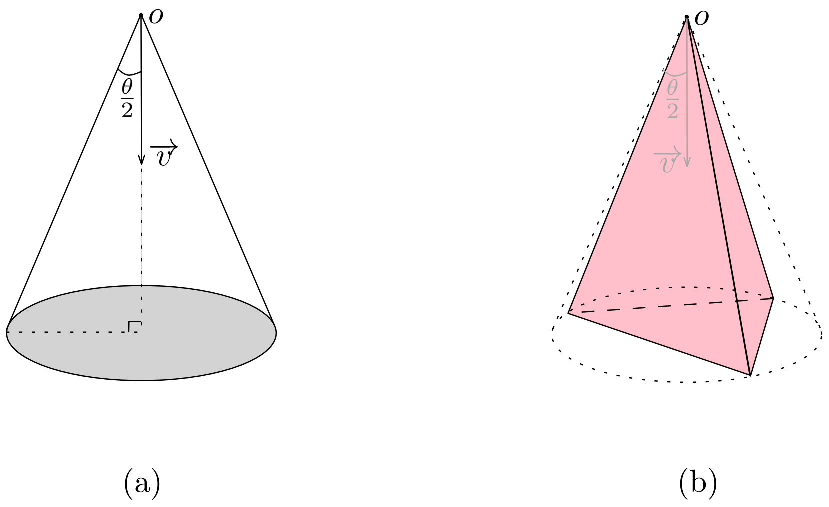


Figure 5.1: An infinite right circular cone and a polyhedral cone.

angle θ with respect to $\overrightarrow{v}$ to be the intersection of d nonparallel half-spaces such that the intersection is contained in an infinite right circular cone K with respect to $\overrightarrow{v}$ and θ, and such that all the half-spaces contain the origin o; Figure 5.1(b) depicts a polyhedral cone in $\mathbb{R}^3$, which is contained in the infinite right circular cone of Figure 5.1(a). The angle between any two rays inside a polyhedral cone of opening angle θ emanating from o is at most θ.

Lemma 5.1. [2] *The d-dimensional space around a point can be covered by a collection of $c = O(1/\theta^{d-1})$ interior-disjoint polyhedral cones of opening angle θ.*

Let $\mathcal{C} = \{C_0, ..., C_{c-1}\}$ be a set of polyhedral cones of opening angle θ with their apex at the origin o that together cover $\mathbb{R}^d$. We assume d is arbitrary but fixed, so c is a constant. Consider a polyhedral cone $C_l \in \mathcal{C}$ with respect to $\overrightarrow{v}$. Denote by x_l the vector in the direction of the unit vector $\overrightarrow{v}$ of C_l, $0 \leq l \leq c-1$. Denote by $f_1, ..., f_d$ the bounding half-spaces of C_l. Let u_i be the normal to f_i, $1 \leq i \leq d$. Figure 5.2 depicts u_1 and u_2 for the half-spaces f_1 and f_2 of a polyhedral cone $C_l \in \mathcal{C}$ in $\mathbb{R}^2$. Let $C_l(p)$ denote a translated copy of C_l with apex at p.

Now we consider a construction of the Semi-Yao graph in $\mathbb{R}^d$ for the set $\mathcal{C}$ of polyhedral cones as follows. Connect each point $p \in P$ to the point in $P \cap C_l(p)$, $0 \leq l \leq c-1$, whose x_l-coordinate is minimum. Figure 5.3 depicts some edges incident to the point p in the Semi-Yao graph in $\mathbb{R}^2$, where $\theta = \pi/3$; the dotted lines are orthogonal to the cone axes $x_0, ..., x_5$; here $x_0 = -x_3$, $x_1 = -x_4$, and $x_2 = -x_5$.

Lemma 5.2. `(Lemma 8.1. of [10])` *Let p be the nearest point to q and let $C_l(p)$*

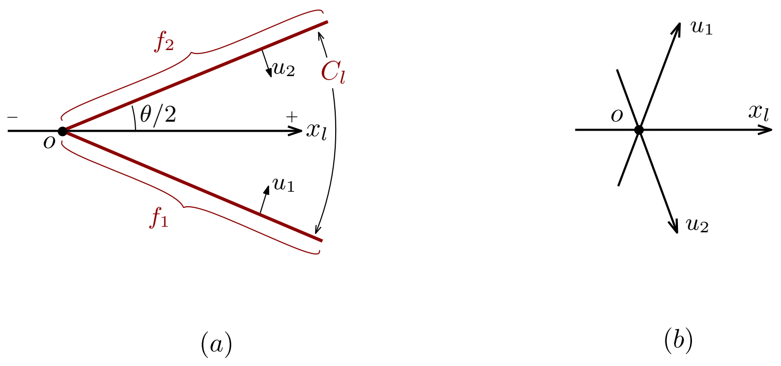


Figure 5.2: The cone C_l with apex at o.

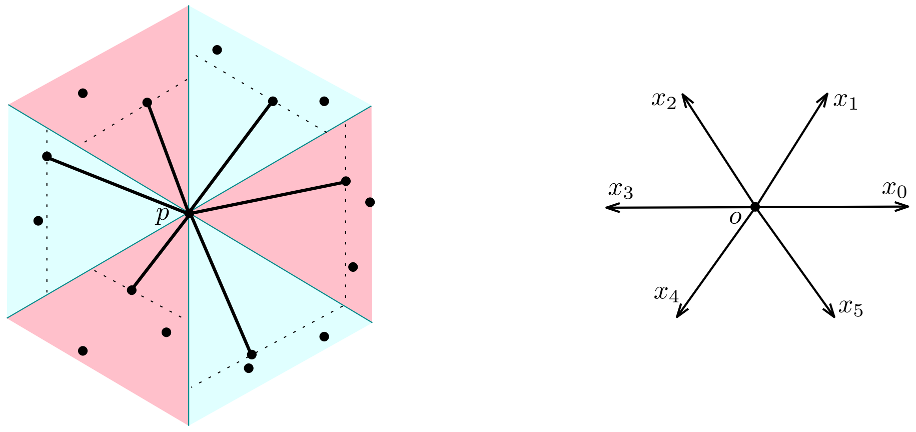


Figure 5.3: The case $d = 2$ and $c = 6$. For each l, $0 \geq l \geq c-1$, the apex point p is connected to the point in $P \cap C_l(p)$ that has the minimum x_l-coordinate.

be a cone of opening angle $\theta \leq \pi/3$ that contains q (see Figure 5.4). *Then q has the minimum x_l-coordinate among the points in $P \cap C_l(p)$.*

For a set of points in $\mathbb{R}^2$, in Section 3.1, we used a 2-dimensional version of Lemma 5.2 to show that the Semi-Yao graph is a super-graph of the nearest neighbor graph. It is easy to see the same result for a set of points in higher dimensions.

Lemma 5.3. *The Semi-Yao graph of a set of points in $\mathbb{R}^d$ is a super-graph of the nearest neighbor graph.*

Proof. Let (p, q) be an edge in the nearest neighbor graph such that p is the nearest neighbor to q. The point q is in some cone $C_l(p)$. The restriction $\theta \leq \pi/3$ of the cone $C_l(p)$ together with Lemma 5.2 imply that the point q has the minimum length projection on x_l among the points in $P \cap C_l(p)$. Thus (p, q) is an edge of the Semi-Yao graph. □

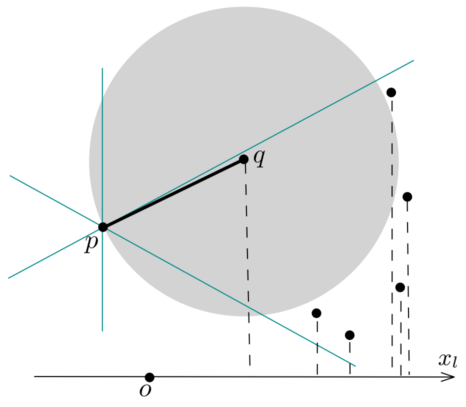


Figure 5.4: The point p is the nearest neighbor to q, so q has the minimum x_l-coordinate among the points in $P \cap C_l(p)$.

By Theorem 2.5, we can obtain the following, which gives the construction time of the Semi-Yao graph in higher dimensions.

Lemma 5.4. *The Semi-Yao graph of a set of n points in $\mathbb{R}^d$ can be constructed in time $O(n \log^{d-1} n)$.*

Now we obtain the following.

Lemma 5.5. *Given the Semi-Yao graph, the all nearest neighbors problem in $\mathbb{R}^d$ can be solved in $O(n)$ time.*

Proof. The Semi-Yao graph is a supergraph of the nearest neighbor graph (by Lemma 5.3). Thus, by examining the edges incident to each point in the Semi-Yao graph, we can find the nearest neighbor to the point. Since the Semi-Yao graph has $O(n)$ edges, all the nearest neighbors can be reported in linear time. $\square$

5.2 Kinetic Semi-Yao Graphs

Here we first in Section 5.2.1 provide data structures for maintaining the Semi-Yao graph, and then in Section 5.2.2 we show how to update these data structures when the points move.

5.2.1 Preprocessing Step

The Semi-Yao graph remains unchanged as long as the order of the points in each of the coordinates $u_1, ..., u_d$, and x_l associated to each cone $C_l \in \mathcal{C}$ remains unchanged.

Thus to maintain the Semi-Yao graph over time, we distinguish between two types of events:

- *u-swap event:* Such an event occurs if two points exchange their order in the u_i-coordinate.
- *x-swap event:* This event occurs if two points exchange their order in the x_l-coordinate.

The u-swap events can be tracked by maintaining the sorted lists $L(u_1), ..., L(u_d)$ of the points in each of the coordinates $u_1, ..., u_d$. In addition, we maintain a list $L(x_l)$ of the points sorted according to their x_l-coordinates to track the x-swap events.

We use kinetic ranked-based range trees (RBRTs) $\mathcal{T}_l$, for $l = 0, ..., c-1$ (see Section 2.2) as basic data structures for maintaining the Semi-Yao graph. Note that, from Section 2.2, we also use the sorted lists $L(u_1), ..., L(u_d)$ to apply changes to a RBRT $\mathcal{T}_l$ when the points move.

For a fixed cone $C_l \in \mathcal{C}$, denote by $\Psi_l = \{(B_1, R_1), ..., (B_m, R_m)\}$ the corresponding cone separated pair decomposition (CSPD) to $\mathcal{T}_l$ (see Section 2.2). Let r_j be the point with minimum x_l-coordinate among the points in R_j. Denote by $\ddot{w}_l$ the point in $P \cap C_l(w)$ with minimum x_l-coordinate. Note that to maintain the Semi-Yao graph, for each point $w \in P$, in fact we must track $\ddot{w}_l$; $\ddot{w}_l$ is the point with the minimum x_l-coordinate among the points r_j's, where the subscripts j are such that $P \cap C_l(w) = \bigcup_j R_j$. To apply required changes to $\ddot{w}_l$ for all $w \in P$, when an event occurs, in addition r_j, we need to maintain more information for each subscript j (*i.e.*, at each internal node v at level d of $\mathcal{T}_l$). The next paragraph describes the extra information.

Allocate a *label* to each point in P. Let $B'_j = \{(w, \ddot{w}_l) | \ w \in B_j\}$ and let $L(B'_j)$ be a sorted list of the pairs of B'_j according to the labels of the second components $\ddot{w}$ of the pairs $(w, \ddot{w}_l)$. This sorted list is used to answer the following query while processing x-swap events: Given a query point p, find all the points $w \in B_j$ such that $\ddot{w}_l = p$. Since we perform updates (insertions/deletions) to the sorted lists $L(B'_j)$ over time, we implement them using a dynamic binary search tree (*e.g.*, a *red-black tree*); each update is performed in worst-case time $O(\log n)$. Furthermore, we maintain a set of links between each point $w \in P$ and the pair $(w, \ddot{w}_l)$ in the sorted lists $L(B'_j)$ where $w \in B_j$; denote this set by $Link(w)$; we use this set to efficiently delete the pair $(w, \ddot{w}_l)$ from the sorted lists $L(B'_j)$ when we are handling the events.

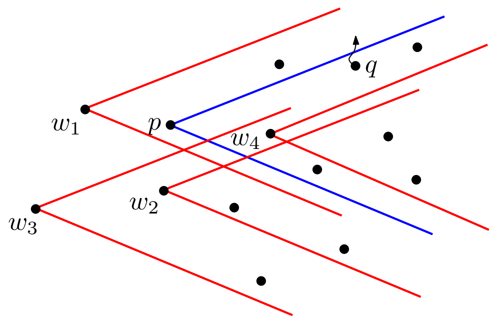


Figure 5.5: A u-swap between p and q does not change the points in other cones.

In the preprocessing step before the motion, for any subscript j and for any point $w \in P$, we find r_j and $\ddot{w}_l$, and then we construct $L(B'_j)$ and $Link(w)$.

Lemma 5.6. *Our KDS uses $O(n \log^d n)$ space and $O(n \log^{d+1} n)$ preprocessing time.*

Proof. By Theorem 2.6, each point $p \in P$ is in at most $O(\log^d n)$ sets B_j's, and $O(\log^d n)$ sets R_j's, so the cardinality of each set $Link(p)$ is $O(\log^d n)$, and the number of the members in the sets B_j and R_j, for all j's, is $O(n \log^d n)$. This implies that (i) the KDS uses $O(n \log^d n)$ storage, (ii) we can find all the r_j and $\ddot{w}$ in time $O(n \log^d n)$, and (iii) we can sort all the B'_j according to the labels of the second components in $O(n \log^{d+1} n)$ time, and then using the sorted lists $L(B'_j)$, we can create $Link(p)$ for all $p \in P$ in the same time $O(n \log^{d+1} n)$. $\square$

5.2.2 Processing the Events

Now let the points move. The following shows how to maintain and reorganize $Link(w)$, $L(B'_j)$ and r_j, for any subscript j and for any point $w \in P$, when a u-swap event or an x-swap event occurs. Note that maintenance of the sets $Link(w)$, for all $w \in P$, gives a kinetic maintenance of the Semi-Yao graph.

Handling u-swap events. Consider a u-swap between p and q. Without loss of generality, assume $q \in C_l(p)$ before the event; see Figure 5.5. After the event, q moves outside the cone $C_l(p)$. Note that this event does not change the points in cones $C_l(w)$ of other points $w \in P$. Therefore, the only change that can happen to the Semi-Yao graph is deletion of an edge incident to p inside the cone $C_l(p)$ and addition of a new one.

In particular, when two points p and q exchange their order in the u_i-coordinate, we perform the following steps.

U1) We swap p and q in the sorted list $L(u_i)$ and update the invalid certificates with new valid ones.

U2) A u-swap event may change the structure of the RBRT $\mathcal{T}_l$, so we update the RBRT T_l.

U3) We update the values in $\{r_j \mid p \in R_j \ \vee \ q \in R_j\}$.

U4) We delete the pairs $(p, \ddot{p}_l)$ of the sorted lists $L(B'_j)$ where $p \in B_j$.

U5) We delete the members of $Link(p)$.

U6) We find the point $\ddot{p}_l$ in $P \cap C_l(p)$ whose x_l-coordinate is minimum among all the r_j such that $p \in B_j$.

U7) We add the pair $(p, \ddot{p}_l)$ into all the sorted lists $L(B'_j)$ according to the label of $\ddot{p}_l$. Then we construct $Link(p)$ of the new links between p and the pair $(p, \ddot{p}_l)$ in the sorted lists $L(B'_j)$.

The following lemma gives the complexity of the steps U1,...,U7 above.

Lemma 5.7. *For maintenance of the Semi-Yao graph, our KDS handles $O(n^2)$ u-swap events, each in worst-case time $O(\log^{d+1} n)$.*

Proof. For a fixed dimension d, the number of swaps between the points in the sorted lists $L(u_i)$, $1 \leq i \leq d$, is $O(n^2)$.

Applying a constant number of changes to the priority queue takes $O(\log n)$ time (Step U1). By Theorem 2.6, an update to the RBRT T_l takes $O(\log^d n)$ (Step U2), and all the R_j can be found in $O(\log^d n)$ time, so the values r_j can be updated in worst-case time $O(\log^d n)$ (Step U3).

By using the links in $Link(p)$, Step U4 can be done in time $O(\log^{d+1} n)$. Each point is in $O(\log^d n)$ sets B_j, so Step U6 takes $O(\log^d n)$ time. Since each operation in a sorted list $L(B'_j)$ can be done in $O(\log n)$ time, Step U7 takes $O(\log^{d+1} n)$ time. □

Handling x-swap events. The structure of RBRT $\mathcal{T}_l$ remains unchanged when an x-swap event between p and q occurs. Such an event might change the second components of the pairs in some sorted lists $L(B'(.))$ and if so, we must apply the changes to the Semi-Yao graph.

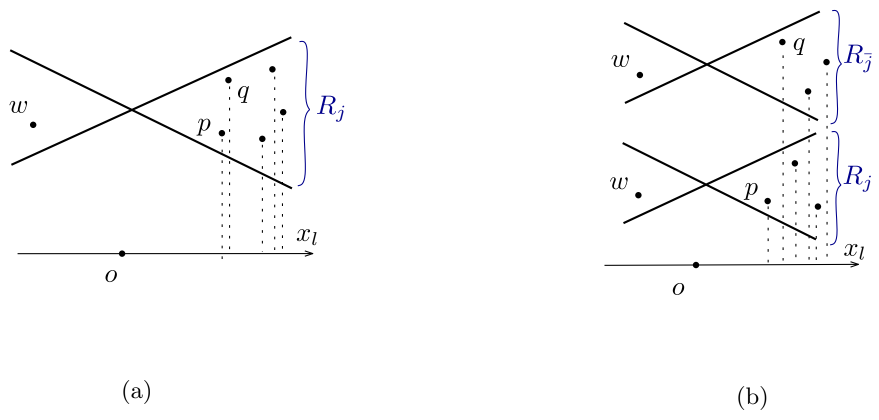


Figure 5.6: Two cases when an x-swap between p and q occurs.

Let $x_l(p)$ be the x_l-coordinate of p. Let p and q be two consecutive points with p preceding q in the sorted list $L(x_l)$ before the event ($i.e.$, $x_l(p) < x_l(q)$). When p and q exchange their order, we first perform the following step.

`X1)` We swap p and q in $L(x_l)$ and update the invalid certificates with new valid ones.

The number of all changes to the Semi-Yao graph depends on how many points $w \in P$ have both p and q in their cones $C_l(w)$. While reporting the points in $P \cap C_l(w)$, note that w can have both p and q in the same set R_j (see Figure 5.6(a)) or in two different sets R_j and $R_{\bar{j}}$ (see Figure 5.6(b)). To find such points w, when an x-swap event between p and q occurs, we seek (I) subscripts j where $\{p, q\} \subseteq R_j$, and (II) subscripts j and $\bar{j}$ where $p \in R_j$ and $q \in R_{\bar{j}}$. In the first case, we must find any point $w \in B_j$ such that p is the point with minimum x_l-coordinate in the cone $C_l(w)$, meaning that $\ddot{w}_l = p$. Then we replace p by q after the event ($\ddot{w}_l = q$). This means that we replace the edge wp of the Semi-Yao graph with wq.

Note that in the second case there is no point $w_1 \in B_j$ such that $\ddot{w}_{1l} = q$, because $x_l(p) < x_l(q)$. Also note that if there is a point $w_2 \in B_j$ such that $\ddot{w}_{2l} = p$, we change the value of $\ddot{w}_{2l}$ to q if $q \in C_l(w_2)$; in the case that $q \in C_l(w_2)$, we can find w_2 in $B_{\bar{j}}$ and we do not need to check whether such points w_2 are in B_j or not. Therefore, for the second case, we only need to check whether there is a point $w \in B_{\bar{j}}$ such that $\ddot{w}_l = p$; if so, we change the value of $\ddot{w}_l$ to q ($\ddot{w}_l = q$).

From the above discussion, the following three steps, together with Step `X1`, summarize the update mechanism of our KDS for maintenance of the Semi-Yao graph when an x-swap event occurs.

X2) We find all the subscripts j such that $\{p, q\} \subseteq R_j$ and $r_j = p$. Also, find all the subscripts j where $r_j = q$ (see Figure 5.6).

X3) For each subscript j (from Step X2), we find all the pairs $(w, \ddot{w}_l)$ in the sorted list $L(B'_j)$ where $\ddot{w}_l = p$.

X4) For each w (from Step X3), using the links in *Link*(w), we find all the corresponding sorted lists $L(B'_j)$ where $w \in B_j$, delete the pair $(w, \ddot{w}_l)$ from them, change the value of the second component $\ddot{w}_l$ to q, and add $(w, \ddot{w}_l)$ into the sorted lists according to the label of q.

The number of edges incident to a point p in the Semi-Yao graph is $O(n)$. Thus when an x-swap event between p and some point q occurs, it might cause $O(n)$ changes to the Semi-Yao graph. The following lemma shows that an x-swap event can be handled in polylogarithmic amortized time.

Lemma 5.8. *For maintenance of the Semi-Yao graph, our KDS handles $O(n^2)$ x-swap events with total processing time $O(n^2\beta_{s+2}(n)\log^{d+1} n)$.*

Proof. Step X1 takes $O(\log n)$ time. By Theorem 2.6, all the subscripts j at Step X2 can be found in $O(\log^d n)$ time.

For each j of Step x3, the update mechanism spends $O(\log n + k_j)$ time where k_j is the number of all the pairs $(w, \ddot{w}_l) \in B'_j$ such that $\ddot{w}_l = p$. For all the subscripts j, the second step takes $O(\log^{d+1} n + \sum_j k_j)$ time. Note that $\sum_j k_j$ is equal to the number of exact changes to the Semi-Yao graph. Since the number of changes to the Semi-Yao graph of a set of n moving points in a fixed dimension d is $O(n^2\beta_{s+2}(n))$ (see Theorem 3.3), the total processing time of Step X3 for all the $O(n^2)$ x-swap events is $O(n^2\log^{d+1} n + n^2\beta_{s+2}(n)) = O(n^2\log^{d+1} n)$.

For each w of Step X4, the processing time to apply changes to the KDS, which in fact is a change to the Semi-Yao graph, is $O(\log^{d+1} n)$. Thus the update mechanism spends $O(n^2\beta_{s+2}(n)\log^{d+1} n)$ time to handle all the $O(n^2)$ events.

Hence the total processing time for all the x-swap events is $O(n^2\beta_{s+2}(n)\log^{d+1} n)$. □

Now we state the main result of this section, which summarizes the complexity of the proposed KDS for the Semi-Yao graph.

Theorem 5.1. *Our KDS for maintenance of the Semi-Yao graph of a set of n moving points in $\mathbb{R}^d$, where the coordinates of each point are polynomial functions of at* *most constant degree s, uses $O(n \log^{d+1} n)$ preprocessing time, $O(n \log^{d} n)$ space and handles $O(n^2)$ events with a total cost of $O(n^2 \beta_{s+2}(n) \log^{d+1} n)$. The KDS is compact, efficient, responsive (in an amortized sense), and local.*

Proof. From Lemma 5.6, the KDS uses $O(n \log^{d+1} n)$ preprocessing time and $O(n \log^{d} n)$ space. The toral cost to process all the $O(n^2)$ events is $O(n^2 \beta_{2s+2}(n) \log^{d+1} n)$ (see Lemmas 5.7 and 5.8); this implies that the KDS is responsive in an amortized sense.

Since the number of the certificates is $O(n)$, the KDS is compact.

A particular point in a sorted list $L(u_i)$ participates in two certificates, one created with the previous point and one with the next point. Thus the number of certificates associated to a particular point is $O(1)$, which implies that the KDS is local.

Since the number of the external events is $O(n^2 \beta_{s+2}(n))$ and the number of the events that the KDS processes is $O(n^2)$, the KDS is efficient. □

Remark 6.1. We have provided two KDS's for maintenance of the Semi-Yao graph. The KDS in Section 5.2 generalizes the KDS in Section 3.2 that only works in $\mathbb{R}^2$.

Also, our KDS in Section 5.2 yields improvements in the number of events and the locality of the KDS: In the KDS of Section 5.2, each point participates in $O(1)$ certificates, but each point in the KDS of Section 3.2 participates in $O(n)$ certificates. The KDS of Section 5.2 handles $O(n^2)$ events (see Theorem 5.1), but the KDS of Section 3.2 handles $O(n^2 \beta_{s+2}(n))$ events (see Theorem 3.3).

5.3 Kinetic All Nearest Neighbors

Given the kinetic Semi-Yao graph, a super-graph of the nearest neighbor graph over time, in the same way to our approach in Section 3.3, we can easily maintain the nearest neighbor to each point $p \in P$. We use a dynamic and kinetic tournament tree to maintain the nearest neighbor to p over time, where the elements of the tournament tree are edges in the Semi-Yao graph incident to the point p.

The following theorem gives the complexity of our KDS for maintenance of all the nearest neighbors in a fixed dimension.

Theorem 5.2. *Our KDS for maintenance of all the nearest neighbors of a set of n moving points in $\mathbb{R}^d$, where the coordinates of each point are polynomial functions of at most constant degree s, has the following properties.*

1. *The KDS uses $O(n \log^d n)$ space and $O(n \log^{d+1} n)$ preprocessing time.*
2. *It processes $O(n^2)$ u-swap events, each in worst-case time $O(\log^{d+1} n)$.*
3. *It processes $O(n^2)$ x-swap events, for a total cost of $O(n^2 \beta_{2s+2}(n) \log^{d+1} n)$.*
4. *The KDS processes $O(n^2 \beta^2_{2s+2}(n) \log n)$ tournament events, and processing all the events takes $O(n^2 \beta^2_{2s+2}(n) \log^2 n)$ time.*
5. *The KDS is compact, efficient, responsive in an amortized sense, and local on average, meaning that each point participates in $O(1)$ certificates on average.*

Proof. Theorem 5.1 gives the statements $1-3$. The proofs of 4 and 5 can be obtained using the same analysis as that of Theorem 3.4. □

Remark 6.2. We have provided two KDS's for maintenance of all the nearest neighbors. Our KDS in Section 5.3 generalizes our KDS in Section 3.3 that only works in $\mathbb{R}^2$. Also, for $\mathbb{R}^d$, we can handle the same number of events as our KDS in $\mathbb{R}^2$ does. In particular, both our KDS's handle $O(n^2 \beta^2_{2s+2}(n) \log n)$ events to maintain all the nearest neighbors.

5.4 Kinetic All $(1 + \epsilon)$-Nearest Neighbors

Let q be the nearest neighbor of p and let $\hat{q}$ be some point such that $|p\hat{q}| < (1+\epsilon) * |pq|$. We call $\hat{q}$ a $(1 + \epsilon)$-*nearest neighbor* of p. Here we maintain some $(1 + \epsilon)$-nearest neighbor for each point $p \in P$.

Consider a collection $\mathcal{C}$ of polyhedral cones with apex at the origin o and opening angle θ that together cover $\mathbb{R}^d$. Let x_l denote a vector inside the infinite right circular cone of the polyhedral cone $C_l \in \mathcal{C}$ with apex at o. Recall the CSPD $\Psi_l = \{(B_1, R_1), ..., (B_m, R_m)\}$ for P with respect to the cone C_l. Figure 5.7 depicts the cone C_l and a pair $(B_j, R_j) \in \Psi_l$. Let b_j (resp. r_j) be the point with the maximum (resp. minimum) x_l-coordinate among the points in B_j (resp. R_j). Let $E_l = \{(b_j, r_j) | \ j = 1, ..., m\}$. We call the graph $G(P, E_l)$ the *relative nearest neighbor graph* with respect to C_l (or RNN$_l$ graph for short), and $G(P, \cup_l E_l)$ the *RNN graph*.

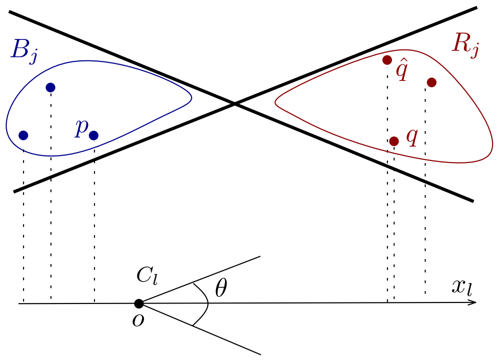


Figure 5.7: The point $\hat{q}$ (resp. p) has minimum (resp. maximum) x_l-coordinate among the points in R_i (resp. B_i).

The RNN graph has the following interesting properties:

- It can be constructed in $O(n \log^d n)$ time by using a d-dimensional RBRT of $O(n \log^d n)$ storage.
- It has $O(n \log^{d-1} n)$ edges.
- The degree of each point is $O(\log^d n)$.

Lemma 5.10 below shows another property of the RNN graph which leads us to find some $(1 + \epsilon)$-nearest neighbor each point $p \in P$; Lemma 5.9 is used in its proof.

Lemma 5.9. `(Lemma 2.1. of [2])` *Let $\mathcal{C}$ be a collection of polyhedral cones of opening angle θ, where* $\cos 2\theta - \sin 2\theta \geq 1/(1 + \epsilon)$. *Let $C_l \in \mathcal{C}$, and let q and r*

be two points in $C_l(p)$ such that the x_l-coordinate of r is less than or equal to the x_l-coordinate of q. Then $|pr| + (1+\epsilon) * |rq| \leq (1+\epsilon) * |pq|$.

Lemma 5.10. *Among all the $O(\log^d n)$ edges incident to a point p in the RNN graph, there exists an edge $(p, \hat{q})$ such that $\hat{q}$ is some $(1+\epsilon)$-nearest neighbor to p.*

Proof. Let q be the nearest neighbor to p and let $q \in C_l(p)$. By Theorem 2.6, for p and q there exists a unique pair $(B_j, R_j) \in \Psi_l$ such that $p \in B_j$ and $q \in R_j$. By Lemma 5.2, p has the maximum x_l-coordinate among the points in B_j.

Let $\hat{q}$ be the point with the minimum x_l-coordinate among the points in R_j. By Lemma 5.9, for any $\epsilon > 0$, there exist an appropriate angle θ and a vector x_l such that $|p\hat{q}| + (1+\epsilon) * |\hat{q}q| \leq (1+\epsilon) * |pq|$; this satisfies the constraint that $|p\hat{q}| \leq (1+\epsilon) * |pq|$. Thus $(p, \hat{q})$, which is an edge of the RNN graph, gives some $(1+\epsilon)$-nearest neighbor to p. □

Consider the set E_l of the edges of the RNN$_l$ graph. Let $N_l(p) = \{r_i | \ (b_i, r_i) \in E_l \ and \ b_i = p\}$. Denote by $n_l(p)$ the point in $N_l(p)$ whose x_l-coordinate is minimum. Let $L(N_l(p))$ be a list of the points in $N_l(p)$ sorted in ascending order according to their x_l-coordinates; the first point in $L(N_l(p))$ gives $n_l(p)$.

From Lemma 5.10, if the nearest neighbor of p is in some set R_j, then r_j gives some $(1+\epsilon)$-nearest neighbor to p. Note that we do not know which cone $C_l(p)$, $0 \leq l \leq c-1$, of p contains the nearest neighbor of p; but it is obvious that the nearest point to p among these c points $n_0(p), ..., n_{c-1}(p)$ gives some $(1+\epsilon)$-nearest neighbor of p. Thus for all $l = 1, ..., c$, we track the distances of all the $n_l(p)$ to p over time, so all we need to do is to maintain $n_l(p)$, $0 \leq l \leq c-1$.

Handling events. Similar to Section 5.2 we handle two types of events, *u-swap events* and *x-swap events*. Note that we do not need to define a certificate for each two consecutive points in $L(N_l(.))$. The following shows how to apply changes (*e.g.*, insertion, deletion, and exchanging the order between two consecutive points) to the sorted lists $L(N_l(.))$ when an event occurs.

Each event makes $O(\log^d n)$ updates to the edges of E_l. Consider an updated pair (b_j, r_j) such that the value of r_j (resp. b_j) changes from p to q. For this update, we must delete p (resp. r_j) from the sorted list $L(N_l(b_j))$ (resp. $L(N_l(p))$) and insert q (resp. r_j) into $L(N_l(b_j))$ (resp. $L(N_l(q))$).

Note that if the event is an x-swap event, we must find all the subscripts j where $r_j = q$ and check whether $n_l(b_j) = p$ or not; if so, p and q are in the same set $N_l(.)$ and we need to exchange their order in the corresponding sorted list $L(N_l(.))$.

Now the following theorem summarizes the complexity of our KDS for maintenance of all the $(1+\epsilon)$-nearest neighbors; as opposed to the all nearest neighbors KDS in Section 5.3, each event in this KDS can be handled in a polylogarithmic worst-case time, and the KDS is local.

Theorem 5.3. *Our KDS for maintenance of all the $(1+\epsilon)$-nearest neighbors of a set of n moving points in $\mathbb{R}^d$, where the coordinates of each one are polynomial functions of constant degree s, uses $O(n \log^d n)$ space and $O(n \log^d n)$ preprocessing time, and handles $O(n^2 \log^d n)$ events, each in worst-case time $O(\log^d n \log \log n)$. It is compact, efficient, responsive, and local.*

Proof. The proof of the preprocessing time and space (compactness) follows from the properties of an RNN graph.

A kinetic sorted list (or a tournament tree) of size c with $O(1)$ certificates is used to maintain the nearest point to p among $n_0(p), ..., n_{c-1}(p)$. Since each event makes $O(\log^d n)$ changes to the values of $n_l(.)$, and since the size of each kinetic sorted list is constant, the number of all events/changes to maintain all the $(1+\epsilon)$-nearest neighbors is $O(n^2 \log^d n)$. This implies that the KDS is efficient, and each point participates in $O(\log^d n)$ certificates (locality).

Each update to a sorted list $L(N_l(.))$ can be done in time $O(\log \log n)$ as $|L(N_l(.))| = O(\log^d n)$, so an event can be handled in $O(\log^d n \log \log n)$ time (responsiveness). □

Remark 6.3. For maintaining all the nearest neighbors, neither the KDS in Section 5.3 nor the KDS in Section 3.3 is local, and furthermore, each event in both KDS's is handled in a polylogarithmic amortized time. To deal with this problem, we have provided a KDS in Section 5.4 for maintenance of all the $(1+\epsilon)$-nearest neighbors, which satisfies all the KDS performance criteria. In particular, the KDS of Section 5.4 is local, responsive, compact, and efficient.

Chapter 6

Kinetic Reverse k-Nearest Neighbor Problem in Higher Dimensions

In this chapter, we provide the first solution to the kinetic reverse k-nearest neighbor (RkNN) problem, for any $k \geq 1$, in any fixed dimension d [1].

In Section 6.1, we show an approach for answering RkNN queries about a set P of stationary points. This section provides a simple method for reporting all the *k-nearest neighbors* in the stationary case. Then, in Section 6.2, we answer RkNN queries for moving points, where the trajectory of each point is a polynomial function of constant maximum degree. We provide the first KDS for maintenance of all the k-nearest neighbors in order to answer RkNN queries at any time.

The results of this chapter were published as a paper in the Proceedings of the 25th International Workshop on Combinatorial Algorithms (IWOCA 2014) [77].

6.1 RkNN Queries on Stationary Points

Here we use similar terminology and notation as in Chapters 3 and 4 for a set P of points in $\mathbb{R}^2$, and as in Chapter 5 for a set P of points in $\mathbb{R}^d$. We denote by W a wedge (cone) which is bounded by two half-spaces in the plane, and by C a cone in d-dimensional space which is bounded by d half-spaces.

[1]The results of this chapter are submitted to a conference.

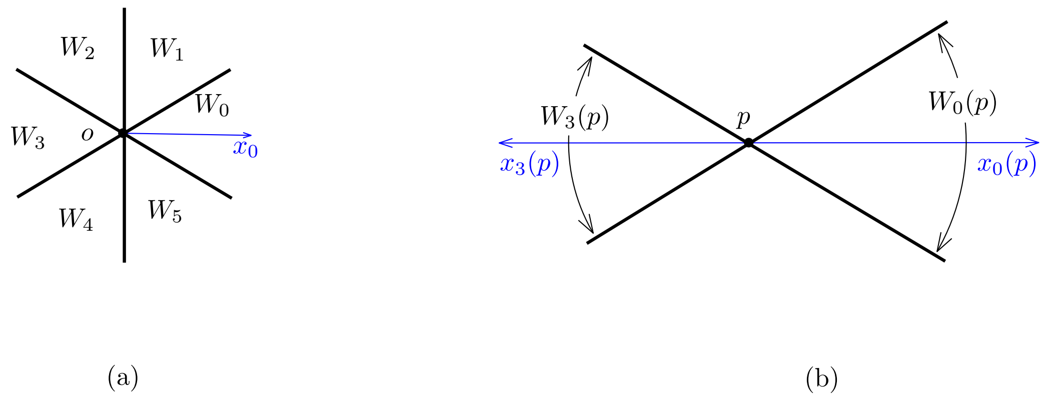


Figure 6.1: (a) A partition of the plane into six wedges with common apex at o. (b) A translation of W_0 that moves the apex to p.

6.1.1 Key Lemmas

From Section 3.1, recall the partition of the plane around the origin o into six wedges, $W_0, ..., W_5$, each of apex angle $\pi/3$ (see Figure 6.1(a)), and recall $W_l(p)$, the translation of wedge W_l, $0 \leq l \leq 5$, such that its apex moves from o to point p (see Figure 6.1(b)). Also, recall x_l (resp. $x_l(p)$), the vector along the bisector of W_l (resp. $W_l(p)$) directed outward from the apex at o (resp. p). Let $W_{l'}(p)$ be the reflection of $W_l(p)$ through p.

Consider the i^{th} nearest neighbor p_i of p. Denote by $L(P \cap W_l(p_i))$ the list of the points in $P \cap W_l(p_i)$, sorted by increasing order of their x_l-coordinates (orthogonal projections to x_l). The following lemma provides the key insight we need to obtain a simple solution for answering RkNN queries.

Lemma 6.1. *Let p_i be the i^{th} nearest neighbor of p among a set P of points in $\mathbb{R}^2$, and let $W_l(p_i)$ be the wedge of p_i that contains p. Then point p is among the first i points in $L(P \cap W_l(p_i))$.*

Proof. Let $P' = P \backslash \{p_1, ..., p_{i-1}\}$. Then the point p_i is the closest point to p among the points in P'; see Figure 6.2. By Lemma 5.2, p has the minimum x_l-coordinate among the points in $P' \cap W_l(p_i)$.

Now we add back the points $p_1, ..., p_{i-2}$, and p_{i-1} to the point set P'. Consider the worst case scenario that all these $i-1$ points insert inside the wedge $W_l(p_i)$, and that the x_l-coordinates of all these points are less than the x_l-coordinate of p. Then the point p is still among the first i points in the sorted list $L(P \cap W_l(p_i))$. $\square$

For example, consider the 3rd nearest neighbor p_3 of p in Figure 6.2. If we ignore

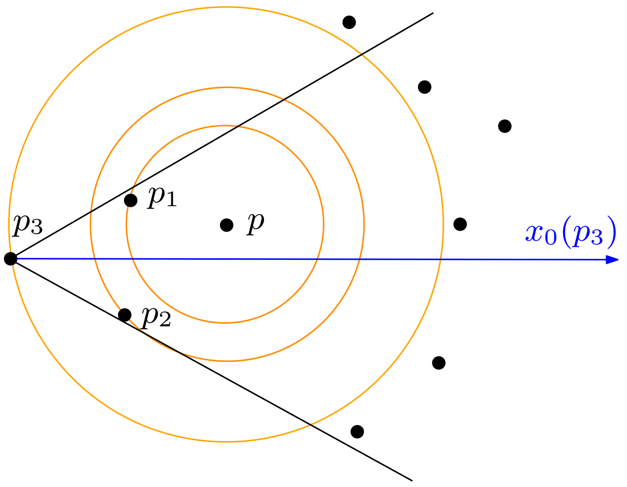


Figure 6.2: Point p_3 is the 3rd nearest neighbor of p, and p is among the first three points in $L(P \cap W_0(p_3))$.

p_1 and p_2, then p_3 is the closest point to p, and p has the minimum x_0-coordinate among the points in $(P \backslash \{p_1, p_2\}) \cap W_0(p_3)$. If we add back the two points p_1 and p_2 to $P \backslash \{p_1, p_2\}$, then p is still among the first three points in $L(P \cap W_0(p_3))$, even though p_1 and p_2 belong to $W_0(p_3)$ and have smaller x_0-coordinates than does p.

If we connect each point $p \in P$ to the first k points in the sorted list $L(P \cap W_l(p))$, for $l = 0, ..., 5$, we obtain what we call the *k-Semi-Yao graph* (k-SYG). The k-SYG has the following interesting property.

Lemma 6.2. *The k-NNG of a set P of points in $\mathbb{R}^2$ is a subgraph of the k-SYG of the set P.*

Proof. Lemma 6.1 gives a necessary condition for p_i to be the i^{th} nearest neighbor of p: The point p is among the first i points in $L(P \cap W_l(p_i))$, where l is such that $p \in W_l(p_i)$. Therefore, the edge set of the k-SYG covers the edges of the k-NNG. □

6.1.2 Computing the k-SYG and All k-Nearest Neighbors

Here, we first compute the k-SYG, and then via a construction of the k-SYG, we give a simple method for reporting all the k-nearest neighbors.

To construct the k-SYG efficiently, we need a data structure to perform the following operation efficiently: For each $p \in P$ and any of its wedges $W_l(p)$, $0 \leq l \leq 5$, find the first k points in $L(P \cap W_l(p))$. Such an operation can be performed by using

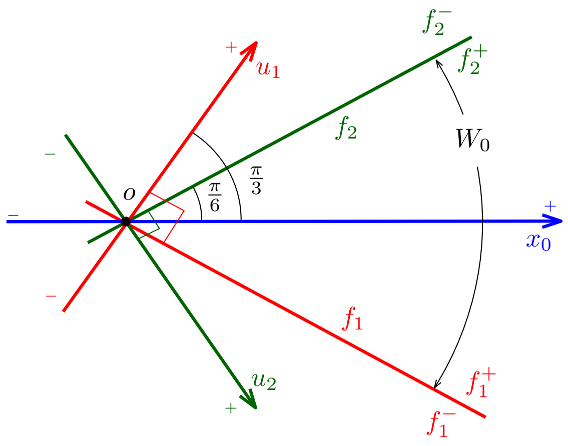


Figure 6.3: The wedge W_0 is bounded by f_1 and f_2. The coordinate axes u_1 and u_2 are orthogonal to f_1 and f_2.

range tree data structures (see Section 2.2). For each wedge W_l with apex at origin o, we construct an associated 2-dimensional range tree $\mathcal{T}_l$ as follows.

Consider a particular wedge W_l with apex at o. The wedge W_l is the intersection of two half-spaces f_1^+ and f_2^+ bounded by f_1 and f_2; see Figure 6.3. Let $\hat{u_1}$ (resp. $\hat{u_2}$) denote the normal to f_1 (resp. f_2) pointing to f_1^+ (resp. f_2^+). We define two coordinate axes u_1 and u_2 through $\hat{u_1}$ and $\hat{u_2}$, where $\hat{u_1}$ and $\hat{u_2}$ give the respective directions of increasing u_1- and u_2-coordinate values[2].

The range tree $\mathcal{T}_l$ is a regular 2-dimensional range tree based on the u_1- and u_2-coordinates. The points at level 1 (resp. level 2) are sorted at the leaves according to their u_1-coordinates (resp. u_2-coordinates) (for more details about range trees, see Section 2.2). From Theorem 2.5, any 2-dimensional range tree, *e.g.*, $\mathcal{T}_l$, uses $O(n \log n)$ space and can be constructed in time $O(n \log n)$, and for any point $r \in \mathbb{R}^2$, the points of P inside the query wedge $W_l(r)$ whose sides are parallel to f_1 and f_2 can be reported in time $O(\log n + z)$, where z is the cardinality of the set $P \cap W_l(r)$.

Now we add a new level to $\mathcal{T}_l$, based on the coordinate x_l. Let $\mathcal{C}_l(p)$ be the set of the first k points in the sorted list $L(P \cap W_l(p))$. To find $\mathcal{C}_l(p)$ in an efficient time, we use the third level of $\mathcal{T}_l$, which is constructed as follows: For each internal node v at level 2 of $\mathcal{T}_l$, we create a list $L(R(v))$ sorted by increasing order of x_l-coordinates of the points in $R(v)$. For the set P of n points in $\mathbb{R}^2$, the modified range tree $\mathcal{T}_l$, which now is a 3-dimensional range tree, uses $O(n \log^2 n)$ space and can be constructed in

[2]Since the normal vectors to the sides of the wedge W_l are parallel to the normal vectors of the wedge $W_{l'}$, where $l' = (l+3) \mod 6$, one can implement three range trees instead of six range trees; one for W_0 and W_3, one for W_1 and W_4, and one for W_2 and W_5.

time $O(n \log^2 n)$ (by Theorem 2.5).

The following establishes the processing time for obtaining a $\mathcal{C}_l(p)$.

Lemma 6.3. *Given $\mathcal{T}_l$, the set $\mathcal{C}_l(p)$ can be found in time $O(\log^2 n + k)$.*

Proof. The set $P \cap W_l(p)$ is the union of $\hat{m} = O(\log^2 n)$ sets $R(v_j)$, where v_j ranges over internal nodes at level 2 of $\mathcal{T}_l$. Consider the associated sorted lists $L(R(v_j))$. Given the $\hat{m}$ sorted lists, the kth point in $L(P \cap W_l(p))$ can be obtained in time $O(\hat{m} + k)$ [46]. By examining the points in each of the $\hat{m}$ sorted lists whose x_l-coordinates are less than or equal to the x_l-coordinate of the kth point, in time $O(k)$ we can find $\mathcal{C}_l(p)$. □

By Lemma 6.3, we can find all the $\mathcal{C}_l(p)$, for all $p \in P$. This gives the following.

Corollary 6.1. *Using a data structure of size $O(n \log^2 n)$, the edges of the k-SYG of a set of n points in $\mathbb{R}^2$ can be reported in time $O(n \log^2 n + kn)$.*

Now we state and prove the main result for reporting all the k-nearest neighbors.

Theorem 6.1. *For a set of n points in the plane, our data structure can report all the k-nearest neighbors, in order of increasing distance from each point, in time $O(n \log^2 n + kn \log n)$. The data structure uses $O(n \log^2 n + kn)$ space.*

Proof. Suppose we are given the k-SYG (from Corollary 6.1), which is a supergraph of the k-NNG (from Lemma 6.2), and we want to report all the k-nearest neighbors.

Let E_p be the set of edges incident to the point p in the k-SYG. By sorting these edges in non-decreasing order according to their Euclidean lengths, which can be done in time $O(|E_p| \log |E_p|)$, we can find the k-nearest neighbors of p ordered by increasing Euclidean distance from p.

Since the number of edges in the k-SYG is at most $6kn$ and each edge pq belongs to exactly two sets E_p and E_q, the time to find all the k-nearest neighbors, for all the points $p \in P$, is $\sum_p O(|E_p| \log |E_p|) = O(kn \log n)$. The proof obtains by combining this with the results of Corollary 6.1. □

6.1.3 RkNN Queries in $\mathbb{R}^2$

Suppose we are given a query point $r \notin P$. For $k = 1$, the number of reverse k-nearest neighbors of r is $O(1)$ [66, 34]. The following gives the upper bound for any given k.

Lemma 6.4. *The number of reverse k-nearest neighbors of a query point is $O(k)$.*

Proof. Consider $\mathcal{C}_l(r)$, the set of the first k points in $L(P \cap W_l(r))$. From Lemma 6.1, $\mathcal{C}_l(r)$ contains the candidate points for r such that r might be one of their k-nearest neighbors. Since $|\mathcal{C}_l(r)| = O(k)$, and since there is a constant number of wedges (cones) around r, the number of reverse k-nearest neighbors of r is $O(k)$. $\square$

Theorem 6.2. *Our data structure uses $O(n \log^2 n + kn)$ space and $O(n \log^2 n + kn \log n)$ preprocessing time to answer RkNN queries with query time $O(\log^2 n + k)$.*

Proof. For the query point $r \notin P$, we have $|\cup_{l=0}^{l=5} \mathcal{C}_l(r)| = O(k)$ candidate points such that r might be one of their k-nearest neighbors.

The set $P \cap W_l(r)$ of the points of P in the wedge $W_l(r)$ of r is the union of $O(\log^2 n)$ sets $R(v_j)$, where v_j ranges over the internal nodes at level 2 of the range tree $\mathcal{T}_l$. Since we have the sorted lists $L(R(v_j))$, from Lemma 6.3, the total time to find the $O(k)$ candidate points ($\mathcal{C}_l(r)$, $l = 0, ..., 5$) is $O(\log^2 n + k)$.

By Theorem 6.1, we can keep the k^{th} nearest neighbor p_k of each $p \in P$; checking a candidate point can be done in $O(1)$ time by comparing distance $|pr|$ to distance $|pp_k|$. Therefore, checking which elements of $\mathcal{C}_l(r)$, for $l = 0, ..., 5$, are reverse k-nearest neighbors of the query point r takes time $O(k)$.

It follows from Theorem 6.1 that our approach uses $O(n \log^2 n + kn)$ space and $O(n \log^2 n + kn \log n)$ preprocessing time. $\square$

6.1.4 RkNN Queries in $\mathbb{R}^d$

Recall from Section 5.1 that $C_0, ..., C_{c-1}$ are the set of polyhedral cones with opening angle θ, where $\theta \leq \pi/3$, that together cover d-dimensional space around the origin o (see Lemma 5.1), and x_l is the cone axis of C_l. Also, recall that $C_l(p)$ is the translation of C_l where o moves to p.

Using a similar proof to that of Lemma 6.1, the following lemma results.

Lemma 6.5. *Let p_i be the i^{th} nearest neighbor of p in a set P of points in $\mathbb{R}^d$, and let $C_l(p_i)$ be the cone of p_i that contains p. Then p is among the first i points in $L(P \cap C_l(p_i))$.*

Since the above lemma gives a necessary condition for p_i to be the i^{th} nearest neighbor of p, we obtain the following, which extends Lemma 6.2 to higher dimensions.

Lemma 6.6. *The k-NNG for a set P of points in $\mathbb{R}^d$ is a subgraph of the k-SYG of the set P.*

In Section 6.1.2 we used a range tree to construct the k-SYG in $\mathbb{R}^2$. Note that each cone (wedge) $W_l(p)$ of the k-SYG in $\mathbb{R}^2$ is bounded by two half-spaces, but each cone $C_l(p)$ of the k-SYG in $\mathbb{R}^d$ is bounded by d half-spaces $f_1, ..., f_d$. To obtain the set $\mathcal{C}_l(p)$ of the first k points in the sorted list $L(P \cap C_l(p))$ for any $p \in P$, we use a *d-dimensional range tree* $\mathcal{T}_l$. For each internal node v at level d of $\mathcal{T}_l$ we create a sorted list of the points in $R(v)$, the set of points at the leaves of the subtree rooted at v. In fact it creates a new level for $\mathcal{T}_l$; this modified range tree $\mathcal{T}_l$ behaves like a $(d+1)$-dimensional range tree. A similar approach and analysis as that in Section 6.1.2 gives the following theorem. This is deriving the known results in a new way [3].

Theorem 6.3. *For a set of n points in $\mathbb{R}^d$, our data structure can report all the k-nearest neighbors, ordered by distance from each point, in time $O(n \log^d n + kn \log n)$. The data structure uses $O(n \log^d n + kn)$ space.*

An analysis similar to that of Lemma 6.3 shows, for a query point r, all the $O(k)$ candidate points can be reported in time $O(\log^d n + k)$. By Theorem 6.3 we can check whether these candidate points are the reverse k-nearest neighbors of the query point r. Therefore, we obtain the followings.

Lemma 6.7. *Given a set of n points in $\mathbb{R}^d$, the number of reverse k-nearest neighbors for a query point is $O(k)$.*

Theorem 6.4. *For a set of n points in $\mathbb{R}^d$, our data structure uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time. A reverse k-nearest neighbor query can be answered in time $O(\log^d n + k)$.*

6.2 RkNN Queries on Moving Points

Suppose we are given a set P of n continuously moving points, where the coordinates of the trajectory of each point in P are given by polynomial functions of bounded degree s. To answer RkNN queries on the moving points, we must keep a valid range tree and track all the k-nearest neighbors during the motion.

In Section 5.2, we used an RBRT to provide a KDS for maintenance of the 1-SYG (a supergraph of the 1-NNG, which is formed by connecting each point $p \in P$ to the point with minimum x_l-coordinate among the points in $P \cap C_l(p)$, $0 \leq l \leq c-1$).

[3] For $k = \Omega(\log^{d-1} n)$, both our data structure and the best previous data structure [42] have the same complexity for reporting all the k-nearest neighbors.

We use a similar approach to that in Section 5.2 to give a KDS for maintenance of the k-SYG, for any $k \geq 1$. Using the kinetic k-SYG, we can easily maintain all the k-nearest neighbors over time.

6.2.1 Kinetic k-SYG

Denote by $\mathcal{T}_l$ the RBRT of the cone C_l, $0 \leq l \leq c-1$, and consider the corresponding CSPD $\psi_l = \{(B_1, R_1), ..., (B_m, R_m)\}$ (see Section 2.2). To maintain the k-SYG, we must track the set $\mathcal{C}_l(p)$ for each point $p \in P$. Therefore, for each subscript $j \in \{1, ..., m\}$, we need to maintain a sorted list $L(R_j)$ of the points in R_j in ascending order according to their x_l-coordinates over time. Note that each set R_j is some $R(v)$, the set of points at the leaves of the subtree rooted at some internal node v at level d of $\mathcal{T}_l$. To maintain these sorted lists $L(R_j)$, we add a new level to the RBRT $\mathcal{T}_l$; the points at the new level are sorted at the leaves in ascending order according to their x_l-coordinates. Therefore, in the modified RBRT $\mathcal{T}_l$, in addition to the u-swap events, we handle new events, called *x-swap events*, that occur when two points exchange their x_l-order. The modified RBRT $\mathcal{T}_l$ behaves like a $(d+1)$-dimensional RBRT. From the last property of an RBRT in Theorem 2.6, when a u-swap event or an x-swap event occurs, the RBRT $\mathcal{T}_l$ can be updated in worst-case time $O(\log^{d+1} n)$.

Denote by $\ddot{p}_{l,k}$ the k^{th} point in $L(P \cap C_l(p))$. To track the sets $\mathcal{C}_l(p)$, for all the points $p \in P$, we need to maintain the followings over time:

- A set of $d+1$ *kinetic sorted lists* $L(u_i)$, $i = 1, ..., d$, and $L(x_l)$ of the point set P. We use these kinetic sorted lists to track the order of the points in the coordinates u_i, $1 \leq i \leq d$, and x_l, respectively;

- For each B_j, a sorted list $L(B'_j)$ of the points in B'_j, where $B'_j = \{(p, \ddot{p}_{l,k}) |\ p \in B_j\}$. The order of the points in $L(B'_j)$ is according to a *label* of the second points $\ddot{p}_{l,k}$. This sorted list $L(B'_j)$ is used to answer the following query efficiently: Given a query point q and a B_j, find all $p \in B_j$ such that $\ddot{p}_{l,k} = q$; and

- The k^{th} point $r_{j,k}$ in the sorted list $L(R_j)$. We maintain the values $r_{j,k}$ in order to make necessary changes to the k-SYG when an x-swap event occurs.

Handling u-swap events. Without loss of generality, let $q \in C_l(p)$ before the u-swap event. When a u-swap event between p and q occurs, the point q moves outside the wedge $C_l(p)$; after the event, $q \notin C_l(p)$. Note that the changes that might occur in the k-SYG are the deletions and insertions of the edges incident to p inside $C_l(p)$.

Whenever two points p and q exchange their u_i-order, we perform the following updates.

`U1)` We update the kinetic sorted list $L(u_i)$.

`U2)` We update the RBRT $\mathcal{T}_l$. If a point is deleted or inserted into a B_j, we update the sorted list $L(B'_j)$.

`U3)` We update the values of $r_{j,k}$. After updating the RBRT $\mathcal{T}_l$, point q might be inserted or deleted from some R_j and change the values of $r_{i,k}$. Thus for all R_j where $q \in R_j$, before and after the event, we perform the following. We check whether the x_l-coordinate of q is less than or equal to the x_l-coordinate of $r_{j,k}$; if so, we take the successor or predecessor point of $r_{j,k}$ in $L(R_j)$ as the new value for $r_{j,k}$.

`U4)` We query to find $\mathcal{C}_l(p)$.

`U5)` If we obtain a new value for $\ddot{p}_{l,k}$, we update all the sorted lists $L(B'_j)$ such that $p \in B_j$.

Now the following gives the complexity of handling u-swap events.

Lemma 6.8. *Our KDS for maintenance of the k-SYG handles $O(n^2)$ u-swap events, each in worst-case time $O(\log^{d+1} n + k \log\log n)$.*

Proof. Each swap event in a kinetic sorted list can be handled in time $O(\log n)$ (Step `U1`). Since each update (insertion/deletion) to $L(B'_j)$ takes $O(\log n)$ time, and since each point is in $O(\log^d n)$ sets B_j, Step `U2` takes $O(\log^{d+1} n)$ time. It is obvious that processing time of Steps `U3` and `U5` is $O(\log^{d+1} n)$. From Lemma 6.3, Step `U4` takes $O(\log^d n + k \log\log n)$ time.

Considering the complexity of each step above, and assuming the coordinates of the trajectory of each point are given by bounded degree polynomials, the proof obtains. □

Handling x-swap events. When an x-swap event between two consecutive points p and q with p preceding q occurs, it does not change the elements of the pairs (B_j, R_j) of the CSPD Ψ_l. Such an event can only change the k-SYG if both p and q are in the same $C_l(w)$, for some $w \in P$, and such that $\ddot{w}_{l,k} = p$.

We perform the following updates to our KDS when two points p and q exchange their x_l-order.

`X1)` We update the kinetic sorted list $L(x_l)$; this takes $O(\log n)$ time.

`X2)` We update the RBRT $\mathcal{T}_l$, which takes $O(\log^{d+1} n)$ time.

`X3)` We find all the sets R_j where both p and q belong to R_j and such that $r_{j,k} = p$. Also, we find all the sets R_j where $r_{j,k} = q$. This takes $O(\log^d n)$ time.

`X4)` For each R_j (from Step `X3`), we extract all the pairs $(w, \ddot{w}_{l,k})$ from the sorted lists $L(B'_j)$ such that $\ddot{w}_{l,k} = p$. Note that each change to the pair $(w, \ddot{w}_{l,k})$ is a change to the k-SYG.

`X5)` For each w (from Step `X4`), we update all the sorted lists $L(B'_j)$ where $(w, \ddot{w}_{l,k}) \in B'_j$: we replace the previous value of $\ddot{w}_{l,k}$, which is p, by the new value q.

Now we obtain Lemma 6.10 below, which summarizes the complexity of handling x-swap events; we use Lemma 6.9 in its proof to obtain an upper bound for χ_k, the number of exact changes to the k-SYG of a set of moving points, where the trajectories are given by bounded degree polynomials.

Lemma 6.9. *The number of changes to the k-SYG of a set of n moving points, where the coordinates of each point are given by polynomial functions of at most constant degree s, is $\chi_k = O(\phi(s, n) * n)$.*

Proof. Fix a point $p \in P$ and one of its cone $C_l(p)$. There are $O(n)$ insertions/deletions into the cone $C_l(p)$ over time. The x_l-coordinates of these points create $O(n)$ partial functions. The k-SYG changes if a change to $\ddot{p}_{l,k}$ occurs. The number of all changes to $\ddot{p}_{l,k}$ is equal to $\phi(s, n)$, the complexity of the k-level of partially-defined polynomial functions of bounded degree s.

Therefore, summing over all the $n = |P|$ points, the number of changes to the k-SYG is within a linear factor of $\phi(s, n)$: $\chi_k = O(\phi(s, n) * n)$. □

Lemma 6.10. *Our KDS for maintenance of the k-SYG handles $O(n^2)$ x-swap events with a total cost of $O(\phi(s, n) * n \log^{d+1} n)$.*

Proof. The complexities of the first three steps are clear. For each found R_j from Step `X3`, Step `X4` takes $O(\log n + \xi_j)$ time, where ξ_j is the number of pairs $(w, \ddot{w}_{l,k}) \in B'_j$ such that $\ddot{w}_{l,k} = p$. Thus, for all the $O(\log^d n)$ sets R_j of Step `X3`, Step `X4` takes $O(\log^{d+1} n + \sum_j \xi_j)$ time, where $\sum_j \xi_j$ is the number of exact changes to the k-SYG when an x-swap event occurs. Therefore, for all the $O(n^2)$ x-swap events, the total processing time for this step is $O(n^2 \log^{d+1} n + \chi_k) = O(\chi_k)$.

The processing time for Step `X5` is a function of χ_k. For each change to the k-SYG, this step spends $O(\log^{d+1} n)$ time to update the sorted lists $L(B'_j)$. Therefore, the total processing time for all the x-swap events in this step is $O(\chi_k * \log^{d+1} n)$. □

Now we can obtain the following.

Theorem 6.5. *For a set of n moving points in $\mathbb{R}^d$, where the coordinates of each point are polynomial functions of at most constant degree s, our k-SYG KDS uses $O(n \log^{d+1} n)$ space and $O(n \log^{d+1} n)$ preprocessing time, and handles $O(n^2)$ events with a total cost of $O(kn^2 \log\log n + \phi(s,n) * n \log^{d+1} n)$.*

Proof. The proof obtains by combining the results of Theorem 2.6 and Lemmas 6.8 and 6.10. □

6.2.2 Kinetic All k-Nearest Neighbors

Given a KDS for maintenance of the k-SYG, a supergraph of the k-NNG (see Theorem 6.5), this section shows how to maintain all the k-nearest neighbors over time. For maintenance of the k-nearest neighbors of each point $p \in P$, we only need to track the order of the edges incident to p in the k-SYG according to their Euclidean lengths. This can easily be done by using a kinetic sorted list. The following gives the complexity of our kinetic approach.

Theorem 6.6. *For a set of n moving points in $\mathbb{R}^d$, where the coordinates of each point are given by polynomials of at most constant degree s, our KDS for maintenance of all the k-nearest neighbors, ordered by distance from each point, uses $O(n \log^{d+1} n + kn)$ space and $O(n \log^{d+1} n + kn \log n)$ preprocessing time. Our KDS handles $O(\phi(s,n) * n^2)$ events, each in $O(\log n)$ amortized time.*

Proof. Let $E_p(t)$ be the set of edges incident to point $p \in P$ in the k-SYG at time t. Let $L(E_p(t))$ denote a kinetic sorted list that maintains the edges in $E_p(t)$ sorted by their Euclidean lengths.

Let m_p be the number of insertions/deletions to the set $E_p(t)$ over time. Since the cardinality of $E_p(t)$ is $O(n)$, each insertion into a kinetic sorted list $L(E_p(t))$ can cause $O(n)$ swaps. Each change (*e.g.*, inserting/deleting an edge pq) to the k-SYG, creates two insertions/deletions in the kinetic sorted lists $L(E_p(t))$ and $L(E_q(t))$; this implies that $\sum_p m_p = O(\chi_k)$. Therefore, from Lemma 6.9, all the kinetic sorted lists handle a total of $O(n \sum_p m_p) = O(\phi(s,n) * n^2)$ events. Each event in a kinetic sorted list is handled in time $O(\log n)$. Combining this and Theorem 6.5 gives the total processing time $O(kn^2 \log\log n + \phi(s,n) * n \log^{d+1} n + \phi(s,n) * n^2 \log n) = O(\phi(s,n) * n^2 \log n)$ for all the events. □

Now we measure the performance of our KDS for maintenance of all the k-nearest neighbors in $\mathbb{R}^d$ by the four standard criteria in the KDS framework.

Lemma 6.11. *The efficiency, responsiveness, locality, and compactness of our KDS are $O(\frac{\phi(s,n)}{k\beta_{2s}(n/k)})$, $O(\log n)$ on average, $O(k)$ on average, and $O(kn)$, respectively.*

Proof. Fix a point $p \in P$. The distances of the $n-1$ points of $P \backslash \{p\}$ to p as functions of time create $2s$-intersecting curves, meaning that each pair intersects at most $2s$ times. The number of changes to the (ordered) k-nearest neighbors $p_1, ..., p_k$ of p is equal to the complexity of the $(\leq k)$-level, which is $O(kn\beta_{2s}(n/k))$. Thus the total for all points $p \in P$ is $O(kn^2\beta_{2s}(n/k))$. Since the number of events in our KDS is $O(\phi(s,n) * n^2)$, the efficiency of our KDS is $O(\frac{\phi(s,n)}{k\beta_{2s}(n/k)})$.

Each event in our KDS can be handled in amortized time $O(\log n)$. Thus the responsiveness of our KDS is $O(\log n)$ on average.

In our KDS, for each two consecutive elements in each of the kinetic sorted lists $L(u_i)$, $L(x_l)$, and $L(E_p(t))$, we have a certificate. This implies that a point participates in a constant number of these certificates in the kinetic sorted lists $L(u_i)$ and $L(x_l)$. Since the number of edges in the k-SYG is $O(kn)$, the number of certificates corresponding to each point, in the kinetic sorted lists $L(E_p(t))$, is $O(k)$ on average. Therefore, the locality of our KDS is $O(k)$ on average.

The number of certificates of the kinetic sorted lists $L(u_i)$ and $L(x_l)$ is $O(n)$, and the number of certificates of the kinetic sorted lists $L(E_p(t))$ is $O(kn)$, so the compactness of our KDS is $O(kn)$. □

6.2.3 RkNN Queries

Consider a query point $r \notin P$ at some time t. A similar approach to that in Section 6.1.3 can report the reverse k-nearest neighbors for r. Note that if one asks the query at time t, which is coincident with the time when an event occurs in the KDS for maintenance of all the k-nearest neighbors, we first handle the event and then answer the query.

The following theorem gives the main results of this section.

Theorem 6.7. *Consider a set P of n moving points in $\mathbb{R}^d$, where the coordinates of each one are given by bounded-degree polynomials. The number of reverse k-nearest neighbors for a query point $q \notin P$ is $O(k)$. Our KDS uses $O(n\log^{d+1} n + kn)$ space, $O(n\log^{d+1} n + kn\log n)$ preprocessing time, and handles $O(\phi(s,n) * n^2)$ events. At any time t, an RkNN query can be answered in time $O(\log^d n + k)$. If an event occurs at time t, the KDS spends amortized time $O(\log n)$ on updating itself.*

Proof. By Theorem 2.6, in time $O(\log^d n)$ we can find a set of R_j where $P \cap C_l(r) = \sum_j R_j$. From Lemma 6.3, and since we have sorted lists $L(R_j)$ at level $d+1$ of $\mathcal{T}_l$, the $O(k)$ candidate points for the query point r can be found in worst-case time $O(\log^d n + k)$. By handling $O(\phi(s,n) * n^2)$ events, we can maintain all the k-nearest neighbors (from Theorem 6.6). Thus we can easily check whether or not these candidate points are the reverse k-nearest neighbors of the query point r at time t.

If one asks a query at time t, which coincides with the time when one of the $O(\phi(s,n) * n^2)$ events occurs, our KDS first spends amortized time $O(\log n)$ to handle the event, and then spends time $O(\log^d n + k\log\log n)$ to answer the query. $\square$

Chapter 7

Kinetic Point Set Embedding for Plane Graphs

In this chapter, we investigate a kinetic version of a point set embedding problem. Given a plane graph $G(V, E)$ where $|V| = n$, and a set P of n moving points, we provide the first kinetic data structure to maintain a point set embedding of G on P with few bends per edge over time. This requires reassigning the mapping of vertices to points from time to time.

As a preliminary step, in Section 7.1, we give a new algorithm to obtain a 2-bend drawing of a given plane graph G on a stationary point set P. This algorithm first draws a 3-bend drawing, and then transforms the 3-bend drawing to a 2-bend drawing. Section 7.2 gives a kinetic data structure to maintain such drawings.

The results of this chapter were published as a paper in the Proceedings of the 20th International Symposium on Graph Drawing (GD 2012) [78].

7.1 Drawing with k bends

In this section we first provide an $O(n \log n)$ algorithm for point set embedding of a given plane graph $G(V, E)$ on P with at most three bends per edge. Then, given this 3-bend drawing, we provide a 2-bend drawing of G on P in linear time, although this is not used in the later sections.

Given a plane graph $G(V, E)$ on $n = |V|$ vertices and a set P of $n = |P|$ points, the point set embedding of G on P with at most k bends is to draw G on P such that each vertex $v \in P$ is mapped to a point $p \in P$, and such that each edge (curve) of G is drawn by a chain of $k + 1$ polyline; the drawn chains representing the edges of G intersect only at common points representing the vertices of G.

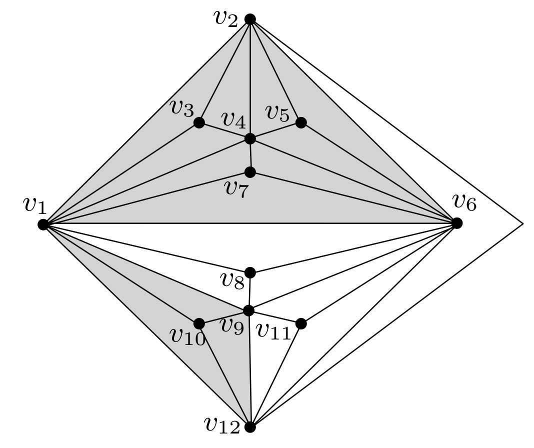
For any given plane graph $G(V, E)$, we add a set of edges E' to the graph G to make it maximally planar [1], and then we embed the graph $G(V, E \cup E')$ on the set of given points P, and finally we remove the extra edges mapped from E'. The remaining drawing is the point set embedding of the graph $G(V, E)$ on the set of points P. From now on, we assume the given plane graph G is a triangulation; Figure 7.1(a) depicts a maximal plane graph (a triangulation).

7.1.1 Drawing with 3 bends

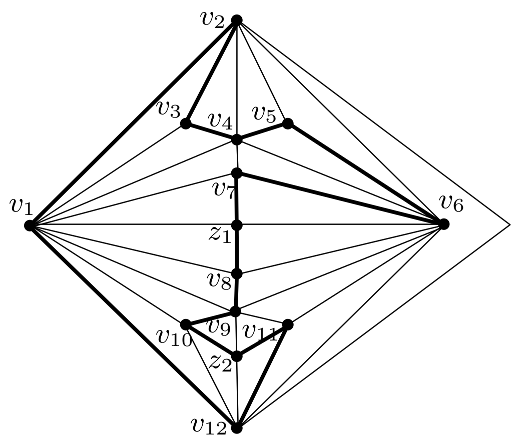
We use a similar approach to that of Kaufmann and Wiese [59] to construct an initial polyline drawing of a plane graph G on a set of stationary points P. Our algorithm draws a point set embedding with at most three bends per edge that we later extend to the kinetic setting. The key insight from [59] is the idea of creating a *Hamiltonian cycle* that has at least one edge, called an *external edge*, on the outer face of G. Such Hamiltonian cycle can be created in linear time [50, 59] by adding *dummy vertices*; we explain the approach of [59] as follows.

Each 4-*connected* planar graph is Hamiltonian [90]. In fact, any maximal plane graph with at most two *separating triangles* is Hamiltonian [33, 54, 95]; a separating triangle is a triangle whose removal separates the graph into more components (Figure 7.1(a), see triangles $v_1v_2v_6$ and $v_1v_9v_{12}$). We now review how to construct a 4-connected graph from any plane graph. Using the algorithm by Chiba and Nishizeki [37], the separating triangles of a maximal plane graph can be found efficiently. Kaufmann and Wiese [59] destroy the separating triangles by adding dummy vertices and edges to create a 4-connected graph. Then, using the algorithm of Chiba and Nishizeki [36], a Hamiltonian cycle of the 4-connected graph with external edge can be found in linear time. To illustrate the main idea, on an example, consider Figure 7.1(a) which depicts a plane graph with separating triangles; $v_1v_2v_6$ and $v_1v_9v_{12}$ are two of the separating triangles. Create a new graph as follows; see Figure 7.1(b). Add two new vertices z_1 and z_2 to destroy the two separating triangles $v_1v_2v_6$ and $v_1v_9v_{12}$. Place z_1 on the edge v_1v_6, which partitions it into two edges v_1z_1 and z_1v_6 that replace v_1v_6. Then create two new edges v_7z_1 and z_1v_8. Similarly, place z_2 on

[1]A maximal plane graph is a triangulation, *i.e.*, a plane graph such that no edge can be added without losing planarity.



(a)



(b)

Figure 7.1: (a) Two separating triangles $v_1v_2v_6$ and $v_1v_9v_{12}$ in a plane graph. (b) Adding two dummy vertices z_1 and z_2 and new edges creates a plane graph with a Hamiltonian cycle (bold edges).

v_9v_{12} and add edges $v_{10}z_2$ and z_2v_{11}. The new graph still has separating triangles (*e.g.*, $v_1v_2v_4$ and $v_2v_4v_6$), but it has a Hamiltonian cycle (bold edges) with external edge v_1v_{12}.

Now we set up some notation and terminology. Let $p_1, p_2, ..., p_n$ be the points of P, ordered by increasing x-coordinate. We assume that no two points have the same x-coordinate. For each dummy vertex z_k we add a dummy point to the given set of points P. As described above a dummy vertex may have been placed on an edge v_iv_j of the given graph G, which partitioned v_iv_j into two edges v_iz_k and z_kv_j. The corresponding dummy point is inserted in the middle of the segment p_ip_j. Let m be the number of dummy vertices, let $C = (u_1, u_2, ..., u_{n+m})$ be the circular sequence of vertices and dummy vertices on the Hamiltonian cycle with external edge u_1u_{n+m}; two dummy vertices z_1 and z_2 of Figure 7.1(b) are called by u_8 and u_{12}, respectively, in Figure 7.2(a). Let chain $Q = \{q_1, q_2, ..., q_{n+m}\}$ be the list of P plus the dummy points sorted in increasing order by their x-coordinates; in particular, for two points $q_i = (x_i, y_i)$ and $q_j = (x_j, y_j)$, if $i < j$ then $x_i < x_j$. In Figure 7.2(b), the dummy point q_8 (resp. q_{12}) is inserted in the middle of q_7q_9 (resp. $q_{11}q_{13}$).

Call the edges on the Hamiltonian cycle of the plane graph G *hull edges*, the edges inside the Hamiltonian cycle *interior edges*, and the edges outside of the Hamilto-

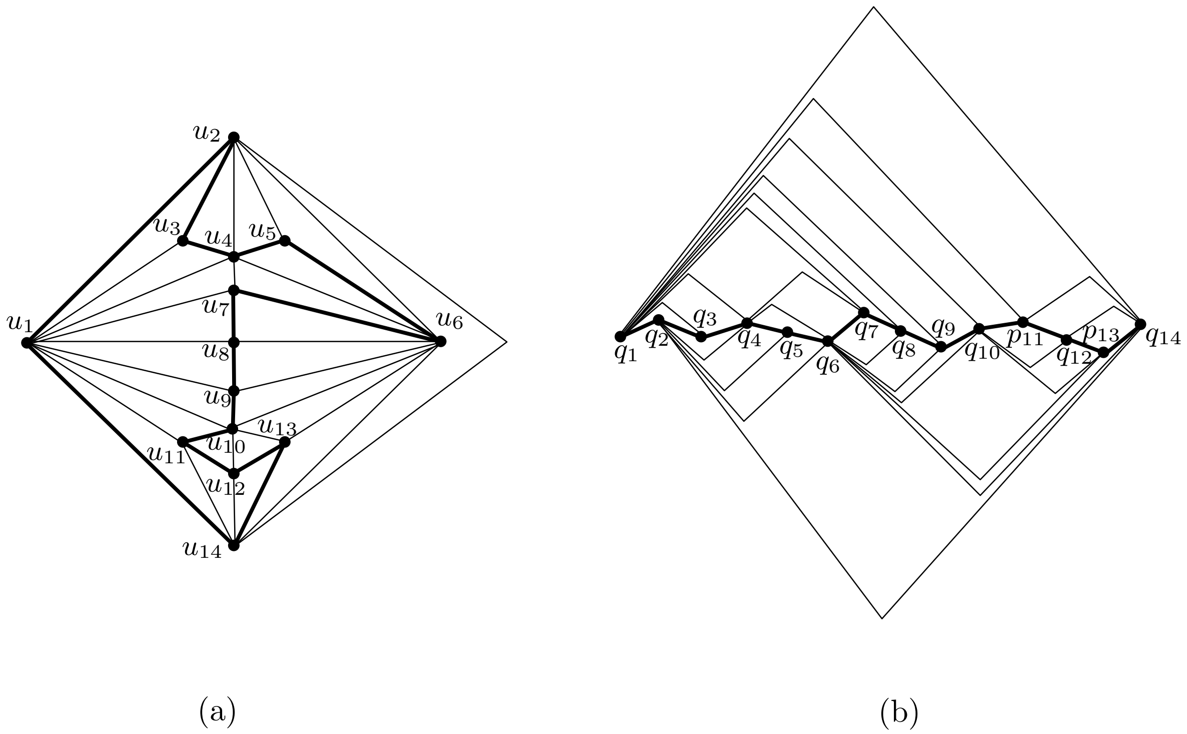


Figure 7.2: Drawing the hull, interior, and exterior edges.

nian cycle *exterior edges*. In order to support kinetic drawing, we assign different slopes to the edges of the polyline drawing than does the algorithm of Kaufmann and Wiese [59]; our slopes prevent intersections between interior edges during the motion of the points; see Section 7.2. Let δ be the maximum absolute slope of the edges of the chain Q. In particular, $\delta = \max_i |\frac{y_{i+1}-y_i}{x_{i+1}-x_i}|$ where $q_i = (x_i, y_i)$ and $q_{i+1} = (x_{i+1}, y_{i+1})$ are consecutive edges of the chain Q. To draw the point set embedding we map the hull edge $u_i u_{i+1}$ to the edge $q_i q_{i+1}$, for $i = 1, ..., n+m-1$. For the external Hamiltonian edge $u_1 u_{n+m}$ and each interior edge $u_i u_j$, where $i < j$, we also draw an edge with one bend b_{ij} at the intersection of two lines, one through u_i with slope $(1 + \frac{j-i}{n+m})\delta$ and the other one through u_j with slope $-(1 + \frac{j-i}{n+m})\delta$; the mapping of the edge $u_i u_j$ is $q_i b_{ij} q_j$ which has one bend at b_{ij}. The interior edges are drawn above the chain $q_1, q_2, ..., q_{n+m}$ and the exterior edges are drawn in a similar way below the chain.

To obtain a point set embedding of the original plane graph G on the original point set P, we regard dummy vertices as bends in the original edges they subdivided, and we remove the newly added edges. For example, if we remove the dummy points q_8 and q_{12} and the corresponding edges q_7q_8, q_8q_9, $q_{11}q_{12}$, and $q_{12}q_{13}$ from the embedding of Figure 7.2(b), we obtain the embedding in Figure 7.3(b), which has at most three

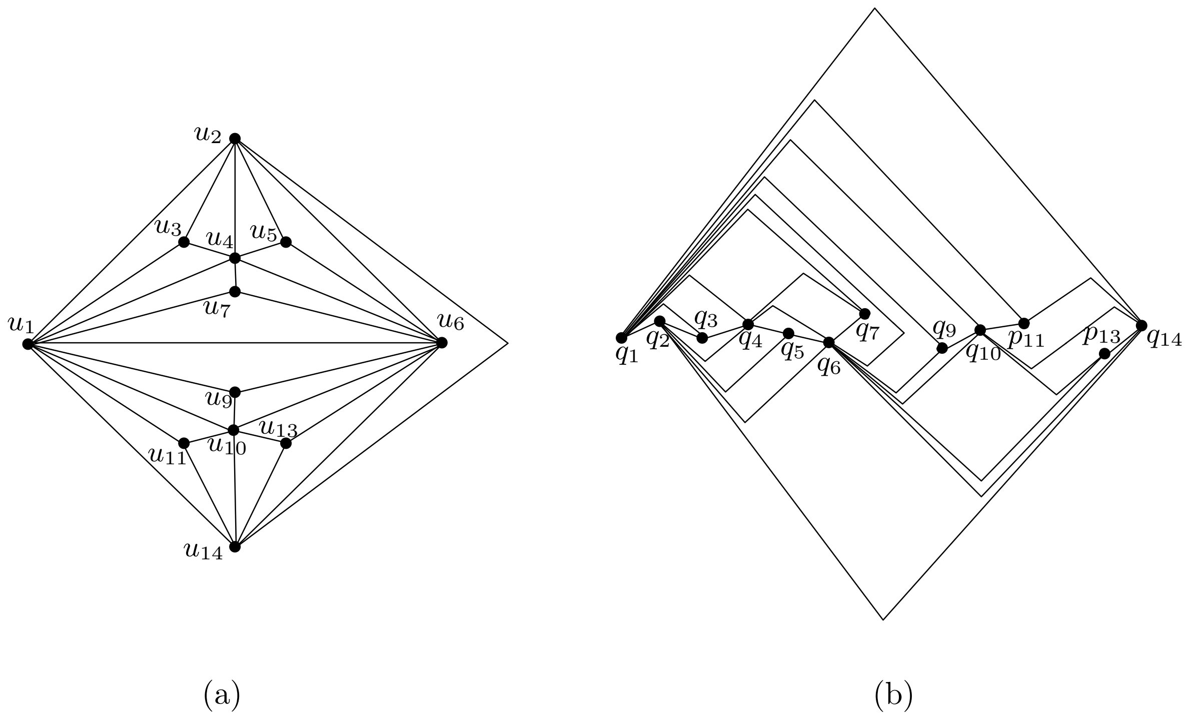


Figure 7.3: Removing the dummy points q_8 and q_{12} and corresponding edges from Figure 7.2.

bends per edge.

Theorem 7.1. *The above point set embedding of plane graph $G(V, E)$ onto the set of $n = |V|$ points P is crossing-free, has at most three bends per edge, and is constructed in $O(n \log n)$ time.*

Proof. After sorting the set of points P by x-coordinate in $O(n \log n)$ time, we map the hull edges to edges of the chain Q plus $q_1 q_{n+m}$. The chain Q separates the interior edges and the exterior edges, and it prevents intersections between these two types. In the following, we consider whether there are intersections among the interior edges; the proof that there is no intersection among the exterior edges is analogous.

Let $q_i q_j$ and $q_k q_l$ be two interior edges; without loss of generality, assume $i \leq k$. There are two possible situations: either $j \leq k$, in which case it is obvious that edge $q_i b_{ij} q_j$ does not cross edge $q_k b_{kl} q_l$ (Figure 7.2(b), see $q_1 q_4$ and $q_4 q_7$), or $j > k$, which implies that $j \geq l$ because the embedded plane graph G has no edge crossing (edge $u_i u_j$ does not cross edge $u_k u_l$). In the second case ($i \leq k < l \leq j$), the slope of $q_i b_{ij}$ (resp. $b_{ij} q_j$) is $(1 + \frac{j-i}{n+m})\delta$ (resp. $-(1 + \frac{j-i}{n+m})\delta$) which is steeper than the slope of $q_k b_{kl}$ (resp. $b_{kl} q_l$) because $j - i > l - k$. Therefore, b_{ij} is above b_{kl} which implies that edge $q_ib_{ij}q_j$ does not cross edge $q_kb_{kl}q_l$ (Figure 7.2(b), see q_1q_7 and q_4q_7).

As we removed the dummy points and the new edges, each edge of the original graph G is mapped to a chain of at most four line segments. If an edge u_iu_j of the original graph is partitioned into two edges u_iu_k and u_ku_j by a dummy vertex u_k, the mapping of u_iu_j has at most three bends (see the mapping of u_1u_6 to q_1q_6 in Figure 7.3); otherwise, the mapping has at most one bend (see the mapping of u_1u_4 to q_1q_4 in Figure 7.3). $\square$

7.1.2 Drawing with 2 bends

Given a 3-bend drawing of a plane graph G on a set P of points (from Section 7.1.1), there exists a way in [59] to obtain a 2-bend drawing of G on P. The method in [59], which we explain in the next paragraph, saves bends at dummy points.

Consider an edge u_iu_j of G with a dummy vertex u_k in the middle of u_iu_j, and consider the drawing q_iq_j of u_iu_j, where $k > \max(i,j)$, with a dummy point q_k in the middle of q_iq_j; see Figure 7.4. The drawing q_iq_j has three bends, one at b_{ik} (the intersection of two lines, one through q_i with slope $(1 + \frac{k-i}{n+m})\delta$ and the other one through q_k with slope $-(1+\frac{k-i}{n+m})\delta)$ which is above the chain Q, one at dummy point q_k, and one at b_{kj} (the intersection of two lines, one through q_k with slope $(1+\frac{k-j}{n+m})\delta$ and the other one through q_j with slope $-(1+\frac{k-j}{n+m})\delta)$ which is below the chain Q. To reduce the number of bends to at most two bends, we replace the chain $b_{ik}q_kb_{kj}$ with a vertical line segment through q_k saving a bend at q_k (see Figure 7.4(b)). The new drawing q_iq_j of the edge u_iu_j may cross other edges and destroy the planarity. Thus we rotate the edges in the point set embedding to avoid crossings (see Figure 7.4(c). Given a 3-bend drawing, the method of [59] takes $O(n^2)$ time.

Next we describe our approach, a linear time drawing algorithm with at most two bends per edge without rotating the edges. We provide our method for the interior edges, which are drawn above the chain $q_1, q_2, ..., q_{n+m}$; the exterior edges are drawn in a similar way below the chain.

Let $r_{ij} = [i, j]$ denote the *range* of subscripts for the edge q_iq_j. The idea behind our 2-bend drawing algorithm is to find the edges whose ranges contain the range r_{ij}, for all edges q_iq_j. If r_{kl} covers r_{ij} we assign slopes to segments of $q_kb_{kl}q_l$ so that they do not intersect the segments of $q_ib_{ij}q_j$. To store these nested layers of ranges we construct a *nested tree* $\mathcal{T}$ data structure. Each node n_{ij} of $\mathcal{T}$ corresponds to an edge q_iq_j, and the subtree rooted at n_{ij} stores all ranges covered by r_{ij} at its nodes.

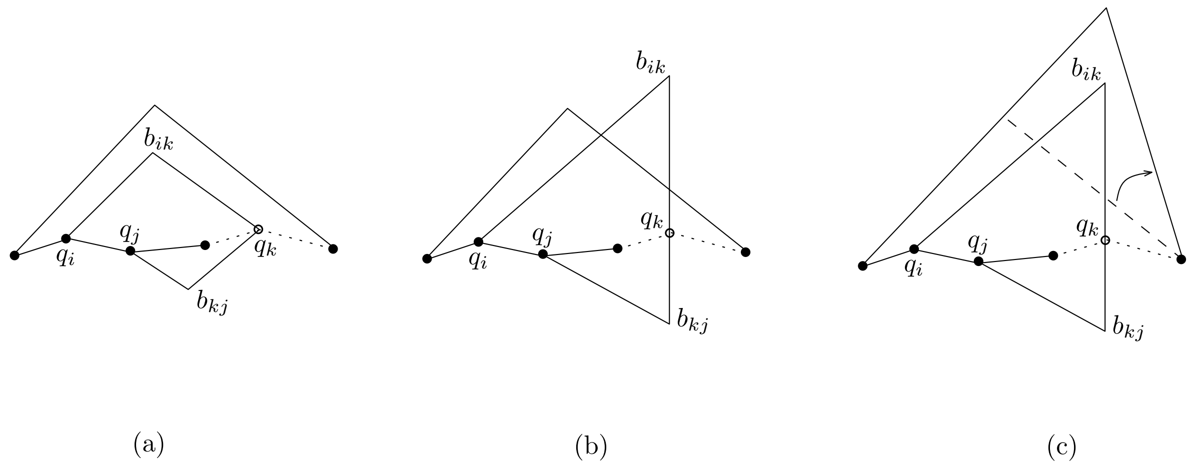


Figure 7.4: Saving bends at dummy points of a 3-bend drawing to obtain a 2-bend drawing.

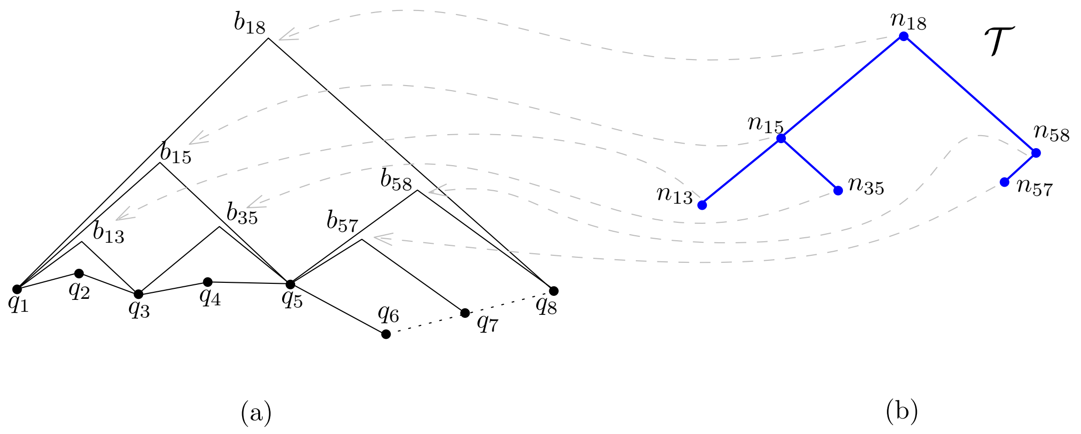


Figure 7.5: (a) The edges above the chain $q_1, q_2, ..., q_8$ of a 3-bend drawing; q_7 is a dummy point. (b) The nested tree $\mathcal{T}$ for the drawing.

Figure 7.5(b) shows the nested tree $\mathcal{T}$ for the graph in Figure 7.5(a).

Lemma 7.1. *Given a 3-bend drawing of a plane graph G on P with at most three bends per edge* (from Theorem 7.1), *the nested tree $\mathcal{T}$ can be built in $O(n)$ time.*

Proof. Using a *stack* we can easily construct the nested tree $\mathcal{T}$ as follows. For the external Hamiltonian edge q_1q_{n+m}, we push q_1 onto the stack, and create a node n_{1n+m} as the root of $\mathcal{T}$ and a pointer pointing to the node n_{1n+m}.

We process the endpoints of edges in order of increasing x-coordinate; if there are two or more edges incident to the same point, then we process these edges by decreasing order of their corresponding ranges. If we encounter the first endpoint of an edge q_iq_j, where $i < j$, we push the point q_i and insert into $\mathcal{T}$ a new rightmost child of the node to which the pointer points; after this the pointer must point to the newly created node. If we encounter the second endpoint q_j of q_iq_j, clearly, the top of the stack is the first endpoint q_i and we pop the point q_i and make the pointer point to the parent of the node n_{ij}. For example, in Figure 7.5(a), after creating the root n_{18}, first we see the point q_1 of the edge q_1q_5, whose range r_{15} includes the range r_{13}, and so we create the node n_{15} as the rightmost child of n_{18}; the pointer then points to n_{15}. After encountering the point q_1 of the edge q_1q_3 and creating the node n_{13}, the pointer points to the node n_{13}. Here, there are three q_1's in the stack corresponding to the three edges q_1q_8, q_1q_5, and q_1q_3. When we encounter q_3 we pop its corresponding q_1 and make the pointer point to n_{15}; continuing this process gives the nested tree $\mathcal{T}$ in Figure 7.5(b).

The running time is clear. □

Next we show how to draw a 2-bend drawing by traversing the nested tree $\mathcal{T}$ from the leaves to the root. For each node n_{ij} of $\mathcal{T}$, where n_{ij} corresponds to the edge q_iq_j, we store two values δ_l and δ_r; the slopes of q_ib_{ij} and $b_{ij}q_j$ will be generated from δ_l and δ_r. If n_{ij} is a leaf and neither endpoint of q_iq_j is a dummy point we set $\delta_l = \delta_r = \delta$, where δ is the maximum slope of the edges of the chain Q. Then we set the slopes of q_ib_{ij} and $b_{ij}q_j$ equal to $(1 + \frac{j-i}{n+m})\delta_l$ and $-(1 + \frac{j-i}{n+m})\delta_r$, respectively. If q_j (resp. q_i) is a dummy point we set $\delta_l = \delta$ (resp. $\delta_r = \delta$), and then b_{ij} is the intersection of the vertical line through q_j (resp. q_i) and the line through q_i with slope $(1+\frac{j-i}{n+m})\delta_l$ (resp. $-(1 + \frac{j-i}{n+m})\delta_r$). Let n_{kl} be the parent of the node n_{ij} corresponding to the edge q_kq_l. In order to assign slopes to the edge q_kq_l whose range r_{kl} covers the range r_{ij}, we find the slope α of the line through q_l (resp. q_k) and b_{ij} (see Figure 7.6) and set $\delta_r = |\alpha|$ (resp. $\delta_l = \alpha$).

After assigning the slopes δ_l and δ_r to all leaves we can find the slopes of the edges corresponding to the internal nodes of $\mathcal{T}$ as follows. For each internal node n_{ij} we set δ_l (resp. δ_r) to be the maximum of the δ_l's (resp. δ_r's) of the children of the node n_{ij}; the slope of q_ib_{ij} (resp. $b_{ij}q_j$) is $(1 + \frac{j-i}{n+m})\delta_l$ (resp. $-(1 + \frac{j-i}{n+m})\delta_r$). If one of the endpoints of the edge q_iq_j corresponding to the internal node n_{ij} is a dummy point we handle n_{ij} as we did for a leaf.

Now we can obtain the following.

Theorem 7.2. *Given a 3-bend drawing of the plane graph G on P, which can be constructed in $O(n \log n)$ time* (see Theorem 7.1)*, a crossing-free 2-bend drawing of*

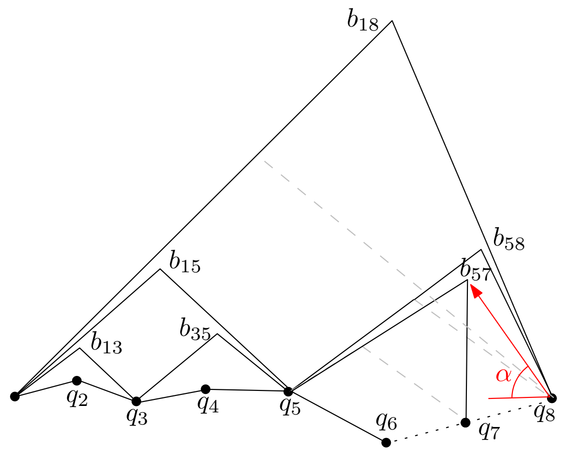


Figure 7.6: The 2-bend drawing of Figure 7.5(a).

G on P can be constructed in linear time.

Proof. We assign slopes to the edges such that if the range r_{kl} covers the range r_{ij} then the slope of $q_k b_{kl}$ (resp. $b_{kl} q_l$) is steeper than the slope of $q_i b_{ij}$ (resp. $b_{ij} q_j$), except when one of the endpoints of $q_i q_j$ is a dummy point. Without loss of generality, assume p_j is a dummy point. In this case, we compute a slope α, and use it to assign a valid slope to $b_{kl} q_l$ so that it does not intersect $b_{ij} q_j$. Thus $q_i q_j$ does not cross $q_k q_l$.

By Lemma 7.1 and the fact that traversing the nodes of $\mathcal{T}$ takes linear time, the construction time of obtaining a 2-bend drawing from a 3-bend drawing is $O(n)$. $\square$

7.2 The Kinetic Drawing

Next we kinetically maintain the drawing we provided in Section 7.1.1. We give a KDS for maintaining the edges above the chain $Q = \{q_1, q_2, ..., q_{n+m}\}$; the edges in the lower part can be maintained analogously.

Each edge $q_i q_j$ above the chain of the 3-bend drawing, in Section 7.1.1, is defined by two line segments $q_i b_{ij}$ and $b_{ij} q_j$ with positive slope $(1 + \frac{j-i}{m+n})\delta$ and negative slope $-(1 + \frac{j-i}{m+n})\delta$, respectively, where δ is the maximum slope of the edges of the chain Q. Since the slopes of each edge are generated from δ, we only need to maintain the maximum slope δ in order to maintain the point set embedding over time. We create a dynamic and kinetic tournament tree $\mathcal{TT}$ to maintain δ, whose elements (leaves) are the edges of the chain Q; the root of $\mathcal{TT}$ always has the steepest slope among all edges of the chain Q.

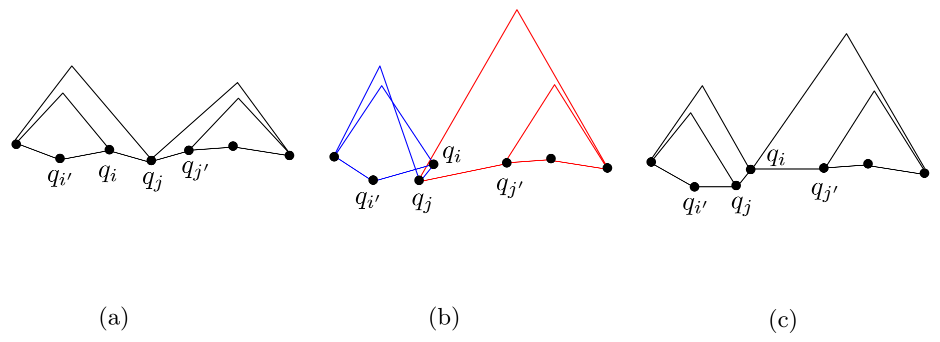


Figure 7.7: (a) Before changing the ordering of q_i and q_j. (b) After points q_i and q_j change their order. (c) Allocating $Inc(q_i)$ to the point q_j and vice versa.

We also maintain a list L_Q of the set of points Q sorted by increasing order of their x-coordinates. When the points move, the order of the x-coordinates of two consecutive points may change. For each pair of consecutive points q_i and q_j, where $j = i + 1$, we maintain a certificate certifying that the x-coordinate of q_i is smaller than the x-coordinate of q_j. The failure time of the certificate is the time t when $x_i(t) = x_j(t)$; we say that an *order event* occurs at time t^+. We put the certificates in a priority queue with the failure times as their keys.

Lemma 7.2. *The dynamic and kinetic tournament tree $\mathcal{TT}$ can be constructed in linear time. It generates $O(n^2\beta_{2s+2}(n)\log n)$ events, each in worst-case time $O(\log^2 n)$.*

Proof. By Theorem 2.3, a dynamic and kinetic tournament tree on $O(n)$ elements can be constructed in linear time.

Since the coordinate functions for each point are polynomials of at most constant degree s, the order of the points according to their x-coordinates changes $\bar{m} = O(n^2)$ times, which is equal to the number of all insertions and deletions into the tournament tree $\mathcal{TT}$. The functions of each pair of edges at leaves of $\mathcal{TT}$ intersect at most $2s$ times. Therefore, from Theorem 2.3, the $\mathcal{TT}$ generates $O(\bar{m}\beta_{2s+2}(n)\log n) = O(n^2\beta_{2s+2}(n)\log n)$ events, each in time $O(\log^2 n)$. □

Let $q_{i'}$, q_i, q_j, and $q_{j'}$ be four consecutive points of L_Q. Whenever an order event between q_i and q_j occurs, we update the dynamic and kinetic tournament tree $\mathcal{TT}$ as follows. We delete two edges $q_{i'}q_i$ and $q_jq_{j'}$ from $\mathcal{TT}$ and add two new edges $q_{i'}q_j$ and $q_iq_{j'}$ into $\mathcal{TT}$. Then we delete the certificates corresponding to q_i and q_j, and replace them with new ones certifying that the order of the x-coordinates of $q_{i'}$, q_j, q_i, and $q_{j'}$ is in increasing order; the failure times are the times when two consecutive points change their x-coordinate ordering.

Note that when an order event between q_i and q_j occurs, the embedding might have edge crossings (see Figure 7.7). Thus we must repair the embedding to avoid edge crossings. Let $Inc(q_i)$ be the set of edges incident to q_i. When such event occurs, some edges in the set $Inc(q_i)$, if non-empty, might cross some edges in the set $Inc(q_j)$, if non-empty; see Figure 7.7(b). To remove the edge crossings and restore the embedding, we allocate $Inc(q_i)$ to the point q_j; similarly, we allocate $Inc(q_j)$ to the point q_i; see Figure 7.7(c).

Now we can state and prove the main result of this section.

Theorem 7.3. *Given an initial point set embedding of a plane graph $G = (V, E)$ with at most three bends per edge on a set P of $n = |V|$ points, where the trajectory of each point is given by a polynomial function of constant degree at most s, there is a KDS that maintains the embedding and that satisfies the following properties. The KDS has linear size, and processes $O(n^2\beta_{2s+2}(n)\log n)$ events, each in $O(\log^2 n)$ time. The KDS is efficient, responsive, compact and local. At any time, the mapping of each edge of G can be obtained in worst-case time $O(\log^2 n)$.*

Proof. Since the number of dummy vertices is $O(n)$, the size of the $\mathcal{TT}$ together with the number of certificates is $O(n)$. Thus the KDS uses $O(n)$ space, and it is compact.

When an order event between q_i and q_j occurs, we first allocate $Inc(q_i)$ to the point q_j and $Inc(q_j)$ to the point q_i, and then apply a constant number of changes to the priority queue which takes $O(\log n)$ time. Next a constant number of insertions and deletions is made in $\mathcal{TT}$; each change to $\mathcal{TT}$ can be processed in time $O(\log^2 n)$. This implies that the KDS is responsive.

The number of changes to the maximum slope δ, which is equal to the number of all changes at the root of $\mathcal{TT}$, is $O(n^2\beta_{2s+2}(n))$. By Lemma 7.2 the number of internal events is $O(n^2\beta_{2s+2}(n)\log n)$. Therefore, the ratio of the number of internal events to the number of external events is polylogarithmic in n, so the KDS is efficient.

Each point involves in a constant number of order events in L_Q, and $O(\log n)$ tournament events in $\mathcal{TT}$, so the number of all certificates associated with a particular point is polylogarithmic in n. Thus the proposed KDS is *local*.

At any time t, we can obtain the slopes of a given edge in an efficient time. If an event occurs at time t, we spend $O(\log^2 n)$ time to handle the event by updating the maximum slope δ, and then generating the slopes of the given edge from δ; otherwise, it takes $O(1)$ time. $\square$

Remark 5.1. Recall from Section 7.1.2 that the slopes of edge q_iq_j, for a 2-bend per edge drawing, arise from two values δ_l and δ_r stored at node n_{ij}. In order to maintain these values in the kinetic setting we define two dynamic kinetic tournament trees $\mathcal{TT}_l$ and $\mathcal{TT}_r$ whose roots store δ_l and δ_r, respectively. The values δ_l (resp. δ_r) stored at the children of n_{ij} are stored at the leaves of $\mathcal{TT}_l$ (resp. $\mathcal{TT}_r$); the root of $\mathcal{TT}_l$ (resp. $\mathcal{TT}_r$) maintains the larger value δ_l (resp. δ_r) of the children.

Let n_{kl} be the parent of node n_{ij} and let $\mathcal{P}_{kl}$ be the path from n_{kl} to the root of the nested tree $\mathcal{T}$. When the root of the tournament tree $\mathcal{TT}_l$ (resp. $\mathcal{TT}_r$) corresponding to n_{ij} changes, the δ_l (resp. δ_r) of a child of n_{kl} is replaced by a new one. Thus an insertion and a deletion are done in the corresponding tournament tree of n_{kl}; this tournament tree may cause an insertion and a deletion in the parent of n_{kl} and hence updates to all corresponding tournament trees on the path $\mathcal{P}_{kl}$.

Therefore, when two points q_i and q_j change their x-coordinate ordering, we update the values δ_l and δ_r of the nodes corresponding to these points, up to the root of $\mathcal{T}$. Using this process we obtain a compact KDS for a 2-bend drawing.

Chapter 8

Conclusions and Open Problems

In this chapter, we briefly review the known results for the problems discussed in this dissertation, and then pose several open problems.

Kinetic All Nearest Neighbors. In Chapter 3, we have provided a kinetic data structure for the all nearest neighbors problem for a set of moving points in the plane. We have applied our structure to maintain the closest pair as the points move. Comparison of our algorithm with the algorithm of Agarwal *et al.* [10] shows that in $\mathbb{R}^2$, our deterministic algorithm is simpler and more efficient than their randomized algorithm for maintaining all the nearest neighbors. In higher dimensions, our deterministic method for maintaining the Equilateral Delaunay graph does not satisfy all four kinetic performance criteria in Section 1.1. For example, in $\mathbb{R}^3$, the number of edges of the Equilateral Delaunay graph is $O(n^2)$, and so for maintenance of all the nearest neighbors, our kinetic approach needs $O(n^2)$ space. By contrast, the randomized kinetic data structure by Agarwal *et al.* [10] uses $O(n \log^3 n)$ space. Thus, for higher dimensions ($d \geq 3$), their approach is asymptotically more efficient, but the simplicity of our algorithm may make it more attractive.

In Chapter 5, we have provided a new method for finding a deterministic kinetic algorithm to maintain all the nearest neighbors in any dimension $d \geq 2$, that does satisfy the performance criteria of the efficiency and the compactness. In fact, Chapter 5 gives deterministic KDS's for maintenance of both the Semi-Yao graph and all the nearest neighbors in $\mathbb{R}^d$. These KDS's are responsive in an amortized sense.

Therefore, open directions include:

Open Problem 1. *Design KDS's for the Semi-Yao graph and all the nearest neigh-*

bors such that each event can be handled in a polylogarithmic worst case time.

Open Problem 2. *Design an exact KDS for maintenance of all the nearest neighbors that is local in the worst-case.*

In Chapter 6, we have provided a KDS for maintenance of all the k-nearest neighbors in order to answer RkNN queries over time. This is the first KDS for maintenance of all the k-nearest neighbors in $\mathbb{R}^d$, for any $k \geq 1$. It processes $O(\phi(s,n) * n^2)$ events, each in time $O(\log n)$ in an amortized sense. Thus an open problem, where each point moves along a bounded degree polynomial function, is:

Open Problem 3. *Design a KDS for maintaining all the k-nearest neighbors that processes less than $O(\phi(s,n) * n^2)$ events.*

Arya *et al.* [13] have a kd-tree implementation to approximate the nearest neighbors of a query point that is in use by practitioners [39] who have found it challenging to implement the theoretical algorithms [91, 27, 38, 42]. To report all the k-nearest neighbors ordered by distance from each point, our method uses multidimensional range trees, which can be easily implemented. Consequently, we believe our method may be useful in practice.

Kinetic EMST. In Chapter 4, we have provided a KDS for maintenance of the EMST and the Yao graph on a set of n moving points in the plane. Our EMST KDS processes $O(n^3\beta^2_{2s+2}(n)\log n)$ events, which improves the previous $O(n^4)$ bound of Rahmati and Zarei [79]. The kinetic algorithm of Rahmati and Zarei results in a KDS having $O(n^{3+\epsilon})$ events, for any $\epsilon > 0$, under the assumption that any four points can be co-circular at most twice [81], or at most three times where each point moves along a straight line at unit speed [80]. Our KDS further improves the upper bound $O(n^{3+\epsilon})$ under the above assumptions. Our kinetic approach can also be used to maintain an L_1-MST and an L_∞-MST. In addition, by defining the Pie Delaunay graph and the Yao graph in higher dimensions, one might use our approach to give a KDS for maintenance of the EMST in $\mathbb{R}^d$, but this approach does not satisfy all the performance criteria.

For the number of combinatorial changes of the EMST (resp. L_1-MST and L_∞-MST) of linearly moving points in the plane, Katoh *et al.* [58] proved an upper bound of $O(n^3 2^{\alpha(n)})$ (resp. $O(n^{5/2}\alpha(n))$. Here, $\alpha(n)$ is the inverse Ackermann function. The upper bound was later improved to $O(\lambda_{ps+2}(n)n^{2-1/(9.2^{ps-3})}\log^{2/3} n)$ for the L_p-MST in $\mathbb{R}^d$, where the coordinates of the points are polynomial functions of constant maximum degree s by Chan [28]; for $p = 2$ and $s = 1$, this formula gives the first improvement $O(n^{25/9} 2^{\alpha(n)} \log^{2/3} n)$ over Katoh *et al.*'s $O(n^3 2^{\alpha(n)})$ bound. An even better bound $O(n^{8/3} 2^{\alpha(n)} \log^{4/3} n)$ can be obtained by combining the results of Chan [28] with those of Marcus and Tardos [67].

Therefore, for a set of n moving points where the trajectory of each point is a polynomial function of bounded degree, future directions are:

Open Problem 4. *Find a tight upper bound for the number of combinatorial changes of the EMST.*

Open Problem 5. *Find a KDS for the EMST that processes a sub-cubic number of events, and such that it is responsive, local, and compact.*

Kinetic Point Set Embedding. In Chapter 7, we have provided a KDS for maintaining a drawing for plane graph G on a set of moving points with at most three bends per edge. In terms of the standard evaluation criteria in the KDS framework, the KDS is efficient, responsive, local, and compact. We can kinetically maintain a 2-bend drawing, but while this KDS is compact, it does not satisfy the other three performance criteria. Thus some future continuations of the new research area of kinetic graph drawing include:

Open Problem 6. *Find a KDS for a point set embedding of a plane graph on a set of points with at most two bends per edge that satisfies all four performance criteria.*

Open Problem 7. *Find a KDS for maintenance of straight-line, crossing-free drawings on moving points for some special graphs like outerplanar graphs and trees.*

The *k-colored Point Set Embedding* problem [14, 48, 49, 93] is a general version of the point set embedding problem, and is defined as follows. Given: a planar graph G of n vertices with a partition of the vertex set into subsets $V_0, ..., V_{k-1}$, and a point set P of n points in the plane with a partition of the point set into $P_0, ..., P_{k-1}$. To find: a drawing such that the vertices of V_i are mapped to the points P_i, respectively, and such that the number of bends along each edge is kept small. Another future direction in the area of kinetic graph drawing would be:

Open Problem 8. *Find a kinetic algorithm for maintenance of a k-colored point set embedding of a planar graph G on a point set P, such that the number of crossings and the number of bends per edge are kept small.*

Bibliography

[1] Mohammad Ali Abam. *New Data Structures and Algorithms for Mobile Data.* PhD Thesis, Eindhoven University of Technology, 2007.

[2] Mohammad Ali Abam and Mark de Berg. Kinetic spanners in $\mathbb{R}^d$. *Discrete & Computational Geometry*, 45(4):723–736, 2011.

[3] Mohammad Ali Abam, Mark de Berg, and Joachim Gudmundsson. A simple and efficient kinetic spanner. *Computational Geometry: Theory and Applications*, 43:251–256, 2010.

[4] Mohammad Ali Abam, Zahed Rahmati, and Alireza Zarei. Kinetic pie Delaunay graph and its applications. In *Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT '12)*, volume 7357 of *Lecture Notes in Computer Science*, pages 48–58. Springer-Verlag, 2012.

[5] P. K. Agarwal, B. Aronov, T. M. Chan, and M. Sharir. On levels in arrangements of lines, segments, planes, and triangles. *Discrete & Computational Geometry*, 19(3):315–331, 1998.

[6] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66:207–243, 2003.

[7] Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6(5):407–422, 1991.

[8] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS '98)*, pages 596–605, Washington, DC, USA, 1998. IEEE Computer Society.

[9] Pankaj K. Agarwal, Jie Gao, Leonidas Guibas, Haim Kaplan, Vladlen Koltun, Natan Rubin, and Micha Sharir. Kinetic stable Delaunay graphs. In *Proceedings of the 26th ACM Symposium on Computational Geometry (SoCG '10)*, pages 127–136, New York, NY, USA, 2010. ACM.

[10] Pankaj K. Agarwal, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Transactions on Algorithms*, 5:4:1–37, 2008.

[11] Gerhard Albers, Joseph S.B. Mitchell, Leonidas J. Guibas, and Thomas Roos. Voronoi diagrams of moving points. *International Journal of Computational Geometry and Applications*, 8:365–380, 1998.

[12] Giora Alexandron, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. *Computational Geometry: Theory and Applications*, 36(2):144–158, 2007.

[13] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[14] Melanie Badenta, Emilio Di Giacomo, and Giuseppe Liotta. Drawing colored graphs on colored points. *Theoretical Computer Science*, 408:129 – 142, 2008.

[15] Julien Basch. *Kinetic Data Structures*. PhD Thesis, Stanford University, 1999.

[16] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 747–756, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[17] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–19, 1999.

[18] Julien Basch, Leonidas J. Guibas, and Li Zhang. Proximity problems on moving points. In *Proceedings of the 13th Annual Symposium on Computational Geometry (SoCG '97)*, pages 344–351, New York, NY, USA, 1997. ACM.

[19] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB Journal*, 15(3):229–249, 2006.

[20] Jon Louis Bentley and Michael Ian Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing (STOC '76)*, pages 220–230, New York, NY, USA, 1976. ACM.

[21] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[22] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and David Ilcinkas. Connections between theta-graphs, Delaunay triangulations, and orthogonal surfaces. In *Proceedings of the 36th International Conference on Graph-theoretic Concepts in Computer Science (WG'10)*, volume 6410 of *Lecture Notes in Computer Science*, pages 266–278, Berlin, Heidelberg, 2010. Springer-Verlag.

[23] Prosenjit Bose. On embedding an outer-planar graph in a point set. *Computational Geometry: Theory and Applications*, 23(3):303–312, 2002.

[24] Prosenjit Bose, Michael McAllister, and Jack Snoeyink. Optimal algorithms to embed trees in a point set. *Journal of Graph Algorithms and Applications*, 1, 1997.

[25] Sergio Cabello. Planar embeddability of the vertices of a graph using a fixed point set is NP-hard. *Journal of Graph Algorithms and Applications*, 10(2):353–363, 2006.

[26] Paul B. Callahan and S. Rao Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 291–300, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[27] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

[28] Timothy M. Chan. On levels in arrangements of curves. *Discrete and Computational Geometry*, 29:375–393, 2003.

[29] Timothy M. Chan. On levels in arrangements of curves, ii: A simple inequality and its consequences. *Discrete & Computational Geometry*, 34(1):11–24, 2005.

[30] Timothy M. Chan. On levels in arrangements of curves, iii: further improvements. In *Proceedings of the 24th annual Symposium on Computational Geometry (SoCG '08)*, pages 85–93, New York, NY, USA, 2008. ACM.

[31] Timothy M. Chan and Sunil Arya. Better ϵ-dependencies for offline approximate nearest neighbor search, Euclidean minimum spanning trees, and ϵ-kernels. In *Proceedings of the 30th Annual Symposium on Computational Geometry (SoCG '14)*, New York, NY, USA, 2014. ACM.

[32] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the 27th Annual Symposium on Computational Geometry (SoCG '11)*, pages 1–10, New York, NY, USA, 2011. ACM.

[33] Chiuyuan Chen. Any maximal planar graph with only one separating triangle is hamiltonian. *Journal of Combinatorial Optimization*, 7(1):79–86, 2003.

[34] Otfried Cheong, Antoine Vigneron, and Juyoung Yon. Reverse nearest neighbor queries in fixed dimension. *International Journal of Computational Geometry and Applications*, 21(02):179–188, 2011.

[35] L. Paul Chew and Robert L. (Scot) Drysdale, III. Voronoi diagrams based on convex distance functions. In *Proceedings of the 1st Annual Symposium on Computational Geometry (SoCG '85)*, pages 235–244, New York, NY, USA, 1985. ACM.

[36] N. Chiba and T. Nishizeki. The hamiltonian cycle problem is linear-time solvable for 4-connected planar graphs. *Journal of Algorithms*, 10(2):187–211, 1989.

[37] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[38] Kenneth L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS '83)*, pages 226–232, Washington, DC, USA, 1983. IEEE Computer Society.

[39] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):599–608, 2010.

[40] Artur Czumaj, Funda Ergün, Lance Fortnow, Avner Magen, Ilan Newman, Ronitt Rubinfeld, and Christian Sohler. Approximating the weight of the euclidean minimum spanning tree in sublinear time. *SIAM Journal on Computing*, 35(1):91–109, 2005.

[41] Erik D. Demaine, Joseph S. B. Mitchell, and Joseph ORourke. *The Open Problems Project*. http://www.cs.smith.edu/orourke/TOPP.

[42] Matthew T. Dickerson and David Eppstein. Algorithms for proximity problems in higher dimensions. *International Journal of Computational Geometry and Applications*, 5(5):277–291, 1996.

[43] Robert L. (Scot) Drysdale, III. A practical algorithm for computing the Delaunay triangulation for convex distance functions. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 159–168, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[44] Tobias Emrich, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, Naixin Xu, and Andreas Züfle. Reverse k-nearest neighbor monitoring on mobile objects. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '10)*, pages 494–497, New York, NY, USA, 2010. ACM.

[45] Jeff Erickson. On the relative complexities of some geometric problems. In *Proceedings of the 7th Canadian Conference on Computational Geometry (CCCG '95)*, pages 85–90, 1995.

[46] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in x + y and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208, 1982.

[47] Jyh-Jong Fu and R. C. T. Lee. Minimum spanning trees of moving points in the plane. *IEEE Transactions on Computers*, 40(1):113–118, 1991.

[48] Emilio Giacomo, Giuseppe Liotta, and Francesco Trotta. Drawing colored graphs with constrained vertex positions and few bends per edge. *Algorithmica*, 57:796–818, 2010.

[49] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, Henk Meijer, Francesco Trotta, and Stephen K. Wismath. k-colored point-set embeddability of outerplanar graphs. *Journal of Graph Algorithms and Applications*, 12(1):29–49, 2008.

[50] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Stephen K. Wismath. Curve-constrained drawings of planar graphs. *Computational Geometry*, 30(1):1 – 23, 2005.

[51] P. Gritzmann, B. Mohar, Jnos Pach, and Richard Pollack. Embedding a planar triangulation with vertices at specified points. *American Mathematical Monthly*, 98:165–166, 1991.

[52] Leonidas J. Guibas. Kinetic data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, pages 23–1–23–18. Chapman and Hall/CRC, 2001.

[53] Leonidas J. Guibas and Joseph S. B. Mitchell. Voronoi diagrams of moving points in the plane. In *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '91)*, Lecture Notes in Computer Science, pages 113–125. Springer, 1991.

[54] Guido Helden. Each maximal planar graph with exactly two separating triangles is hamiltonian. *Discrete Applied Mathematics*, 155(14):1833 – 1836, 2007.

[55] Yoshiko Ikebe, Micha A. Perles, Akihisa Tamura, and Shinnichi Tokunaga. The rooted tree embedding problem into points in the plane. *Discrete & Computational Geometry*, 11:51–63, 1994.

[56] James M. Kang, Mohamed F. Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE '07)*, pages 806–815, 2007.

[57] Menelaos I. Karavelas and Leonidas J. Guibas. Static and kinetic geometric spanners with applications. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*, pages 168–176, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[58] Naoki Katoh, Takeshi Tokuyama, and Kazuo Iwano. On minimum and maximum spanning trees of linearly moving points. *Discrete & Computational Geometry*, 13:161–176, 1995.

[59] Michael Kaufmann and Roland Wiese. Embedding vertices at points: Few bends suffice for planar graphs. *Journal of Graph Algorithms and Applications*, 6(1):115–129, 2002.

[60] J. Mark Keil and Carl A. Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete & Computational Geometry*, 7:13–28, 1992.

[61] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[62] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 201–212, New York, NY, USA, 2000. ACM.

[63] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, 1956.

[64] Yokesh Kumar, Ravi Janardan, and Prosenjit Gupta. Efficient algorithms for reverse proximity query problems. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '08)*, pages 39:1–39:10, New York, NY, USA, 2008. ACM.

[65] Jessica Lin, David Etter, and David DeBarr. Exact and approximate reverse nearest neighbor search for multimedia data. In *Proceedings of the 2008 SIAM International Conference on Data Mining (SDM '08)*, pages 656–667. SIAM, 2008.

[66] Anil Maheshwari, Jan Vahrenhold, and Norbert Zeh. On reverse nearest neighbor queries. In *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG '02)*, pages 128–132, 2002.

[67] Adam Marcus and Gábor Tardos. Intersection reverse sequences and geometric applications. *Journal of Combinatorial Theory, Series A*, 113(4):675–691, 2006.

[68] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer Verlag, Berlin, 1984.

[69] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[70] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[71] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, New York, NY, USA, 2nd edition, 1998.

[72] Seth Pettie. Sharp bounds on Davenport-Schinzel sequences of every order. In *Proceedings of the 29th Annual Symposium on Computational Geometry (SoCG '13)*, pages 319–328, New York, NY, USA, 2013. ACM.

[73] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, 1957.

[74] Michael O. Rabin. Probabilistic algorithms. In *Algorithms and Complexity: New Direction and Results*, pages 21–39. Academic Press, 1976.

[75] Zahed Rahmati, Mohammad Ali Abam, Valerie King, and Sue Whitesides. Kinetic data structures for the Semi-Yao graph and all nearest neighbors in $\mathbb{R}^d$. In *Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG '14)*, 2014.

[76] Zahed Rahmati, Valerie King, and Sue Whitesides. Kinetic data structures for all nearest neighbors and closest pair in the plane. In *Proceedings of the 29th Symposium on Computational Geometry (SoCG '13)*, pages 137–144, New York, NY, USA, 2013. ACM.

[77] Zahed Rahmati, Valerie King, and Sue Whitesides. Kinetic reverse k-nearest neighbor problem. In *Proceedings of the 25th International Workshop on Combinatorial Algorithms (IWOCA '14)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[78] Zahed Rahmati, Sue Whitesides, and Valerie King. Kinetic and stationary point-set embeddability for plane graphs. In *Proceedings of the 20th International Symposium on Graph Drawing (GD '12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 279–290. Springer-Verlag, 2013.

[79] Zahed Rahmati and Alireza Zarei. Kinetic Euclidean minimum spanning tree in the plane. *Journal of Discrete Algorithms*, 16(0):2–11, 2012.

[80] Natan Rubin. On kinetic Delaunay triangulations: A near quadratic bound for unit speed motions. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS '13)*, pages 519–528, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[81] Natan Rubin. On topological changes in the Delaunay triangulation of moving points. *Discrete & Computational Geometry*, 49(4):710–746, 2013.

[82] Daniel Russel. *Kinetic Data Structures in Practice.* PhD Thesis, Stanford University, 2007.

[83] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Proceedings of the 16th IEEE Symposium on Foundations of Computer Science (FOCS '75)*, pages 151–162, 1975.

[84] Micha Sharir. On k-sets in arrangements of curves and surfaces. *Discrete & Computational Geometry*, 6(1):593–613, 1991.

[85] Micha Sharir and Pankaj K. Agarwal. *Davenport-Schinzel Sequences and their Geometric Applications.* Cambridge University Press, New York, NY, USA, 1995.

[86] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[87] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proceedings of the 2000 ACM SIGMOD*

Workshop on Research Issues in Data Mining and Knowledge Discovery, pages 44–53, 2000.

[88] Yufei Tao, Dimitris Papadias, Xiang Lian, and Xiaokui Xiao. Multidimensional reverse knn search. *The VLDB Journal*, 16(3):293–316, 2007.

[89] Yufei Tao, Man Lung Yiu, and Nikos Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 18(9):1239–1252, 2006.

[90] W.T. Tutte. A theorem on planar graphs. *Transactions of the American Mathematical Society*, 82:99–116, 1956.

[91] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(2):101–115, 1989.

[92] Pravin M. Vaidya. Minimum spanning trees in k-dimensional space. *SIAM Journal on Computing*, 17(3):572–582, 1988.

[93] Mereke van Garderen, Giuseppe Liotta, and Henk Meijer. Universal point sets for 2-coloured trees. *Information Processing Letters*, 112:346 – 350, 2012.

[94] Shengsheng Wang, Qiannan Lv, Dayou Liu, and Fangming Gu. Efficient filter algorithms for reverse k-nearest neighbor query. In *Proceedings of the 12th International Conference on Web-Age Information Management (WAIM '11)*, volume 6897 of *Lecture Notes in Computer Science*, pages 18–30. Springer Berlin Heidelberg, 2011.

[95] H Whitney. A theorem on graphs. *Annals of Mathematics*, 32:378–390, 1931.

[96] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

[97] Wei Wu, Fei Yang, Chee-Yong Chan, and K.-L. Tan. Continuous reverse k-nearest-neighbor monitoring. In *Proceedings of the 9th International Conference on Mobile Data Management (MDM '08)*, pages 132–139, 2008.

[98] Tian Xia and Donghui Zhang. Continuous reverse nearest neighbor monitoring. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, pages 77–77, 2006.

[99] Andrew Chi-Chih Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982.

[100] Man Lung Yiu and Nikos Mamoulis. Reverse nearest neighbors search in ad hoc subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):412–426, 2007.

[101] Ming Zhang and Reda Alhajj. Effectiveness of NAQ-tree in handling reverse nearest-neighbor queries in high-dimensional metric space. *Knowledge and Information Systems*, 31(2):307–343, 2012.