KINETIC DATA STRUCTURES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Julien Basch
June 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Leonidas J. Guibas
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Hershberger

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Rajeev Motwani

Approved for the University Committee on Graduate Studies:

Abstract

Modeling the physical world in the computer raises problems that intertwine discrete and continuous aspects. For example, physical objects move along continuous trajectories; yet every so often discrete events occur, such as collisions between objects.

In a model of objects in space, there are many discrete attributes that one may want to compute: the closest pair, the convex hull, the minimum spanning tree, etc. When the objects are in motion, the values of these attributes change over time, and it becomes necessary to keep track of them as the objects move.

In this thesis, we introduce a general approach, and an analysis framework, for solving this type of problems. To keep track of a discrete attribute, we create a new type of data structure, called a kinetic data structure. A kinetic data structure is made of a proof of correctness of the attribute which is animated through time by a discrete event simulation.

Acknowledgements

First and foremost, I wish to thank my seniors Sanjeev Khanna and Ramkumar Gurumurthy. They helped me start on a research track during my first few years at Stanford.

I am greatly endebted to my advisor Leo Guibas, who is in more ways than it is possible to say a great source of inspiration.

I wish to thank John Hershberger for reading so carefully this thesis, helping me to remove a great many errors from the final version.

The work presented in this thesis wouldn't have existed without the essential participation of a number of co-authors. I wish to thank first Leo Guibas and John Hershberger, and Harish Devarajan, Ramkumar Gurumurthy, Piotr Indyk and Li Zhang. I was also very glad to interact professionally with Rajeev Motwani, Sanjeev Khanna, Lyle Ramshaw, Eric Veach, Olaf Hall-Holt, Jeff Erickson, Marc de Berg, Jorge Stolfi and Pankaj Agarwal.

Stanford would not have been Stanford for me without Chandra, Chiaki, Harish, Mélanie, Piero, Ramkumar, Sanjeev, Sean and Young. My family in France was always supportive over the phone or by email, in particular my sister Sandra and my aunt Judith.

Finally, Pankaj Agarwal and Terry Vance were of tremendous help as I was struggling to finish this thesis.

Contents

List of Figures

Chapter 1

Introduction

Modeling the physical world in the computer raises problems that combine discrete and continuous aspects. For instance, one may want to compute the smallest distance between all pairs of objects in a set. This distance is a real number, but its combinatorial description (the identity of the pair that realize this distance) is a *discrete attribute* that we call the ***closest pair***. The closest pair depends on some continuously defined parameters (the coordinates of the objects).

The continuous and discrete aspects are even more intertwined in physical simulations or real-time settings, in which the objects' positions depend on time. As physical objects move along continuous trajectories, the smallest distance changes continuously with time. The closest pair, on the other hand, changes only at discrete times. Moreover, in the course of a physical simulation, the continuous trajectories themselves are subject to occasional discrete changes: when a collision occurs, for instance, the instantaneous velocities of the two objects colliding have to be recomputed.

One can often reduce the treatment of motion to a sequence of static problems. Consider a physical simulation of a gas, in which each molecule is represented as a tiny sphere. To perform the simulation, one first selects a certain global *time step* Δt. Given the state of the simulation S_t at a certain time t, one computes the new positions of all the molecules at time $t + \Delta t$, checks for possible collisions, and updates the velocities of all the colliding pairs, to obtain the new state $S_{t+\Delta t}$. One then repeats this step forever.

How to choose Δt? If it is too large, then we will observe some aberrant behavior

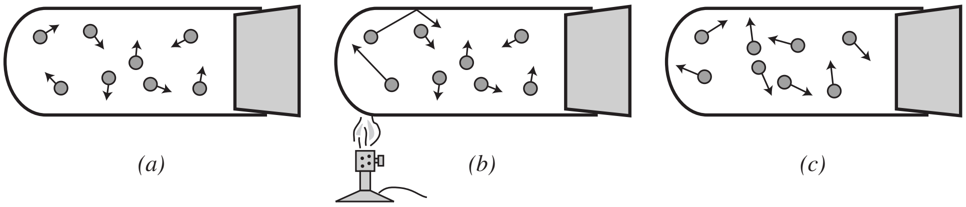


Figure 1.1: When simulating a system with objects going at vastly varying speeds, a global time step is not desirable.

(bullets will start going through windows without breaking the glass). If it is too small, a lot of computation time will be wasted to check for collisions at times when none occurred.

Let's imagine now that Δt has been chosen just the right way: between S_t and $S_{t+\Delta t}$, just a few molecules have collided, and only by a small amount. What this means is that the rest of the simulation has barely changed. In other words, there is a *temporal coherence* between steps, and it seems wasteful to restart a full collision checking algorithm from scratch, while a lot of information could be gained from the previous computation that was done on a very similar input.

Many existing collision detection systems attempt to take advantage of this temporal coherence by keeping some data structures from one time step to the next. This thesis, on the other hand, proposes a new approach that completely gets rid of this global time step approach altogether.

In a simulation, the trajectory of each object is typically given by a partial differential equation. Numerical methods are used to integrate this equation over time, with a fixed or adaptive time step. So, as a time step is needed to perform this motion integration, why should we attempt to get rid of it for the collision detection? To answer this question, let's consider the specific scenario of a cold gas in a long horizontal tube. At some instant t_0, the left end of the tube is heated during one second, and we would like to simulate the propagation of the heat along the tube (Figure 1.1). In this case, we want to use very different time steps at the left end of the tube, where molecules are suddenly starting to move very fast, and at the right end of the tube, where they will keep their initial slow speed for a while. Hence, even though a time step is needed for each molecule to perform their motion integration, a global time step should be avoided.

One way to perform collision detection is to keep track of the closest pair of molecules, but this is only one of many discrete attributes that could be desirable to maintain efficiently. For instance, in a flight simulator game, an important computational task is to render a complex 3D scene on the computer screen. For this purpose, fast rendering hardware has been designed to which one directly sends a set of triangles in 3D. The hardware automatically clips the invisible parts at the pixel level. The excellent performance of the rendering hardware shifted the bottleneck at the communication level: it takes too much time to send all the triangles to the hardware. To overcome this bottleneck, special *visibility data structures* are added, which allow a program to cull most of the invisible polygons, and to send only a very small set to the rendering hardware. Such data structures, being the result of a discrete computation, are also discrete attributes of the input scene. In our flight simulator, many objects may move at the same time at vastly different speeds (planes, tanks). Once again, we would like to take advantage of the temporal coherence to avoid recomputing the visibility data structures before every frame is displayed on the screen.

In general, we would like to associate with any given attribute that is defined on static data a so called *kinetic* equivalent on moving data, that is, a data structure with two fundamental operations:

1. A ***query*** operation, which returns the attribute of interest (say, the closest pair), or some value depending on the attribute being maintained (say, the area of the convex hull);

2. An ***update*** operation[1], which updates the structure's internal contents to reflect the positions of the data at the current time.

A typical use of such a data structure, e.g., in the flight simulator example, would be to perform sixty times per second: (1) an update call for the visibility structure, and (2) a query that returns all currently visible polygons (to send them to the rendering hardware). This thesis proposes general techniques to design and analyze data structures specially suited for keeping track of discrete attributes of moving data. These data structures are called *kinetic data structures*.

[1]The update operation is different from what is usually understood in the context of dynamic data structures, in which objects don't move but can be inserted or deleted.

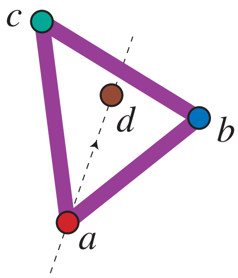


Figure 1.2: The convex hull diagram of four items in a given configuration.

1.1 Kinetic Data Structures

We begin with an informal discussion of the fundamental idea of a kinetic data structure, or ***KDS*** for short. In summary, a kinetic data structure is obtained by taking an algorithm for computing a discrete attribute, turning it into a proof that this attribute is correct, and *animating this proof through time.*

Our running example will be the convex hull of a small set of points in the plane. The convex hull is an infinite set: it is the set of all convex combinations of the input points. Its combinatorial description, however, is finite: it is the (counter-clockwise oriented) circular list of all input points that appear on the convex hull boundary.

Let's make this distinction between geometric objects and their combinatorial descriptions even more explicit, and say that we have a set of ***items***, which are abstract entities. A ***configuration*** assigns a ***position*** in the plane to each item. The list of items whose positions constitute the vertices of the convex hull boundary in counter-clockwise order is called the ***convex hull diagram*** of the item set for a given configuration.

Consider four items a, b, c, d in the configuration depicted in Figure 1.2. Their convex hull diagram is (a, b, c).

A possible proof of this fact involves four tests:

a is to the left of bc
d is to the left of bc
b is to the right of ad
c is to the left of ad

What this means is that if we change the positions of a, b, c, d anywhere in the plane,

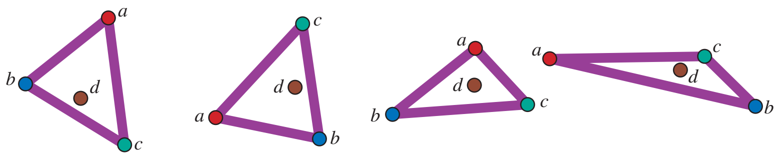


Figure 1.3: Equivalent convex hull diagrams

their convex hull diagram will not change unless at least one of the sideness tests fails (Figure 1.3). Here, we consider *oriented* lines: a is to the left of bc iff it is to the right of cb.

This sideness test ("left of" or "right of") can be expressed as the sign of a determinant dependent on the items' positions:

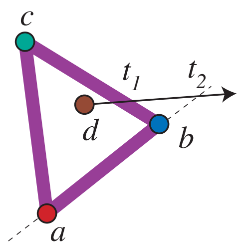
$$c \text{ left of } ab \quad := \quad \begin{vmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{vmatrix} > 0$$

This determinant is in fact twice the signed area of the triangle abc. We denote this determinant, as a function of the configuration, by $\texttt{CCW}(a, b, c)$, because the triangle abc is oriented counter-clockwise exactly when $\texttt{CCW}(a, b, c) > 0$.

We consider a model of motion in which the position of each item is a known function of time. The coordinates of item a are $(x_a(t), y_a(t))$, where x_a, y_a are continuous functions of t. In this model, it is possible to compute the time at which a test fails. In the case of $\texttt{CCW}(a, b, c)$, it is precisely the time t after the current time for which

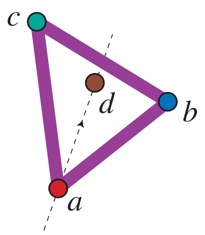
$$\begin{vmatrix} 1 & x_a(t) & y_a(t) \\ 1 & x_b(t) & y_b(t) \\ 1 & x_c(t) & y_c(t) \end{vmatrix} = 0\,.$$

In Figure 1.4, we restart from Figure 1.2. We let d move to the right and compute the failure time of each certificate of our proof. As the certificates prove the correctness of the convex hull diagram, it is unnecessary to do any computation until one of them fails. Hence, we can put all the certificates in a priority queue, ordered by failure time. The failure of a certificate is called an ***event***, and requires some processing: we need to update the convex hull diagram, and to devise a new proof of correctness. In Figure 1.5, we have a new proof of correctness after time t_1.



Certificate	Failure time
a left of bc	never
d left of bc	t_1
b right of ad	t_2
c left of ad	never

Figure 1.4: If d moves at constant speed, some certificate failures happen when it becomes collinear with certain pairs.



Old proof	New proof
a left of bc	a left of bc
d **left** of bc	d **right** of bc
b right of ad	b right of ad
c left of ad	c left of ad

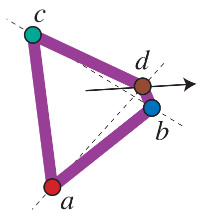


Figure 1.5: A proof update after a certificate failure.

It appears that there are very few differences between the old proof and the new proof. Hence, an event doesn't require a full recomputation, but only a small ***proof update***. This is, in essence, where the *temporal coherence* obtained by continuous motion is exploited in the framework of kinetic data structures.

A ***kinetic data structure*** is therefore simply made of two structures: a proof of correctness of an attribute, and a priority queue called the ***event queue***. The continuous problem has been replaced by a discrete event simulation. The event loop starts from a correct proof, considers the first event, updates the attribute and the proof depending on the certificate that failed, and loops (Figure 1.6). At each step, some certificates are deleted (both from the proof of correctness and from the event queue), and some new certificates are created (and scheduled in the event queue). In the case of a real-time application, there is nothing to do until the current time is past the first failure time in the event queue. In the case of a physical simulation, there is no global time step anymore.

As we mentioned in the second paragraph, there is an essential *on-line* component in many situations: the velocity of an item may change at arbitrary moments. This may be

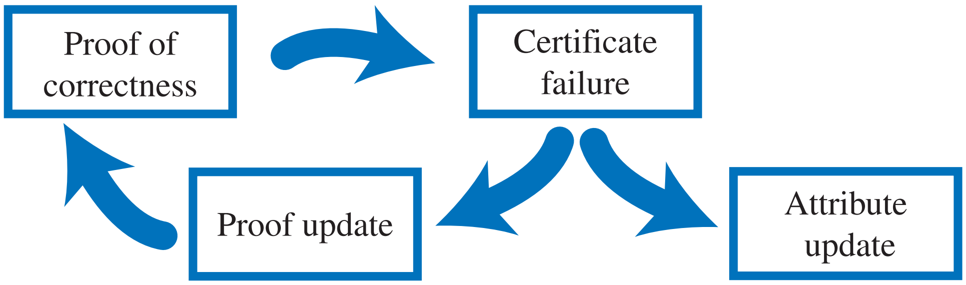


Figure 1.6: The event loop of a kinetic data structure.

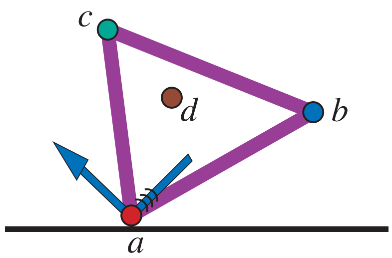


Figure 1.7: If a point bounces, all certificates that depend on it need to have their failure time recomputed. In the proof of Figure 1.2, all certificates except $\texttt{CCW}(b, c, d)$ depend on a.

due to collisions in a physical simulation (Figure 1.7), or to a user action in an interactive environment. There are some certificates whose failure times depend on that item's equation of motion. When it changes its equation of motion, we need to recompute all those failure times. This is straightforward to do with a kinetic data structure, provided we keep for each item a list of the certificates it is involved in. We call such an event a ***motion plan update***.

* * *

How do we get a proof of correctness of the convex hull diagram? Well, an algorithm that computes the convex hull diagram performs a number of tests, and these tests are precisely a proof of correctness of the output: this is what it means, after all, to be a correct algorithm. Therefore, for every *static* convex hull diagram algorithm, we obtain a *proof of correctness*, for which it is possible to compute all certificate failure times. It remains to devise a way to update the proof upon a certificate failure. The process of transforming a static algorithm or data structure into a kinetic data structure is called ***kinetization***.

If we have several kinetic data structures that perform the same function, we need a measure of quality to be able to compare them. The diagram of Figure 1.6 suggests three measures of quality that we review here informally.

First, the proof of correctness should be small: we'll call a KDS ***compact*** if its size is not too much more than the size of the smallest proof of correctness of the attribute we wish to maintain. For instance, any proof of correctness of the closest pair or the convex hull diagram requires linearly many certificates.

Second, a proof update should be fast upon a certificate failure. We call a KDS ***responsive*** if the worst-case cost of processing a certificate failure is small—this is the cost of discovering how to update the proof and (possibly) the discrete attribute.

A third key performance measure for a KDS is the worst-case number of events processed. In our example, each certificate failure changed the convex hull, but this might not be the case in a more complicated setting. We make a distinction between ***external events***, i.e., those affecting the configuration function we are maintaining (e.g., convex hull or closest pair), and ***internal events***, i.e., those processed by our structure because of its internal needs, but not affecting the desired configuration function. External events are those that we *have to pay for* in any kinetic data structure. Our aim will be to develop kinetic data structures for which the total number of events processed by the structure in the worst case is asymptotically of the same order as, or only slightly larger than, the number of external events in the worst case. A KDS meeting this condition will be called ***efficient***. Note that in order to compute this worst case, we need to restrict our attention to some specific classes of motion. For instance, Attalah [14] showed that, when considering items whose positions are low-degree polynomial functions of time, the closest pair and the convex hull diagram can change at most roughly quadratically many times in the worst case. Thus, our goal will be to design kinetic data structures for these attributes that process roughly that many internal events for the same class of motions.

Finally, we call a KDS ***local*** if, at any one time, the maximum number of events in the event queue that depend on a single object is small. As we noticed a few paragraphs above, this property is crucial for fast handling of motion plan updates (Figure 1.7).

1.2 Framework

In this section, we define some vocabulary. We will assume here that the reader has a basic knowledge of computational geometry. If this is not the case, the reader is encouraged to read first the background material of Chapter 2, and, if this is not enough, one of the several excellent books now available on the subject [27, 35, 44, 79, 84].

Given a set S of items, a ***configuration*** π associates with each item a point in the plane (or on the real line for one-dimensional configurations), called its ***position***. The position of an item s under configuration π is denoted $s(\pi)$, or simply s if the configuration is clear from the context. An ***attribute*** is a function that associates with each configuration a combinatorial structure based on S. For an attribute A, we denote by $A(\pi)$ its value at configuration π. For instance, the convex hull diagram, the closest pair, the Voronoi diagram, taken as functions of the configuration, are all attributes.

A ***certificate*** based on a tuple of items is a continuous function that associates a real number with each configuration of these items. When this real number is positive for a given configuration, we say that the certificate is ***valid***. When it is negative, we say it is ***invalid***, and when it is zero, we say it is ***degenerate***. We write $[a < b]$ for the certificate that associates with a configuration π the quantity $b(\pi) - a(\pi)$.

Given a set of certificates that are valid in a certain configuration π, what does it mean for this set to be a proof of correctness of an attribute at π? There are in fact two distinct notions of proof, *strong* and *weak*. We say that a set $\mathcal{L}$ of certificates ***strongly certifies*** an attribute A at π if, for any other configuration π' in which all certificates of $\mathcal{L}$ are valid, we have $A(\pi') = A(\pi)$.

Define a ***scenario*** to be a path $(\pi_t)_{t=0}^{1}$ in the space of configurations, that is to say, the position $a(\pi_t)$ of an item a as a function of t is continuous. We say that the set $\mathcal{L}$ of certificates ***weakly certifies*** A at π if, for any scenario (π_t) with $\pi_0 = \pi$ such that all certificates of $\mathcal{L}$ are valid for all t, we have $A(\pi_t) = A(\pi)$ for all t.

If one looks at the space of all configurations of S, a set $\mathcal{L}$ of certificates defines a subset of configurations in which all certificates are valid. For strong certification, we request that the attribute be constant over this whole subset, whereas for weak certification, we request that the attribute be constant only over one path-connected component. This calls for a few examples.

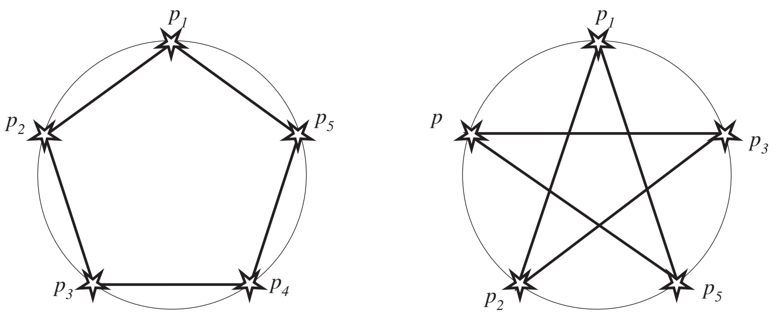


Figure 1.8: The certificates of the left configuration are valid in the right configuration, although the cyclic orders differ. The certificates weakly certify the cyclic order because it is impossible to move continuously from one configuration to the other without having at least one certificate failure.

Example 1.1. Let's first consider a one-dimensional configuration of a set of items, i.e., a configuration that associates with each item a real number. The ***sorted order*** is an attribute, as it is a permutation of S that depends on the positions. For a given configuration π in which the sorted order is $(s_1, \ldots, s_n)$, the set of certificates:

$$\{[s_i < s_{i+1}] \mid i = 1, \ldots, n-1\}$$

strongly certifies the sorted order. Note that the number of certificates is only linear although it requires $n \log n$ tests in the worst case to *compute* the sorted order in the comparison model of computation.

Example 1.2. Let's again take a one-dimensional configuration for S, but this time, the range is the unit circle. The ***cyclic order*** is an attribute that associates with a configuration the cyclic list of items around the circle. More precisely, for a configuration π, the cyclic order is $(s_1, \ldots, s_n)$ if the items positions appear in this order counter-clockwise along the unit circle. In this case, the set of certificates (with indices modulo n)

$$\{\mathtt{CCW}(s_i, s_{i+1}, s_{i+2}) \mid i = 1 \ldots n\}$$

only weakly certifies the cyclic order. In both configurations of Figure 1.8, all certificates are valid, but the cyclic order is not the same in both configurations.

This example can also be used to show that the set of certificates devised by Roos [89] only weakly certifies the Voronoi diagram.

Often, in geometric algorithms, one assumes that a configuration is in *general position* for the geometric tests of interest. What this means is that the configuration is such that none of the geometric test/certificate used is degenerate in this configuration. This assumption is reasonable in a measure-theoretic sense, as the set of configurations that are not in general position has Lebesgue measure zero in the space of all configurations. In other words, an infinitesimal random perturbation of the input will almost surely guarantee that it is in general position. There are delicate problems that intertwine the general position assumption and the numerical inaccuracies due to floating point arithmetic, but we will not try to address them here. In the remainder of this thesis, we use the term "in general position" to mean "in general position with respect to the set of certificates we consider."

When dealing with objects in motion, the assumption of general position cannot hold at all times, as it is precisely when certificates become degenerate (fail) that interesting things happen with kinetic data structures. Our general position assumption therefore has to be "one dimension higher": We will always assume that, in any scenario, two certificates (amongst those that we consider) cannot be degenerate at the same time.

A ***proof scheme*** for an attribute A associates a set of certificates with each configuration in general position. The ***locality*** of the proof scheme is the worst case number of certificates any given item is involved in. When using a proof scheme for a kinetic data structure, the motion plan update of an item can be handled with a computational cost at most equal to the locality times the cost of inserting and deleting an event in the event queue.

Operationally, in a kinetic data structure, we add a data structure to a proof, to help in the proof update when a certificate fails. The ***responsiveness*** of a KDS is the worst case computational cost of processing one event.

Example 1.3. The proof schemes for the sorted order and circular lists have locality $O(1)$, but the one for the Delaunay triangulation [89] has locality $\Theta(n)$. Both schemes have responsiveness $O(\log n)$.

The attributes that we have just seen have a very nice property: there is a (nearly) one-to-one correspondence between the combinatorial elements of the attribute and the certificates needed to certify it. We call such attributes ***self-certifying***. There are other examples of self-certifying structures: the vertical decomposition of a set of segments, for instance, or their constrained Delaunay triangulation. A triangulation of a point set is also self-certifying, but in this latter case, we can have many triangulations, and therefore many proofs, for a given configuration.

When considering a single scenario (π_t) that is clear in the context, we will denote by $S(t)$ the configuration at t and by $s(t)$ the position of item s at t. We will also often need to examine times "just before" or "just after" an event. For a given time t, we define t^+ to be a time after t arbitrarily close to t, and t^- to be a time before t arbitrarily close to t. If we denote by $K(t)$ the status of a kinetic data structure at time t, then processing an event at time t means that we update the structure from $K(t^-)$ to $K(t^+)$.

* * *

It is now time to define classes of allowable motions for analysis purposes. Let us stress the fact that these definitions are not meant to restrict the use of kinetic data structures to certain classes of motion.

A (δ, n)***-scenario*** is a scenario (π_t) on a set S of n items such that, for every item $s \in S$, the function $t \mapsto \pi_t(s)$ is a vector of polynomials in t of degree at most δ, and such that no two certificates are degenerate at the same time.

Sometimes, we would also like a kinetic data structure to be *dynamic*, i.e., we would like to be able to insert and delete items dynamically from time to time. Therefore, we need a model of motion to analyze data structures that are both kinetic and dynamic. Let $\bar{\mathbb{R}}^2 = \mathbb{R}^2 \cup \{\omega\}$, where ω is a special symbol that signifies that the item is "hidden". A (δ, n, m)***-dynamic scenario*** is a scenario in which the positions are in $\bar{\mathbb{R}}^2$ instead of $\mathbb{R}^2$, such that for each $s \in S$, the function $t \mapsto \pi_t(s)$ is a polynomial of degree at most δ on each interval where it is not equal to ω, and with the two additional global requirements that the total number of such intervals (over all items) is at most m, and that at most n items are visible at any time. Using the vocabulary of dynamic data structures, we say talk about deletions and insertions when items switch from "real" positions to ω and back.

The ***efficiency*** of a kinetic data structure K in a scenario is the computational cost of processing this scenario. This cost includes all schedulings and deschedulings in the event queue, as well as any additional time needed to update the supporting structure of the KDS. In a dynamic scenario, we assume that each item knows when it is inserted and deleted, so that it is not necessary to schedule all insertions and deletions in advance.

Example 1.4. The kinetic sorted list has efficiency $O(n^2 \log n)$ in a (δ, n)-scenario, and efficiency $O(mn \log n)$ in a (δ, n, m)-dynamic scenario. The same bounds hold for a *kinetic dictionary*, where the structure maintained also supports logarithmic-cost searches at any time.

1.3 Thesis Overview

In this introduction, we have presented the essential idea of kinetic data structures, which is, once again, *animating proofs through time*. The remainder of this thesis is devoted to demonstrating the practicality of this idea, by applying it to several fundamental problems in computational geometry. The second chapter gives the required background in computational geometry and surveys related work on motion.

In the third chapter, we focus on a simple yet rich one-dimensional problem: that of keeping track of the maximum of one-dimensional items. In the fourth chapter, we consider two fundamental two-dimensional attributes: the convex hull and the closest pair; we provide efficient kinetic data structures in both cases. In the fifth chapter, we partially describe an implementation of the kinetic data structures presented in this thesis. In the sixth chapter, we present a probabilistic model for combinatorial problems involving motion.

* * *

Most of the work presented in this thesis has appeared in conference publications. Kinetic data structures were first introduced at the Eighth Symposium on Discrete Algorithms [17], with the examples of the kinetic tournament, convex hull and closest pair. The other solutions for the kinetic priority queue appeared at the Fourth European Symposium on Algorithms [19] and at the Thirteenth Symposium on Computational Geometry [18]. Preliminary results for the probabilistic analysis of Chapter 6 appeared in the Thirteenth Symposium on Computational Geometry [15]. The implementation chapter is based on an implementation that was the subject of a poster at the same conference [20].

Chapter 2

Background and Related Work

The approach of a Computational Geometer who wants to compute a discrete output from geometric input of finite description complexity consists of several steps:

1. identify some discrete predicates, whose truth values are in general given by the signs of certain polynomials,

2. show how the output depends on the truth values of these predicates only,

3. exploit the constraints given by the fact that the predicates come from a certain space to use only a subset of the predicates defined.

For instance, the convex hull of a finite set of points in the plane is an infinite set, but it has a finite description: the counter-clockwise ordered list of vertices appearing on its boundary. We call this the ***convex hull diagram***. In order to compute this diagram, one first notices that it depends only on the predicate "r is to the left of line (pq)" for all triplets of points of the input set. Next one avoids looking at all $\binom{n}{3}$ such predicates by exploiting the structure of the plane, and one finally obtains an $O(n \log n)$ algorithm to compute the convex hull. The last phase often requires looking at the combinatorial complexity of certain geometric arrangements.

In this chapter, we review some basic material used in or related to this thesis.

2.1 Background

There are a few discrete attributes that are the object of this thesis. The convex hull diagram and the closest pair have been introduced in the previous chapter. Both can be computed in $O(n \log n)$ time for a set of n items positioned in the plane using a variety of different techniques [57, 82, 83, 84]. Data structures exist to perform efficient updates of these attributes upon item insertion and/or deletion [25, 29, 55, 64, 71, 81, 84].

Another classical structure is the ***Voronoi diagram***, which encodes the nearest item to each point in the plane. The Voronoi diagram is a planar map that can also be computed in $O(n \log n)$ time [53, 93]. The geometric dual of the Voronoi diagram is called the ***Delaunay triangulation*** and is also a planar map; it is a triangulation when the items are in general position.

2.1.1 Plane Sweep Methods

Consider a set of line segments in the plane. How can we find whether there exists an intersecting pair? More generally, how can we report all intersecting pairs? These two questions are the simplest ones in the category of *intersection problems*, and were the motivation for the creation of the powerful *space sweep* paradigm, which is an algorithm design technique specific to Computational Geometry.

The idea of a plane sweep method is to reduce the computation of a d-dimensional arrangement of curves and surfaces to that of keeping track of a $(d-1)$-dimensional arrangement that changes over time. Here is how this works for the original problem addressed by Bentley and Ottmann [24], which is that of reporting all pairwise intersections in a family S of line segments in the plane. We present the plane sweep paradigm here in the vocabulary of kinetic data structures. Denote by s_0, s_1 the left and right endpoints of a segment s. We assume here that the input is non-degenerate, i.e., that no three segments intersect at a common point and that no two endpoints or intersections lie on a common vertical line.

One imagines a vertical *sweep line*, first placed at the extreme left of the plane, which will move to the right by jumping between endpoints and intersections. This is done as follows. At a given position x of the sweep line, there is a set of segments S_ℓ to the left of the sweep line, a set of segments S_r to the right of the sweep line, and a set of segments S_i that cross the sweep line. For a segment s in the last set, define $\mathrm{up}(s)$ to be the segment just above s at the current position of the sweep line. The configuration is *certified* by a set of comparisons:

$$\begin{cases} \forall s \in S_\ell, & s_1 < x \\ \forall s \in S_r, & s_0 > x \\ \forall s \in S_i, & s_0 < x \wedge s_1 > x \\ \forall s \in S_i, & s < \mathrm{up}(s) \end{cases}$$

The last comparison is between the y-coordinates of the intersections of the segments with the sweep line.

Each comparison has a *failure time*, i.e., a position of the sweep line at which it becomes invalid. For the left segments, this never happens. A y-coordinate comparisons fails when there is an intersection.

The gist of the method relies on the fact that, as the sweep line moves from left to right, when a comparison fails, there is very little to do to create a new valid set of comparisons that certifies the new configuration. Hence, the algorithm works as follows: put each comparison in an event queue, ordered by failure time, and move the sweep line from failure time to failure time. At each step, update the list of comparisons and the event queue.

Theorem 2.1 ([24, 28]). *The plane sweep method finds all intersections between a set of n line segments in $O((n+k)\log n)$ time and $O(n)$ space, where k is the number of pairwise intersections.*

It is quite obvious that kinetic data structures owe a lot to plane sweep methods. In effect, a kinetic data structure is nothing more than a sweep of space/time along the time direction that keeps track of partial structures instead of a full arrangement.

The plane sweep paradigm is a very general way to approach many problems in computational geometry. To give an idea of its generality, the convex hull diagram, the closest pair, and the Voronoi diagram can all be computed in $O(n \log n)$ time using plane sweep techniques [53]. Edelsbrunner and Guibas [45] have replaced the straight sweep line by a curvy line in what they called a “topological sweep” of the plane.

2.1.2 Arrangements of Curves and Surfaces

Definition 2.2. The ***lower envelope*** of a family $(f_i)_i$ of functions from R^d to R is the point-wise minimum:

$$F(X) = \min_i f_i(X)\,.$$

If one is interested in computing the lower envelope, one needs a combinatorial description of it, that is, of which function realizes the minimum at which point. This is called the ***minimization diagram***. For univariate functions, this diagram is simply a sequence of function indices that realize the minimum from left to right. For bivariate functions, this diagram is a planar map made of connected regions (faces, edges, and vertices) for which the minimum is realized by a fixed set of functions. The ***upper envelope*** and ***maximization diagram*** can be defined similarly.

The lower and upper envelopes are natural constructions that appear in many contexts. A point can be associated with a line by geometric duality. The dual of a convex hull diagram is a maximization diagram. The Voronoi diagram of a set of items whose positions are points in d dimensions is the minimization diagram of a set of paraboloids centered at each item in $d+1$ dimensions. The quadratic term is the same for all paraboloids and can be eliminated. Hence, the Voronoi diagram is the minimization diagram of a set of planes in $d+1$ dimensions.

* * *

In two dimensions, nearly exact results are known for the worst-case complexity of the minimization diagram of curves of fixed degree. Even more interestingly, the question has been reduced to a completely combinatorial question, and the results hold for any family of curves that abide by some strictly combinatorial conditions that capture all the possible behavior of algebraic curves.

We say that a sequence of integers $\sigma = (\sigma_1, \dots, \sigma_n)$ is ***non-repeating*** if $\sigma_i \neq \sigma_{i+1}$ for each i.

Definition 2.3 ([96]). Let n, s be two positive integers. A sequence $\sigma = (\sigma_1, \dots, \sigma_m)$ is an (n, s)-***Davenport-Schinzel sequence*** if it is non-repeating and satisfies the following conditions:

1. $1 \leq \sigma_i \leq n$ for each i .

2. Any non-repeating subsequence of σ made of only two integers is of length at most $s+1$.

We denote by $\lambda_s(n)$ the length of the longest (n,s)-Davenport-Schinzel sequence

Tight bounds on $\lambda_s(n)$ involve the Ackermann function. This function is defined by diagonalization. First, we define inductively a family $(A_k)_{k=1}^{\infty}$ of integer functions as:

$$\begin{aligned} A_1(n) &= 2n \\ A_k(n) &= A_{k-1}^{(n)}(1) \quad (k \geq 2) \end{aligned}$$

where $f^{(n)}$ is the n-fold composition. Then we define the Ackermann function A to be

$$A(n) = A_n(n)$$

The inverse of the Ackermann function, denoted $\alpha(n)$, appears in the bounds for $\lambda_s(n)$. It grows extremely slowly (it is less than 4 for practical values of n).

Theorem 2.4 ([96]).

$$\begin{aligned} \lambda_1(n) &= n \\ \lambda_2(n) &= 2n-1 \\ \lambda_3(n) &= \Theta(n\alpha(n)) \\ \lambda_s(n) &\leq n2^{(1+o(1))\alpha(n)^{\frac{s-2}{2}}} && \textit{if } n \textit{ is even,} \\ \lambda_s(n) &\leq n2^{(1+o(1))\alpha(n)^{\frac{s-2}{2}}\log\alpha(n)} && \textit{if } n \textit{ is odd} \end{aligned}$$

If $(f_i)_{i=1}^{n}$ is a family of polynomials of degree at most s, then its minimization diagram, seen as the sequence of indices of the functions that realize the minimum, is a Davenport-Schinzel sequence. Indeed, any two functions of the family intersect at most s times, and therefore, the number of times they can alternate on the upper envelope is at most $s+1$.

The notion of upper envelope and maximization diagram can be generalized. If we replace the maximum by the k-th order statistic in the definition of the upper envelope we obtain the k-level.

Definition 2.5. Given a set Γ of arcs in the plane, the ***level*** of a point (x, y) in Γ is the number of curves of Γ that intersect the relatively open vertical ray $\{(x, v) \mid v > y\}$.

The k-***level*** of an arrangement of arcs is the sequence of vertices that are at level k. Levels in arrangements of lines and curves are a well-studied topic in computational geometry [5, 78]. Although estimating the exact number of vertices at level ℓ has proven difficult, a simple bound on the number of vertices of level at most ℓ can be obtained using standard random sampling techniques [31].

Theorem 2.6 ([94]). *Let Γ be a set of n arcs in the plane such that any two arcs intersect at most δ times. Denote by κ_ℓ the number of vertices that have level exactly ℓ. Then:*

$$\sum_{i \leq \ell} \kappa_i \leq (\ell+1)^2 \lambda_{\delta+2}\left(\left\lfloor \frac{n}{\ell+1} \right\rfloor\right).$$

* * *

We now jump one dimension higher. In this context, many results are known, although most are not as tight as in the case of curves.

Theorem 2.7 ([95]). *The maximization diagram of a family of n algebraic surfaces of fixed degree has complexity $O(n^{2+\epsilon})$ for any $\epsilon > 0$.*

The ***overlay*** of two planar maps is the planar map defined as follows: there is a face f in the overlay if there are two faces in the original maps whose intersection if f. If the original maps have total complexity m and each pair of edges has a constant number of intersections, then the overlay has complexity $\Theta(m^2)$ in the worst case. We just saw that a maximization diagram is roughly of quadratic complexity in the worst case (Theorem 2.7). What about the overlay of two maximization diagrams?

Theorem 2.8 ([8]). *Let F, G be two families of bivariate algebraic functions. The overlay of their maximization diagrams has complexity $O(n^{2+\epsilon})$ for any $\epsilon > 0$.*

This combinatorial result gives a very simple and nearly optimal $O(n^{2+\epsilon})$ divide-and-conquer algorithm for computing the upper envelope of a set of surfaces. Once two maximization diagrams have been computed in the divide step, they can be merged using a plane sweep method very similar to the one described in Section 2.1.1 [8].

Davenport-Schinzel sequences were first introduced by Davenport and Schinzel [34], and rediscovered in computational geometry by Atallah [14]. Many problems can be reduced to combinatorial questions about arrangements [62].

2.1.3 Probabilistic Analysis for Geometric Structures

In discrete algorithms, average case calculations on the performance of algorithms were routinely done until the seventies, in particular for Quicksort [66]. Nowadays, they are unfashionable, and there is a good reason for this. Consider the case of a sorting algorithm. A reasonable algorithm on a reasonable input distribution of n numbers runs in $\Theta(n \log n)$ time. However, there are algorithms that run in this amount of time in the worst case. And finally, the average case can often be transformed into a worst case by use of randomization [76]. Hence, the average case is completely subsumed by other measures.

This is not the case in computational geometry. Consider, for instance, the convex hull diagram of a set of n items with positions in a d-dimensional vector space. In the worst case, its combinatorial complexity can be as much as $\Theta\left(n^{\lfloor \frac{d}{2} \rfloor}\right)$ [74, 92], and take that much time to compute. However, if the positions are independently uniformly distributed in a convex polytope, the expected asymptotic complexity of their convex hull diagram is only $\Theta\left(\log^{d-1} n\right)$ [23]. Moreover it can be computed in linear expected time [22].

It is somewhat surprising to learn that one gets very different answers depending on the input distribution. If the items' positions are drawn independently uniformly at random from a unit d-dimensional ball, the convex hull diagram has expected complexity $\Theta\left(n^{\frac{d-1}{d+1}}\right)$ [42]. Some papers have addressed these questions for more general spherically symmetric distributions [42, 49, 86], some for a uniform distribution in a convex polytope [1, 40]. A good summary can be found in [41] to complement this partial list of references.

Although the Voronoi diagram in d dimensions is the dual of a convex hull in one dimension higher, this doesn't help in the probabilistic model. Dwyer [43] shows that the Voronoi diagram of n items whose positions are independently and uniformly distributed in the unit d-dimensional ball has linear complexity and can be computed in linear expected time. In contrast with the convex hull diagram, this result seems to be very robust across distributions.

In general, one would like to design algorithms that are both optimal in the worst case and in the average case.

For the closest pair in the static case, the description complexity is of no interest, but Efron computes the distribution of the closest distance:

Theorem 2.9 ([49]). *Let S be a set of n independent, identically distributed random variables in d-dimensional Euclidian space, whose common distribution is given by a bounded density function f with respect to the Lebesgue measure. Let $M_S = \min\{\|x-y\| \mid x, y \in S, x \neq y\}$. Then:*

$$\lim_{n\to\infty} (M_S > r) = \exp[-c_f n^2 r^d]$$

where

$$c_f = \frac{\pi^{d/2}}{2\Gamma(\frac{1}{2}d+1)} \int_{R^d} f^2(x)dx$$

is minimized for the uniform distribution.

* * *

Rényi and Sulanke [87] initiated the study of the average complexity of geometric structures. We briefly recall their approach to compute the expected complexity of the convex hull of n items with positions independently and uniformly distributed in the unit square. The idea is to consider a given pair of items p, q, and to compute the probability that this pair forms an edge of the convex hull diagram, i.e., that all other items $r_1, \cdots, r_{n-2}$ are to the left of the oriented line passing through (p, q) and denoted pq.

$$\begin{aligned}
\Pr[(p,q) \text{ on convex hull}] &= \Pr[\forall i, r_i \text{ left of } pq] \\
&= E[\Pr[\forall i, r_i \text{ left of } pq \mid p, q]] \\
&= E\left[\prod_i \Pr[r_i \text{ left of } pq \mid p, q]\right] \\
&= E\left[(1 - A(pq))^{n-2}\right]
\end{aligned}$$

where, for convenience of notation, p denotes the random position of item p, and where $A(pq)$ is the area of the piece of the square to the left of the oriented line pq, which is defined almost surely. The crucial step is from line 2 to line 3: the events $[r_i$ left of $pq]$ are not independent, but they are independent conditionally on p, q.

To compute the distribution of $A(p, q)$, one introduces a map Φ that gives the positions of p and q from the parameters of the line ℓ they define and their abscissae s_p, s_q on this line. The change of variable formula cited below (Theorem 2.10) gives:

$$\Pr[(p,q) \text{ on convex hull}] = \int (1 - A(\ell))^{n-2} |J\Phi(\ell, s_p, s_q)| \, d\ell \, ds_p \, ds_q$$

where $A(\ell)$ is the area of the square to the left of ℓ, $J\Phi$ is the Jacobian of Φ, and the integration is taken over the space of parameters such that p and q fall in the unit square. It is then necessary to carefully bound the Jacobian to finish the computation.

Let us recall the change of variable formula. (See, e.g., Billingsley [26], Theorem 17.2.):

Theorem 2.10. *Let $\Phi : V \mapsto \Phi V$ be a one-to-one mapping of an open set V onto an open set ΦV. Suppose that Φ has continuous partial derivatives ϕ_{ij}. Then*

$$\int_V f(\Phi(x)) |J\Phi(x)| \, dx = \int_{\Phi V} f(y) \, dy$$

where $J\Phi(x)$ is the Jacobian of Φ at x, i.e., the determinant of the matrix of first derivatives $(\phi_{i,j})_{i,j}$.

2.1.4 Combinatorial Aspects of Motion

So far, we have looked at discrete attributes of fixed sets of points in the plane. The closest pair is such an attribute. Let's now consider what happens if the points start moving. As in the introduction, we imagine that we have a set of n *items*, each of which has a position continuously dependent on time. In this setting, although the distance between the closest pair varies continuously, the pair of items that realizes this distance changes only at specific times. Such a change is called an ***event*** for the closest pair.

Let us now assume that each item moves at constant velocity: item i has position $p_i + tv_i$ at time t (where p_i, v_i are fixed two-dimensional vectors). How many events can there be between time 0 and $+\infty$? Can there be infinitely many?

If we plot the distances between each pair of items as a function of time, we obtain a set of $\binom{n}{2}$ curves, and the closest pair at time t is the pair that corresponds to the lowest curve.

A change of the closest pair is then precisely a vertex in the minimization diagram of these curves. Hence, the combinatorial results on arrangements of curves can be directly applied. Here, each pair of curves can intersect at most twice (this is easily seen by taking the squares of the distances instead of the distances), and the minimization diagram therefore has $O(n^2)$ vertices by Theorem 2.4. Moreover, it is easy to construct an example in which this bound is attained.

This type of questions was first examined by Atallah [14], who drew the connection with combinatorial questions about arrangements of curves, and introduced Davenport-Schinzel sequences to the computational geometry community. We use here the vocabulary that was defined in the introduction.

Theorem 2.11 ([14]). *In the plane, the closest pair changes $O(n\lambda_{2\delta}(n))$ times in a (δ, n)-scenario. There is a $(1, n)$-scenario in which the closest pair changes $\Omega(n^2)$ times.*

The same question can be asked for any discrete attribute. For instance, how many events can there be for the convex hull diagram of n items in δ-motion? Once again, a Davenport-Schinzel argument can be applied.

Theorem 2.12 ([6, 14]). *In the plane, the convex hull diagram changes $O(n\lambda_{2\delta}(n))$ times in a (δ, n)-scenario. There is a $(1, n)$-scenario in which it changes $\Omega(n^2)$ times.*

The Delaunay triangulation contains the convex hull as a substructure, and hence the same lower bound applies, but the best known upper bound is roughly cubic.

For the purpose of this thesis, it is important to know the above bounds. When we construct a kinetic data structure that keeps track of a given fixed attribute, we will be satisfied if the worst-case number of events for this structure is not too much larger than the worst-case number of events for the attribute it maintains. Our methods of analysis are also often similar to those that were used to prove the theorems in this section: we identify each event with a vertex in a certain arrangement in space/time, and use theorems like those of Section 2.1.2 to bound the number of such vertices.

2.2 Related Work on Motion

Since the pioneering work of Atallah [13], many results have been obtained on the combinatorial number of changes to attributes based on moving data. This type of study is typically called “dynamic computational geometry”.

Voronoi diagrams of moving points have raised considerable interest in the past few years. Several papers simultaneously gave bounds better by one order of magnitude than the naïve bounds [54, 58, 69, 88]. Similar bounds have also been obtained in higher dimensions [90, 11]. The Voronoi diagram can be defined for any metric, and Chew [30] gave roughly quadratic bounds for the number of changes of the L_1-Voronoi diagram in a (δ, n)-scenario.

Edelsbrunner and Welzl [48] ask for a bound on the number of changes to the k-th order statistic of a set of items with changing one-dimensional position. This is exactly the k-level of the curves described by the items in space/time. Recently, Dey [37] showed that the k-level of a set of lines has complexity at most $O(nk^{1/3})$.

If we are given a fixed graph with edge weights varying continuously with time, we can ask how many times the structure of the minimum spanning tree changes. Gusfield [61], in a paper that predates the work of Atallah, shows that if the edge weights are changing at constant speed, the minimum spanning tree changes $O(m\sqrt{n})$ for a graph with n vertices and m edges, but these bounds are not tight. This question is intimately related to the k-level, and Dey's bound applies here also: The new bound is $O(mn^{1/3})$. If items are moving in the plane, the minimum spanning tree can be defined with respect to any metric. Katoh, Tokuyama, and Imano [72] showed that in a $(1, n)$-scenario (items moving at constant velocity), the L_1-minimum spanning tree changes at most $O(n^{5/2}\alpha(n))$. In the L_2 metric, the bound is only $O(n^2\lambda_4(n))$. The maximum spanning tree, on the other hand, changes $O(n^2)$ times, and this is tight.

* * *

Atallah [14] also considers some algorithmic problems associated with motion, but his solutions are essentially *off-line*. For instance, he shows how to compute all the changes to the convex hull diagram of a set of items for which the motion is known in advance. It is the distinction between *off-line* and *on-line* that characterizes best the difference between these questions and kinetic data structures.

Ottmann and Wood [80], in a paper that bears a name very similar to that of Atallah, also consider some algorithmic problems involving motion. They ask how to compute the first collision time in a set of moving items. They provide a plane-sweep solution for one-dimensional data (this amounts to finding the leftmost intersection of a set of lines).

Edelsbrunner and Welzl [48] motivate their interest in the k-level by asking how to compute the k-th order statistic of points moving along a line. Their method to compute the k-level is the first non-trivial on-line method for maintaining a discrete attribute of moving data, although it works only for constant velocity motion. It is generalized to arbitrary motion in [19].

Roos [88] shows how to keep track of the Voronoi diagram of moving points. The scheme is exactly the same as that of a kinetic data structure, except that, in the case of the Voronoi diagram, there are no issues of design as the structure is "self-certifying" (except that the structure is not local). Devillers and Golin [36] make the observation that the previous algorithm can also be used to maintain a point location structure on the changing Voronoi diagram, using the dynamic point location structure of Goodrich and Tamassia [56].

* * *

Kahan [70] proposes a computational model for continuously moving data, with a radically different assumption from ours. In many situations, he argues, only an upper bound on the speed of the objects is known, and it is necessary to perform some costly operation (like reading off a radar) to know the precise position of an object. He imagines that a user is going to perform a set of queries at different times, and that an algorithm, in his model, will have to request the exact position of a number of items in order to answer the query exactly. The performance of an algorithm is measured by comparing the number of position requests to the minimum number required to answer a query sequence. Hence, this measure is akin to the competitive ratio used in the evaluation of on-line algorithms. In this model, Kahan exhibits an algorithm for answering maximum queries that needs to perform only one more position request than the minimum required to certify that the queries are correct. No result is known for attributes based on data in dimension higher than one.

* * *

A number of works have focused on using elaborate range searching data structures to find the first collision time between a set of objects in the plane or in space, moving with known motion laws. The goal here is to beat the trivial quadratic bounds obtained by computing the collision time of every pair of objects or features.

Here are some results obtained with this type of approach: Gupta, Janardan and Smid [60] find the first collision time of n points each moving at constant velocity in the plane, with a computational cost of $O\big(n^{5/3}(\log n)^{6/5}\big)$. Schömer and Thiel [91] answer the same question for two polyhedra translating in space with a computational cost of $O(n^{8/5+\epsilon})$. Erickson and Eppstein [50] consider repeated collision detection amongst a set of unit disks each moving at constant velocity. They create a dynamic data structure such that, after a collision, they can update the structure in order to find the next collision. The construction of the data structure on n disks takes $O(n^{49/29})$, and provides $O(n^{20/29})$ time for query and update.

All these solutions are interesting from a theoretical point of view, but are unlikely to be implementable. Moreover, they are restricted to simple classes of motion. They may, however, provide some useful guidelines for the design of practical heuristics in implementations.

* * *

The idea of kinetic data structures is likely to find applications for collision detection in dynamic simulation systems. There are a number of results in the computational geometry community about the static problem of deciding whether there are collisions amongst a set of objects: for an arbitrary number of spheres in three dimensions [67], for two convex polyhedra with preprocessing [38], or for simple polygons with preprocessing [77].

In the context of objects in motion, collision detection tests can be performed at every time step, but the *temporal coherence* between each step means that it is wasteful to restart each test from scratch. Many systems take advantage of this coherence by using computations done at the previous step to speed up the computation at the new step. For instance, Lin and Canny [73], in their algorithm that finds the closest pair of features between two convex polyhedra, initialize the initial guess to the closest pair found at the previous time step. The number of papers that deal with collision detection is too large to review here, but the reader is referred to Hubbard [68] for a good discussion of the issues involved in the possible uses of temporal coherence, and to Hayward et al. [63] for a recent literature review of existing approaches.

Chapter 3

Maximum Maintenance

In this chapter, we focus on the simplest attribute of a set of real numbers: the maximum. The maximum is a sub-attribute of the sorted list, so that a kinetic sorted list is a possible way to maintain the maximum. However, this solution has a drawback: as we saw in Theorem 2.4, the maximum of a set of n items can change at most $\lambda_\delta(n)$ times in a (δ, n)-scenario, a quantity that is nearly linear in n. The kinetic sorted list, on the other hand, needs to process quadratically many events in the worst case. The purpose of this chapter is to seek better solutions, in terms of efficiency, to the problem of maximum maintenance. A kinetic data structure that maintains the maximum will be called a ***kinetic priority queue***.

There are numerous algorithms to compute the maximum, and many data structures to maintain it upon insertion and deletion. All of them can be considered for kinetization. For instance, we can compute the maximum by organizing a *tournament tree*: the items are divided into two sets of roughly equal size, the maximum of each subset is recursively computed, and the two maxima are compared to get the winner. A classical data structure that maintains the maximum is a *binary heap*: a binary tree in which each node stores a distinct item, such that the item stored at a node is greater than any item stored in its subtree. The proof scheme induced by a tournament contains one certificate per internal node; it is of linear size and has $O(\log n)$ locality. The proof scheme induced by a binary heap contains one certificate per edge; it is also of linear size and has $O(1)$ locality. Figure 3.1 shows examples of both proof schemes. In the remainder of this chapter, we present different update rules for these proof schemes, and compute efficiency bounds for each of them.

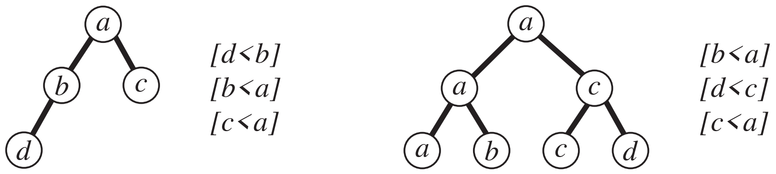


Figure 3.1: Two ways to certify the maximum item in a configuration in which $d < c < b < a$. The left structure is a *binary heap*. There is one certificate per edge, comparing the parent and the child. The right structure is a *tournament*. There is one certificate per internal node, comparing the two children.

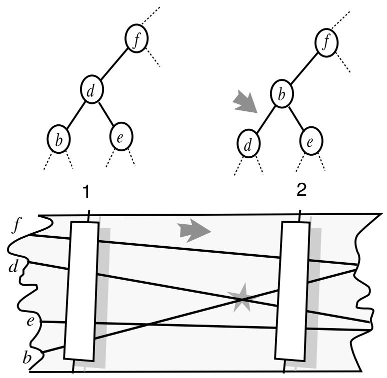


Figure 3.2: At time $t = 1$ items b and e are not in the same subtree, hence their intersection is not scheduled as an event. The first intersection to be scheduled is bd. At this point, the heap property is maintained by a swap between b and d. The events corresponding to the four edges around this pair are descheduled before the swap and rescheduled with the changed items after the swap.

3.1 Kinetic Swapping Heap

We consider here a binary heap. The simplest way to restore the heap property after the failure of an edge certificate is to swap the parent and child of the failing edge (Figure 3.2). We call this a ***kinetic swapping heap***. The responsiveness of this update rule is $O(\log n)$: A swap changes at most four certificates (the edge above the parent and the two children of the child), and each of these changes takes $O(\log n)$ to deschedule and reschedule in the event queue.

One particularity of this method—compared to all other kinetic data structures presented in this thesis—is its *path dependence*: the structure of the binary heap at a given time depends not only on the values of the keys at that time, but also on the specific history

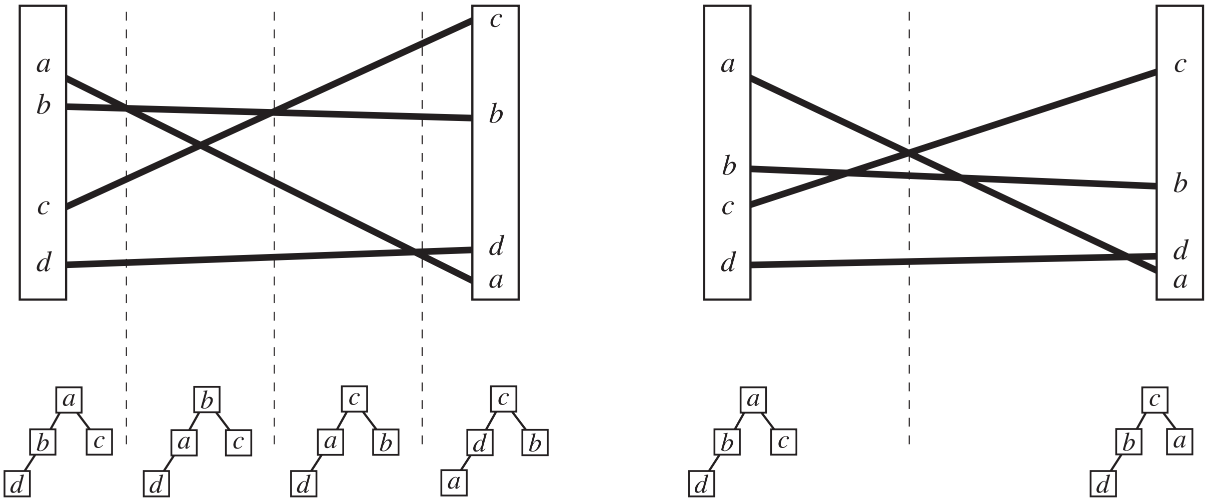


Figure 3.3: Two scenarios in which we start and end with the same configuration, and start with the same heap. The structure of the heap at the end depends upon the full scenario and not only upon the end configuration.

of intersections before that time (Figure 3.3). This particularity may be the reason why the kinetic swapping heap is extremely difficult to analyze. As a matter of fact, we obtained only partial and unsatisfactory results on the efficiency of this method. We show that for items moving at constant velocity, the efficiency is $O\big(n \log^3 n\big)$, one log short of the lower bound. In a $(1, n, m)$-dynamic scenario the best bound we obtain is $O\Big(m\sqrt{n} \log^{\frac{3}{2}} n\Big)$. No result is known for non-linear motion.

* * *

We now consider a kinetic swapping heap in a $(1, n)$-scenario starting at time 0. The tree structure underlying the heap remains fixed during the sweep, even though the node contents change over time. The ***level*** of an item a at time t, denoted by $\ell_t(a)$, is the distance from the node containing a at t to the bottom level. As the tree structure of the heap is balanced, the level of an item is an integer between 0 and $\lceil \lg n \rceil$. We denote by $\hat{\ell}_t(a)$ the highest level ever attained in the heap by item a between time 0 and t.

Lemma 3.1. *Let K be a kinetic swapping heap in a $(1, n)$-scenario. If $[a > b]$ is a certificate in $K(t)$ that fails after time t, then $\hat{\ell}_t(a) \geq \hat{\ell}_t(b) + 1$.*

Proof. Consider the last time τ^- before t when b is at its maximum level $\hat{\ell}_t(b)$. At time

τ^-, item b is in some node ν of the heap and from then on, item b only moves within the subtree U rooted at ν.

If b is in ν at t, then a is in the parent of ν at that time, and the claim trivially holds.

Otherwise, as $[a > b]$ fails at a time later than t and can only fail once (because items have constant velocities), we have $b(\tau^-) < a(\tau^-)$, and item a is not in U at τ. Therefore, the path followed by a in the heap from τ^- to t enters U at some time before t, and thus passes through the parent of ν. The conclusion follows. □

Theorem 3.2. *A kinetic swapping heap has $O(n \log^3 n)$ efficiency in a $(1, n)$-scenario.*

Proof. We use a potential function argument. For an item a in the heap, define its potential $\phi_t(a)$ at time t to be

$$\phi_t(a) = \hat{\ell}_t(a)(\ell_t(a) - \hat{\ell}_t(a)) \, .$$

The potential of the entire heap is the sum of the potentials of all the items. Note that this potential is 0 at time 0 (for every a, $\ell_0(a) = \hat{\ell}_0(a)$); it is non-positive for all t (because $\ell_t(a) \leq \hat{\ell}_t(a)$). At the end of the scenario, its absolute value is at most $O(n \log^2 n)$. We now show that an event decreases the total potential by at least 1.

Consider a swap at time t between a parent a and a child b in the heap. The potential of any item other than a or b doesn't change. Also, the quantity $\hat{\ell}_{t^-}(a)$ doesn't change, so that the potential change for a is $\phi_{t^+}(a) - \phi_{t^-}(a) = -\hat{\ell}_{t^-}(a)$.

There are two cases for b: either it reaches a new highest level at t^+ (i.e. $\ell_{t^-}(b) = \hat{\ell}_{t^-}(b)$), or it does not. In the first case, $\hat{\ell}_{t^+}(b) = \ell_{t^+}(b)$, so $\phi_{t^-}(b) = \phi_{t^+}(b) = 0$, and b's potential doesn't change. The decrease in potential is therefore $\hat{\ell}_{t^-}(a)$, which is at least 1 because a is not on the bottom level before the swap.

In the second case, the potential of b increases by $\hat{\ell}_{t^-}(b)$, and the net potential change is $\hat{\ell}_{t^-}(b) - \hat{\ell}_{t^-}(a)$, which corresponds to a decrease of at least 1 by Lemma 3.1.

Hence, the number of events is $O(n \log^2 n)$, the absolute value of the potential at the end of the scenario. An additional log comes from the insertions and deletions in the event queue. □

* * *

Alas, this analysis completely breaks down in the case of line segments, as Lemma 3.1 no longer holds. Instead, we use another potential argument reminiscent of the one used independently by several authors [47, 51, 61] for proving upper bounds on the k-level of an arrangement of lines.

Lemma 3.3. *The kinetic swapping heap has* $O\left(m\sqrt{n}\log^{\frac{3}{2}} n\right)$ *efficiency in a* $(1, n, m)$*-dynamic scenario.*

Proof. Let us first assume that we have only n insertions/deletions, but that we may start with a non-empty heap (with at most n items), and end with a non-empty heap. Let us order these (at most $2n$) items by increasing (signed) velocity, and denote by $r(a)$ the rank of an item a in this ordering. Let $H(\nu)$ be the subtree rooted at a node ν, and let $r_t(\nu)$ be the rank of the segment in node ν at time t.

For a node ν in the heap, define its potential $\phi_t(\nu)$ at time t to be the sum of the ranks in its subtree:

$$\phi_t(\nu) = \sum_{\mu \in H(\nu)} r_t(\mu)\,.$$

The potential of the entire heap is the sum of the potentials of all the nodes of the heap. As each item's rank is counted in at most $\log n$ potentials, the initial potential is at most $2n^2 \log n$.

If two nodes μ, ν are parent and child, a swap of their contents at t changes the potential by $r_{t^-}(\mu) - r_{t^-}(\nu)$, which is at most n. In a binary heap, an insertion is implemented by the addition of a leaf at the bottom of the heap, followed by $O(\log n)$ swaps. The insertion at the bottom of the heap also increases the potential by $O(n \log n)$ and each swap increases the potential by at most $O(n)$. Hence, an insertion increases the potential by at most $2n \log n$, and the same bound holds for deletions. The total potential *increase* due to insertion/deletions is $O\left(n^2 \log n\right)$.

Let R be the set of swap events. At an event, the potential can only decrease, as it is the higher ranked item (the one with greater velocity) that goes up one level in the heap. Hence, if we denote by $\delta(e)$ the absolute difference in rank between the two items that exchange position ordering at an event denoted by e, we have:

$$\sum_{e \in R} \delta(e) = O\big(n^2 \log n\big) \ .$$

Therefore, for any fixed B, the number of events $e \in R$ with $\delta(e) \geq B$ is $O\Big(\frac{n^2 \log n}{B}\Big)$. Moreover, each item has at most B items that differ in rank by less than B, so there are at most nB events with $\delta(e) \leq B$. Choosing appropriately $B = \sqrt{n \log n}$, the number of events is $O\big(n\sqrt{n \log n}\big)$.

If we have m insertions/deletions, we batch them as m/n groups of n, to which we apply the argument above. The total number of events is therefore $O\big(m\sqrt{n \log n}\big)$, and an extra log comes from the cost of scheduling and descheduling events in the event queue. □

Both arguments extend to motions whose trajectories in time/space create an arrangement of pseudo-lines or pseudo-segments (see [2]).

3.2 Kinetic Heater

If each item in a set is given two numbers, a *key* and a *priority*, there is a unique binary tree that is both a search tree on the keys and a heap on the priorities (the tree is not necessarily balanced). Such a tree is well-known and called a *treap*—Aragon and Seidel [12] used it to create their popular randomized search tree data structure, which is a treap on items with a given key and a randomly assigned priority. The randomization guarantees that this structure is balanced with high probability.

Let's turn things around, and define a ***heater***: a heater is like a treap, but this time, priorities are given and keys are random. When an element with a given priority is inserted in a heater, it is first assigned a random key, and inserted at the appropriate leaf of the heater. It then bubbles up with a sequence of rotations until it reaches a position consistent with its priority. Deletions are implemented in an analogous way. The structure of a heater is the same as that of a binary heap, although it might not be balanced. It is certified by one certificate per edge.

In a scenario in which the priorities change continuously, the heater needs to be updated from time to time and becomes a ***kinetic heater***. This is where it differs from the kinetic

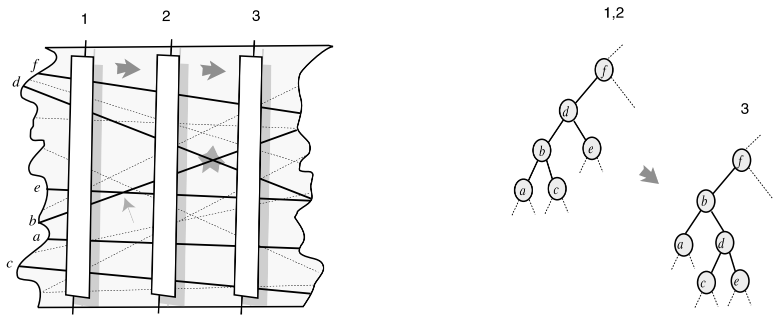


Figure 3.4: Three times and the corresponding heaters with attention focused on items with random keys a through f (which we don't distinguish from the items' names). Between time 1 and time 2, the heater remains unchanged although there is a change in the order of the priorities of b and e. Between time 2 and time 3, a rotation is performed that swaps the priorities of b and d but otherwise preserves the in-order sequence of the items' keys.

swapping heap. When the priorities of a parent and a child become equal, we do not swap their contents. Instead, we perform a rotation along the edge that links them. This operation preserves the key order and adjusts the heater to the new priorities. After the rotation, two parent-child relationships change in the heater. Thus up to two existing events in the event queue may have to be descheduled, and two new events scheduled. See Figure 3.4. The responsiveness of a kinetic heater is $O(\log n)$, as is the case for the kinetic swapping heap.

We consider a heater H in a (δ, n, m)-dynamic scenario for a set S of m items; each item is assigned a random key. We recall that the heater structure is at the same time a heap on the position of each item, and a search tree on random keys assigned to each item. Expectations are taken over a uniform distribution of all $m!$ ordering of the keys. The good behavior of a heater relies on two facts that we proceed to prove: (i) its depth at time t is logarithmic in expectation, which allows efficient insertion/deletion of elements (Lemma 3.4), and (ii) there are not too many events (Lemma 3.7).

Lemma 3.4. *The expected depth of a kinetic heater on n items at a given time is $O(\log n)$.*

Proof. The random keys define a bijection between the ordering of the priorities and the ordering of the keys. This bijection is drawn uniformly at random from the space of all $n!$ bijections. Thus, the structure of the tree is the same as if the keys were given and the priorities were random. Seen this way, the result is an immediate consequence of the original analysis of treaps by Aragon and Seidel [12]. ☐

Corollary 3.5. *The expected cost of all insertions and deletions in a kinetic heater in a (δ, n, m)-dynamic scenario is $O(m \log^2 n)$.*

Proof. By the lemma above, an insertion of an element in the heater causes an expected $O(\log n)$ rotations. Each rotation disturbs $O(1)$ parental relationships, and requires $O(1)$ updates of the event queue. The expected time of an insertion/deletion is thus $O(\log^2 n)$. □

The rest of this section focuses on bounding the expected number of internal heater updates. It makes crucial use of the randomness of the structure. The analysis proceeds as follows. Given a dynamic scenario on a set S of items, we construct an arrangement of arcs Γ in two-dimensional space-time, by associating to each item its trajectory. An event in the kinetic heater corresponds to a vertex in Γ, but not all vertices are events. We then observe that the probability that a given intersection causes an update in the heater is related only to its *level* (Definition 2.5). Finally, we perform a standard computation *à la* Clarkson-Shor [31], based on the combinatorial result bounding the complexity of the upper envelope of a set of curves (Section 2.1.2).

Lemma 3.6. *Let Γ be the arrangement of arcs associated with a dynamic scenario, and let v be a vertex in Γ at level ℓ. The probability that v corresponds to an event for a kinetic heater is exactly $\frac{2}{\ell+2}$.*

Proof. Let a, b, t denote the two items and time that correspond to vertex v, that is, we have $b(t) = a(t)$. Without loss of generality, let's assume that $b(t^-) < a(t^-)$. We consider the set S' made of a, b, and the ℓ items whose priorities are greater than that of a and b at t. The set S' forms a contiguous group at the top of the heater, as all other items have a lower priority at that time. We restrict our attention to the pruned tree on S', as well as to the induced random ranking χ amongst the keys of those items.

The certificate $[b < a]$ exists at time t^- if and only if a is the parent of b. Item b has lowest priority in S', so it is a leaf of the subtree. Moreover, a has lowest priority amongst the remaining items, so it is either a leaf of the subtree, or the parent of b. In the latter case, a and b are contiguous in χ. In the former case, their common ancestor has an intermediate key, so that a and b are not contiguous in χ. Hence, the certificate exists at time t^- if and only if $|\chi(b) - \chi(a)| = 1$

Now, the permutation χ is a uniform random variable on all permutations of $\ell + 2$ elements. Hence:

$$\begin{aligned} P[|\chi(b) - \chi(a)| = 1] &= P[\chi(b) = \chi(a) + 1] + P[\chi(b) = \chi(a) - 1] \\ &= 2\left(1 - \frac{1}{\ell+2}\right)\frac{1}{\ell+1} \end{aligned}$$

Thus the probability that a, b are contiguous is $2/(\ell + 2)$. $\square$

Lemma 3.7. *The kinetic heater has expected efficiency $O\big(\lambda_{\delta+2}(m)\log^2 n\big)$ in a (δ, n, m)-dynamic scenario.*

Proof. Let us denote by κ_ℓ the number of vertices at level ℓ in the arrangement of arcs Γ induced by the scenario. By linearity of expectation and Lemma 3.6, the expected number of events is:

$$\sum_{\ell=0}^{n-2} \kappa_\ell \frac{2}{\ell+2} .$$

Using summation by parts, we replace κ_ℓ in this expression by its partial sum $K_\ell = \sum_{i \le \ell} \kappa_i$. The bound of Theorem 2.6 implies the (very slightly) weaker bound:

$$K_\ell = O((\ell+1)\lambda_{\delta+2}(m)) .$$

A standard calculation gives:

$$\begin{aligned} \sum_{\ell=0}^{n-2} \kappa_\ell \frac{2}{\ell+2} &= 2\frac{K_\ell}{\ell+2}\bigg|_0^{n-1} - 2\sum_{\ell=0}^{n-2} K_{\ell+1}\frac{-1}{(\ell+2)(\ell+3)} \\ &\le O(\lambda_{\delta+2}(m)) + 2\sum_{\ell=0}^{n-2}\frac{\lambda_{\delta+2}(m)}{(\ell+3)} \\ &= O(\lambda_{\delta+2}(m)\log n) . \end{aligned}$$

A certificate failure is implemented by a single rotation, which disturbs the parental relationship of $O(1)$ nodes, and each of those takes $O(\log n)$ time for scheduling the associated failure time. The total expected processing cost is $O\big(\lambda_{\delta+2}(m)\log^2 n\big)$. By Corollary 3.5, the additional expected cost of the $2m$ insertions/deletions is $O\big(m\log^2 n\big)$. $\square$

3.3 Kinetic Tournament

We now turn to a deterministic structure that realizes the same efficiency bounds (with a much simpler proof). It is based on the kinetization of a ***tournament tree*** (Figure 3.1). A tournament tree on a set of n values is a balanced tree. Each leaf is one of the values or a special symbol $-\infty$ (defined to be less than any other value), and each node is filled from the bottom up with the higher value of the two children. The certificates are not the same as those used to certify a heap: for each internal node, a comparison between the two children certifies which one is the winner.

When an item becomes invisible, we replace it with $-\infty$. When an item becomes visible, we put it in the first available leaf containing $-\infty$. If no such leaf is available, we extend the tournament tree by adding a leaf and up to $\lceil \log n \rceil$ internal nodes to connect this leaf with the rest of the tree.

When an event happens at an internal node of the tree, the winner at that node changes, and the new winner has to be percolated up the tree. Similar percolations need to happen when an item becomes visible or invisible. Thus, unlike in the heater case, tournament events can cause a number of deschedulings and reschedulings proportional to the height of the tournament tree. The responsiveness of this structure is $O\big(\log^2 n\big)$.

Lemma 3.8. *The kinetic tournament has $O\big(\lambda_{\delta+2}(m) \log^2 n\big)$ efficiency in a (δ, n, m)-dynamic scenario.*

Proof. When we process an event, the number of certificates we have to schedule and deschedule in the event queue is proportional to the number of nodes whose contents change during the processing.

Consider a node ν in the tournament tree and denote by n_ν the number of items that ever appear in the subtree rooted at ν. Its content—the maximum item in the subtree—changes at most $\lambda_{\delta+2}(n_\nu)$ times.

If L_i denotes the set of nodes at level i, we have

$$\sum_{\nu \in L_i} n_\nu \le m\,.$$

Therefore, the total number of changes at level i is at most

$$\sum_{\nu \in L_i} \lambda_{\delta+2}(n_\nu) \leq \lambda_{\delta+2}(m)\,.$$

Summing over the whole tournament tree of depth $\log n$, the total number of changes is $O(\lambda_{\delta+2}(m)\log n)$. Hence, the efficiency is $O\big(\lambda_{\delta+2}(m)\log^2 n\big)$ □

Thus, although the kinetic tournament is not as responsive as our two other kinetic priority queues, it is as efficient as a kinetic heater, and is deterministic. The only possible weakness of this structure is that its locality is not optimal.

3.4 Conclusion

We presented several solutions for the kinetic priority queue. Although the kinetic tournament is close to optimal, it doesn't achieve optimal locality or responsiveness. It would be nice to obtain an update rule for a deterministic binary heap, with optimal locality, which would allow us to prove strong bounds similar to those obtained for the kinetic tournament. On the other hand, the kinetic tournament seems more natural to implement on a parallel architecture.

The kinetic priority queue is useful to perform a plane sweep in a case in which it is not necessary to keep track of the exact structure of the sweep line. For instance, we used it to obtain an output-sensitive algorithm that reports all red-blue intersections between two sets of connected line segments, and another one that computes the k-level of an arrangement of curves [19].

Chapter 4

Two-Dimensional Problems

In this chapter, we design kinetic data structures for two fundamental two-dimensional problems: the convex hull and the closest pair. These two problems give representative examples of the kinetization process. The structure for the convex hull (Section 4.1) is based on a classic divide-and-conquer algorithm, but the proof of efficiency calls upon some deep theorems of combinatorial geometry. For the closest pair (Section 4.2), we need to develop a new static algorithm, as existing algorithms do not kinetize well.

4.1 Convex hull

As we have seen in the introduction, it is possible to obtain a proof scheme for an attribute from a static algorithm that computes this attribute. One way to compute the convex hull is to use a divide and conquer strategy: we arbitrarily divide our set of items into two subsets, recursively compute the convex hulls of each of them, and merge the result. This merging step is accomplished by walking around the two hulls with parallel tangents, a step we will describe in the dual setting.

We focus here on computing the upper convex hull, and dualize a point (p, q) to the line $y = px + q$. In the dual, the goal is to maintain the upper envelope of a family of lines whose parameters change in a continuous, predictable fashion. We will perform the kinetization of a divide-and-conquer algorithm: we divide the set of n items into two subsets of roughly equal size, compute their upper envelopes recursively, and then merge the two envelopes. To focus on the merge step, we first study how to maintain the upper envelope of two convex piecewise linear univariate functions.

4.1.1 Proof Scheme for the Upper Envelope of Two Chains

We represent a convex piecewise linear function by a doubly linked list of edges and vertices ordered from left to right, and we call this representation a *chain*. In this section, we consider two chains—a red and a blue—and present a KDS to maintain the purple chain that represents the upper envelope of the two input chains.

As the supporting lines are the primary objects in our problem, we denote by a lowercase letter an edge or its supporting line, and by ab the vertex between edges a and b. A configuration is a placement of all the edges/vertices that makes both chains convex. For a vertex ab, the edge from the other chain that is above or below ab is called the *contender edge* of ab and denoted $\texttt{ce}(ab)$; we add to each vertex a pointer to its contender edge. We denote by $\chi(\cdots)$ the color (red or blue) of an input vertex or edge. Finally, we denote by $ab.\texttt{prev}$ (resp. $ab.\texttt{next}$) the red or blue vertex closest to ab on its left (resp. right). This is easily found by comparing the x-coordinate of the neighbor vertex in the chain to which ab belongs with that of one of the endpoints of the contender edge of ab.

The comparisons done by a standard sweep for merging the red and blue chains lead to certificates of two types: x-certificates proving the horizontal ordering of vertices, denoted by $<_x$, and y-certificates proving the vertical position of a vertex with respect its contender edge, denoted by $<_y$. Unfortunately, if we were to keep all these comparisons as certificates, the kinetic data structure thus obtained would not be local, as a given edge could be the contender of linearly many vertices from the other envelope. We thus build an alternative list of certificates that also involves comparisons between line slopes, denoted by $<_s$.

Table 4.1 gives this modified list of certificates (See also Figure 4.1). The first column contains the name of a certificate, the second column contains the comparison that this certificate guarantees, and the third column contains additional conditions for this certificate to be present in the KDS. For instance, the first line in the table says that there is a certificate called $\texttt{x}[ab, cd]$ in the KDS only when ab and cd are neighbors and are of different colors (the *condition*). In this case, the *comparison* certifies the local x-ordering. The equation associated with this comparison has to be solved for t in order to find the first time at which the certificate fails.

Name	Comparison	Condition(s)
$\mathtt{x}[ab]$	$[ab <_x cd]$	$cd = ab.\mathtt{next}$ $\chi(ab) \neq \chi(cd)$
$\mathtt{yli}[ab]$	$[ab <_y \text{ or } >_y \mathtt{ce}(ab)]$	$b \cap \mathtt{ce}(ab) \neq \emptyset$
$\mathtt{yri}[ab]$	$[ab <_y \text{ or } >_y \mathtt{ce}(ab)]$	$a \cap \mathtt{ce}(ab) \neq \emptyset$
$\mathtt{yt}[ab]$	$[\mathtt{ce}(ab) <_y ab]$	$a <_s \mathtt{ce}(ab) <_s b$ $\mathtt{ce}(ab) <_y ab$
$\mathtt{slt}[ab]$	$[a <_s \mathtt{ce}(ab)]$	
$\mathtt{srt}[ab]$	$[\mathtt{ce}(ab) <_s b]$	
$\mathtt{sl}[ab]$	$[b <_s \mathtt{ce}(ab)]$	$b <_s \mathtt{ce}(ab)$ $ab <_y \mathtt{ce}(ab)$ $\chi(ab) \neq \chi(ab.\mathtt{next})$
$\mathtt{sr}[ab]$	$[\mathtt{ce}(ab) <_s a]$	$\mathtt{ce}(ab) <_s a$ $ab <_y \mathtt{ce}(ab)$ $\chi(ab) \neq \chi(ab.\mathtt{prev})$

Table 4.1: Certificates for the upper envelope of two convex chains

The certificates have the following meaning: (1) The exact x-ordering of vertices is recorded with $\mathtt{x}[\cdots]$ certificates. (2) Each intersection is surrounded by $\mathtt{yli}[\cdots]$ and $\mathtt{yri}[\cdots]$ certificates ("y left/right intersection"). (3) If an edge is not part of the upper envelope, the certificates place its slope in the sequence of slopes of the edges covering it: either three "tangent" certificates ($\mathtt{yt}[\cdots], \mathtt{slt}[\cdots], \mathtt{srt}[\cdots]$), or one certificate proving there is no tangent ($\mathtt{sl}[\cdots]$ or its symmetric $\mathtt{sr}[\cdots]$). Illustrations of the certificates appear in Figure 4.1. To be complete, we need to add slope certificates between the two leftmost edges and between the two rightmost edges.

Lemma 4.1. *The locality of the proof scheme above is $O(1)$.*

Proof. Consider a given edge a. It can be involved in a certificate because one of its endpoints is. But each vertex can be involved in only two $\mathtt{x}[\cdots]$ certificates, and one of every other type. We therefore only need to consider the number of certificates a is involved in as a contender edge.

If a is cut by the other convex chain, it can be involved in $\mathtt{yli}[\cdots]$ and $\mathtt{yri}[\cdots]$ certificates as the contender edge, and there can be at most two such occurrences as a is intersected at most twice by the other convex chain.

Next, if a doesn't intersect with the other convex chain, it can be involved in at most

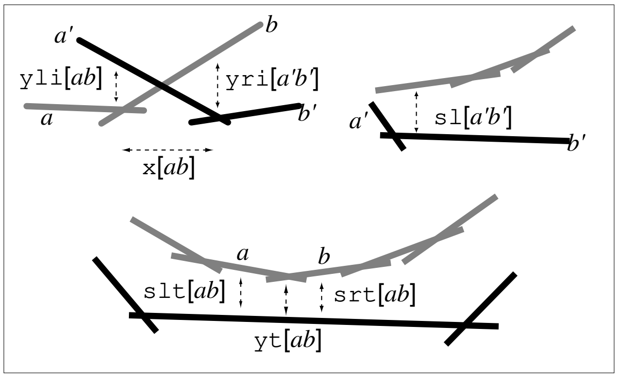


Figure 4.1: Depending on the relative positions of the red and blue convex chains, different certificates are used to certify the intersection structure (top left case) or the absence of intersection (top right and bottom cases). Arrows point to the elements being compared (vertices or edges).

two triplets of tangent certificates if it is above the other convex chain, and one if it is below.

Finally, suppose that a is the contender edge of many vertices. These vertices are all of the same color. Thus, only the leftmost and rightmost can involve a as the contender edge in a $\texttt{sl}[\cdots]$ or $\texttt{sr}[\cdots]$ certificates. Hence the locality of the proof scheme is $O(1)$. $\square$

Lemma 4.2. *The set of certificates described above strongly certifies the upper envelope.*

Proof. Let π be a configuration with certificate set $\mathcal{L}$, and π' be another configuration in which these certificates are valid. We show that the upper envelope of π' has the same vertices as the upper envelope of π.

First, the x-certificates prove the correctness of the contender edge pointers. Any vertex that has a y-certificate in $\mathcal{L}$ is also guaranteed to be placed in π' as in π. It remains to show that those vertices without a y-certificate cannot be placed differently in π and π'.

We consider a maximal contiguous sequence S of vertices without y-certificates in $\mathcal{L}$, and we assume without loss of generality that the red function is above the blue function in this stretch in π. Let $\Delta(x)$ be the difference in π between the red slope and the blue slope at x. This function increases at red vertices and decreases at blue vertices. It doesn't change sign at a red vertex of S, as this would imply a $\texttt{yt}[\cdots]$ certificate in $\mathcal{L}$.

Hence, on the interval defined by S, $\Delta(x)$ is positive until a certain blue vertex, and

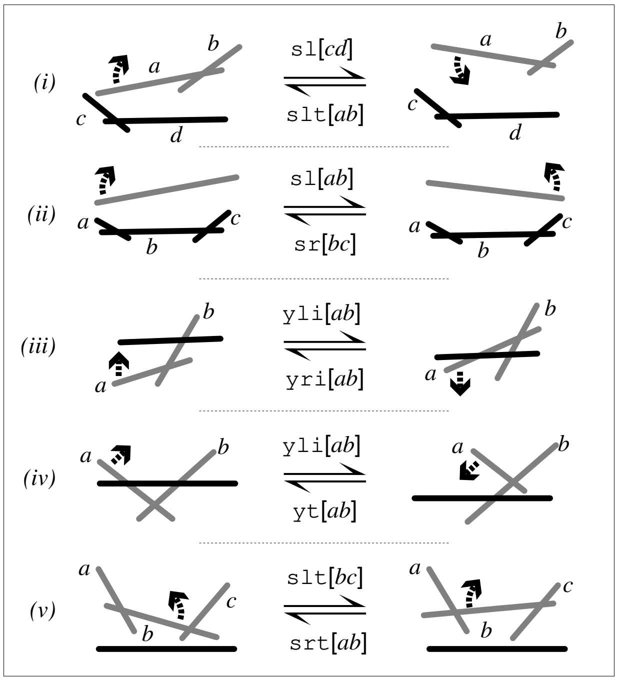


Figure 4.2: A partial list of events. The certificate that changes sign is indicated for each transition. There are three additional cases not shown: *(i)* and *(iii)* in mirror image, and the event corresponding to the x-certificate.

then is negative. If the contender edge of this vertex is a, there will be $\texttt{sl}[\cdots]$ certificates to the left of the left endpoint of a and $\texttt{sr}[\cdots]$ certificates to the right of the right endpoint of a. These certificates guarantee that the red function remains above the blue function on the interval defined by S. $\square$

4.1.2 Maintenance

Lemma 4.2 shows that the certificate list described above is sufficient to maintain the upper envelope. As in the case of any kinetic data structure, all these certificates are placed in a global event queue, where each certificate is stamped with its failure time. When it is time to process the first event in the queue, we need to update the certificate list. Below is the list of changes that need to be performed for each type of event. A pictorial description is shown in Figure 4.2.

Event: failure of $yli[ab]$

Delete `yri`$[ab.$`next`$]$, `yli`$[ab]$;
if $\exists$`yri`$[ab]$
then {
Delete `yli`$[ab.$`prev`$]$, `yri`$[ab]$
Create `slt`$[ab]$, `srt`$[ab]$, `yt`$[ab]$
Remove `ce`(ab) from output
}
else {
Create `yri`$[ab]$, `yli`$[ab.$`prev`$]$
Add a or remove b in output
}

Event: failure of $yt[ab]$

Delete `yt`$[ab]$, `slt`$[ab]$, `srt`$[ab]$;
Create `yli`$[ab]$, `yri`$[ab.$`next`$]$;
Create `yri`$[ab]$, `yli`$[ab.$`prev`$]$;
Add `ce`(ab) to output

Event: failure of `slt`$[ab]$

Delete `slt`$[ab]$, `srt`$[ab]$, `yt`$[ab]$;
$cd \leftarrow ab.$`prev`;
if $d = a$ /* i.e., same color */
then Create `slt`$[cd]$, `srt`$[cd]$, `yt`$[cd]$;
else Create `sl`$[cd]$;

Event: failure of `sl`$[ab]$

Delete `sl`$[ab]$;
$cd \leftarrow ab.$`next`;

if $\chi(cd) \neq \chi(ab)$
  then Create $\texttt{slt}[cd], \texttt{srt}[cd], \texttt{yt}[cd]$;
  else if $\texttt{ce}(cd) \neq \texttt{ce}(ab)$ **then Create** $\texttt{sr}[cd]$;

Event: failure of $\texttt{x}[ab, cd]$
  Delete x[ab,cd]; **Create** x[cd,ab];
  if $\exists \texttt{x}[ab.\texttt{prev}, ab]$
    then Delete $\texttt{x}[ab.\texttt{prev}, ab]$;
    else Create $\texttt{x}[ab.\texttt{prev}, cd]$;
  if $\exists \texttt{x}[cd, cd.\texttt{next}]$
    then Delete $\texttt{x}[cd, cd.\texttt{next}]$;
    else Create $\texttt{x}[ab, cd.\texttt{next}]$;
  $cd.\texttt{prev} \leftarrow ab.\texttt{prev}$; $ab.\texttt{next} \leftarrow cd.\texttt{next}$;
  $cd.\texttt{next} \leftarrow ab$; $ab.\texttt{prev} \leftarrow cd$;
  $\texttt{ce}(ab) \leftarrow d$; $\texttt{ce}(cd) \leftarrow a$;
  /* Now update intersection certificates */
  if $\exists \texttt{yri}[ab]$
    then Delete $\texttt{yri}[ab]$; **Create** $\texttt{yri}[cd]$;
  if $\exists \texttt{yli}[cd]$
    then Delete $\texttt{yli}[cd]$; **Create** $\texttt{yli}[ab]$;
  /* Update slope certificates if ab is below cd */
  if $ab <_y d$
    then {
      if $\exists \texttt{yt}[cd]$ **then Delete** $\texttt{slt}[cd], \texttt{srt}[cd], \texttt{yt}[cd]$;
      if $\exists \texttt{sl}[ab]$ **then Delete** $\texttt{sl}[ab]$;
      if $\chi(ab) \neq \chi(ab.\texttt{next})$ & $b <_s d$ **then Create** $\texttt{sl}[ab]$
      if $\chi(cd.\texttt{prev}) \neq \chi(cd)$ & $cd.\texttt{prev} <_y \texttt{ce}(cd.\texttt{prev})$ & $a <_s c$ **then Create** $\texttt{sl}[cd.\texttt{prev}]$
      elseif $\exists \texttt{sr}[ab]$ **then if** $a <_s d$
        then Delete $\texttt{sr}[ab]$; **Create** $\texttt{slt}[cd], \texttt{srt}[cd], \texttt{yt}[cd]$;
        else Update $\texttt{sr}[ab]$ to point to new $\texttt{ce}(ab)$;
    }
  /* Symmetric treatment if cd is below ab (not shown) */

As an example, consider the event yt$[ab]$, which corresponds to case (iv), right to left, of Figure 4.2: a red edge moves above a blue vertex. In this case, we remove the three certificates proving that the red edge was below the blue chain (bottom case of Figure 4.1), and add certificates to bracket the two newly formed intersections. Finally, the new edge on the upper envelope is added to the output.

Events corresponding to certificates yri$[ab]$, sr$[ab]$, and srt$[ab]$ are exactly symmetric to yli$[ab]$, sl$[ab]$, and slt$[ab]$.

In general, when a y-certificate changes sign, this modifies the output: either two neighbor vertices merge into one, or the reverse. For the purpose of the recursive construction, it is therefore necessary to be able to handle such local structural changes in the input. Consider the case in which edge b disappears between a and c. Just before this happens, b is not the contender edge of any vertex. Hence, only a constant number of certificates may need to be removed: those whose name includes ab or bc between the brackets if they exist (Table 4.1); they are replaced by certificates involving ac instead. In the reverse direction, edge b appears and breaks up vertex ac into two vertices ab and bc, and once again only certificates with ac between the brackets need to be changed. A slope comparison between b and the contender edge of ac may be required to decide which of the two newly created vertices should inherit the tangent certificates if there are any.

4.1.3 Divide and Conquer Upper Envelope

To kinetize the divide and conquer algorithm, we keep a record of the entire computation in a balanced binary tree. A node in this tree is in charge of maintaining the upper envelope of the two convex chains computed by its children. If an event triggers a change in the output of a node, this node passes on the event to its parent, as a local structural change to the input, and so on to upper levels of the computation tree while this change remains visible. This structure has $O(\log n)$ locality and $O\big(\log^2 n\big)$ responsiveness.

As in the case of the one-dimensional kinetic tournament data structure (Section 3.3), we analyze efficiency by considering time as an additional static dimension and charging each event to a feature of a three-dimensional structure with known worst-case complexity. The primal version of the problem is ill-suited for such an analysis, as the static structure described by the convex hull over time is not the convex hull of the trajectories of the underlying points. On the other hand, in the dual, the structure described by the upper envelope over time is exactly the upper envelope of the surfaces described by the underlying lines. We can thus use results proving near-quadratic complexity for the upper envelope of algebraic surfaces [95]. We also make use of the recent result of Agarwal, Schwarzkopf, and Sharir [8] about the near-quadratic complexity of the overlay of the projections of two upper envelopes to obtain sharp bounds on the number of events due to x-certificates.

Theorem 4.3. *The KDS for maintaining the convex hull has $O(n^{2+\epsilon})$ efficiency for any $\epsilon > 0$ in a (δ, n)-scenario. The hidden constant depends on δ and ϵ.*

Proof. We first focus on the events attached to a specific node of the computation tree that involves a total of n red and blue lines. Consider time as a static third dimension: a line whose parameters are polynomial functions of time describes an algebraic surface in three dimensions. The blue (red) family of lines is now a family of bivariate algebraic functions. Looking at the upper envelopes of the blue and red families, and at their joint upper envelope in turn, we observe that a purple vertex on the associated maximization diagram corresponds to a change of sign of a y-certificate (a "y-event") in the kinetic interpretation (Figure 4.3). A monochromatic vertex corresponds to the appearance/disappearance of an edge triggered by some descendant in the computation tree. As our surfaces are algebraic of bounded degree, their maximization diagram has complexity $O(n^{2+\epsilon})$ for any $\epsilon > 0$ by Theorem 2.7, and therefore the number of events due to y-certificate sign changes is bounded by this quantity[1].

Consider now the events corresponding to the x-reordering of two vertices of different colors (called "x-events"). In the 3-dimensional setting, a blue envelope vertex becomes an edge of the blue maximization diagram. Hence, an x-event corresponds to a point (x, t) above which there is an edge in both the blue and the red upper envelopes (Figure 4.4). In other words, each x-event is associated with a bichromatic vertex in the overlay of the maximization diagrams of the red and blue upper families. If we have are n bivariate algebraic surfaces of bounded degree, the complexity of this overlay is also $O(n^{2+\epsilon})$ for any $\epsilon > 0$ by Theorem 2.8. Hence, there are at most that many x-events.

Finally, each pair of lines becomes parallel a constant number of times, so there are $O(n^2)$ slope events attached to the node we have been focusing on up to now.

[1] The best known bound for this specific problem is tighter [6], but this bound is sufficient for our purposes.

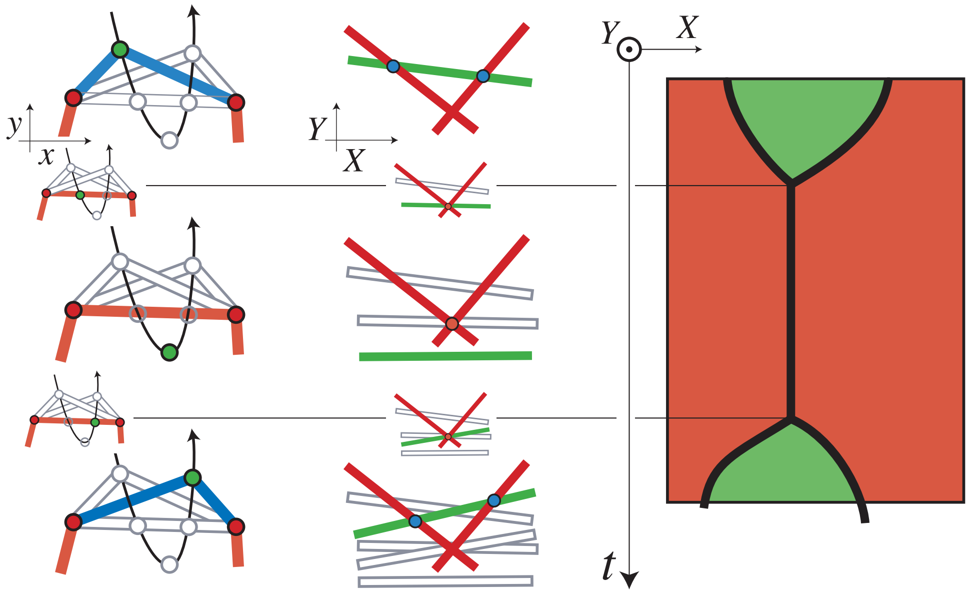


Figure 4.3: A vertex passes through an edge. In the dual space-time view, this corresponds to a vertex in the upper envelope of the surfaces described by the dual lines.

Getting back to the full computation tree, we conclude that the total number of events $C(n)$ satisfies the recurrence $C(n) = 2C(n/2) + O\left(n^{2+\epsilon}\right)$, and therefore $C(n) = O\left(n^{2+\epsilon}\right)$. □

In the worst case, the convex hull of n points in linear or higher order motion changes $\Omega(n^2)$ times [6]; hence our KDS is efficient.

The kinetic data structure for the convex hull, as is, is not very useful if it doesn't have a query data structure built on top of it. This is, of course, easy to obtain: just keep the list of vertices and edges in a dynamic binary tree instead of keeping them in a circular list. This doesn't change the efficiency, and supports ray shooting and point location queries.

4.2 Closest Pair

It is rather simple to create a kinetic data structure that maintains the closest pair of a set of points: create a kinetic priority queue on all possible pairwise distances. The problem with this solution is that it is neither local nor compact (each point is involved in linearly many certificates). To obtain a compact kinetic data structure, we can start from an efficient algorithm algorithm that computes the closest pair.

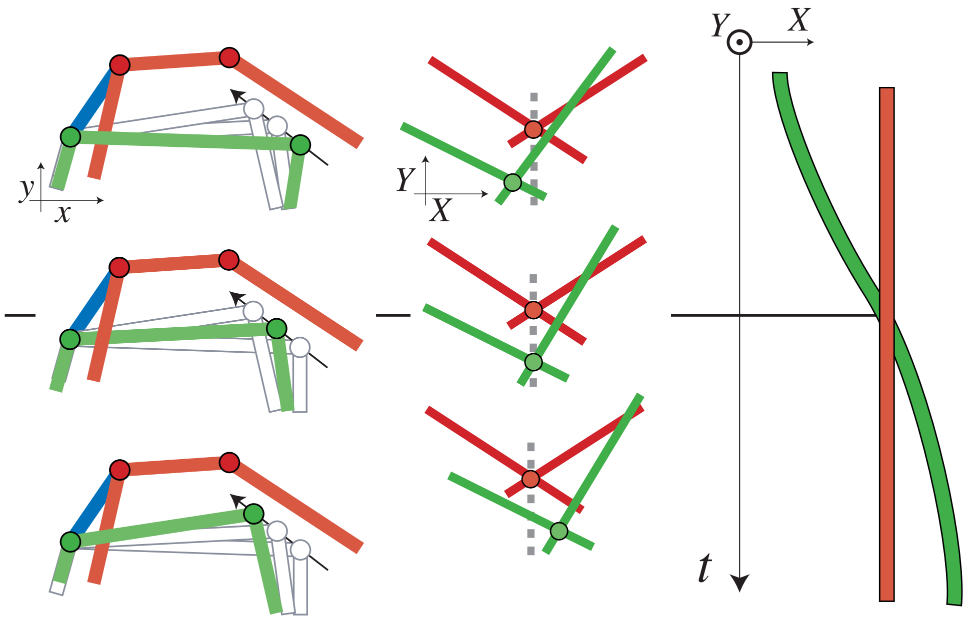


Figure 4.4: Two edges become parallel in the primal. In the dual, this means that two vertices cross each other in the x-coordinate. We have a vertex in the overlay of two upper envelopes in the space-time view.

Consider the following classic divide-and-conquer algorithm due to Shamos [84]: divide the points into the left half and the right half and recursively compute the closest distances δ_L and δ_R within each half. Then check all pairs that are within distance $\delta = \min(\delta_L, \delta_R)$ of a vertical median line $x = x_0$ (Figure 4.5). A kinetic version of this algorithm would require, for each point p on the left side, a certificate of the form $[x_p < x_0 - \delta]$ or the converse. The resulting KDS would not even be responsive: when the identity of the pair that realizes δ changes, all certificates of the form above need to be updated, and there might be a linear number of them.

This example shows that not all algorithms are well suited for kinetization. Surprisingly, in the case of the closest pair, we were not able to find a good kinetization for any known algorithm.

In this section, we describe a new static algorithm for computing the closest pair of a set of points in the plane. The algorithm is based on the plane sweep paradigm. We then consider the kinetic setting: we have a set of items, and each item has a position that is a continuous function of time. We add some data structures to the static algorithm to record

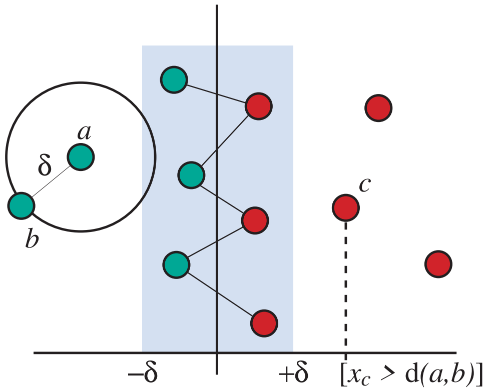


Figure 4.5: A traditional divide-and-conquer algorithm for finding the closest pair. The kinetization of this algorithm is neither local nor responsive, as the current closest monochromatic pair ab is involved in $\Theta(n)$ comparisons.

its history, and show that these data structures can be maintained as the items move. The data structures at a given time t always reflect the history that would result if the plane sweep algorithm were applied to the positions of the items at t. We show that the resulting kinetic data structure has the qualities that we required in the introduction.

4.2.1 Static Algorithm and Proof Scheme

The static closest-pair algorithm is based on the idea of dividing the space around each point into six 60° wedges. It is a trivial observation that the nearest neighbor of each point is the closest of the nearest neighbors in the six wedges. We show that an approximate definition of nearest neighbor in each wedge (based on one-dimensional projections) is still sufficient to find the closest pair. The relaxed definition lets us compute neighbors efficiently, and aids in the kinetization of the algorithm.

In this section, we consider a set of items S in a fixed configuration. For simplicity of notation, we don't distinguish between the items with their positions in this configuration. The distance between two items p and q is denoted by $\mathrm{d}(p,q)$. The x-ordering of S is the ordering of the projections of the positions on the x-axis. We also define the $+60°$-ordering as the ordering of the orthogonal projection of the positions on an oriented line that makes a $+60°$ angle with the x-axis, and similarly the $-60°$-ordering. We write $<_0, <_+$ and $<_-$ for these three orderings. The general position assumption means (1) that no items are equal in any of these three orderings, and (2) that no two pairs of items are at the same distance. With these assumptions, the structures described in this section and the closest pair are uniquely defined.

Lemma 4.4. *Let T be an equilateral triangle. Let u be one of the vertices of its boundary, and v be a point lying on the boundary edge of T opposite to u. Then for every point w in the interior of T,*

$$\mathrm{d}(w, v) < \mathrm{d}(u, v)$$

Proof. Let a be the length of a side of T, and $b = \mathrm{d}(u, v)$. The boundary edge that contains v has two end vertices, and we let z be the one farther from v. Then we can write $\mathrm{d}(v, z) = \frac{a}{2} + c$, for some c with $0 \leq c \leq \frac{a}{2}$. By the Pythagorean theorem, $b^2 = c^2 + \frac{3}{4}a^2$, and therefore:

$$\begin{aligned}(b - (c + \frac{a}{2}))(b + (c + \frac{a}{2})) &= b^2 - c^2 - ac - \frac{a^2}{4} \\ &= \frac{a^2}{2} - ac \\ &\geq 0\end{aligned}$$

Thus, $b \geq c + \frac{a}{2}$; the circle centered at v and of radius $\mathrm{d}(u, v)$ contains all three vertices of T's boundary, so it strictly contains all the points interior to T. □

We define the ***dominance wedge*** of an item p, call it $Dom(p)$, to be the right-extending wedge bounded by the lines through p that make $\pm 30°$ angles with the x-axis. The item q is in $Dom(p)$ iff $p <_- q$ and $p <_+ q$.

We define the ***cover*** of an item q, denoted $Cover(q)$, as the rightmost item that contains q in its dominance wedge (we add a special item at $(-\infty, 0)$ as a sentinel) . The ***candidates set*** of item p, denoted $Cands(p)$, is the set of items whose cover is p, and the leftmost of those, denoted $lcand(p)$, is called the ***left candidate*** of p. A pair $(p, lcand(p))$ is called a ***candidate pair***. See Figure 4.6.

Lemma 4.5. *Let S be a set of items in general position. If (p, q) is the closest pair in S and $q \in Dom(p)$ then q is the left candidate of p.*

Proof. First, suppose p is not the cover of q. This means another item $r \in S$ is to the right of p and such that $q \in Dom(r)$. In this case, r is contained in the interior of an

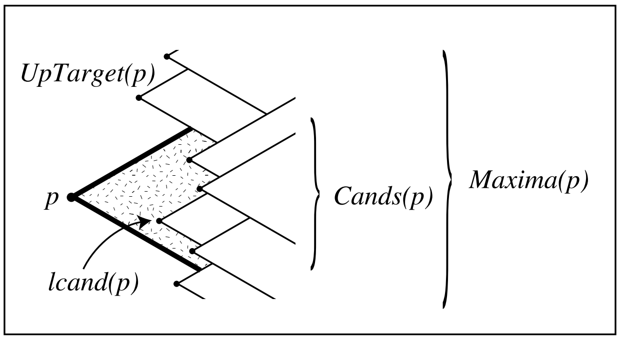


Figure 4.6: The sets of points $Maxima(p)$ and $Cands(p)$, the leftmost candidate $lcand(p)$, and the upper target $UpTarget(p)$.

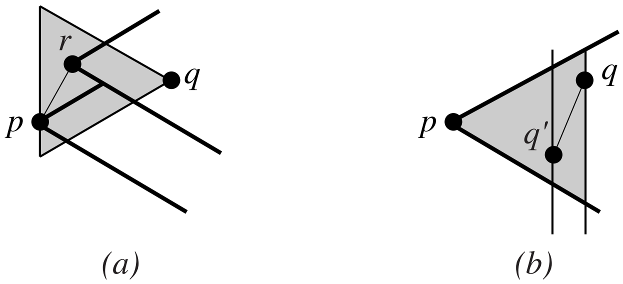


Figure 4.7: If (p, q) is the closest pair and $q \in Dom(p)$, then (a) p is the cover of q; (b) q is the left candidate of p (Lemma 4.5).

equilateral triangle with a vertex at q and extending leftwards, with p on its left boundary (Figure 4.7a). By Lemma 4.4, this contradicts the hypothesis that (p, q) is the closest pair. Hence $p = Cover(q)$.

Now, assume that there is a point $q' \in S \cap Dom(p)$ that is to the left of q. This time, q' is in the interior of the equilateral triangle starting at p and extending rightwards with q on its right boundary. Thus $\mathrm{d}(q, q') < \mathrm{d}(p, q)$ by Lemma 4.4 which contradicts the hypothesis once again. Hence q is the left candidate of p. See Figure 4.7b. $\square$

We now have at most n candidate pairs for a set of n items. We say that these are the candidate pairs associated with the $0°$ direction. In order to catch the closest pair, we need to repeat the same definitions for two other directions $\pm 60°$. The dominance wedge associated with an angle Δ is simply the dominance wedge that would be obtained from the definitions above if the plane were rotated by $-\Delta$ around the origin.

Corollary 4.6. *The closest pair of a set of items S in general position is one of the (at most) $3n$ candidate pairs associated with the three directions $0°, 60°, -60°$.*

Proof. Say that (p, q) is the closest pair, with p to the left of q. The three dominance wedges of p span the whole half-plane to the right of the vertical line passing through p. Hence, item q is contained in one of them. By the previous lemma, (p, q) is a candidate pair for this direction. $\square$

We need a few more definitions. For a given item p, we consider all the items to its right, denoted S_p. The set of items of S_p whose cover is not in S_p is denoted by $Maxima(p)$ (Figure 4.6).

Proposition 4.7. *If $q, r \in Cands(p)$ then*

$$q <_+ r \Leftrightarrow r <_- q \Leftrightarrow \ q \text{ is smaller than } r \text{ in the } y\text{-ordering.}$$

Proof. As q, r are not in each other's dominance wedge, r is either above or below the double wedge delimited by the $\pm 30°$ lines passing through q. Whether it is above or below, the $<_+$ and $<_-$ orderings are opposite of one another. Also, the $<_+$ ordering is the same as the y-ordering. $\square$

The dominance wedge of p cuts $Maxima(p)$ in three sets: the central set, those items in the dominance wedge, are the candidates of p. The lowest item of the set above $Dom(p)$ is called the ***upper target*** of p, denoted $UpTarget(p)$. The highest item below $Dom(p)$ is called the ***lower target*** of p, denoted $LowTarget(p)$. The upper target of p is exactly the left candidate of p for direction $+60°$. See Figure 4.6 again. We also add here sentinels at infinity above and below the plane so that the targets are well-defined for all items.

Proposition 4.8. *If $UpTarget(q) = UpTarget(r)$ and $q <_0 r$, then $q <_- r$. If $LowTarget(q) = LowTarget(r)$ and $q <_0 r$, then $q <_+ r$.*

Proof. We only prove the first statement. Let p be the common target of q and r, and assume that q is to the left of r. If $r <_- q$, then r is above the dominance wedge of q. But r is below p, so p cannot be the upper target of q, a contradiction. $\square$

We now sketch the static algorithm to compute all the candidate pairs associated with the horizontal direction. We insert the items from right to left. Before inserting item p, the set $Maxima(p)$ is stored in a binary tree $Maxima$, sorted by increasing y-coordinate (or, equivalently, by $\pm 60°$-ordering). The tree is augmented at each node with a field that contains the leftmost item in the subtree rooted at that node.[2] Here is the algorithm:

1. Initialize $Maxima$ to $\emptyset$.
2. For each point $p \in S$ from right to left,
   (a) Find $UpTarget(p)$ and $LowTarget(p)$ in $Maxima$.
   (b) Set $Cands(p) = Maxima \cap Dom(p)$.
   (c) Set $lcand(p)$ to be the leftmost element of $Cands(p)$.
   (d) Replace the items of $Cands(p)$ by p in $Maxima$.

It is clear that the plane sweep algorithm can be implemented to run in $O(n \log n)$ time. Sorting the points of S in preparation for sweeping takes $O(n \log n)$ time. We store $Maxima$ in an augmented balanced binary tree structure that supports logarithmic-time searches, insertions, deletions, splits, and joins [33]. Computing $Cands(p)$ requires two $O(\log n)$ time searches on $Maxima$, since $Cands(p)$ is a consecutive subsequence of $Maxima$. Splitting $Cands(p)$ out of $Maxima$ and inserting p in its place (with the update of the "leftmost" field) takes $O(\log n)$ per item p. Thus the total running time is $O(n \log n)$.

The same algorithm is applied with the plane rotated $+60°$ and $-60°$ to obtain all the candidate pairs referred to in Corollary 4.6. A tournament on the distances between the candidate pairs thus obtained allows us to select the closest pair in additional linear time. Hence, we obtain the closest pair in worst-case total time $O(n \log n)$.

The algorithm to compute the candidate pairs uses only comparisons in the three directions $0°, 60°$ and $-60°$. Hence, the cover, target, and left candidate of each point are certified by the orders along these three directions. We refer to all these attributes as the ***cone structure*** of S.

Proposition 4.9. *The following set of certificates on a set of items S:*

[2] The "leftmost" field is not needed for the static algorithm because we can afford to look at all elements of $Cands(p)$ explicitly without increasing the running time. It will be useful later for the kinetization.

1. *those that certify the* $0°$ *order of* S,
2. *those that certify the* $60°$ *order of* S,
3. *those that certify the* $-60°$ *order of* S,
4. *those that certify a tournament on the candidate pairs of* S,

are enough to certify the cone structure and the closest pair of S.

This proof structure has linear size and $O(\log n)$ *locality.*

Proof. The validity of the proof is given by the correctness of the algorithm.

As for locality, each item is involved in at most 2 certificates per ordered list. Moreover, for each direction, it is involved in at most two candidate pairs: with the item that is its left candidate, and with the item that it is the left candidate of. Therefore, in the tournament, a given item appears in $O(1)$ leaves, and in $O(\log n)$ internal nodes. Hence, the proof structure has $O(\log n)$ locality. □

4.2.2 Kinetization

We now have a structure that contains the closest pair, and a set of certificates that prove that this structure is valid. We now need to examine how this structure needs to change upon a certificate failure, and count the worst case number of events that can happen in the polynomial model of motion.

We already know how to update the tournament on the candidate pairs (Section 3.3). We need to describe here how to update the cone structure when the ordering of two items changes along one of the three privileged directions. When two items p, q interchange their x-ordering, this can trigger a change in the cover of linearly many other items (Figure 4.8). Similarly, a change in the $+60°$ ordering can change linearly many targets (Figure 4.9). In order to obtain a responsive kinetic data structure, we represent the targets and covers implicitly using three binary trees:

1. *Cands*(p) contains the candidates of p, as a sequence of items sorted in the y-ordering. (This order is the same as the $\pm 60°$-orderings by Proposition 4.7.) This sequence is stored in a balanced binary tree and supports the usual searching and update

operations. In addition, each node of the tree has a pointer to its parent in the tree, and the root of the tree for $Cands(p)$ points to p. Thus each item $q \in S$ can find its cover in $O(\log n)$ time. Each node in a $Cands()$ tree also keeps track of the leftmost (in x-order) item in its subtree, and so the root of $Cands(p)$ records $lcand(p)$. The parent pointers can be maintained as part of the standard tree update operations, within the same asymptotic time bound, as can the "leftmost" fields. As part of our algorithm, we will make sure that the "leftmost" fields are maintained correctly whenever the x-order of items changes.

2. $Hits_u(p)$ records all the items for which p is an upper target, sorted in the $-60°$-ordering (or, equivalently by Proposition 4.8, in the x-ordering). The sequence is stored once again in a balanced binary tree, with a parent pointer at each node and a pointer to p at the root, so that the upper target of any item can be found in logarithmic time.

3. $Hits_\ell(p)$ records all the items for which p is a lower target in a similar fashion.

These are the only data structures needed for the kinetization. In particular, we don't use the *Maxima* data structure described in the static case. The static algorithm is useful to create these data structures from scratch in $O(n \log n)$ time (the *Hits* structure can be built incrementally each time a target is found).

The following algorithmic sketch shows how to update all the affected $Cands()$, $Hits()$, and $lcand()$ fields when two items p and q exchange positions in the x-order of S. Without loss of generality, assume that p is left of q before the exchange. Furthermore, assume that p is below q at the instant of exchange (similar pseudo-code applies if p is above q).

1. If $q = UpTarget(p)$, then
   (a) Split off the portion of $Cands(q)$ inside $Dom(p)$ and join it to the top of $Cands(p)$.
   (b) Let $w = LowTarget(q)$. Delete q from $Hits_\ell(w)$ and insert it into $Hits_\ell(p)$.
   (c) Let v be the new bottom item of $Cands(q)$, if any, or else the upper target of q. Delete p from $Hits_u(q)$ and insert it into $Hits_u(v)$.
2. Let $p' = Cover(p)$ and $q' = cover(q)$. If $p' = q'$, then update $lcand(p')$ starting from the common ancestor of p and q in the tree for $Cands(p')$.

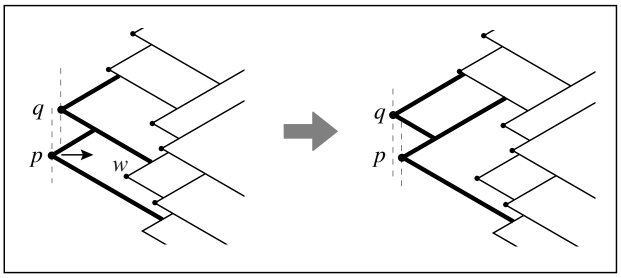


Figure 4.8: An x event and the change in the *Cands* sets.

A few explanations and Figure 4.8 might help the reader to follow this algorithm. If q is a target for p, then their x-exchange changes the cone structure. Specifically, the items of $Maxima(q)$ in $Dom(q) \cap Dom(p)$ are transferred from $Cands(q)$ to $Cands(p)$. Item p gets a new target; the new point of contact between the segment from p and its target lies inside $Dom(q)$. Likewise q gets p as a target. Step 1 handles these changes.

The only edges of the maxima diagram that change are those that extend to the right from p and q—there are no target changes for items either right or left of $\{p, q\}$—so the operations of Step 1 suffice to update the maxima diagram.

If neither p nor q is a target for the other, then the maxima diagram does not change—the $Cands()$ and $Hits()$ fields do not need to be updated.

Whether or not the maxima diagram changes, one $lcand()$ field may change. If $p, q \in Cands(u)$ for some item u, we need to ensure that the "leftmost" fields are updated in the binary tree representing $Cands(u)$, so that any comparison of p and q in that tree is re-evaluated; this may cause $lcand(u)$ to change. Step 2 takes care of this.

Each step can be accomplished in logarithmic time. In particular, step 1a can be done because $Cands(q)$ is sorted in the $-60°$-ordering. The rescheduling of the modified certificates for the ordering that changed also takes logarithmic time.

* * *

The following pseudo-code updates the affected fields when two items p and q exchange positions in the $+60°$-order of S (at the instant of exchange, the line through p and q makes an angle of $-30°$ with the x-axis). Without loss of generality, assume that p is to the left of q and above q. There are two cases, depending on whether q enters or exits from $Dom(p)$.

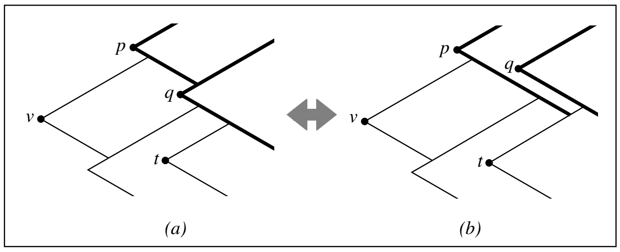


Figure 4.9: A 60° event. (a→b) q enters $Dom(p)$; (b→a) q exits $Dom(p)$.

In the first case, q enters $Dom(p)$. Update the data structures thus:

1. If $q = LowTarget(p)$ then
   (a) Let $v = Cover(q)$. Delete q from $Cands(v)$ and insert it into $Cands(p)$.
   (b) Let t be the leftmost item in $Hits_u(q)$ that is to the right of p, if any, or else the lower target of q. Delete p from $Hits_\ell(q)$ and insert p into $Hits_\ell(t)$.
   (c) Split off the subsequence of $Hits_u(q)$ whose items are to the left of t (and hence left of p) and join it onto the bottom of $Hits_u(p)$.

In the second case, q exits $Dom(p)$. The pseudo-code in this case just inverts the action performed in the first case:

1. If $q \in Cands(p)$ then
   (a) Let $t = LowTarget(p)$. Delete p from $Hits_\ell(t)$ and insert p into $Hits_\ell(q)$.
   (b) Split off from $Hits_u(p)$ the items larger than q according to $<_-$, and join them onto the top of $Hits_u(q)$.
   (c) Let v be the new rightmost item of $Hits_u(p)$, if any, or else the cover of p if $Hits_u(p)$ is empty. Delete q from $Cands(p)$ and insert q into $Cands(v)$.

Here are a few explanations to be read with Figure 4.9. Consider first the case in which q enters $Dom(p)$. If q is not the lower target of p, then $q \notin Maxima(p)$ and the exchange of p and q in the $+60°$-order does not affect the cone structure. No data structure updates are necessary.

If q is the lower target of p, the cone structure changes, but only in the vicinity of p and q. The exchange of p and q does not change the targets of items to the right of p. Only the lower target of p needs to be updated; Step 1b takes care of this. Of the items to the left of p, only those with q as their upper target (i.e., members of $Hits_u(q)$) need to have their targets changed to p. Step 1c does this. The $Cands()$ set changes only for p (because q enters it) and for the item v whose $Cands(v)$ set q leaves; Step 1a does this. The "leftmost" fields are updated in the $Cands()$ binary trees during the modification, so $lcand(p)$ and $lcand(v)$ are correctly maintained. The $Cands()$ and $Hits()$ lists are enough to specify the combinatorial structure of the maxima diagram; since they are correctly maintained, so is the maxima diagram.

In the case in which q exits $Dom(p)$, the changes to the maxima diagram are the inverse of those in the first case. The update procedure for this case inverts the action of the first update procedure.

All the operations above can be done in logarithmic time. In particular, step 1b of the first case can be done because the items in $Hits_u(q)$ are in x-order. The $O(1)$ certificates that change can be updated in the event queue in logarithmic time also.

The procedure for exchanging two items in the $-60°$-order is symmetric to the one for $+60°$-order exchanges.

* * *

The final element of our kinetic data structure is a kinetic tournament on the $3n$ distances corresponding to $(p, lcand(p))$ pairs (this adds $3n$ certificates to our KDS). The root of the tournament tree contains the closest pair at any time. Note that when $lcand(p)$ changes, it triggers a discontinuity of the associated distance in the kinetic tournament. Hence, although we start with a non-dynamic scenario, the kinetic tournament is acting on a dynamic scenario.

Theorem 4.10. *The KDS for the closest pair has efficiency $O\big(n\lambda_{2\delta+2}(n)\log^2 n\big)$ in a (δ, n, m)-dymanic scenario.*

Proof. An event for the cone structure is an exchange of order of two items along one of the three orders. As a pair of items can exchange their order at most δ times, there are $O\big(n^2\big)$ such events, and we saw earlier that each such event is processed in $O(\log n)$ time.

The square of the distance between two items is a polynomial of degree 2δ. Hence the scenario induced for the kinetic tournament is a $(2\delta, n, n^2)$-dynamic scenario and therefore has efficiency $O\big(\lambda_{2\delta+2}(n^2)\log^2 n\big)$ (Lemma 3.8). $\square$

We therefore have a kinetic data structure for the closest pair. Its locality is constant, it has linear size, it has $O\big(\log^2 n\big)$ responsiveness, and has near optimal efficiency.

4.3 Conclusion

In this chapter, we presented kinetic data structures for some fundamental computational geometry problems. We hope that these examples will have convinced the reader that there is a craft in both the design and the analysis of kinetic data structures. In particular, kinetic data structures can be combined, as in the closest pair problem, and it is therefore important to study carefully kinetic data structures for the basic problems, as we can expect that they will be useful in more elaborate setups.

It is possible to insert and delete items from the convex hull KDS, but this might require linearly many changes in the worst case. To obtain a kinetic data structure with good dynamic properties, it would be tempting to kinetize the dynamic data structure of Overmars and Van Leeuwen [81], which maintains the convex hull of a set of points with $O\big(\log^2 n\big)$ cost per insertion or deletion. Proving that such a scheme is efficient (if it is so) seems challenging as part of the structure requires the maintenance of the median of n moving values, a problem for which no tight combinatorial bounds are currently known. The closest pair can be made dynamic with polylogarithmic cost per insertion/deletion if we use multi-level search trees based on the three orderings.

Chapter 5

Implementation

The kinetic data structures described in Chapter 4 have been implemented [20], with the addition of two kinetic data structures: one for the Voronoi Diagram and another one for the minimum spanning tree. The implementation (***Demokin***) is intended as a proof of concept. It comes with a user interface specifically built to demonstrate how kinetic data structures work.

In this chapter, we do not intend to describe a full implementation. Our goal is rather to give a high-level description of three aspects of this implementation: an overview of the user-interface (Section 5.1.1); an overview of the basic structure of a KDS implementation (Section 5.1.2); and a case for a "certificate-centered" implementation (Section 5.1.3). In Section 5.2, we delve into two specific aspects of the implementation that are the most important: the computation of the failure times of certificates (which poses specific problems of robustness), and the software system issue of combining existing kinetic data structures as black boxes.

We describe exactly the user interface that we implemented. As for the program structure, it is close to what was implemented in spirit, but we take the liberty of describing what we think the architecture ought to have been, based on the experience gained during the implementation.

5.1 Overview

5.1.1 User Interface

The main window of Demokin (Figure 5.1, left) contains the display of the moving items, as well as a thermometer (vertical left) that indicates the amount of processing time used for the kinetic data structure maintenance exclusively (the drawing time is not included). At the bottom of the window, a tape runs from right to left. Each time an event happens in one of the active KDSs, a tick mark is put at the right end of the tape. An external event is indicated by a large red tick mark, an internal event is indicated by a small blue tick mark.

The control window (Figure 5.1, right) contains a few buttons to start and stop a simulation and to control the appearance of the moving items (points or disks). Each kinetic data structure is controlled by a line of three checkboxes. The first checkbox indicates whether the KDS is active or not, the second one controls whether to display the certificates or not, and the third box allows the user to interrupt the simulation at every external event.

With the first box, we can show that several kinetic data structures are able to run in parallel, but we can also select only one KDS in order to examine the frequency of events (on the bottom tape) and the processing cost in the thermometer.

Each KDS presents its certificates in its own specific way. For instance, for the Voronoi diagram (the prettiest KDS if not the most algorithmically elaborate), we choose to display the Delaunay triangulation as the certificates. For the convex hull, we display the two subhulls, and the triangulation of the sleeve between them. An extra panel allows the user to recursively inspect the substructures (Figure 5.2). For the closest pair, the user can display each of the three cone structures or simply all the candidate pairs (Figure 5.3).

Finally, the control panel includes a button "Brute force" that inactivates all kinetic data structures, and replaces them by a recomputation from scratch at a fixed time interval. The recomputation frequency can be varied, to show that a high frequency is computationally expensive, while a low frequency is inaccurate (Figure 5.4).

An important aspect of this implementation cannot be conveyed by these screen snapshots: the structures shown are redrawn continuously as the objects move, allowing the user to understand the distinction between the continuously-changing positions of the items and

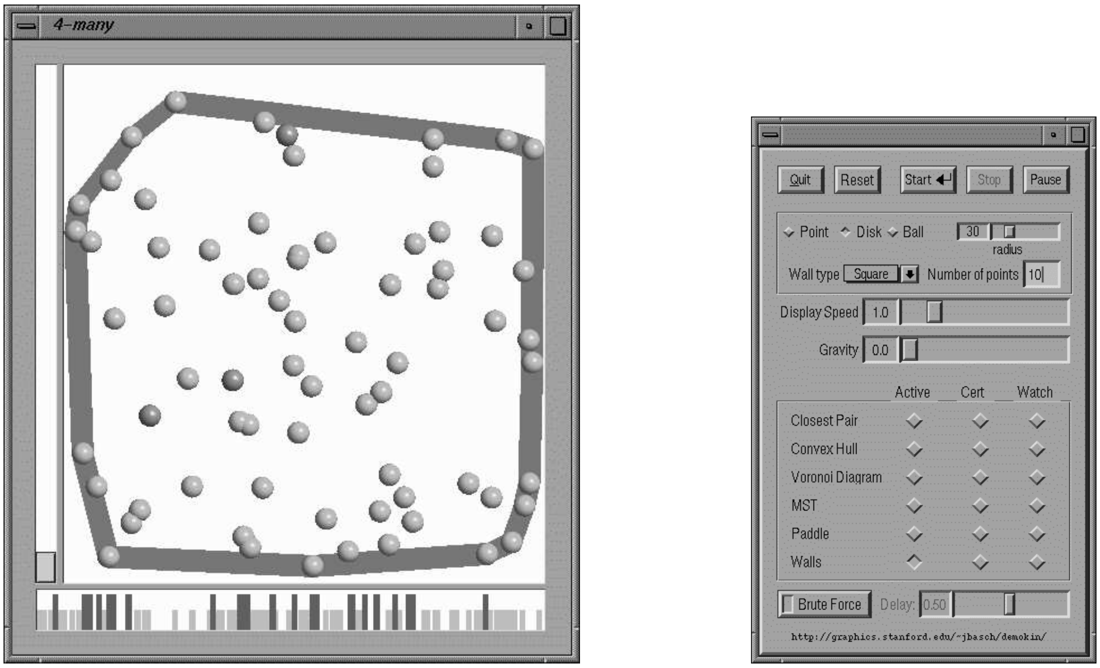


Figure 5.1: The main window and control panel of Demokin.

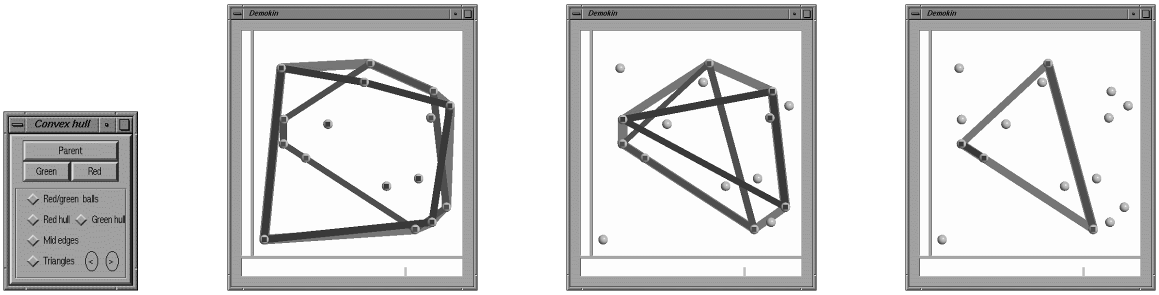


Figure 5.2: Certificate structure for the convex hull diagram. The control panel on the left allows the user to navigate in the recursive structure of the convex hull proof.

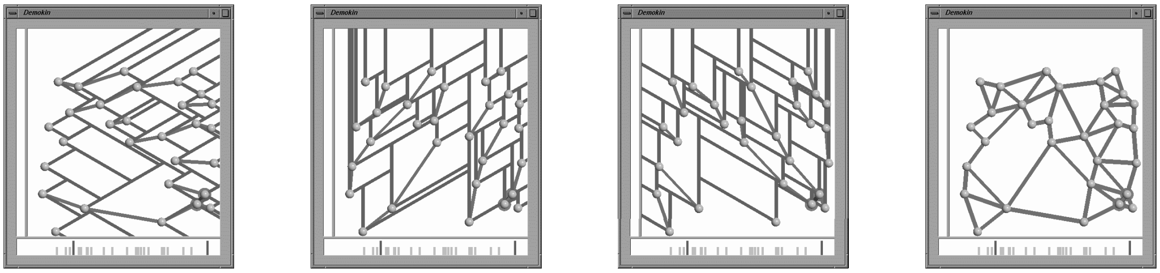
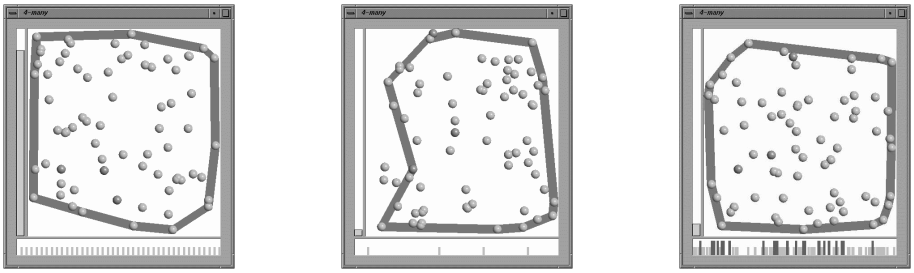


Figure 5.3: The three cone structures of the closest pair KDS, and the set of candidate pairs selected. The closest pair is emphasized by circles.



(a) (b) (c)

Figure 5.4: Comparison between the kinetic data structure for the convex hull diagram and a method that recomputes the convex hull diagram every Δt. Three fast items have been put in a large collection of slow items. (a) If Δt is small, the result is accurate but the computation cost is huge, as indicated by the vertical bar on the left. (b) With a larger Δt, the computation cost is small, but the convex hull diagram is most often wrong. (c) The kinetic data structure keeps track of the convex hull diagram exactly (each event requires very little computation).

the discretely-changing discrete attributes and kinetic structures based on these items.

5.1.2 Structure of a KDS Implementation

We now consider how to implement a new KDS. There are three classes that a KDS implementation should subclass: `kds`, `certificate`, and `item`. In fact, we will see later that the KDS will subclass some predefined subclasses of `item` and `certificate`. For instance, there is a subclass of `item` for each dimension (1 or 2), and there is a subclass of `certificate` for each type of geometric test. We will refer here to the subclasses by the names above, as there is no confusion possible.

Only one object of the subclass of `kds` is created: it is the one in charge of the creation of the structures, of the management of the event queue, and the like. This object is also the owner of all the `certificate` and `item` objects that are created. In this section, we refer to the unique `kds` object as "the KDS". It is created by a ***client*** who is interested in keeping track of the discrete attribute the KDS is able to maintain. Note that when we combine kinetic data structures, the client may be a kinetic data structure itself. All KDSs keep a pointer to a global event queue and schedule their certificate failures in this event queue.

For each moving item, the client asks the KDS to create an `item` object with `item* kds::newItem()`, then sets its equation of motion. If the KDS is dynamic, this request can be done at any time, otherwise it should be done only when the KDS is inactive. Similarly, the client can ask the KDS to destroy the `item` objects previously created. The reason why it is the KDS that should create the object is so that it can create an object of the appropriate subclass. In general, it will be necessary to store some information specific to the KDS in an `item`.

When the KDS is inactive, no particular structure is created apart from all the items requested. When the KDS is made active (the "current time" is given as a parameter), it creates whatever structures are necessary, as well as all certificates that certify the kinetic structure at the current time. A certificate is where everything happens. For each certificate subclass, we need to implement two methods: `failureTime` and `fails`. The first method is in charge of computing the failure time of the certificate. In general, this method is already implemented by the superclass, unless we have very exotic certificates. The method `fails` is in charge of updating the KDS structures when the certificate fails. Finally, an `item` keeps track of the certificates in which it is involved, and requests these certificates to recompute their failure times upon a motion plan update.

To summarize, the `kds` object is in charge of creating the global structures, and the `certificates` compute their failure times and change the global structure when they fail. The `item` is in charge of handling a motion plan update.

* * *

The description above indicates that there are a number of procedures common to all KDS implementations. These procedures can be handled by the base classes:

1. The global event queue has a method `update(time t)` that takes care of the event scheduling. It processes all events until time t to bring the kinetic data structures up-to-date.

2. An `item` keeps a list of all the certificates it is involved in. When an item updates its motion plan, it automatically asks all the certificates that it partakes in to recompute their failure time.

3. A mechanism (described in Section 5.2.2) is provided to combine kinetic data structures, either to have them run in parallel on the same input, or to use the output of one as the input of another.

5.1.3 Certificate-centered Implementation

The most important lesson of our implementation is this: when implementing a kinetic data structure, one is tempted to make it *event-centered*, but it should be *certificate-centered.* Here is an example to illustrate this cryptic statement.

Suppose that we want to create a kinetic data structure that keeps all items inside the unit square. We call this kinetic data structure a `kWall`. In order to reverse the velocity of an item when it reaches the boundary of the unit square, we create an object called `bounceEvent`. Its failure time is the earliest time at which the item reaches one of the four walls, and we store in the object the wall that is going to be hit. When it is time for this event to happen, depending on the wall that is hit, we change the velocity of the object appropriately, and recompute the new failure time. This is what we call an event-centered implementation.

In a certificate-centered implementation, we create four certificates per item: one for each side of the square (which we call *walls*). That is, we have a certificate that guarantees that the item is below the top wall, one that guarantees that the item is above the bottom wall, and two more for the side walls. Each certificate has a distinct failure time, and is put in the event queue independently, thus using more memory, even for certificates that we know will never fail (something will happen to them before they fail). This is the approach we advocate. We saw that it consumes more memory. Is it more time-efficient? Not quite so: when a certificate fails, the equation of motion of the item involved changes, and we therefore cannot avoid recomputing the failure time of all four certificates dependent on this item.

Let's consider the event-centered solution in a context in which an item's equation of motion includes an acceleration due to gravitation. When the item hits the bottom wall, it bounces against it, but the bottom wall might be hit again before any other wall. As we will see in Section 5.2.1, we have to be very careful to discard the first root when computing the next time the item will hit the bottom wall, but the first root should not be discarded when computing the time at which the item bounces against the other walls. In a certificate-centered solution, this problem is taken care of automatically.

In general, the creation of one object per certificate allows us to write more elegant code with fewer special cases.

5.2 Implementation Details

In this section, we provide a few implementation details regarding questions that are not of a combinatorial nature, but that deserve some attention.

The first question that we address here is about robustness. In geometric algorithms, one often assumes general position as a way to obtain simpler code. To obtain general position, the careful implementor uses a symbolic perturbation scheme [46], while the less careful one readily perturbs all inputs by some small random amount. In a kinetic data structures, an event is precisely a degeneracy, and this leads to the most important robustness problem that we address below.

The second question is in the realm of software systems. We show how to combine kinetic data structures. Indeed, an important aspect of a good software library is the possibility of plugging together different data structures in a seamless fashion. In the second part of this section, we give some directions on how to do the same thing with kinetic data structures. The efficiency-minded reader will notice that this system is likely to be fairly slow, in particular due to the heavy usage of memory allocation, but our goal here is only to give a good prototyping environment.

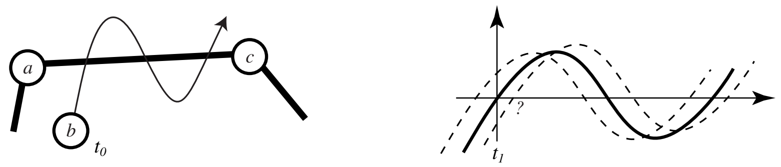


Figure 5.5: If a certificate fails at time t_1, special care needs to be taken to compute the failure time of its opposite.

5.2.1 Computing Failure Times

In our implementation, all the items are given a velocity that is either constant or changing with a constant downward acceleration (gravitation). All functions are therefore polynomials, and the failure times of all certificates are roots of polynomials. The highest degree polynomials we need to solve are of degree 7: they correspond to the `InCircle` test used for the Voronoi diagram. If the acceleration is the same for all points, the polynomials are only of degree 6.

To find the roots of polynomials of degree more than two, we use the Laguerre method as implemented in "Numerical Recipes in C" [85]. The Laguerre method returns all (real and complex) roots of a polynomial, and is certainly not the most efficient method for our problem, but it is the solution we adopt in order to make sure that no root is missed (which we could not guarantee if we had used a method that finds a single root).

Let us not discuss in any more detail the exact way we compute the roots of our polynomials. This is not our field of expertise, and the model of motion—polynomials—is anyhow too restrictive for any real application. However, whatever way is used to compute the roots, there are robustness issues that influence the combinatorial questions. Mostly, robustness problems come from the fact that we cannot find roots very accurately.

To fix ideas, let us consider the convex hull diagram. A certificate c is a `CCW` test involving three items, and has an associated function $f_c(t)$ that is zero at times when the test fails. When computing the failure time, we need to find the smallest root of f_c that is *greater than* t_0, if t_0 is the current time of the simulation. Now, when this certificate c fails at time t_1, its opposite usually remains in the proof after t_1, and the associated polynomial is $-f_c(t)$. Due to numerical inaccuracies, the quantity $-f_c(t_1)$ is not exactly zero, but can be of either sign. Finding the first root of $-f_c$ that is greater than t_1 will sometimes spuriously return t_1 again (or, rather, a quantity slightly greater than t_1). See Figure 5.5.

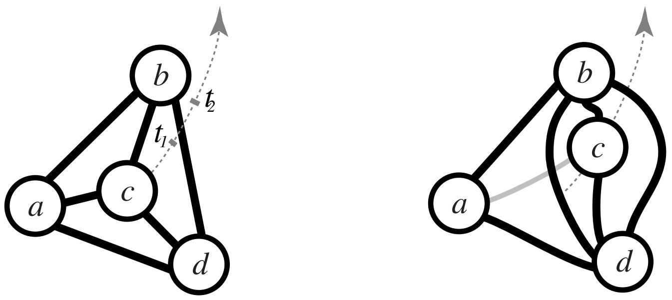


Figure 5.6: This is a Delaunay triangulation. Due to precision errors, we mistakenly think that *abcd* become cocircular (at t_1) before *bcd* become collinear (at time t_2. The result of this out-of-order scheduling is a corrupted structure at t_1^+. It can be detected by checking the sign of `CCW`(b, c, d) when we deschedule it.

There are a number of ways to address this problem. The simplest is as follows: when a KDS creates a function $f_c(t)$ associated with a certificate c, it creates it so that the sign of $f_c(t)$ at the current time is *positive*. What we mean is that it creates the function that would be positive at t^+ if there was no numerical inaccuracy. Then the root-finding method can be modified to find, instead of the first root greater than t_0, the first root greater than t_0 where the function becomes negative. This automatically discards the spurious root mentioned above.

There is another problem that may occur due to numerical inaccuracies: two events might happen out of order in the event queue, leading to an invalid structure (Figure 5.6). Some other swaps in the event queue might be harmless if the events involved are not "connected" in the geometric structure as they are in Figure 5.6. This is an issue that deserves much more careful study. In our implementation, we have a simple mechanism to detect harmful swaps, which is as follows: whenever we schedule or deschedule a certificate in the event queue, we verify that its sign is positive at the current time[1]. If it is not, we stop the simulation immediately. In an application setting, we could instead restart the KDS from scratch at the current time. This would be reasonable, as we observed that the harmful swaps were extremely rare.

The first technique we describe uses the sign of a certificate to decide whether to discard the first root or not. The second technique uses the same sign to detect some dramatic problem. How do we do both at once? Here is a solution: when processing an event,

[1]In the existing implementation, we check this sign only upon scheduling and not upon descheduling.

we keep a pointer to the failing certificate that triggered this event. When scheduling or descheduling a certificate, we compare this certificate with the failing certificate. If they are the same (and by this, we mean *symbolically* the same, and not only physically), only then do we use the sign to discard the potentially spurious first root. Otherwise, we use the sign to detect a potentially harmful swap.

As with all algorithms that involve algebraic computations, there are important robustness issues that require more careful scrutiny. In our implementation, we only addressed them up to the point at which we could obtain accurate simulations for a few thousand items. Guibas and Karavelas [59] replaced the root-finding method with a bracketing method using Sturm sequences, a technique that should allow them to use exact arithmetic and guarantee a robust implementation.

5.2.2 Combining Kinetic Data Structures

We now switch gears radically and address a program design issue, that of combining kinetic data structures. There are essentially two contexts in which we need to combine KDSs.

Let's consider the closest pair kinetic data structure (Section 4.2). We can think of it as three kinetic data structures that maintain cone structures and candidate pairs. Each candidate pair defines a one-dimensional moving item (its separation), and all these items form the input of a kinetic tournament. When an item changes its equation of motion, it is necessary to find all certificates and items that depend directly upon this item.

Sometimes, it is necessary to have different kinetic data structures running on the same data. In an action game, we may want to keep track of visibility structures while performing collision detection, and we would like to run transparently in parallel two kinetic data structures that we could have created earlier independently. The issue is more subtle than it seems, mostly due to changes of motion. Indeed, it might happen that the two structures use exactly the same certificate, say $\texttt{CCW}(p,q,r)$. Suppose that upon failure of this certificate, the collision detection KDS decides that it is necessary to change the motion of p. In this case, the visibility KDS should be warned of this fact. Moreover, the KDS should be aware that the certificate $\texttt{CCW}(p,q,r)$ that it is asked to recompute the failure time of is currently failing. This is necessary for the reasons surveyed in Section 5.2.1. Hence, the integration of different kinetic data structures has to be done carefully.

* * *

Here is our approach to these problems. To be able to run several KDSs on the same data, we create a special KDS, called a `flock`, whose purpose is to distribute the data to several slave KDSs. As we said that an `item` belongs to the KDS that created it, we need to be able to create `clones` of items, that is, items that don't keep any equation of motion but only a pointer to another item that holds the equation of motion. Conversely, the original item needs to be able to access and warn its clones when it updates its motion plan. This also means that any KDS is defined as a template: the name of the `item` class that it has to subclass is a parameter to the template.

```
class item {
  private:
    list<item> dependentItems;
    list<certificate> dependentCertificates;
    kds *owner;
  public:
    registerDependentItem(item *i);
    motionChanged();
      // Warns all dependent items and asks all dependent certificates
      // to recompute their failure times.
    ...

class clone2d: public item {
  item2d *getOriginal();
    // Returns the original item this is the clone of
  ...
}

class flock: public kds<item2d> {
  register(kds<clone2d> *k);
    // Each item2d will be distributed to all registered KDSs.
  ...
}
```

Having a KDS be a template based on a variable `item` class is natural and arises in the serial way to combine KDSs also. As we saw, the candidate pairs selected by the cone structures have to be fed into a kinetic tournament. Hence, the kinetic tournament should be able to be based either an `item1d`, or what we call an `itemEdge`.

```
class item1d: public item {
  setMotionEquation(const polynomial &p);
  ...
}

class itemEdge: public item {
  private:
    item2d *fromItem, *toItem;
  public:
    setEndPoints(item2d *i1, item2d *i2);
  ...
}
```

Here is the way we define the closest pair KDS. It needs to be a flock of three cone structures, and the cone structures feed their candidate pairs to a kinetic tournament. We assume we already have a cone structure KDS called `kConeStructure`; it has virtual methods `doCreatePair` and `doDeletePair` that are called whenever a candidate pair appears or disappears. We first subclass it to allow it to output its candidate pairs to a tournament:

```
class kMyConeStructure: public kConeStructure {
private:
  kTournament<itemEdge> *tournament;
  doCreatePair(coneItem *i1, coneItem *i);
  doDeletePair(item *);
public:
  setOutput(kTournament<itemEdge> *);
}

kMyConeStructure::doCreatePair(coneItem *i1, coneItem *i2)
```

```
{
  itemEdge *e = tournament->newItem();
  e->setEndPoints(i1->getOriginal(),i2->getOriginal());
  i1->registerDependentItem(e);
  i2->registerDependentItem(e);
}
```

Once we have this subclass, the closest pair KDS can take three of these cone structures and a minimum-keeping tournament, and plug them together.

```
class kCP: public kFlock {
  kMyConeStructure<clone2d> cones[3];
  kMinTournament<itemEdge>   tournament;
}

kCP::kCP()
{
  for (i = 0; i <= 2; i++) {
    register(&cones[i]);
    cones[i].setOutput(&tournament);
    }
  ...
}
```

If the closest pair is used for collision detection between sets of disks of equal size, the tournament can be subclassed so as to be warned when the minimum changes using a mechanism similar to `doCreatePairs`. We can then recompute the time at which the closest pair is scheduled to collide and insert an event at that time.

5.3 Tests

It is not easy to compare meaningfully a kinetic data structure with any other method, in particular with a fixed time-step method. Indeed, it is always possible create a scenario

n	t_{kin}	t_{min}	t_{avg}
1000	3.7	189.4	0.34
2000	10.3	574.5	0.86
4000	32.2	4756.8	1.97
8000	110.7	14643.6	4.44
16000	559.7	67074.1	11.41
32000	3251.0	372959.4	34.63

Table 5.1: Comparison between the convex hull KDS and a fixed time-step recomputation.

with one very fast object and many very slow objects: a fixed time-step method has to look at all objects at a frequency adapted to the fastest object. The notion of a "typical" scenario is not well defined either.

For lack of a better choice, we consider random items as in Chapter 6: each item is given an initial position and velocity independently drawn from the unit square. We run a kinetic data structure on these items between time 0 and time 1, and record the processing time required to perform the simulation. In order to compare it with a fixed-time-step simulation, we also record the average and the minimum time between two external events. We then calculate the user time required for a fixed-time-step simulation that uses each of these as time-steps. For these latter experiments, we use the ready-to-use routines provided with LEDA 3.6.1 [75].

Results are summarized in Table 5.1 for the convex hull (all quantities are averaged over 10 runs) and in table 5.2 for the Voronoi diagram (all quantities are averaged over 4 runs). All experiments were done on an O2 workstation with a R10000 processor and *-O2* optimized compiling. The meaning of the columns are as follows (all times are in seconds):

n	t_{kin}	t_{min}	t_{avg}
1000	159	1.8E+8	5,166
2000	453	2E+9	14,552
4000	1,328	8E+9	40,967
8000	3,934	1.2E+11	115,498
16000	11,820	5.4E+11	326,737
32000	36,733	5E+12	925,271

Table 5.2: Comparison between the Voronoi diagram KDS and a fixed time-step recomputation.

n Number of points

t_{kin} Time of the KDS,

t_{min} Time necessary to recompute the convex hull/Voronoi diagram from scratch at each time step, if the time step is the minimum time between two external events,

t_{avg} Time necessary to recompute the convex hull/Voronoi diagram from scratch at each time step, if the time step is the average time between two external events.

As can be seen in the tables, a fixed time-step simulation can give widely varying ranges of processing time depending on the time-step parameter. In order to improve a fixed-time-step method, one can think of several modifications to the naïve recompute-from-scratch method. Let us consider the convex hull.

First, one can keep the convex hull at a time step t and verify at the next step if it is still correct. This use of temporal coherence can be found for instance in the collision detection package I-Collide [32]. However, as all the data needs to be examined at each time step, it amounts to replacing an $O(n \log n)$ algorithm by a linear-time verification in most of the cases. This improvement can be significant but the fundamental problem of choosing the time step remains.

Second, one can imagine a number of heuristics to predict an interval during which the attribute under consideration will not change. But isn't this exactly what is done with kinetic data structures?

5.4 Conclusion

As we mentioned in the introduction of this chapter, there is an implementation of all the kinetic data structures described in this thesis, although it does not conform to the descriptions of this chapter. Nevertheless, the implementation gives us a good idea of where we currently stand in terms of two major indicators of the experimental quality of our algorithms: speed and robustness.

Chapter 6

Probabilistic Analysis

Given a set of n points, what is the description complexity of their convex hull? In the world of analysis of algorithms, this question is understood with an implicit "in the worst case", and the answer is $O\big(n^{\lfloor d/2 \rfloor}\big)$, where d is the dimension of the underlying space. This is not entirely satisfactory, as this description complexity can vary tremendously depending on the positions of the points. Another type of answer is to look at the expected description complexity when the points are drawn from a given distribution. This type of analysis, initiated by Rényi and Sulanke [87], is valuable because this expectation is in general much smaller than the worst case, and, more importantly, because it often allows one to design algorithms whose expected running times are much better than the worst case [22, 43].

In this chapter, we study theoretical bounds for the expected number of changes of combinatorial functions of moving points drawn from prescribed distributions, as well as expected time bounds for kinetic data structures that maintain these combinatorial functions.

* * *

We first need to choose a probabilistic model from which to draw a random scenario. The first assumption we make is that the motion parameters of all items are identically and independently distributed. Then comes the choice of distribution of random motion parameters for each item. We consider that the position of an item p at time t is given by

$$p(t) = s_p + tv_p$$

where s_p (the *initial position*) and v_p (the *velocity*) are independently and uniformly distributed according to a probability density g. We say that an item with this random equation of motion is a g-***random item***. In this case, $p(t)$ is also a random variable in the plane whose density is

$$g_t(z) = \int_{\mathbb{R}^2} g(v)g(z - vt)dv \,. \tag{6.1}$$

Let us discuss some other possible choices of probabilistic models. First, we could use a density g' that could be a translation, rotation or uniform scaling of g, but the expectations we are concerned with wouldn't change, because the attributes themselves are invariant under such transformations.

Instead of defining the motion of an item by its initial position and velocity, we could have instead decided to select the initial position (at time 0) and final position (at time 1) of each item according to the density g. This choice makes sense, but it is in fact exactly equivalent to ours. Indeed, the position of an item at time t with initial position p_0 and final position p_1 is:

$$p(t) = (1 - t)p_0 + tp_1$$

If we let $\tilde{p}(u) = (1 + u)p\left(\frac{u}{1+u}\right)$, and let u vary from 0 to $+\infty$, we have:

$$\tilde{p}(u) = p_0 + up_1$$

Therefore, if there is an event at a time t in a model in which each item is defined by an initial position and a final position, there is also an event at time $t/(1 - t)$ in the model in which the final position is transformed into a velocity.

By symmetry, the expected number of events in the initial/final position model is the same during the time interval $[0, 1/2]$ and during the time interval $[1/2, 1]$. Hence, in our model, the expected number of events is the same during time interval $[0, 1]$ and $[1, +\infty]$. In the remainder of this chapter, we consider only events happening between time 0 and 1.

* * *

The general method for the three forthcoming sections is as follows. We describe it for the convex hull for definiteness. We first focus on three items in order to compute the expected number of times they are involved in a change to the convex hull. We identify the condition of this event (no other item to the right of the line defined by the three items when they become collinear), and introduce a change of variable Φ to condition upon the line of collinearity. We bound the Jacobian of the transformation and perform routine integration.

There is a technical detail: the change of variable formula requires Φ to be one-to-one, but the parameterization will cover each triplet of items up to twice (this is because the time at which three items become collinear is the solution of a quadratic equation). To avoid this, we introduce an intermediary space $\mathcal{M}$ on which Φ is one-to-one.

* * *

For completeness, let us start by examining one-dimensional problems. It is straightforward to show that in the present probabilistic setting, and for a set S of n items with random initial position and random velocity in the unit interval, the expected number of changes to a sorted order is $\Theta(n^2)$: indeed, by symmetry, a given pair has probability $\frac{1}{2}$ of ever crossing, and this bound holds for any distribution (it is based only on the ordering of the initial positions and velocities, and relies only on independence).

The case of the maximum is once again more difficult, and the exact bounds depend on the distribution. However, a duality argument allows us to link it directly with existing static results. Indeed, the trajectory of an item $s \in S$ in time/space is a line $y = a_s t + b_s$, where a_s, b_s are independently drawn from a fixed distribution. By a standard duality, the upper envelope of all the lines becomes the upper convex hull of their dual points. Each item, in the dual, is a point (a_s, b_s), and each coordinate is independently distributed according to the original distribution. There are two cases in which we know the result directly: if the distribution is uniform on an interval, then the convex hull has complexity $\Theta(\log n)$. If the distribution is a Gaussian, then the complexity of the convex hull is $\Theta(\sqrt{\log n})$. This technique does not generalize to the convex hull in two dimensions, and this case will be treated in Section 6.3.

We note also that, in one dimension, the Voronoi diagram is nothing but the sorted order, and it therefore changes quadratically many times in our model.

6.1 Closest Pair

In this section, we assume that we have a bounded density g with compact support.

A change in the closest pair description happens at a time t if the distance between two items p_1, q_1 is equal to the distance between two items p_2, q_2, and there is no pair at a smaller distance at t. We first focus on a given quadruplet $P = (p_1, q_1, p_2, q_2)$ of independent g-random items, and an additional tuple U of n other independent g-random items. We compute the probability that the quadruplet P defines an event for the closest pair.

More precisely, the equation $\mathrm{d}(p_1(t), q_1(t)) = \mathrm{d}(p_2(t), q_2(t))$ is quadratic in t, so it can have up to two solutions. Hence, we are not going to compute the probability that P triggers a change in the closest pair, but the expected number of times it can do so, i.e.:

$$E\Big[\big|\{t \mid \mathrm{d}(p_1(t), q_1(t)) = \mathrm{d}(p_2(t), q_2(t)) \wedge \mathrm{d}(p_1(t), q_1(t)) = M_t(P, U)\}\big|\Big] \tag{6.2}$$

where $M_t(P, U)$ is the smallest pairwise distance amongst $P \cup U$ at t.

It is inconvenient to work with the expression above. A more convenient replacement for (6.2) can be obtained if we define:

$$\mathcal{M} = \big\{(p_1, p_2, q_1, q_2, t) \in \mathbb{R}^{16} \times [0, 1] \mid \mathrm{d}(p_1(t), q_1(t)) = \mathrm{d}(p_2(t), q_2(t))\big\}$$

For $\xi \in \mathcal{M}$, denote by P_ξ the associated quadruplet of item parameters, t_ξ the associated time, and r_ξ the common distance between p_i and q_i at time t_ξ. We equip $\mathcal{M}$ with the measure μ induced from $\mathbb{R}^{16}$, i.e.,

$$\int_{\mathcal{M}} f(\xi) d\mu = \int_{\xi \in \mathcal{M}} f(\xi) g(P_\xi)\, dP_\xi\,,$$

where $g(P_\xi)$ is a shorthand for the product of the densities of each item of the tuple P_ξ. In this case, the expected number of changes to the closest pair due to P is:

$$\Lambda = \int_{(\xi, U) \in \mathcal{M} \times \mathbb{R}^{4n}} [M_{t_\xi}(P_\xi, U) \geq r_\xi] g_{t_\xi}(P_\xi) g_{t_\xi}(U)\, dP_\xi\, dU \tag{6.3}$$

The computations to come are rather involved but the idea is simple. We compute the expectation conditioned upon the distance r between the closest pair when the event happens. For the upper bound, we replace $M_{t_\xi}(P_\xi, U)$ by $M_{t_\xi}(U)$ and integrate first with respect to U. The quantity

$$\int_U [M_{t_\xi}(U) > r_\xi] g_{t_\xi}(U)\, dU\,,$$

in which r_ξ and t_ξ are fixed, is the probability that the closest distance of a set of n random points is greater than a given quantity at a time t_ξ. It is directly given by Theorem 2.9, from which we obtain

$$\int_U [M_{t_\xi}(U) > r_\xi] g_{t_\xi}(U)\, dU = \exp(-c_{g_{t_\xi}} n^2 {r_\xi}^2)(1 + o(1))\,, \tag{6.4}$$

where $c_{g_{t_\xi}}$ is a constant that depends on t_ξ. We let

$$C = \min_t c g_t\,.$$

The quantity (6.4) is at most

$$\exp\left(-Cn^2 {r_\xi}^2\right)(1 + o(1))\,.$$

If we put this bound in (6.3), we obtain:

$$\Lambda \le (1 + o(1)) \int_{\xi \in \mathcal{M}} \exp\left(-Cn^2 {r_\xi}^2\right) g_{t_\xi}(P_\xi)\, dP_\xi\,. \tag{6.5}$$

The positions of p_1, q_1, p_2, q_2 are not independent conditioned on r_ξ, so it is necessary to do a change of variable to account for the dependencies. We express the position of q_i in terms of its distance to p_i at time t. That is, we replace the variables (s_{q_1}, s_{q_2}) in the integral by the variables $(t, r, \theta_1, \theta_2)$, with the following transformation Φ (Figure 6.1):

$$\Phi : \begin{cases} s_{q_1} = s_{p_1} + tv_{p_1} + r \binom{\cos\theta_1}{\sin\theta_1} - tv_{q_1} \\ s_{q_2} = s_{p_2} + tv_{p_2} + r \binom{\cos\theta_2}{\sin\theta_2} - tv_{q_2} \end{cases}$$

Note that although two points in the domain of Φ can give the same quadruplet of item parameters, they map to different points on $\mathcal{M}$, so that Φ is one-to-one. This is the main reason to introduce the space $\mathcal{M}$. The Jacobian of this transformation, obtained by

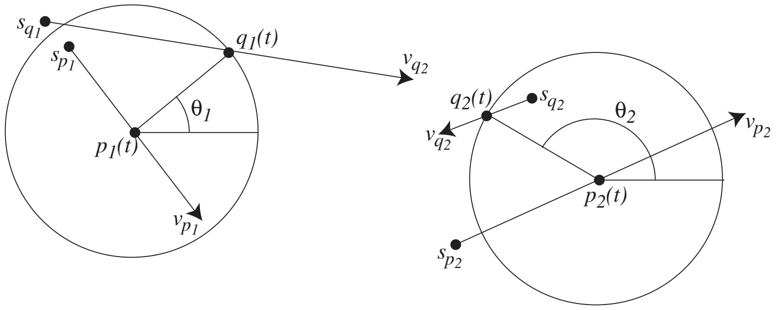


Figure 6.1: Parameterization for the closest pair.

computing all partial derivatives, is:

$$J\Phi = r^2 \begin{vmatrix} v^x_{p_1} - v^x_{q_1} & \cos\theta_1 & -\sin\theta_1 & 0 \\ v^y_{p_1} - v^y_{q_1} & \sin\theta_1 & \cos\theta_1 & 0 \\ v^x_{p_2} - v^x_{q_2} & \cos\theta_2 & 0 & -\sin\theta_2 \\ v^y_{p_2} - v^y_{q_2} & \sin\theta_2 & 0 & \cos\theta_2 \end{vmatrix}$$

As all terms in the determinant are $O(1)$, we obtain:

$$|J\Phi| = O(r^2)$$

We restart from (6.5) and apply the parameterization just described with the change of variable formula (Theorem 2.10). We have:

$$\begin{aligned} \Lambda &\leq (1+o(1)) \int |J\Phi| e^{-Cn^2r^2} \prod_{i=1,2} \left(g(s_{p_i})g(v_{p_i})g(s_{q_i})g(v_{q_i})\, ds_{p_i}\, dv_{p_i}\, dv_{q_i}\, d\theta_i\right)\, dt\, dr \\ &= O(1) \int_r r^2 e^{-Cn^2r^2}\, dr \end{aligned} \tag{6.6}$$

The passage to the last line is done by integrating with respect to all variables except r. The last integral is $\Theta(1/n^3)$ by Proposition 6.5 (see Appendix).

* * *

This upper bound is matched by a lower bound up to a constant. To show this, our goal is to replace $[M_{t_\xi}(P_\xi, U)]$ by $[M_{t_\xi}(U)]$ because Theorem 2.9 gives a lower bound on integrals involving the latter quantity. To do this, we focus on a subdomain in which the two quantities are the same, making sure that the subdomain is large enough to capture a positive fraction of the probability content we want to estimate.

Let us restart from (6.3) and restrict once and for all the domain of integration to

$$r_\xi \leq \frac{1}{n}$$

In this domain, we can write the following logical inclusion:

$$[M_{t_\xi}(U, P_\xi) \geq r_\xi] \supset [M_{t_\xi}(U) \geq r_\xi] \wedge [\forall u \in U, \mathrm{d}(u, p_i) \geq \frac{3}{n}] \wedge [\mathrm{d}(p_1, p_2) \geq \frac{3}{n}]. \tag{6.7}$$

In words, if point u is more than $3/n$ away from p_i, then it is more than r_ξ away from q_i. We'll see that we don't loose much if we replace the left hand side by the right hand side in (6.3). Consider now, for a fixed ξ, the integral

$$\begin{aligned} & \int_U [M_{t_\xi}(U, P_\xi) \geq r_\xi][\forall u \in U, \mathrm{d}(u, p_i) \geq \frac{3}{n}] g_{t_\xi}(U)\, dU \\ = \; & \Big(\int_U [\forall u \in U, \mathrm{d}(u, p_i) \geq \frac{3}{n}]\, dU\Big) \int_U [M_{t_\xi}(U, P_\xi) \geq r_\xi] h_\xi(U)\, dU \end{aligned} \tag{6.8}$$

where

$$h_\xi(U) = \frac{[\forall u \in U, \mathrm{d}(u, p_i) \geq \frac{3}{n}] g_{t_\xi}(U)}{\int_U [\forall u \in U, \mathrm{d}(u, p_i) \geq \frac{3}{n}]\, dU}$$

is the density for U at t_ξ conditionally on the fact that no item is too close to any point of p_i or q_i.

The right term in (6.8) is the probability that $M_{t_\xi}(U)$ is greater than r_ξ conditioned upon no point of U being too close to p_i or q_i at t. By Theorem 2.9, this quantity is lower-bounded by

$$exp\big(-c'_\xi n^2 {r_\xi}^2\big)(1 + o(1)).$$

The support of h_ξ is bounded and its maximum value is also bounded. Hence, c'_ξ is bounded from above by a constant, which we denote C'. Using (6.7), we have:

$$\Lambda \geq \int_\xi e^{-C' n^2 {r_\xi}^2} [\mathrm{d}(p_1, p_2) > \frac{3}{n}] g(P_\xi)\, dP_\xi$$

where the domain of integration is, let's repeat it, restricted to $r_\xi < \frac{1}{n}$. We can further restrict the domain so as to make $[\mathrm{d}(p_1, p_2) > \frac{3}{n}]$ true. For instance, we can look only at those samples in which p_1 is in the top third of the support of g_t and p_2 in the bottom third. After a change of variable, we have a Jacobian in the integral, but this Jacobian vanishes on a set of measure 0. By symmetry, the domain of integration can be further refined so that the Jacobian is always $\Omega(r^2)$. The result is an expression like (6.6), which gives the bound of $\Theta\left(\frac{1}{n^3}\right)$. In summary, we obtain the lemma:

Lemma 6.1. *Let $P = (p_1, p_2, q_1, q_2)$ be a quadruplet of independent g-random items and U be a set of n more. The expected number of times $\mathrm{d}(p(t), q_1(t)) = \mathrm{d}(p(t), q_2(t))$ with no other pair at a smaller distance is $\Theta(1/n^3)$.*

There is another case that is not addressed in the above: when the closest pair (p, q_1) changes to the closest pair (p, q_2). We need another lemma for this case. The proof proceeds similarly to the one above.

Lemma 6.2. *Let $P = (p, q_1, q_2)$ be a triplet of independent g-random items and U be a set of n more. The expected number of times $\mathrm{d}(p(t), q_1(t)) = \mathrm{d}(p(t), q_2(t))$ with no other pair at a smaller distance is $\Theta(1/n^3)$.*

We now use linearity of expectation to sum over all possible triplets and quadruplets of items to obtain:

Theorem 6.3. *Let U be a set of n independent g-random items in the plane, where g is bounded with compact support. The expected number of changes to their closest pair is $\Theta(n)$.*

6.2 Voronoi Diagram

The Voronoi diagram of a set of items in motion changes when 4 items become cocircular, and no other item is inside their circle of cocircularity at that time [58]. We consider once again four distinguished g-random items (p_1, p_2, p_3, p_4), and compute the expected number of changes to the Voronoi diagram due to these items, in the presence of a set U of n other g-random items.

The ideas are very similar to that of the previous section, and even more similar to the original treatment of Rényi and Sulanke mentioned in Section 2.1.3. Let us give here the general idea. We condition the probability that our quadruplet generates an event upon the circle along which the cocircularity happens. Say that the radius of this circle is r. Most cases happen when the circle of cocircularity is very small and around the center of the distribution. In this case, the probability that an item u is inside a circle of radius r is πr^2, and hence the probability that this circle is empty is roughly $e^{-\pi n r^2}$ by our independence assumption. The probability that each item p_i is "on" the circle is proportional to its perimeter, that is to say, it is $2\pi r$. Hence the conditional density of r, the radius of the circle of cocircularity, is about $\Theta\left(r^4\right)$, and the probability that our quadruplet generates an event is about

$$\Theta(1) \int e^{-\pi n r^2} r^4 \, dr \tag{6.9}$$

This integral is $\Theta\left(\frac{1}{n^{5/2}}\right)$ by Proposition 6.5. Summing over all $\Theta\left(n^4\right)$ quadruplets, we obtain that the expected number of changes to the Voronoi diagram is $\Theta\left(n^{3/2}\right)$.

From this back of the envelope calculation, we have the following conjecture:

Conjecture 6.4. *Let g be a bounded density with compact support. The Voronoi diagram of n independent g-random items changes $\Theta\left(n^{\frac{3}{2}}\right)$ times.*

Alas, the computations are much heavier than this synopsis suggests. There are two main reasons for this. First, we need to perform a parameterization as in the closest pair problem. Second, we need to treat separately a lot of boundary cases. For instance, if we let g be uniform on the unit square $[0,1]^2$, the density g_t is defined by different formulas in the corners, the sides, and the center. More precisely,

$$g_t(x,y) = h_t(x) h_t(y) \tag{6.10}$$

where:

$$h_t(z) = \begin{cases} \frac{z}{t} & \text{if } z \leq t \\ 1 & \text{if } t \leq z \leq 1 \\ \frac{1+t-z}{t} & \text{if } 1 \leq z \geq 1+t \end{cases}$$

Hence, we need to treat separately many cases, depending on whether the circle of cocircularity touches the lines $x = t$, $y = t$, $x = 1$, $y = 1$ and/or the boundary of the distribution

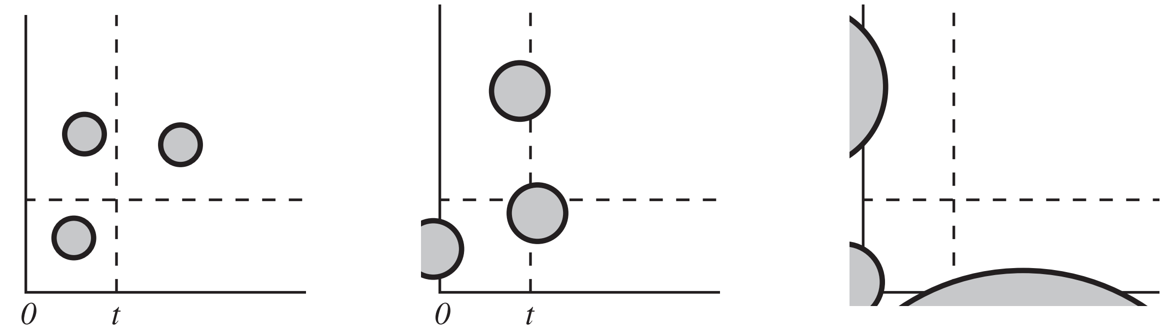


Figure 6.2: Some of the numerous cases for the circle of cocircularity. For those circles that touch the boundary, the probability content can be arbitrarily small even if their radius is large.

(Figure 6.2).

We lack a general way to take care of these boundary cases all at once. This also happens in the static setting [43], although the special cases are less numerous in this latter setting. In this section, we consider a specific distribution (the unit square), and we do not treat the boundary cases. Therefore, we obtain only a partial result. Treating all the boundary cases would add an unreasonable amount of technical details to this thesis. Moreover, Dwyer [39] has very recently obtained the full result for the disk distribution, which makes a full treatment of the square case less interesting.

* * *

Let D be a disk of center (x, y) and radius r. Let $G_t(x, y, r)$ denote the probability content of D at time t:

$$G_t(x, y, r) = \int [z \in D] g_t(z) \, dz \, .$$

The cocircularity test for four items is a degree 4 algebraic equation in t. To deal with the multiplicity of events triggered by the same quadruplet, we define once again:

$$\mathcal{M} = \left\{ (p_1, p_2, p_3, p_4, t) \in \mathbb{R}^{16} \times [0, 1] \mid p_1(t), p_2(t), p_3(t), p_4(t) \text{ cocircular} \right\} \, .$$

We parameterize $\mathcal{M}$ as follows. We replace the initial positions of the four items (8

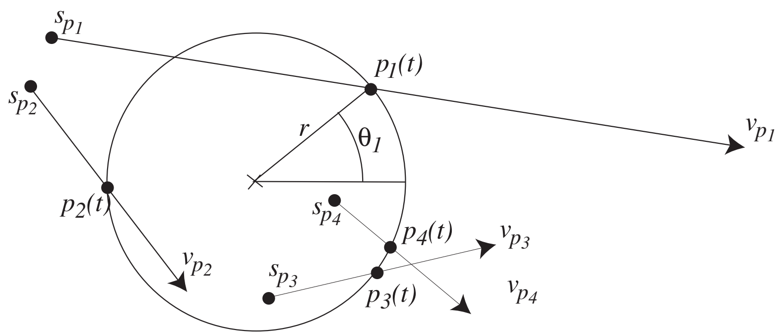


Figure 6.3: Parameterization of each moving item with respect to the disk of cocircularity.

dimensions) by (Figure 6.3):

$$\begin{cases} x, y & \text{positions of the center of the circle of cocircularity,} \\ r & \text{radius of the circle of cocircularity,} \\ t & \text{time of the event,} \\ \theta_i & \text{angle between the horizontal and the vector } p_i(t) - (x, y)\,. \end{cases}$$

More precisely, we have:

$$s_{p_i} = \begin{pmatrix} x \\ y \end{pmatrix} + r \begin{pmatrix} \cos\theta_i \\ \sin\theta_i \end{pmatrix} - t v_{p_i}$$

where $v_{p_i} = (v^x_{p_i}, v^y_{p_i})$ is the velocity of item p_i. We denote by Φ this parameterization. It is one-to-one, and we can apply the change of variable formula (Theorem 2.10). The expected number of events is

$$\Lambda = \int (1 - G_t(x, y, r))^n |J\Phi| \prod_i \left(g(v_{p_i}) g(s_{p_i})\, dv_{p_i}\, d\theta_i\right)\, dx\, dy\, dr\, dt \tag{6.11}$$

where $J\Phi$ and s_{p_i} depend on all the parameters of integration.

What is the Jacobian of Φ? Let $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$. Computing all the first derivatives gives:

$$
J\Phi = \begin{vmatrix}
1 & 0 & -v_1^x & c_1 & -rs_1 & 0 & 0 & 0 \\
1 & 0 & -v_2^x & c_2 & 0 & -rs_2 & 0 & 0 \\
1 & 0 & -v_3^x & c_3 & 0 & 0 & -rs_3 & 0 \\
1 & 0 & -v_4^x & c_4 & 0 & 0 & 0 & -rs_4 \\
0 & 1 & -v_1^y & s_1 & rc_1 & 0 & 0 & 0 \\
0 & 1 & -v_2^y & s_2 & 0 & rc_2 & 0 & 0 \\
0 & 1 & -v_3^y & s_3 & 0 & 0 & rc_3 & 0 \\
0 & 1 & -v_4^y & s_4 & 0 & 0 & 0 & rc_4
\end{vmatrix}
$$

If we group line i with line $i+4$, we obtain:

$$
J\Phi = r^4 \begin{vmatrix}
v_1^x c_1 + v_1^y s_1 & s_1 & c_1 & 1 \\
v_2^x c_2 + v_2^y s_2 & s_2 & c_2 & 1 \\
v_3^x c_3 + v_3^y s_3 & s_3 & c_3 & 1 \\
v_4^x c_4 + v_4^y s_4 & s_4 & c_4 & 1
\end{vmatrix}
$$

We now restrict our domain of integration to one representative of the most common case. We consider only events happening between time 0 and time 1/2, whose circle of cocircularity is entirely contained in the square $[1/2, 1]^2$. The density g_t on this square is equal to 1 on the time interval considered (Equation 6.10), therefore, $G_t(x, y, t)$ is equal to πr^2.

Note also that the Jacobian can be bounded by $O\big(r^4\big)$. Hence, restarting from (6.11), and denoting by $\bar{\Lambda}$ the part of Λ restricted to our domain of integration defined above, we have:

$$
\bar{\Lambda} = O(1) \int (1 - \pi r^2)^n r^4 \prod_i \left(g(v_{p_i}) g(s_{p_i})\, dv_{p_i}\, d\theta_i\right)\, dx\, dy\, dr\, dt
$$

After integration along v_{p_i}, θ_i, x, y and t, we are left with an integral like (6.9). Table 6.1 gives an indication that the total number of events is indeed of the order of $n^{3/2}$.

n	nb events	$1.29n^{3/2}$
$1,000$	$40,899$	$40,793$
$2,000$	$116,993$	$115,381$
$4,000$	$328,558$	$326,347$
$8,000$	$919,680$	$923,048$
$16,000$	$2,602,173$	$2,610,776$
$32,000$	$7,388,052$	$7,384,391$

Table 6.1: Average number of changes to the Voronoi diagram. The "nb events" column gives the number of changes to the Voronoi diagram obtained by simulation. The third column gives the predicted number of events (the constant is adjusted to match the last line approximately).

6.3 Convex Hull

The techniques for the convex hull are very similar to the ones used above for the Voronoi diagram, but here, it is the boundaries that play the crucial role. Hence, the results are very dependent on the distribution the items are drawn from. In this section, we consider that all items' initial position and velocity are drawn uniformly independently at random from the unit square.

Here again, there are many special cases, and, as for the Voronoi diagram, we compute only one of them, which we believe is the dominant one: the case when the items are very close to a corner. This is the dominant case in the static setting.

An event for the convex hull happens when three items become collinear, and all other items are on the same side of the line defined by the three items. By symmetry, it is enough to consider only events that happen in the lower left corner, i.e., events such that the line of collinearity cuts the two axes at positive coordinates, and such that all other items are above that line. We consider three specific items p_1, p_2, p_3 and a set U of n other g-random items. We proceed to compute the expected number of changes caused by the collinearity of the p_i's.

As is now usual, we define the space

$$\mathcal{M} = \left\{(p_1, p_2, p_3, t) \in \mathbb{R}^{12} \times [0, 1] \mid p_1(t), p_2(t), p_3(t) \text{ collinear}\right\}$$

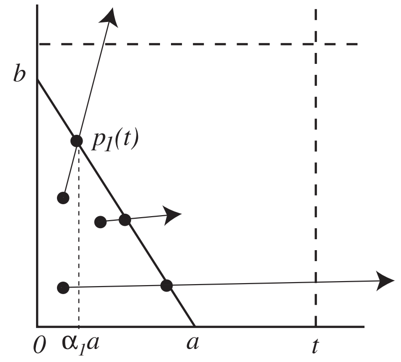


Figure 6.4: Parameterization of each moving item with respect to the line of collinearity.

and parameterize it as follows. We replace the initial position of the three items by (Figure 6.4):

$$\begin{cases} a & \text{intersection of the line of collinearity with } (Ox), \\ b & \text{intersection of the line of collinearity with } (Oy), \\ \alpha_i & \text{position of } p_i \text{ on the line of collinearity.} \end{cases}$$

More precisely, we define:

$$s_{p_i} = \begin{pmatrix} \alpha_i a \\ (1-\alpha_i)b \end{pmatrix} - tv_{p_i} \,,$$

where $v_{p_i} = (v_{p_i}^x, v_{p_i}^y)$ is the velocity of item p_i. We denote this transformation by Φ.

Given two reals a, b, denote by ℓ_{ab} the line that intersects (Ox) at a and (Oy) at b. Let $G_t(a,b)$ be the probability content below ℓ_{ab} at t, and $A_t(a,b)$ the probability content on ℓ_{ab}.

$$\begin{aligned} G_t(a,b) &= \int [(x,y) \text{ below } \ell_{ab}] g_t(x,y)\, dx\, dy \\ A_t(a,b) &= \int g_t(\alpha a, (1-\alpha)b)\, d\alpha \end{aligned}$$

The expected number of events due to p_1, p_2, p_3 is:

$$\Lambda = \int (1 - G_t(a,b))^n |J\Phi| \prod_i (g(s_{p_i}) g(v_{p_i})\, d\alpha_i\, dv_{p_i})\, da\, db\, dt$$

Let us assume for now that we have an upper bound $B(a,b,t)$ for the Jacobian of Φ that depends only on a,b,t. Once we replace $|J\Phi|$ by its upper bound, the only terms dependent on v_{p_i} are $g(s_{p_i})$ and $g(v_{p_i})$. Moreover,

$$\begin{aligned}\int g(s_{p_i})g(v_{p_i})\,dv_{p_i} &= \int g(p_i(t)-v_{p_i}t)g(v_{p_i})\,dv_{p_i}\\ &= g_t(p_i(t)) \quad \text{(Formula 6.1)}\\ &= g_t(\alpha_i a,(1-\alpha_i)b)\,.\end{aligned}$$

Hence, integrating with respect to each v_{p_i}, then with respect to each α_i, gives:

$$\begin{aligned}\Lambda &\le \int (1-G_t(a,b))^n B(a,b,t)\prod_i \left(g_t(\alpha_i a,(1-\alpha_i)b)\,d\alpha_i\right)\,da\,db\,dt\\ &= \int (1-G_t(a,b))^n B(a,b,t)A_t(a,b)^3\,da\,db\,dt\\ &\approx \int e^{-nG_t(a,b)}B(a,b,t)A_t(a,b)^3\,da\,db\,dt\,. \end{aligned} \tag{6.12}$$

We proceed to compute an upper bound on the Jacobian of Φ, which is

$$J\Phi = \begin{vmatrix} \alpha_1 & 0 & -v_1^x & a & 0 & 0\\ \alpha_2 & 0 & -v_2^x & 0 & a & 0\\ \alpha_3 & 0 & -v_3^x & 0 & 0 & a\\ 0 & 1-\alpha_1 & -v_1^y & -b & 0 & 0\\ 0 & 1-\alpha_2 & -v_2^y & 0 & -b & 0\\ 0 & 1-\alpha_3 & -v_3^y & 0 & 0 & -b \end{vmatrix}.$$

If we group the i-th row with the $i+3$-th row, we obtain:

$$J\Phi = ab\begin{vmatrix} \alpha_1 & 1-\alpha_1 & bv_1^x+av_1^y\\ \alpha_2 & 1-\alpha_2 & bv_2^x+av_2^y\\ \alpha_3 & 1-\alpha_3 & bv_3^x+av_3^y \end{vmatrix} = ab\left(b\begin{vmatrix} \alpha_1 & 1 & v_1^x\\ \alpha_2 & 1 & v_2^x\\ \alpha_3 & 1 & v_3^x \end{vmatrix} + a\begin{vmatrix} \alpha_1 & 1 & v_1^y\\ \alpha_2 & 1 & v_2^y\\ \alpha_3 & 1 & v_3^y \end{vmatrix}\right).$$

We wish to obtain an upper bound on the Jacobian. To this end, notice that, as $p_i^x(t)=\alpha_i a$, the x-speed is at most $\alpha_i a/t$ (and also bounded by 1), and $\alpha_i \le 1$. Therefore, bounding a determinant by the product of the Euclidian norms of the column vectors:

$$\begin{vmatrix} \alpha_1 & 1 & v_1^x \\ \alpha_2 & 1 & v_2^x \\ \alpha_3 & 1 & v_3^x \end{vmatrix} \leq 3\sqrt{3}\max(v_1^x, v_2^x, v_3^x) \leq 3\sqrt{3}\min(a/t, 1)$$

Hence the determinant is at most:

$$\begin{aligned} |J\Phi| &\leq 3\sqrt{3}ab\,(b\min(a/t,1) + a\min(b/t,1)) \\ &= O\left(ab\min(\frac{ab}{t}, a+b)\right) . \end{aligned}$$

As mentioned above, we consider only one case: when both a and b are less than t. By symmetry, we can assume that $0 < a < b < t$, and we denote by $\bar{\Lambda}$ the integral restricted to this domain. In this case, integrating 6.10 gives:

$$\begin{aligned} G_t(a,b) &= \Theta\left(\frac{a^2b^2}{t^2}\right) , \\ A_t(a,b) &= \Theta\left(\frac{ab}{t^2}\right) . \end{aligned}$$

Then, putting the appropriate bounds in (6.12):

$$\bar{\Lambda} = \int e^{-\Theta(1)n\frac{a^2b^2}{t^2}}\frac{a^2b^2}{t}\left(\frac{ab}{t^2}\right)^3 [0 < a < b < t < 1]\, da\, db\, dt$$

Integrating with respect to a (Proposition 6.5), we obtain

$$\bar{\Lambda} = O(1)\int \frac{b^5}{t^7}\min\left(\left(\frac{t^2}{nb^2}\right)^3, b^6\right) [0 < b < t < 1]\, db\, dt$$

which gives $\Theta(\log^2 n/n^3)$ (see the Appendix).

Summing over all triples of items, we obtain the result that the expected number of changes to the convex hull in a corner is $O(\log^2 n)$.

* * *

The results we obtained for the expected number of changes to the convex hull of items

n	Nb events	Nb ext	$5.28n$	$1.57\ln^2 n$
1,000	5,193	51	5,280	75
2,000	10,537	67	10,560	91
4,000	21,207	97	21,120	108
8,000	42,650	125	42,240	127
16,000	84,804	132	84,480	147
32,000	169,052	169	168,960	169

Table 6.2: Using the implementation described in Chapter 5, we compute the number of external and internal events for the convex hull, and compare them with the theoretical bounds obtained in this chapter. The constants are adjusted to make the last line match approximately.

drawn from the uniform square distribution directly imply an expected linear bound for the convex hull kinetic data structure described in Section 4.1. This data structure divides the point set arbitrarily into a blue and a red half, recursively computes the blue and red convex hulls, and maintains a set of certificates between edge-vertex pairs and edge-edge pairs to certify the convex hull of the whole set. If N_b and N_r are the (random) total number of edges that ever appear on the blue and red convex hulls, the number of events involved at the top level is $O(N_b N_r)$. As these two quantities are independent, the expectation is $E(N_b N_r) = E(N_b)E(N_r) = \Theta(\log^4 n)$ in the case of the square distribution. The expected running time $T(n)$ of the whole data structure thus obeys the recurrence $T(n) = O(\log^4 n) + 2T(n/2)$, which solves to $T(n) = O(n)$. These results make it clear that it is not a good idea to use the Delaunay triangulation (the dual of the Voronoi diagram) to maintain the convex hull. Table 6.2 gives the experimental number of events, both external and internal, showing a close match with the predicted values.

6.4 Conclusion

In this chapter we initiated the study of average case behavior of some combinatorial properties defined on points moving on random trajectories. It remains to see whether it is possible to develop a technique that gives full results for the Voronoi diagram and the convex hull without the need of a cumbersome case analysis. It would also be interesting to generalize these results to d dimensions. Preliminary calculations show that the closest pair changes $\Theta\left(n^{\frac{2}{d}}\right)$ times, the Voronoi diagram changes $\Theta\left(n^{1+\frac{1}{d}}\right)$ times, and the convex hull changes $\Theta\left(\log^d n\right)$ times for the square distribution.

6.5 Appendix

Proposition 6.5. *Let*

$$W = \int_0^a x^\mu e^{-cx^\nu}\, dx$$

Then,

$$\begin{aligned} W &\leq \frac{1}{\nu}\Gamma\left(\frac{\mu+1}{\nu}\right)\left(c^{-\frac{\mu+1}{\nu}}\right) \\ W &\leq \frac{a^{\mu+1}}{\mu+1} \end{aligned}$$

Proof. By a simple change of variable. Let $y = cx^\nu$. Then $x = (y/c)^{1/\nu}$ and $dx = (y/c)^{1/\nu} dy/(y\nu)$. Hence we have:

$$\begin{aligned} W &= \int_0^{ca^\nu} \left(\frac{y}{c}\right)^{\mu/\nu} e^{-y} \left(\frac{y}{c}\right)^{1/\nu} \frac{dy}{y\nu} \\ &= \frac{1}{\nu c^{\frac{\mu+1}{\nu}}} \int_0^{ca^\nu} e^{-y} y^{\frac{\mu+1}{\nu}-1}\, dy \end{aligned}$$

And the integral is now the incomplete Gamma function. For the second inequality, simply drop the exponential term. ☐

We also have:

$$I = \int \frac{b^5}{t^7} \min\left(\left(\frac{t^2}{nb^2}\right)^3, b^6\right) [0 < b < t < 1]\, db\, dt = \Theta\left(\log^2 n/n^3\right) .$$

We compute the integral in three pieces:

$$
\begin{aligned}
I_1 &= \int \cdots [0 < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < b < t < 1] \\
I_2 &= \int \cdots [0 < b < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < t < 1] \\
I_3 &= \int \cdots [0 < b < t < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < 1]
\end{aligned}
$$

$$
\begin{aligned}
I_1 &= \int \frac{b^5}{t^7} \frac{t^6}{n^3 b^6} [\frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < b < t < 1] \, db \, dt \\
&= \frac{1}{n^3} \int \frac{1}{tb} [\frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < b < t < 1] \, db \, dt \\
&= \frac{1}{n^3} \int \frac{1}{t} (\ln t - \ln \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}}) [\frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < t < 1] \, dt \\
&= \frac{1}{n^3} \int \left(\frac{\ln t}{2t} + \frac{\ln n}{4t} \right) [\frac{1}{\sqrt{n}} < t < 1] \, dt \\
&= \frac{1}{n^3} \left(\frac{1}{4} \ln^2 t |_{\frac{1}{\sqrt{n}}}^1 + \frac{\ln n}{4} \ln t |_{\frac{1}{\sqrt{n}}}^1 \right) \\
&= \frac{\ln^2 n}{16 n^3}
\end{aligned}
$$

$$
\begin{aligned}
I_2 &= \int \frac{b^5}{t^7} b^6 [0 < b < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < t < 1] \, db \, dt \\
&= \int \frac{b^{11}}{t^7} [0 < b < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < t < 1] \, db \, dt \\
&= \frac{1}{12} \int \frac{1}{t^7} \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}}^{12} [\frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < t < 1] \, dt \\
&= \frac{1}{12 n^3} \int \frac{1}{t} [\frac{1}{\sqrt{n}} < t < 1] \, dt \\
&= \frac{\ln n}{24 n^3}
\end{aligned}
$$

$$\begin{aligned}
I_3 &= \int \frac{b^5}{t^7} b^6 [0 < b < t < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < 1] \, db \, dt \\
&= \int \frac{1}{t^7} \frac{t^{12}}{12} [t < \frac{t^{\frac{1}{2}}}{n^{\frac{1}{4}}} < 1] \, dt \\
&= \int \frac{t^5}{12} [t < \frac{1}{\sqrt{n}}] \, dt \\
&= \frac{1}{72n^3}
\end{aligned}$$

Chapter 7

Conclusion

A wide range of problems involving motion requires the computation of certain attributes of the moving data at discrete time steps. For instance, to render a dynamic scene, it is necessary to compute all visible objects at every frame. In a physical simulation, we need to detect collisions, and we do this by checking as often as possible if any pair of objects intersect.

More generally, questions involving motion can often be worded as "keeping track of a discrete attribute depending on the objects' positions." The discrete attribute might be the closest pair of objects (for collision detection purposes) or some visibility data structure (for rendering purposes). For such problems, the goal is to exploit the time coherence given by the continuity of motion, to avoid recomputations from scratch at every time step.

In this thesis, we presented a general approach to address such problems. We exhibited a ***kinetization strategy*** based on animating a proof of correctness of a discrete attribute. We also defined a framework for analysis, in which different solutions to a given problem can be compared on theoretical grounds. More practically, as in traditional algorithm design, the analysis framework guides the design of efficient kinetic data structures.

Essentially, kinetic data structures are a new type of data structures, for which there is a craft of design, as exhibited by the closest pair problem (Section 4.2), and general analysis techniques, as shown by the questions involving the maximum maintenance (Chapter 3) and the convex hull problem (Section 4.1).

In the same way as one can ask how to maintain dynamically a discrete attribute upon item insertion and deletion, or how one can obtain an efficient algorithm to compute this discrete attribute in a parallel model of computation, it is now possible to ask how to keep track of such a discrete attribute *kinetically*, in the precise sense described in this thesis. We summarize recent results at the end of this conclusion.

There are many issues that require further research in this domain, and we start exploring a few of them here.

7.1 Competitive Ratio

The notion of competitive ratio underpins the field of on-line algorithms. The idea is to design an algorithm whose output is provably within a certain factor of an unattainable optimal solution. In on-line algorithms, the optimal solution is unattainable because it requires knowing the future. There is a similar notion in approximation algorithms, in which the output is unattainable because it is hard to compute.

It is tempting to define the same competitive ratio for the running time of a kinetic data structure. Consider an attribute A, and a kinetic data structure K. The competitive ratio of K for a given class of motion is given by

$$\max \frac{\text{events}K}{\text{events}A}$$

where the max is taken over all possible scenarios. In words, we wish to bound the maximum of the ratio instead of the ratio of the maxima—the latter is what we settled for in this thesis. Clearly, it would be marvelous to obtain a local KDS with small competitive ratio.

Why didn't we use this superb measure of quality in this thesis? There is one simple reason: it proved impossible to design a competitive KDS. Apart from the self-kinetizing attributes, all KDSs we could obtain have the same horrible competitive ratio, attained when the worst-case number of changes to the KDS is balanced by no change in the attribute to be maintained. Because of that, the competitive ratio has so far proven to be useless.

Is this due to an oversight on our part? Maybe. For instance, Edelsbrunner and Welzl [48] show a way to compute the median of n numbers that change at constant velocity with an algorithm that can be seen as competitive: after $O(n \log n)$ preprocessing, there are only external events, which are processed in $O(\log^2 n)$ time a piece. Moreover, the structure is local, in that a change of velocity takes only $O(\log^2 n)$ time. We therefore have a truly competitive kinetic data structure, which was invented more than ten years ago.

However, this kinetic data structure is based on the maintenance of the convex hull of the dual of the straight line trajectories followed by the moving values in space-time. It therefore doesn't generalize to algebraic motion. With today's knowledge, we see this example as an isolated lucky case.

It may be the case, on the other hand, that certain kinetic data structures always have more events than others. For instance, the Delaunay triangulation KDS can be used to keep track of the convex hull. What is the competitive ratio between this KDS and the KDS we presented in Chapter 4? Alas, once again, it is possible to construct an example for which the Delaunay triangulation barely changes while our KDS changes quadratically many times, and vice versa.

The question of the existence of a competitive kinetic data structure can be asked non-constructively in an off-line setting. For a certain attribute A and a scenario (π_t), is it the case that *some* kinetic data structure exists that keeps track of A competitively? Conversely, can we construct a scenario in which A changes few times but in which any kinetic data structure within a certain class changes many times? Results of this latter form have recently been obtained for kinetic binary space partitions and kinetic triangulations [4].

7.2 Kinetization and Parallelization

We can view our kinetization process as starting from a proof of correctness of a static configuration function, and then "animating this proof through time." Not all proofs are equally good for this use. Our locality requirement favors proofs that have a small number of predicates involving each item. Thus it will generally be advantageous to start with "shallow proofs"—proofs of small depth—for the static problem, such as one gets, for example, from *parallel* algorithms for solving the static version. Techniques already developed in parallel computational geometry [10] or in parametric searching [9] may prove to be useful.

Taking the question from the other side, is it possible to design parallel kinetic data structures? Apparently, there is a coordination problem: the time is a global variable, and we therefore need a global event queue to schedule the events. Shouldn't every event pass through the top of the event queue, and hence be handled by a unique processor?

Let us get back to our divide and conquer convex hull diagram algorithm (Section 4.1). To avoid this global event queue, we can change the structure so that each node of the computation tree contains its own event queue. An event queue at a node contains the normal events for all certificates of that node, but it also contains events corresponding to times when a vertex appears or disappears from one of the two input convex chains. When a node is asked to update itself up to time t, it recursively (and in parallel) sends this request to its own children, waits until they are done, then processes the events in its own event queue. Therefore, in the case of the convex hull, there is some parallelism that can be exploited.

To get further parallelism, it might be possible to get rid completely of a global clock. The kinetic structure could be required to process only events that have some causal dependencies in the original order. In a sense, this would amount to performing a kind of topological sweep [45] of time/space. Would this allow us to get rid of the event queue altogether?

7.3 Robustness

Because out-of-order events might completely destroy a kinetic data structure, the issue of robustness of a KDS implementation becomes unavoidable. In Section 5.2.1, we saw a number of heuristics to treat the near-zero cases, which significantly improve the robustness of a kinetic data structure. There is a question, however, of whether it is possible to implement kinetic data structures in exact arithmetic efficiently.

Some encouraging work has been done in this direction by Guibas and Karavelas [59]. They don't compute numerically as we do the roots of the polynomial associated with certificates. Instead, they maintain for each polynomial its so called *Sturm sequence*, which brackets the roots in intervals of varying width. When two failure times need to be compared, the intervals are updated just enough to be able to give an exact answer to the comparison. Although the current implementation uses floating point arithmetic, it seems that this method can be used with exact arithmetic. This method is of course rather expensive as it requires maintaining some extra information for each certificate, but we can expect that some gain will be achieved for the following reason: a root doesn't need to be computed with too much precision, but with only enough precision to answer a comparison.

7.4 Models of Motion

In a practical setting, items don't necessarily follow algebraic trajectories. For instance, the coordinates of a point rotating at constant velocity involve transcendental functions of time. In this case, finding the exact failure time of a certificate might become an expensive task. In interactive contexts, it might not even be possible to know the exact equations of motion of certain items. For instance, in a context where a computer simulation is interacting with the real world, the positions and velocity of the real objects cannot be known at all times.

Clearly, the model of motion that we have adopted in this thesis—all coordinates are polynomial functions of time—is too restrictive to be of much use in applications, although it is perfectly adequate for theoretical purposes. Is it possible to adapt kinetic data structures to some other models of motion?

In fact, a small addition to the framework will allow us to treat these questions in a uniform way. Instead of creating only events for failures of certificates, we can introduce *warning events*, set to happen at times when a certificate is not guaranteed to be valid anymore. In the case of a system in which some motions are only partially known, it is possible to compute a lower bound on the failure time of a certificate given some bounds on the speed or the acceleration of the objects involved. If the equations of motion are complicated, we can imagine a numerical scheme that converges to a certificate failure time from below. With such a scheme, we would perform an iteration only at a warning event, instead of performing all the iterations when the certificate is created.

In fact, the framework precisely allows for a completely orthogonal treatment of the combinatorial questions—those are addressed in this thesis—and of the numerical questions or those raised by uncertain knowledge of motion. Further work needs to be focused on these latter questions. For instance, here is a simple and fundamental numerical question: assume that we have three items whose positions and velocities are known at time t_0, and for which we have bounds on their accelerations. Compute the farthest time t_1 at which we are guaranteed that the triangle defined by the three items doesn't change orientation.

In the context of partially known motion, a theoretical issue also arises that is again combinatorial and algorithmic: how best to spend "sensing dollars" in order to acquire the information about the moving objects that is necessary to detect the events of interest for a kinetic data structure. This question was partially addressed by Kahan [70].

7.5 Recent Developments

There are many attributes in computational geometry that we know how to compute efficiently. For many of them, we also know how to *dynamize* them, i.e., to create data structures to maintain them efficiently upon insertion and deletion of data. The question is now how to *kinetize* them.

Following the publication of the original paper [17], kinetic data structures have been developed for the maintenance of a variety of structures: binary space partitions [3, 7], closest pair and minimum spanning trees in arbitrary dimensions [21], diameter and width in the plane [6], rectangle connectivity [65], and collision detection between polygons [16, 52].

Bibliography

[1] F. Affentranger and J. A. Wieacker. On the convex hull of uniform random points in a simple d-polytope. *Discrete Comput. Geom.*, 6:291–305, 1991.

[2] P. Agarwal, B. Aronov, and M. Sharir. On levels in arrangements of lines, segments, planes, and triangles. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 30–38, 1997.

[3] P. Agarwal, J. Erickson, and L. Guibas. Kinetic binary space partitions for triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, Jan. 1998.

[4] P. K. Agarwal, J. Basch, M. de Berg, L. J. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 1999. To appear.

[5] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 67–75, 1994.

[6] P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach. Maintaining the extent of a moving point set. In *Proc. 5th Workshop Algorithms Data Struct.*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 31–44. Springer-Verlag, 1997.

[7] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[8] P. K. Agarwal, O. Schwarzkopf, and M. Sharir. The overlay of lower envelopes and its applications. *Discrete Comput. Geom.*, 15:1–13, 1996.

[9] P. K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *J. Algorithms*, 17:292–318, 1994.

[10] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry.* Prentice Hall, 1993.

[11] G. Albers and T. Roos. Voronoi diagrams of moving points in higher dimensional spaces. In *Proc. 3rd Scand. Workshop Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 399–409. Springer-Verlag, 1992.

[12] C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 540–545, 1989.

[13] M. J. Atallah. Dynamic computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 92–99, 1983.

[14] M. J. Atallah. Some dynamic computational geometry problems. *Comput. Math. Appl.*, 11:1171–1181, 1985.

[15] J. Basch, H. Devarajan, P. Indyk, and L. Zhang. Probabilistic analysis for combinatorial functions of moving points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 442–444, 1997.

[16] J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection for two simple polygons. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 102–111, 1999.

[17] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.

[18] J. Basch, L. J. Guibas, and G. Ramkumar. Sweeping lines and line segments with a heap. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 469–471, 1997.

[19] J. Basch, L. J. Guibas, and G. D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proc. 4th Annu. European Sympos. Algorithms*, volume 1136 of *Lecture Notes Comput. Sci.*, pages 302–319. Springer-Verlag, 1996.

[20] J. Basch, L. J. Guibas, C. D. Silverstein, and L. Zhang. A practical evaluation of kinetic data structures. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 388–390, 1997.

[21] J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 344–351, 1997.

[22] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 179–187, 1990.

[23] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25:536–543, 1978.

[24] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[25] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 152–161, 1995.

[26] P. Billingsley. *Probability and measure.* John Wiley & Sons, 2nd edition, 1986.

[27] J.-D. Boissonnat and M. Yvinec. *Géométrie algorithmique.* Ediscience international, Paris, 1995.

[28] K. Q. Brown. Comments on “Algorithms for reporting and counting geometric intersections”. *IEEE Trans. Comput.*, C-30:147–148, 1981.

[29] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms (SODA '95)*, pages 263–272, 1995.

[30] L. P. Chew. Near-quadratic bounds for the L_1 Voronoi diagram of moving points. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 364–369, Waterloo, Canada, 1993.

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

[32] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scaled environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, Monterey, CA, 1995.

[33] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[34] H. Davenport and A. Schinzel. A combinatorial problem connected with differential equations. *Amer. J. Math.*, 87:684–689, 1965.

[35] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[36] O. Devillers and M. Golin. Dog bites postman: Point location in the moving Voronoi diagram and related problems. In *Proc. 1st Annu. European Sympos. Algorithms (ESA '93)*, volume 726 of *Lecture Notes in Computer Science*, pages 133–144. Springer-Verlag, 1993.

[37] T. K. Dey. Improved bounds on planar k-sets and related problems. *Discrete Comput. Geom.*, 19:373–382, 1998.

[38] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. In *Proc. 9th Internat. Colloq. Automata Lang. Program.*, volume 140 of *Lecture Notes Comput. Sci.*, pages 154–165. Springer-Verlag, 1982.

[39] R. Dwyer. Voronoi diagram and convex hull of moving points, 1998. Manuscript.

[40] R. A. Dwyer. On the convex hull of random points in a polytope. *J. Appl. Probab.*, 25(4):688–699, 1988.

[41] R. A. Dwyer. Kinder, gentler average-case analysis for convex hulls and maximal vectors. *SIGACT News*, 21(2):64–71, 1990.

[42] R. A. Dwyer. Convex hulls of samples from spherically symmetric distributions. *Discrete Appl. Math.*, 31(2):113–132, 1991.

[43] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete Comput. Geom.*, 6:343–367, 1991.

[44] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[45] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.

[46] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[47] H. Edelsbrunner and E. Welzl. On the number of line separations of a finite set in the plane. *J. Combin. Theory Ser. A*, 40:15–29, 1985.

[48] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.

[49] B. Efron. The problem of the two nearest neighbors (abstract). *Ann. Math. Statist.*, 38:298, 1967.

[50] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. In *Proc. 14th Symp. Computational Geometry*, pages 58–67. Assoc. Comput. Mach., June 1998.

[51] P. Erdős, L. Lovász, A. Simmons, and E. Straus. Dissection graphs of planar point sets. In J. N. Srivastava, editor, *A Survey of Combinatorial Theory*, pages 139–154. North-Holland, Amsterdam, Netherlands, 1973.

[52] J. Erickson, L. J. Guibas, J. Stofi, and L. Zhang. Separation-sensitive kinetic collision detection for convex objects. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 327–336, 1999.

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

[54] J.-J. Fu and R. C. T. Lee. Voronoi diagrams of moving points in the plane. *Internat. J. Comput. Geom. Appl.*, 1(1):23–32, 1991.

[55] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 301–310, 1993.

[56] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[57] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[58] L. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In *Proc. 17th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, volume 570 of *Lecture Notes in Computer Science*, pages 113–125. Springer-Verlag, 1991.

[59] L. J. Guibas and M. Karavelas. Interval methods for kinetic simulations. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 1999. To appear.

[60] P. Gupta, R. Janardan, and M. Smid. Fast algorithms for collision and proximity problems involving moving geometric objects. *Comput. Geom. Theory Appl.*, 6:371–391, 1996.

[61] D. Gusfield. Bounds for the parametric spanning tree problem. *Utilitas Math.*, pages 173–183, 1975. Proc. Humboldt Conf. Graph Theory Combin. Comput.

[62] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 21, pages 389–412. CRC Press LLC, Boca Raton, FL, 1997.

[63] V. Hayward, S. Aubry, A. Foisy, and Y. Ghallab. Efficient collision prediction among many moving objects. *International Journal of Robotics Research*, 14(2):129–143, 1995.

[64] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[65] J. Hershberger and S. Suri. Kinetic connectivity of rectangles. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 1999. To appear.

[66] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[67] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *Internat. J. Robot. Res.*, 2(4):77–80, 1983.

[68] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Trans. Visualization and Computer Graphics*, 1(3):218–230, Sept. 1995.

[69] K. Imai, S. Sumino, and H. Imai. Minimax geometric fitting of two corresponding sets of points. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 266–275, 1989.

[70] S. Kahan. A model for data in motion. In *Proc. 23th Annu. ACM Sympos. Theory Comput.*, pages 267–277, 1991.

[71] S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1994.

[72] N. Katoh, T. Tokuyama, and K. Iwano. On minimum and maximum spanning trees of linearly moving points. *Discrete Comput. Geom.*, 13:161–176, 1995.

[73] M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *Proc. IEEE Internat. Conf. Robot. Autom.*, volume 2, pages 1008–1014, 1991.

[74] P. McMullen. The maximal number of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[75] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, New York, 1998.

[76] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[77] D. M. Mount. Intersection detection and separators for simple polygons. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 303–311, 1992.

[78] K. Mulmuley. On levels in arrangements and Voronoi diagrams. *Discrete Comput. Geom.*, 6:307–338, 1991.

[79] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[80] T. Ottmann and D. Wood. Dynamical sets of points. *Comput. Vision Graph. Image Process.*, 27:157–166, 1984.

[81] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[82] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Commun. ACM*, 22:402–405, 1979.

[83] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

[84] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[85] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1993.

[86] H. Raynaud. Sur l'enveloppe convexe des nuages de points aléatoires. *J. Appl. Prob.*, 7:35–48, 1970.

[87] A. Rényi and R. Sulanke. Über die konvexe Hülle von n zufällig gewählten Punkten I. *Z. Wahrsch. Verw. Gebiete*, 2:75–84, 1963.

[88] T. Roos. Tighter bounds on Voronoi diagrams of moving points. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 358–363, 1993.

[89] T. Roos. Voronoi diagrams over dynamic scenes. *Discrete Appl. Math.*, 43, 1993.

[90] T. Roos and G. Albers. Maintaining proximity in higher dimensional spaces. In *Proc. 17th Internat. Sympos. Math. Found. Comput. Sci.*, volume 629 of *Lecture Notes Comput. Sci.*, pages 483–493, Aug. 1992.

[91] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 51–60, 1995.

[92] R. Seidel. The upper bound theorem for polytopes: an easy proof of its asymptotic version. *Comput. Geom. Theory Appl.*, 5:115–116, 1995.

[93] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[94] M. Sharir. On k-sets in arrangements of curves and surfaces. *Discrete Comput. Geom.*, 6:593–613, 1991.

[95] M. Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete Comput. Geom.*, 12:327–345, 1994.

[96] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.