New Data Structures and Algorithms for Mobile Data

Mohammad Ali Abam

New Data Structures and Algorithms for Mobile Data

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 13 november 2007 om 16.00 uur

door

Mohammad Ali Abam

geboren te Teheran, Iran

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.T. de Berg

Copromotor:
dr. B. Speckmann

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Abam, Mohammad Ali

New Data Structures and Algorithms for Mobile Data / by Mohammad Ali Abam.
Eindhoven: Technische Universiteit Eindhoven, 2007.
Proefschrift. ISBN 978-90-386-1134-1
NUR 993
Subject headings: computational geometry / data structures / algorithms
CR Subject Classification (1998): I.3.5, E.1, F.2.2

Promotor: prof.dr. M.T. de Berg
faculteit Wiskunde & Informatics
Technische Universiteit Eindhoven

copromotor: dr. Bettina Speckmann
faculteit Wiskunde & Informatics
Technische Universiteit Eindhoven

Kerncommissie:
prof.dr. P.K. Agarwal (Duke University)
prof.dr. L.J. Guibas (Stanford University)
prof.dr. G. Woeginger (Technische Universiteit Eindhoven)

The work in this thesis is supported by the Netherlands' Organization for Scientific Research (NWO) under project no. 612.065.307.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

© Mohammad Ali Abam 2007. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Cover Design: S.E. Baha
Printing: Eindhoven University Press

به نام دوست

و به یاد او که خواهد آمد.

تقدیم به خانواده عزیزم.

Contents

Preface iii

1 Introduction 1
1.1 Kinetic data structures . . . 3
1.2 Results in this thesis . . . 6

2 Kinetic sorting and kinetic convex hulls 9
2.1 Introduction . . . 10
2.2 The kinetic sorting problem . . . 12
2.2.1 The lower-bound model . . . 12
2.2.2 A lower bound in the comparison-graph sorting model . . . 13
2.2.3 A lower bound in the algebraic decision-tree model . . . 16
2.2.4 Upper bounds for kinetic sorting . . . 19
2.3 Gift-wrapping and convex-hull containment queries . . . 19
2.4 Conclusions . . . 23

3 Out-of-order event processing 25
3.1 Introduction . . . 26
3.2 Our model . . . 29
3.3 Kinetic sorting . . . 31
3.4 Kinetic tournaments . . . 38
3.5 Kinetic range trees . . . 41
3.6 Experiments . . . 42
3.7 Conclusions . . . 47

4 Kinetic kd-trees and longest-side kd-trees 49
4.1 Introduction . . . 50
4.2 Rank-based kd-trees . . . 52
4.3 Rank-based longest-side kd-trees . . . 58
4.4 Conclusions . . . 65

5 Kinetic collision detection for fat objects 67
5.1 Introduction . . . 68

5.2 Balls rolling on a plane . . . 70
5.2.1 Detecting collisions between small and large balls . . . 70
5.3 Free-flying fat objects in 3-space . . . 74
5.3.1 Similarly sized objects . . . 74
5.3.2 Arbitrarily sized objects . . . 75
5.4 Conclusions . . . 82

6 Streaming algorithms for line simplification **83**
6.1 Introduction . . . 84
6.2 A general algorithm . . . 88
6.3 The Hausdorff error function . . . 89
6.3.1 The error oracle for convex paths . . . 91
6.3.2 The error oracle for xy-monotone paths . . . 92
6.4 The Fréchet error function . . . 93
6.4.1 The error oracle . . . 93
6.5 The Hausdorff error function for general paths . . . 99
6.6 Conclusions . . . 100

7 Concluding remarks **103**

References **107**

Summary **115**

Curriculum Vitae **117**

Preface

It is not surprising that arriving at this momentous time of my life would have been impossible without the support, enthusiasm and encouragement of many incredibly precious people. Hence, I devote this preface to them.

First and foremost, I would like to thank Mark de Berg for giving me the opportunity to work with him and under his supervision. I am very grateful, not only for his invaluable guidance, our discussion and his insightful comments on my manuscripts, but also for his friendship and concern about things not related to work. I am sure I would have been unable to finish this thesis without his help and remarkable ideas concerning the publications that I co-authored with him. For all this and more, I gratefully thank him.

The results in this thesis are the fruits of joint work with my distinguished co-authors: Pankaj Agarwal, Mark de Berg, Peter Hachenberger, Sheung-Hung Poon, Bettina Speckmann, Hai Yu and Alireza Zarei. So my best thanks go to them. I would also like to express my great appreciation to the rest of my co-authors: Mohammad Farshi, Mohammad Ghodsi and Joachim Gudmundsson with whom I worked on papers that are not in this thesis. It was my pleasure to work with all of them, and it made me realize the value of working together as a team. Thank you all!

The great working atmosphere in the Algorithms group at Technische Universiteit Eindhoven is certainly never forgotten. I express my best thanks to all members of the group for being so friendly, helping me from time to time, organizing enjoyable excursions and listening to my talks in the noon-seminar (sometimes afternoon-seminar). I have been fortunate to be an office mate of many nice people in the group. I would like to thank all my office mates (in particular, my three-year office mate, Micha Streppel) for their help, conversations and discussions. My great thanks go to Mohammad Farshi (my compatriot colleague). He has been too friendly and kind to be just a colleague. I will never forget his help and kindness and our discussions during tea breaks.

I gratefully thank Bettina Speckmann who kindly supported me as a co-promoter. The members of my thesis committee are also gratefully acknowledged for reading the thesis and giving useful comments. It was my pleasure to have Pankaj K. Agarwal, Mark de Berg, Leo J. Guibas, Bettina Speckmann and Gerhard Woeginger on the core committee and Lars Arge and Frank van der Stappen on the defense committee.

I would like to gratefully thank my amiable friends, Frederico Custodio from Portugal and Pawan Kumar Tiwari from India, who were my house mates in the early months of my arrival in the Netherlands. Having discussions and cooking together every Thursday was my best opportunity to get to know other cultures. I also would like to highly thank my sympathetic and patient house mates, Amirhossein Ghamarian and Mohammad Mousavi, for their warm company in the last four years. I had a great time with them which is certainly never forgotten. My special appreciation goes to Ehsan Baha for being a good friend and designing the elegant cover of this thesis. I express my best thanks to the Iranian families Baha, Farshi, Fatemi, Mousavi, Rezaeian, Talebi, Vahedi for their help and invitations which have always given me a taste of home. Special thanks go to the members of the International soccer team which has always been at either the top or the bottom of the table. Visiting many countries in Europe would not have been enjoyable without the company of many beloved friends: Mohammad Farshi, Hamed Fatemi and his family, Amirhossein Ghamarian, Abbas Heydarnoori, Mohammd Mousavi, Mehdi Nikbakht, Pooyan Sakian, Hamid Salimi and Mostafa Vahedi. Thank you all! I would also like to acknowledge my senior friends: Omran Ahmadi, Abbas Asadi, Saeid Kasiri and Reza Pourezaei, who have been remaining chums in the last four years. Finally, thanks to many other wonderful friends: Arash Abtahi, Ali Etaati, Majid Ghaderi, Mohammad Ghanehpasand, Hamid Javani, Saman Harati, Pejman khalili, Majid Majdolashrafi, Kamyar Malakpour, Iman Mosavat, Payam Sadeghi, Abdol Saib, Arash Sarhangi, Mehdi Shafiee, Mojtaba Shokri, Ali Tagholi, Ebrahim Talebi and many other good friends.

I would like to express my gratitude to my professors at Sharif University of Technology for their advice and insights. Special thanks go to Mohammad Ghodsi (my ex-supervisor) who introduced me the wonderful world of computational geometry. I also appreciate my colleagues on the committee of ACM international-collegiate-programming contest in Tehran site and my colleagues at Sharif-network-security center for their help, conversations and discussions. I would also like to thank my teachers in the olympiad training courses (in particular, Yahya Tabesh and Ebadollah Mahmoodian) and the members of Iranian international mathematical olympiad team in 1995. Here I would like to reminisce my dearest friend in the team, Reza Sadeghi, who was unfortunately killed in a horrific car accident. I also gratefully appreciate my teachers in Roshd high school (in particular, Heydar Zandiyeh and Moharam Iradmusa) whom I am greatly indebted to them for their help and encouragement that stimulated my interest in mathematics.

I have been very fortunate to receive a tempting offer as a postdoc from the MADALGO center at Aarhus university. Therefore I would like to thank Lars Arge for his offer.

A list can never be complete. I would therefore like to thank all those others as well, who contributed to pushing my cart forward, so that I could reach this point.

I am grateful beyond expression to my dearest family. I think that now, at the end of my PhD studies, is a good moment to express my best thanks to them for unconditional support, encouragement and faith in me throughout my whole life and in particular, during the last four years. I hope to be able to compensate them in the future. I dedicate this thesis to them, *with love and gratitude*.

Chapter 1

Introduction

We live in a three-dimensional world. It is not surprising, therefore, that in many areas of computer science it is necessary to store, analyze, and create or manipulate geometric data. In a geographic information system(GIS), for instance, one may want to select all topographic features within a rectangular query region. Or in robotics applications, one may need to plan a collision-free path for a robot. Other examples of application areas involving geometric data are computer graphics and virtual reality, computer-aided design and manufacturing (CAD/CAM), and integrated circuit design. Computation geometry [42] is the field within algorithms research that deals with geometric problems. The primary goal of research in computation geometry is to design and analyze efficient algorithms and data structures for problems stated in terms of basic geometric objects: points, lines, and other objects in 2-, 3-, or higher dimensional space.

Over the past decade, there has been a lot of interest in computational geometry in developing algorithms for *mobile data*, that is, data about moving objects. mobile data is becoming more and more available in a variety of application areas—air-traffic control, mobile communication, and geographic information systems, for instance—due to the increased availability of GPS systems and to other technological advances. In many of these areas, the data are moving points in 2- or higher-dimensional space. What is needed is to store these points in such a way that either some queries—*range queries*, for instance, or *nearest-neighbor queries*—can be answered efficiently, or some attributes—the convex hull of moving objects, for instance, or the closest pair among moving points—can be maintained efficiently. Within computational geometry, the common model for designing and analyzing data structures for moving objects is the *kinetic-data-structure* framework introduced by Basch *et al.* [23].

In other areas of computer science, a lot of work has been dedicated to mobile data as well. A wide and increasing range of database applications has to deal with objects moving around such as taxis, airplanes, hurricanes or flood areas, to name but a few examples. Therefore, there has been a lot of work on extending the capabilities of existing

database systems to handle continuously changing data such as the position of moving objects. In traditional databases, in order to represent moving objects (e.g air planes) and answer queries concerning their positions (e.g. "retrieve all the airplanes that will come within 30 miles of the airport in the next 10 minutes") the air planes' positions and the indexes storing them must be updated continuously which is unsatisfactory. This leads researchers in the database community to model and index moving objects (see e.g. [26, 61, 89, 90, 93, 94, 98, 99, 100]) in order to efficiently update moving-object databases (MOD) and quickly answer queries concerning moving objects. In the wireless-network community, the study of mobile hosts has received a lot of attention. The hosts in a mobile network move according to various patterns. Realistic models for the motion patterns are needed in simulation in order to evaluate system and protocol performance. Mobility patterns have been used in the study of various problems such handoff, location management, paging, registration, calling time and traffic load. Mobility models have been explored in both cellular networks [21, 80, 81, 82, 105] where communications are point to point rather than among groups, and ad hoc networks [36, 76, 105] where communications are among teams which tend to coordinate their movements (e.g. a firemen rescue team in a disaster recovery situation). Object tracking is an important task within computer-vision community, due to the increased availability of high quality and inexpensive video cameras and the increased need for automated video analysis. What is needed in this area of research is the detection of moving objects, tracking of such objects from frame to frame and analysis of object trajectories to recognize their behavior—for more details see [102] and references therein. In computer graphics, dealing with mobile data is unavoidable. Many applications in computer graphics require fast and robust collision detection algorithms. The algorithm in this area can be grouped into four approaches: space-time volume intersection [28], swept volume interference [46, 64], multiple interference detection [35, 50, 51] and trajectory parametrization [88, 92]. Rendering of moving objects [79] is another example of work that has been dedicated to mobile data in computer graphics. When one is designing algorithms for mobile data, an important issue is how to analyze them in order that their efficiency can be compared. In many of the above-mentioned areas, the analysis is often done experimentally. From an algorithmic-research perspective, however, one would like to do a theoretical analysis. The kinetic-data-structures framework developed in computational geometry provides the tools for such a theoretical analysis, and they form the topic of this thesis. Whether this framework provides the best solution in practice will probably depend on the application at hand, and is a topic for further research.

Kinetic data structures have been the major area of my research as a PhD student in the algorithms group at the TU Eindhoven from 2004 to 2007. In the remainder of this chapter I will first give a short introduction to kinetic data structures and then summarize the results that we obtained. The next five chapters correspond to the papers resulting from my research in this area: four chapter about kinetic data structures and one chapter about streaming algorithms with applications to moving objects. The thesis is concluded with a chapter discussing open problems and directions for future research.

1.1 Kinetic data structures

Algorithms that deal with objects in motion traditionally discretize the time axis and compute or update their structures based on the position of the objects at every time step. A major problem with this approach is the choice of the prefect time interval. If the interval is too large, important events will be missed and the data structure will be invalid for some time. If, on the other hand, the interval is too small, a lot of computation time is wasted: some objects may have moved only slightly, in such a way that the overall data structure is not influenced. One would like to use the temporal coherence to speed up the process—to know exactly at which point we need to take an action. In fact the location of an object should require our attention if and only if it triggers an actual change in the data structure. *Kinetic Data Structures* (KDS for short) introduced by Basch *et al.* [23] do exactly that: they maintain not only the structure itself but also some additional information that helps to find out when the structure will undergo a real combinatorial change.

A KDS is a structure that maintains a certain attribute of a set of continuously moving objects—the convex hull of moving objects, for instance, or the closest distance among moving objects. It consists of two parts: a combinatorial description of the attribute and a set of certificates—elementary tests on the input objects—with the property that as long as the outcomes of the certificates do not change, the attribute does not change. In other words, the set of certificates forms a proof that the current combinatorial description of the attribute is still correct. It is assumed that each object follows a known trajectory so that one can compute the failure time of each certificate. Whenever a certificate fails—we call this an *event*—the KDS must be updated. The KDS remains valid until the next event. To know the next time any certificate fails, the failure time are stored in an event queue. Note that a change in the set of certificates also means a change in the set of failure times, so the event queue has to be updated at each event as well. An important aspect of KDSs is their on-line character: although the positions and the motions (flight plans) of the objects are known at all times, they are not known far in advance. In particular, any object can change its flight plan at any time. The KDS has to be able to handle such changes in flight plans efficiently.

As a concrete example, consider the kinetic priority queue: maintain the rightmost point (that is, the point with maximum x-coordinate) of a set S of continuously moving points on the real line. One simple possibility is to store S in a sorted array $A[1..n]$. The rightmost point is now given by $A[n]$. For each $1 \leqslant i < n$ there is a certificate $\big[A[i] < A[i+1]\big]$. Whenever $A[j] = A[j+1]$ for some j, we have a certificate failure. At such an event we swap $A[j]$ and $A[j+1]$. Furthermore, at most three new certificates arise: $\big[A[j-1] < A[j]\big]$, $\big[A[j] < A[j+1]\big]$, and $\big[A[j+1] < A[j+2]\big]$. We compute the failure time of each of them, based on our knowledge of their current motions, and insert the failure times into the event queue Q. Some certificates may also disappear because the two points involved are no longer neighbors; they have to be deleted from Q. Note that any point is involved in at most two certificates: comparisons with its immediate predecessor and its immediate successor. Hence, a change in flight plan involve the re-

scheduling of at most two failure times, and can thus be handled efficiently.

The performance of a KDS is measured according to four criteria [54].

Responsiveness: One of the most important performance measure for a KDS is the time needed to update it (as well as the event queue) when a certificate fails. This is called the response time. If the response time is polylogarithmic, the KDS is called *responsive*.

Compactness: The compactness of a KDS is the total number of certificates that are stored. Note that this is not necessarily the same as the amount of storage the entire structure needs. For example, it can happen that a certain structure on a set of points in the plane is valid as long as the x- and y-orders of the points do not change—hence, $2(n-1)$ certificate suffice—but that the data structure itself needs more than linear storage. This happens for instance when we want to maintain a 2-dimensional range tree [24]. A KDS is called *compact* if its compactness is always near-linear in the total number of objects.

Locality: The locality of a KDS is the maximum number of certificates any object is involved in. This is an important measure, because when an object changes its flight plan, one has to recompute the failure times of all certificates it is involved in, and update the event queue accordingly. A KDS is called *local* if its locality is polylogarithmic.

Efficiency: The notion of efficiency is slightly more complicated than the previous three performance measures. It deals with the number of events that have to be processed. A certificate failure does not automatically imply a change in the attribute being maintained; it could also be that there is only an internal change in the KDS. In the kinetic priority queue described above, for instance, an event occurs whenever two points change order, but the attribute only changes when the rightmost point is involved in the event. Events where the attribute changes are called *external events*, other event are called *internal event*. The efficiency of a KDS can be defined as the ratio of the maximum total number of internal and external events to the maximum total number of external events. The main difficulty in designing KDSs is to make sure that the efficiency is good: the worst-case number of events handled by the data structure for a given motion is small compared to some worst-case number of "external events" that must be handled for that motion. These worst-case number of events are under certain assumptions on the trajectories of the objects. The most common assumptions are that the motions are linear, or that they can be described by bounded-degree polynomials. The KDS is called *efficient* if its efficiency is polylogarithmic.

Let's analyze the performance of our kinetic priority queue. It is clearly responsive, with a response time of $O(\log n)$. It is also local—each point is involved in at most two certificates—and, hence, compact. Unfortunately it is not efficient: if the points move

linearly, for instance, then the worst-case total number of external events, which occur when the rightmost point changes, is $O(n)$, but the total number of internal events can be $\Theta(n^2)$. Hence, this is not a satisfactory solution.

Previous results

The paper by Basch *et al.* [23], which introduced the KDS framework, gives three examples of KDSs. The first KDS is for the problem that we gave above: maintain the rightmost point of a set S of continuously moving points on the real line. The authors present a kinetic priority queue, which they called a kinetic tournament, that is much more efficient than the simple solution we sketched above. The KDS has size $O(n)$, each point is involved in $O(\log n)$ certificates, its response time is $O(\log^2 n)$, and the total number of events is $O(\lambda_s(n) \log n)$, where s is the number of times any pair of points can swap and $\lambda_s(n)$ is the maximum length of Davenport-Schinzel sequence of order s on n symbols [16]. Another KDS presented by the authors maintains the convex hull of moving points in the plane. They designed a KDS that needs to be updated $O(n^{2+\varepsilon})$ times, assuming the trajectories of the points are algebraic curves described by bounded degree polynomials. Since the convex hull can change $\Omega(n^2)$ times, the KDS is efficient. The KDS is also compact, local, and responsive. Finally, they gave a compact, local, responsive and efficient KDS to maintain the closest pair of a set of moving points in the plane.

The paper of Basch *et al.* sparked a great amount of research activity in the area of kinetic data structures, resulting in many publications. In the following paragraphs we discuss a large and representative sample of these papers.

There are some papers [11, 14, 32, 39] dealing with kinetic binary space partitions (BSPs). That is, given a set S of moving objects in the plane—line segments, for instance—one wants to maintain a BSP [42] on the set S: a recursive partitioning of the plane by lines such that each cell of the final partitioning intersects at most one object. (Some of these papers also have results on 3-dimensional BSPs [11, 32].) The proposed KDSs are local, compact, and their expected [11, 14] or worst-case [39] response time is polylogarithmic. Each of these kinetic BSPs has $O(n^2)$ certificates failures. Whether or not this is efficient is a tricky question, because a BSP is not uniquely defined for a given set of objects. Interestingly, Agarwal *et al.* [9] have shown that there are configuration of n moving segments, such that any BSPs must undergo $\Omega(n\sqrt{n})$ changes during the motions.

Several papers study the problem of maintaining the connected components in a set of moving regions in the plane. This is motivated by *ad hoc* networks [70, 95]. The basic question one wants to be able to answer is here: "are region A and B currently in the same connected component?" Hershberger and Suri [67] do this for the case where the regions are rectangles, and Guibas *et al.* [55] for the case where the regions are unit disks. In both cases the proposed KDS has near-linear size, and the amortized response time is polylogarithmic. The time to answer a connectivity query is $O(\log n / \log \log n)$. Both KDSs have to process roughly a quadratic number of certificate failures; since the

connectivity can change $\Omega(n^2)$ times in the worst case, this means that both KDSs are efficient. Gao *et al.* [49] study the somewhat related problem of maintaining clusters among a set of moving points in the plane.

In the context of kinetic data structure, an interesting open problem is to efficiently maintain a triangulation of n moving points, which was attacked in some papers [10, 18]. The best result so far has been obtained by Agarwal *et al.* [18]. They propose a KDS that processes $O(n^2 2^{\sqrt{\log n \cdot \log\log n}})$ events and almost matches the $\Omega(n^2)$ lower bound [9]. Maintaining the Delaunay triangulation of a set of n moving points in the plane is simple: when four vertices of two adjacent triangles become co-circle, we can simply flip the edge. How many events are processed in the worst case by this KDS is a longstanding open problem. Although it is conjectured that the number of events is $O(n^2)$, the best known upper bound is near-cubic, which is much more than the $\Omega(n^2)$ lower bound.

There are also papers on KDSs for collision detection and range searching problems, and papers dealing with other aspects of KDSs (e.g. how to reliably compare certificate failure times, or how to trade off the number of events against the time needed to report the maintained attribute). Since these are more closely related to the specific problems we studied, we will discuss them more extensively in the corresponding chapters.

1.2 Results in this thesis

Our research in the area of kinetic data structures led to four papers: *Kinetic sorting and kinetic convex hulls* [2], *Out-of-order event processing in kinetic data structures*[1], *Kinetic kd-trees and longest-side kd-trees* [5], *Kinetic collision detection for convex fat objects* [4]. These results are presented in chapters 2-5. We also studied a problem on moving points in a different setting. In this setting the trajectories are not given to us explicitly (as is the case in the KDS framework), but we received a stream of data points describing the location of a moving object at consecutive time instances. The goal is to maintain an approximation of the path traveled by the object, without using too much storage. This research led to a paper *Streaming algorithms for line simplification* [3], which is presented in Chapter 6. Below we give a brief overview of all our results.

In some applications of KDSs it may be necessary to maintain the attribute of interest explicitly. If one uses a KDS for collision detection, for instance, any external event— a collision in this case— must be reported. In other applications, however, explicitly maintaining the attribute at all times may not be necessary; the attribute is only needed at certain times. This leads us to view a KDS as a query structure in Chapter 2: we want to maintain a set S of moving objects in such a way that we can reconstruct the attribute of interest efficiently whenever this is called for. Thus, instead of maintaining the attribute explicitly (which requires us to update the KDS whenever an external event happens) the goal is to maintain some information that needs to be updated less frequently, while it still allows us to reconstruct the attribute quickly. This makes it possible to reduce the maintenance cost (number of events), as it is no longer necessary to update the KDS

whenever the attribute changes. On the other hand, a reduction in maintenance cost will have an impact on the query time, that is, the time needed to reconstruct the attribute. Thus there is a trade-off between maintenance cost and query time, somewhat similar to storage versus query time trade-offs for e.g. range-searching data structures. The main goal of this chapter is to study such trade-offs for kinetic convex hulls. We first study the simpler *kinetic sorting problem*: maintain a KDS on a set of n points moving on the real line such that at any time we can quickly reconstruct a sorted list of the points. we show a lower bound for this problem showing the following: with a subquadratic maintenance cost one cannot obtain any significant speed-up on the time needed to generate the sorted list (compared to the trivial $O(n \log n)$ time), even for linear motions. This negative result gives a strong indication that good trade-offs are not possible for a large number of geometric problems—Voronoi diagrams and Delaunay triangulations, for example, or convex hulls—as the sorting problem can often be reduced to such problems. However, we show that it is possible to get a good trade-off between maintenance and reconstruction time when the number of points on the convex hull is small: For any Q with $1 \leqslant Q \leqslant n$ and any $\varepsilon > 0$ there is a KDS that processes $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events such that one can reconstruct the convex hull of S in $O(hQ \log n)$ time, where δ is the maximum degree of the polynomials describing the motions of the points and h is the number of vertices of the convex hull.

In traditional KDSs it is essential to process events in the correct order. Otherwise, major inconsistencies may arise from which the KDS cannot recover. In Chapter 3, we study the problem of designing KDSs when event times cannot be computed exactly and events may be processed in a wrong order. Indeed, the goal of this chapter is to address the following question: is it possible to do the event scheduling and processing in such a way that the KDS is more robust under errors in the computation of event times? The KDS may process the events in a wrong order and thus may maintain a wrong geometric attribute from time to time, but we would like the KDS to detect these errors and fix them quickly. We present KDSs that are robust against this out-of-order processing, including kinetic sorting, kinetic tournaments and kinetic range searching. Our algorithms are *quasi-robust* in the sense that the maintained attribute of the moving objects will be correct for most of the time, and when it is incorrect, it will not be far from the correct attribute. As a by-product of our approach, degeneracy problems (how to deal with multiple events occurring simultaneously) arising in traditional KDS algorithms naturally disappear, because our KDSs no longer cares about in which order these simultaneous events are processed.

The range searching problem is the subject of Chapter 4: given a set S of n points, the goal is to design a data structure such that we can quickly report all points inside any given region. This is a fundamental problem in computational geometry that arises in many applications. In practice, simple structures such as kd-trees are used. In this chapter we show how to maintain kd-trees and longest-side kd-trees when the points move. We present a new and simple variant of the standard kd-tree, called *rank-based kd-trees*, for a set of n points in d-dimensional space. Our rank-based kd-tree supports orthogonal range searching in time $O(n^{1-1/d} + k)$ and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized efficiently. The rank-based kd-tree processes $O(n^2)$ events

in the worst case if the points follow constant-degree algebraic trajectories and each event can be handled in $O(\log n)$ worst-case time. Moreover, each point is involved only in a constant number of certificates. We also propose the first kinetic variant of the longest-side kd-tree, which we call the *rank-based longest-side kd-tree* (or *RBLS kd-tree*, for short), for a set of n points in the plane. (We have been unable to generalize this result to higher dimensions.) An RBLS kd-tree uses $O(n)$ space and supports approximate nearest-neighbor, approximate farthest-neighbor, and approximate range queries in the same time as the original longest-side kd-tree does for stationary points, namely $O((1/\varepsilon)\log^2 n)$ (plus the time needed to report the answers in case of range searching). The kinetic RBLS kd-tree maintains $O(n)$ certificates, processes $O(n^3 \log n)$ events if the points follow constant-degree algebraic trajectories, each event can be handled in $O(\log^2 n)$ time, and each point is involved in $O(\log n)$ certificates.

Collision detection is a basic problem arising in all areas of computer science involving objects in motion—motion planning, computer-simulated environments, animated figure articulation, or virtual prototyping, to name a few. Therefore it is not surprising that over the years it has attracted a great amount of interest. The main goal of Chapter 5 is to develop KDSs for 3D collision detection that have a *near-linear number of certificates* for *multiple* convex fat objects of *varying sizes*. We start with the special case of n balls of arbitrary sizes rolling on a plane. Here we present an elegant and simple KDS that uses $O(n \log n)$ storage and processes $O(n^2)$ events; processing an event takes $O(\log^2 n)$ time. Then we turn our attention to free-flying convex objects that are *fat*, that is, not very long and skinny. (See Section 5.3 for precise definition.) We first study fat objects that have similar sizes. We give an almost trivial KDS that has $O(n)$ size and processes $O(n^2)$ events; handling an event takes $O(\log n)$ time. Next we consider the much more difficult general case, where the fat objects can have vastly different sizes. Here we present a KDS that uses $O(n \log^6 n)$ storage and processes $O(n^2)$ events; handling an event takes $O(\log^7 n)$ time.

Chapter 2-5 all deal with scenarios where the trajectories of the objects are known in advance (at least in the near future) and are given to us explicitly. In Chapter 6, however, we consider a different scenario: instead of getting an explicit description of a trajectory, we are getting a (possible infinite) stream of points describing consecutive locations of a moving object. As a concrete example, suppose we are tracking one, or maybe many, moving objects. Each object is equipped with a device that is continuously transmitting its position at certain times. Thus we are receiving a stream of data points that describes the path along which the object moves. In this chapter, we study maintaining the path of an object that we are tracking over a very long period of time, as happens for instance when studying the migratory patterns of animals. In this situation it may be undesirable or even impossible to store the complete stream of data points. Instead we have to maintain an approximation of the input path. Here, we present the first general algorithm for maintaining a simplification of the trajectory of a moving object without using too much storage. We analyze the competitive ratio of our algorithms, allowing resource augmentation: we let our algorithm maintain a simplification with $2k$ (internal) points, and compare the error of our simplification to the error of the optimal simplification with k points.

Chapter 2

Kinetic sorting and kinetic convex hulls

Abstract. Let S be a set of n points moving on the real line. The kinetic sorting problem is to maintain a data structure on the set S that makes it possible to quickly generate a sorted list of the points in S, at any given time. We prove tight lower bounds for this problem, which show the following: with a subquadratic maintenance cost one cannot obtain any significant speed-up on the time needed to generate the sorted list (compared to the trivial $O(n \log n)$ time), even for linear motions.

We also describe a kinetic data structure for so-called gift-wrapping queries on a set S of n moving points in the plane: given a point q and a line ℓ through q such that all points from S lie on the same side of ℓ, report which point $p_i \in S$ is hit first when ℓ is rotated around q. Our KDS allows a trade-off between the query time and the maintenance cost: for any Q with $1 \leqslant Q \leqslant n$, we can achieve $O(Q \log n)$ query time with a KDS that processes $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events, where δ is the maximum degree of the polynomials describing the motions of the points. This allows us to reconstruct the convex hull quickly when the number of points on the convex hull is small. The structure also allows us to answer extreme-point queries (given a query direction $\overrightarrow{d}$, what is the point from S that is extreme in direction $\overrightarrow{d}$?) and convex-hull containment queries (given a query point q, is q inside the current convex hull?).

An extended abstract of this chapter was previously published as: M. A. Abam and M. de Berg, Kinetic sorting and kinetic convex hulls, In *Proc. 21st ACM Symposium on Computational Geometry (SCG)*, pages 190–197, 2005. The full paper was published in *Computational Geometry: Theory and Applications*, 37:16–26, 2007. (Special issue on 21st ACM Symposium on Computational Geometry.)

2.1 Introduction

Background. Computing the convex hull of a set of points in the plane is a classic problem in computational geometry. It is therefore not surprising that the kinetic maintenance of the convex hull of a set of n moving points in the plane was already studied by Basch *et al.* [23] in their seminal paper on kinetic data structures. They designed a KDS that needs to be updated $O(n^{2+\varepsilon})$ times, assuming the trajectories of the points are algebraic curves described by bounded degree polynomials.

In some applications of KDSs it may be necessary to maintain the attribute of interest explicitly. If one uses a KDS for collision detection, for instance, any external event—a collision in this case— must be reported. In such cases, the number of changes to the attribute is a lower bound on the number of events to be processed. Since the convex hull of n linearly moving points can change $\Omega(n^2)$ times [13], this means that any KDS that maintains an explicit representation of the convex hull must process $\Omega(n^2)$ events in the worst case. Hence, the convex-hull KDS of Basch *et al.* [23], which indeed maintains the convex hull explicitly, is close to optimal in the worst case.

In other applications, however, explicitly maintaining the attribute at all times may not be necessary; the attribute is only needed at certain times. This is for instance the case when a KDS is used as an auxiliary structure in another KDS. The auxiliary KDS is then used to update the main KDS efficiently when a certificate of the main KDS fails. In this case, even though the main KDS may have to be maintained explicitly, the attribute maintained by the auxiliary KDS only needs to be available at certain times. This leads us to view a KDS as a query structure: we want to maintain a set S of moving objects in such a way that we can reconstruct the attribute of interest efficiently whenever this is called for. This makes it possible to reduce the maintenance cost, as it is no longer necessary to update the KDS whenever the attribute changes. On the other hand, a reduction in maintenance cost will have an impact on the query time, that is, the time needed to reconstruct the attribute. Thus there is a trade-off between maintenance cost and query time, somewhat similar to storage versus query time trade-offs for e.g. range-searching data structures. Our main goal is to study such trade-offs for kinetic convex hulls.

Our results. As just noted, our main interest lies in trade-offs between the maintenance cost of a kinetic convex-hull structure and the time to reconstruct the convex hull at any given time. To this end, we first study the simpler *kinetic sorting problem*: maintain a KDS on a set of n points moving on the real line such that at any time we can quickly reconstruct a sorted list of the points. We prove in Section 2.2 that already for the kinetic sorting problem one cannot get good trade-offs: even for linear motions, the worst-case maintenance cost is $\Omega(n^2)$ if one wants to be able to do the reconstruction in $o(n)$ time. Note that with $\Omega(n^2)$ maintenance cost, we can explicitly maintain the sorted list at all times, so that the reconstruction cost is zero. Thus interesting trade-offs are only possible in a very limited range of the spectrum, namely for reconstruction costs between $\Omega(n)$ and $O(n \log n)$. For this range we also prove lower bounds: we roughly show that one

needs $\Omega(n^2/m)$ maintenance cost if one wants to achieve $o(n \log m)$ reconstruction cost, for any m with $2 \leqslant m \leqslant n$. (See Section 2.2.1 for a definition of our lower-bound model.) We also give a matching upper bound.

The negative results on the kinetic sorting problem make it quite unlikely that one can obtain good trade-offs for the kinetic convex-hull problem. (The results do not give a formal proof of this fact because the comparison-based model we use for the 1D sorting problem does not apply in 2D.) However, we will show that it is possible to get a good trade-off between maintenance and reconstruction time when the number of points on the convex hull is small: For any Q with $1 \leqslant Q \leqslant n$ and any $\varepsilon > 0$ there is a KDS that processes $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events such that one can reconstruct the convex hull of S in $O(hQ \log n)$ time, where δ is the maximum degree of the polynomials describing the motions of the points and h is the number of vertices of the convex hull. We obtain this result by giving a KDS for *gift-wrapping queries*: given a point q and a line ℓ through q such that all points from S lie on the same side of ℓ, report the point $p_i \in S$ that is hit first when ℓ is rotated (in counterclockwise direction, say) around q. Our KDS for this problem has $O(Q \log n)$ query time and it processes $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events. For linear motions, this bound is very close to the lower bounds De Berg [38] proved for the kinetic dictionary problem—see below—which seems an easier problem. Our KDS can also answer *extreme-point queries* (given a query direction $\overrightarrow{d}$, what is the point from S that is extreme in direction $\overrightarrow{d}$?) and *convex-hull containment queries* (given a query point q, is q inside the current convex hull?).

Related work. Some existing KDSs—the kinetic variants of various range-searching data structures [6, 7, 8, 12, 67, 77, 78, 90], for instance—do not maintain a uniquely defined attribute such as the convex hull, but they maintain a query data structure. In this setting the KDS is, of course, a query structure as well. Our setting is different because we are studying the maintenance of a single, uniquely defined, attribute such as the convex hull. This is somewhat similar to the papers by Guibas *et al.* [55] and by Hershberger and Suri [67], who study the kinetic maintenance of the connectivity of moving regions in the plane. Their structures can answer queries of the form: “Are regions A and B in the same connected component of the union of the regions?” However, their structure is updated whenever the connectivity changes—they do not allow for trade-offs between the number of events and the query time. Moreover, their goal is not to be able to quickly reconstruct the entire connectivity information at any given time. Thus their KDS is essentially a kinetic version of a structure for connectivity queries, rather than a kinetic query structure for reconstructing a unique attribute.

One of our main results is a lower bound on the trade-offs between reconstruction time and maintenance cost for the kinetic sorting problem. Lower bounds for trade-offs between query time and maintenance cost were also given by De Berg [38], but he studied the kinetic dictionary problem, where one wants to maintain a dictionary on a set S of n points moving on the real line. He showed that any kinetic dictionary with worst-case query time $O(Q)$ must have a worst-case total maintenance cost of $\Omega(n^2/Q^2)$, even if the

points move linearly. As already remarked, the upper bounds we obtain for gift-wrapping and convex-hull containment queries on moving points in the plane—these problems seem at least as hard as the kinetic dictionary problem—almost match these bounds for linear motions.

A recent paper by Agarwal *et al.* [7] is closely related to Section 2.3, where we describe a KDS for convex-hull containment queries. They describe, besides data structures for various range searching and proximity queries on moving points, a structure for convex-hull containment queries on moving points. Their structure is more powerful since it can answer queries about past or future convex hulls. On the other hand, they only deal with linear motions. Their structure uses $O(n^{2+\varepsilon}/Q^2)$ storage to obtain a query time of $O(Q \text{ polylog } n)$. Since the KDS is precomputed for the complete motions, there are no events. Note that the number of events we process for linear motions is the same as the amount of storage used by their structure. This means we can also obtain their result (namely the ability to answer queries in the past and future as well, at the expense of extra storage): during preprocessing, do a complete simulation based on the motions (that are assumed to be given) and record the changes to the KDS using standard persistency methods.

2.2 The kinetic sorting problem

Let $S = \{x_1, \cdots, x_n\}$ be a set of n point objects[1] moving continuously on the real line. In other words, the value of x_i is a continuous function of time, which we denote by $x_i(t)$. We define $S(t) = \{x_1(t), \cdots, x_n(t)\}$. For simplicity, we write S and x_i instead of $S(t)$ and $x_i(t)$, respectively, provided that no confusion arises. The kinetic sorting problem asks to maintain a structure on S such that at any given time t we can quickly generate a sorted list for $S(t)$. We call such a structure a *sorting KDS*.

We focus on trade-offs between the sorting cost and the maintenance cost: what is the worst-case maintenance cost if we want to guarantee a sorting cost of $O(Q)$, where Q is some parameter, under the assumption that the point objects follow trajectories that can be described by bounded degree polynomials. (In fact, for our lower bounds we will only use linear motions, whereas for our upper bounds we only need the restriction that any pair of points swaps $O(1)$ times.)

2.2.1 The lower-bound model

We shall prove our lower bounds for the kinetic sorting problem in the comparison-graph model introduced by De Berg [38], which is defined as follows. A *comparison graph* for a set S of numbers is defined as a directed graph $\mathcal{G}(S, A)$ such that if $(x_i, x_j) \in A$,

[1] We use "point objects" (or sometimes just "object") for the points in S to distinguish them from other points that play a role in our proofs.

then $x_i < x_j$. The reverse is not true: the fact that $x_i < x_j$ does not mean there must be an arc in $\mathcal{G}$. The idea is that the comparison graph represents the ordering information encoded in a sorting KDS on the set S: if $(x_i, x_j) \in A$, then the fact that $x_i < x_j$ can be derived from the information stored in the KDS, without doing any additional comparisons.

Maintenance cost. The operations we allow on the comparison graph are insertions and deletions of arcs. For the maintenance cost, we only charge the algorithm for insertions of arcs; deletions are free. Note that by doing a single comparison we can sometimes obtain a lot of ordering information by transitivity. Therefore we only charge the algorithm for a new arc in the transitive reduction of the graph, that is, a new arc that is not implied by transitivity. Following De Berg [38], we therefore define the maintenance cost as the total number of such non-redundant arcs ever inserted into the comparison graph, either at initialization or during maintenance operations.

We say that the arc $(x_i, x_j) \in A$ *fails* at time t if $x_i(t) = x_j(t)$. The non-redundant arcs in the comparison graph essentially act as certificates, and their failures trigger events at which the KDS needs to be updated.

Query cost. A query at time t asks to construct a sorted list on the points in the current set S (that is, $S(t)$). We shall consider two different measures for the query cost.

The comparison-graph sorting model: The first measure is in a very weak model, where we only charge for the minimum number of comparisons needed to obtain a sorted list, assuming we have an oracle at our disposal telling us exactly which comparisons to do. This is similar to the query cost used by De Berg when he proved lower bounds for the kinetic dictionary. For the sorting problem this simply means that the query cost is equal to the number of pairs $x_i, x_j \in S$ that are adjacent in the ordering and for which there is no arc in the comparison graph.

The algebraic decision-tree model: In this model we also count the number of comparisons needed to sort the set S, but this time we do not have an oracle telling us which comparisons to do. We shall use the following basic fact: Suppose the number of different orderings of S that are compatible with the comparison graph at some given time is N. Then the cost to sort S in the algebraic decision-tree model is at least $\log N$.

2.2.2 A lower bound in the comparison-graph sorting model

The point objects in our lower-bound instance will move with constant (but different) velocities on the real line. Hence, if we view the line on which the point objects move as the x-axis and time as the t-axis, then the trajectories of the point objects are straight

lines in the tx-plane. We use ξ_i to denote the line in the tx-plane that is the trajectory of x_i. It is somewhat easier to describe the lower-bound instance in the dual plane. We shall call the two axes in the dual plane the u-axis and the v-axis. We use the standard duality transform [42], where a line $\xi : x = at + b$ in the tx-plane is mapped to the point $\xi^* : (a, -b)$ in the dual plane, and a point $p : (a, b)$ in the primal plane is mapped to the line $p^* : v = au - b$ in the dual plane.

Now let $p_1, \ldots, p_n$ be the vertices of a regular n-gon in the dual plane that is oriented such that the diagonal $p_{l-1}p_{l+1}$ connecting the two neighbors of the leftmost vertex p_l is almost parallel to the v-axis and has negative slope—see Figure 2.1. The trajectories $\xi_1, \cdots, \xi_n$ in our lower-bound instance are the primals of the vertices p_i, that is, $\xi_i^* = p_i$. In the remainder of this section we will prove a lower bound on the maintenance cost of any comparison graph for this instance whose sorting cost (in the comparison-graph sorting model) is bounded by Q, where Q is a parameter with $0 \leqslant Q < n$.

For any pair of vertices p_i, p_j, let ℓ_{ij} denote the line passing through p_i and p_j. Since the p_i are the vertices of a regular n-gon, the lines ℓ_{ij} have only n distinct slopes. Indeed, because the polygon is regular, the slope of edge p_ip_{i+1} is the same as the slope of the diagonals $p_{i-1}p_{i+2}$, $p_{i-2}p_{i+3}$, etc. Note that ℓ_{ij} corresponds to the intersection of ξ_i and ξ_j in the tx-plane, with the slope of ℓ_{ij} being equal to t-coordinate of the intersection. This implies that the intersection points of the trajectories in the tx-plane have only n distinct t-values. Let $t_1, \cdots, t_n$ be the sorted sequence of these t-values. The times $t_1, \ldots, t_n$ define $n+1$ open time intervals $(-\infty, t_1), (t_1, t_2), \cdots, (t_n, +\infty)$. Since no two trajectories intersect inside any of these intervals, the order of the point objects is the same throughout any interval. We say that x_i is *directly below* x_j in such an interval if $x_i(t) < x_j(t)$ for times t in the interval and there is no other point object x_k in between them in that interval. Furthermore, we call a vertex p_i a *lower vertex* if it lies on the lower part of the boundary of the n-gon, and we call p_i an *upper vertex* if it lies on the upper part of the boundary of the n-gon; the leftmost and rightmost vertices are neither upper nor lower vertices.

Lemma 2.1

(i) The object x_i is directly below each other point object x_j in at least one time interval and at most three time intervals.

(ii) There are $\lceil n/2 \rceil$ objects x_i such that x_i is directly below each other object x_j in exactly one time interval.

Proof.

(i) Consider two objects x_i, x_j. Since ξ_i and ξ_j intersect, there is at least one interval—just before or just after the intersection—where x_i is directly below x_j. Since there are $n+1$ intervals, and x_i is below each of the $n-1$ other objects in at least one interval, x_i can be directly below x_j in at most three time intervals.

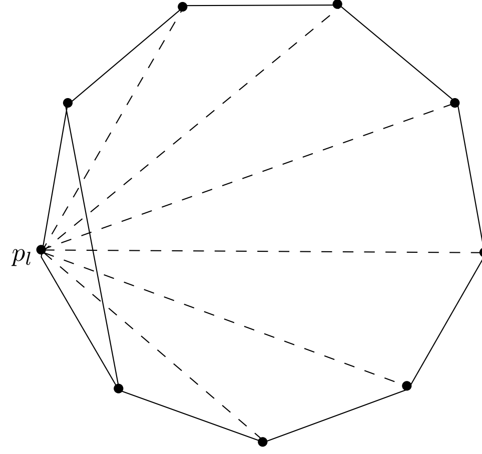


Figure 2.1 The orientation of the regular n-gon.

(ii) First we show that if p_i is a lower vertex or leftmost vertex, x_i is directly below each other object x_j in exactly one time interval. Since there are $n+1$ intervals, and x_i is below each of the $n-1$ other objects in at least one interval, we need to prove that x_i is above all other objects in the two remaining intervals. To prove this, we note that ξ_i appears on the upper envelope of the trajectories in the tx-plane, since p_i is a lower or leftmost vertex. Let p_i be a lower vertex. Assume that ξ_i appears on the upper hull at time t when it intersects ξ_j, and assume it disappears at time t' when it intersects $\xi_{j'}$. Then ξ_j and $\xi_{j'}$ must cross between times t and t', showing that ξ_i is on the upper envelope during two time intervals.

Now consider the leftmost vertex p_l. This vertex is on the upper envelope at time $t = -\infty$. Because of the orientation of the n-gon, the slope of the diagonal connecting the two neighbors of p_l is smaller than the slope of any diagonal (or edge) incident to p_l—see Figure 2.1. This means that the two objects that are initially below ξ_l^* intersect before ξ_l^* disappears from the envelope, which proves x_l is also above all other objects in the two remaining intervals.

When n is even, the number of vertices that are a lower or leftmost vertex is $n/2$, and we are done. For odd n, we will argue that the object x_r corresponding to the rightmost vertex is also directly below each other object exactly once.

To show this, we will argue that ξ_r appears on the upper envelope during two time intervals. Indeed, there are $n+1$ time intervals and the $\lfloor n/2 \rfloor$ leftmost and lower vertices correspond to trajectories appearing twice. Hence there are $n+1-\lfloor n/2 \rfloor = 2$ time intervals where some other trajectory must appear on the upper envelope. This must be ξ_r, because the trajectories of upper vertices will not show up on the upper envelope.

□

We can now prove the lower bound. Suppose that we have a comparison graph on the point objects whose sorting cost is Q during each of the time intervals defined above. This implies that during each such time interval, there must be at least $n - Q - 1$ arcs (x_i, x_j) in the comparison graph such that x_i is directly below x_j, because each of the $n - 1$ adjacent pairs must have an arc, and we are allowed to add only Q arcs to answer the query. In total, $(n + 1)(n - Q - 1)$ arcs are needed over all $n + 1$ time intervals. Some arcs, however, can be used in more than one interval. For x_i, let k_i be the number of arcs of the form (x_i, x_j) that are used. For any of the $\lceil n/2 \rceil$ objects x_i for which case (ii) of Lemma 2.1 applies, all these arcs are distinct. For the remaining $\lfloor n/2 \rfloor$ objects case (i) applies and so at least $k_i - 2$ arcs are distinct. Hence, the total number of arcs inserted over time is at least $(n+1)(n-Q-1) - 2\lfloor n/2 \rfloor \geqslant n(n-Q-2)$. We get the following theorem.

Theorem 2.2 *For any $n \geqslant 1$, there is an instance of n point objects moving with constant velocities on the real line, such that any comparison graph whose worst-case sorting cost in the comparison-graph cost model is Q must have maintenance cost at least $n(n-Q-2)$, for any parameter Q with $0 \leqslant Q < n$.*

2.2.3 A lower bound in the algebraic decision-tree model

In the previous section we gave a lower bound for the maintenance cost for a given sorting cost Q in comparison-graph sorting model. Obviously, this is also a lower bound for the algebraic decision-tree model. Hence, the results of the previous section imply that for any sorting cost $Q = o(n)$ in the algebraic decision-tree model, the worst-case maintenance cost is $\Omega(n^2)$. Since with $O(n^2)$ maintenance cost we can process all swaps—assuming the trajectories are bounded-degree algebraic, so that any pair swaps at most $O(1)$ times—this bound is tight: with $O(n^2)$ maintenance cost we can achieve sorting cost zero. What remains is to investigate the range where the sorting cost is $\Omega(n)$ and $o(n \log m)$, where $1 < m \leqslant n$.

Recall that the sorting cost of a given comparison graph in the algebraic decision-tree model is at least $\log N$, where N is the number of different orderings that are compatible with the comparison graph. We use this to prove the following lemma.

Lemma 2.3 *There is a positive constant c such that if the sorting cost of a comparison graph is at most $cn \log m$, then there is a path in the comparison graph whose length is at least $n/m^{1/3}$.*

Proof. Let k be the length of the longest path in the comparison graph. We define the *level* of the point object x_i as the length of the longest path to x_i in the comparison graph. Let n_j be the number of objects at level j. Since the order of the objects in the same level is not determined by the information in the comparison graph, the number of permutations

compatible with the comparison graph is at least $n_0!n_1!\cdots n_k!$. Note that $\sum_{i=0}^{k} n_i = n$, so $n_0!n_1!\cdots n_k!$ is minimized when the n_i's are all equal to $\lceil n/(k+1)\rceil$ or $\lfloor n/(k+1)\rfloor$. Hence

$$n_0!n_1!\cdots n_k! \geqslant \left(\lfloor \frac{n}{k+1}\rfloor !\right)^{k+1},$$

and the sorting cost in the algebraic decision-tree model is at least $\log((\lfloor n/(k+1)\rfloor !)^{k+1})$, which is at least $c_1 n \log(n/k)$ for some constant c_1. Since the sorting cost of the comparison graph is at most $cn \log m$, we must have $c_1 \log(n/k) \leqslant c \log m$, which implies that $n/k \leqslant m^{c/c_1}$. So for $c = c_1/3$ we have $k \geqslant n/m^{1/3}$, as claimed. □

Next we describe the lower bound construction. As before, it will be convenient to describe the construction in the dual plane. To this end, let $G_a := \{0, 1, \cdots, a-1\}^2$ be the $a \times a$ grid. The trajectories of the point objects in our lower-bound instance will be straight lines in the tx-plane, such that the duals of these lines are the grid points of $G_{\sqrt{n}}$. (We assume for simplicity that n is a square number.) Before we proceed, we need the following lemma.

Lemma 2.4 *Let $p = (p_x, p_y)$ be a grid point of $G_{\sqrt{n}}$ and $p_x, p_y \leqslant a$, where $a \leqslant \sqrt{n}/2$. Let ℓ_p be the line through the origin and p. The number of different lines passing through at least one point of $G_{\sqrt{n}}$ and being parallel to ℓ_p is at most $4a\sqrt{n}$.*

Proof. Let B be the smallest box containing $G_{\sqrt{n}}$ that has one edge (in fact, two) parallel to ℓ_p. Thus B is a bounding box for $G_{\sqrt{n}}$ whose orientation is defined by ℓ_p. Besides the points from $G_{\sqrt{n}}$, the box B will contain more points with integer coordinates; in the remainder of this proof we will call these points grid points as well. The number of grid points inside B is at most $2n$. Moreover, because $p_x, p_y \leqslant a$, any line passing through a grid point and being parallel to ℓ_p contains at least $\lfloor \sqrt{n}/a \rfloor \geqslant \sqrt{n}/(2a)$ grid points inside B. Since two distinct lines parallel to ℓ_p cannot have any grid points in common, this implies that the number of such lines containing at least one grid point is at most

$$\frac{2n}{\sqrt{n}/(2a)} = 4a\sqrt{n}.$$

□

We are now ready to prove the main result of this section.

Theorem 2.5 *For any $n \geqslant 2$, there are positive constants c and c' such that there is an instance of n point objects moving with constant velocities on the real line such that, for any m with $1 < m \leqslant n$, any comparison graph whose worst-case sorting cost in the algebraic decision-tree model is $Q \leqslant cn \log m$, must have maintenance cost at least $c'n^2/m$.*

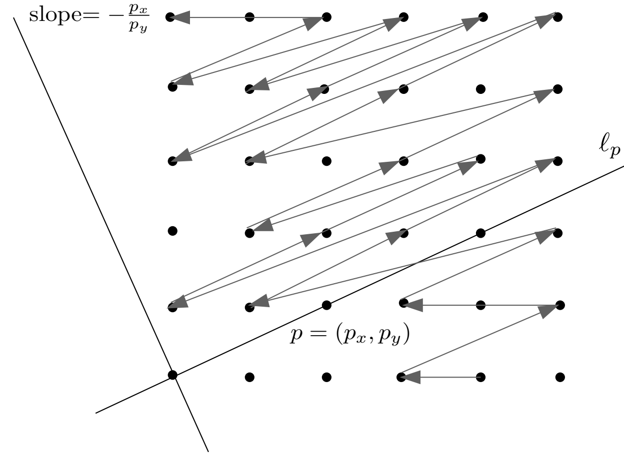


Figure 2.2 A long path uses many arcs parallel to ℓ_p.

Proof. As mentioned above, the trajectories ξ_i of the point objects x_i that constitute our lower-bound instance will be straight lines in the tx-plane, whose dual points ξ_i^* form the grid $G_{\sqrt{n}}$.

Let $a := \sqrt{n}/(8m^{1/3})$ and c be the constant of Lemma 2.3. Consider the comparison graph at some time $s + \varepsilon$ with $s = p_y/p_x$, where $p_x, p_y \leqslant a$ and $\varepsilon > 0$ is sufficiently small. Suppose the sorting cost at time s is at most $cn \log m$. Then by Lemma 2.3 there must be a path in the comparison graph of length at least $n/m^{1/3}$. We claim (and will prove below) that at least half of the arcs in this path are between point objects x_i, x_j such that ξ_i^* and ξ_j^* lie on a common line of slope s—see Figure 2.2. The number of distinct values for s is equal to the number of pairs (p_x, p_y) where p_x and p_y are integer numbers between 0 and $a - 1$ (including 0 and $a - 1$) and $GCD(p_x, p_y) = 1$. Because of symmetry, we count the number of pairs (p_x, p_y) with the property $p_x \leqslant p_y$. For a nonnegative integer i, let $\varphi(i)$ be the number of nonnegative integers that are less than i and relatively prime to i. Then the number of pairs (p_x, p_y) with the desired properties is $\sum_{i=1}^{a-1} \varphi(i)$. It is known [97] that this summation is $\Theta(a^2)$. Then, the total number of arcs needed over all times of the form $p_x/p_y + \varepsilon$ with $p_x, p_y \leqslant a$ is at least

$$n/(2m^{1/3}) \cdot \Theta(a^2) \quad \geqslant \quad c'n^2/m \quad \text{for some constant } c',$$

which proves the theorem.

It remains to prove the claim that at least half of the arcs in the path are between point objects x_i, x_j such that ξ_i^* and ξ_j^* lie on a common line of slope s. Note that the sorted order of the point objects $x_i(s + \varepsilon)$ corresponds to the sorted order of the orthogonal projections of the points ξ_i^* onto a line with slope $-1/(s + \varepsilon)$. If $\varepsilon > 0$ is sufficiently small, then the projections of all the points lying on a common line of slope s will be

adjacent in this order. Let's group the point objects x_i into subsets such that any two point objects x_i, x_j for which ξ_i^* and ξ_j^* lie on a common line of slope s are in the same subset. Then, at time $s+\varepsilon$, any path in the comparison graph can enter and leave a subset at most once. By Lemma 2.4 the number of subsets is at most $4a\sqrt{n}$. Hence, the number of arcs connecting point objects in the same subset is at least

$$n/m^{1/3} - 4a\sqrt{n} = n/m^{1/3} - 4(\sqrt{n}/(8m^{1/3}))\sqrt{n} = n/(2m^{1/3}),$$

as claimed. □

2.2.4 Upper bounds for kinetic sorting

The following theorem shows the bounds in Theorem 2.5 are tight.

Theorem 2.6 *Let S be a set of n point objects moving on the line, such that any pair of points swaps $O(1)$ times. For any m with $1 < m \leqslant n$, there is a data structure whose maintenance cost is $O(n^2/m)$ such that at any time a sorted list of the points in S can be constructed in $O(n \log m)$ time.*

Proof. Partition the set S into m subsets of size at most n/m in an arbitrary manner. For each subset, maintain a sorted array of all its points. When two point objects in the same subset swap, the array can be updated in $O(1)$ time. Since each pair of objects changes order $O(1)$ times, the maintenance cost of each subset is $O(n^2/m^2)$. Since there are m subsets, the total maintenance cost is $O(n^2/m)$. To generate a sorted list of all the point objects, we have to merge the m sorted arrays, which can be done in $O(n \log m)$ time. □

2.3 Gift-wrapping and convex-hull containment queries

Let $S = \{p_1, \cdots, p_n\}$ be a set of point objects moving in the plane such that the position of p_i at time t is $(x_i(t), y_i(t))$, where x_i and y_i are polynomials of degree at most δ. Recall that a *gift-wrapping query* is defined as follows: given a point q and a line ℓ through q such that all points from S lie on the same side of ℓ, report the point object $p_i \in S$ that is hit first when ℓ is rotated (in counterclockwise direction, say) around q. We call a KDS for such queries a *gift-wrapping KDS*. We want to find a gift-wrapping KDS of near-linear size with good trade-offs between the maintenance cost and the query time. We also want to answer *extreme-point queries* (given a query direction $\overrightarrow{d}$, what is the point from S that is extreme in direction $\overrightarrow{d}$?) and *convex-hull containment queries* (given a query point q, is q inside the current convex hull?).

One easy solution is the following: partition the set S into Q subsets of roughly size n/Q and maintain each subset using the kinetic convex-hull structure of Basch *et al.* [23]. Since we can answer a gift-wrapping query on each subset in $O(\log n)$ time (if we have the convex hull in a sorted array or balanced search tree), we can answer gift-wrapping queries on S in $O(Q \log n)$ time. Extreme-point queries and convex-hull containment queries can also be answered with this structure. The total maintenance cost will for this KDS would be $Q \cdot (n/Q)^{2+\varepsilon} = O(n^{2+\varepsilon}/Q)$. Next we describe a KDS that can answer all three types of queries as well, and is more efficient than the easy solution described above.

The data structure. Consider the following transformation on (the trajectories of) the point objects in S: the point object $p_i(t) = (x_i(t), y_i(t))$, where $x_i(t) = x_{i,\delta}t^\delta + \cdots + x_{i,0}$ and $y_i(t) = y_{i,\delta}t^\delta + \cdots + y_{i,0}$, is mapped to the point $p_i^* = (x_{i,\delta}, \cdots, x_{i,0}, y_{i,\delta}, \cdots, y_{i,0})$ in $2(\delta + 1)$-dimensional space.

Lemma 2.7 *If the point object $p_i(t)$ is more extreme than $p_j(t)$ in direction $\overrightarrow{d} = (d_x, d_y)$, then p_i^* is more extreme than p_j^* in direction $\overrightarrow{d^*} = (t^\delta d_x, \cdots, td_x, d_x, t^\delta d_y, \cdots, td_y, d_y)$.*

Proof. If $p_i(t)$ is more extreme than $p_j(t)$ in direction $\overrightarrow{d} = (d_x, d_y)$, then $p_i(t) \cdot \overrightarrow{d} > p_j(t) \cdot \overrightarrow{d}$. Plugging in the polynomials defining the coordinates of $p_i(t)$ we see that this is equivalent to $p_i^* \cdot \overrightarrow{d^*} > p_j^* \cdot \overrightarrow{d^*}$. Hence, p_i^* is more extreme than p_j^* in direction $\overrightarrow{d^*}$. □

Our gift-wrapping KDS is a combination of the data structure for half-space emptiness queries (in $2(\delta+1)$-dimensional space) as described by Matousek and Schwarzkopf [85], and the kinetic convex-hull structure (in the plane) of Basch *et al.* [23]. It is defined recursively, as follows.

Let $S_v \subset S$ be the subset of points for which we are constructing the KDS. Initially, $S_v = S$.

- If $|S_v| \leqslant n/Q^{1+1/\delta}$, where n is the number of points in the original set S, then S_v is stored in the kinetic convex-hull structure of Basch *et al.* [23].
- Otherwise we transform (the trajectories of) the points in S_v to obtain a static set S_v^* of points in $2(\delta + 1)$-dimensional space as described above, and we proceed as Matousek and Schwarzkopf [85]: We construct a simplicial partition Ψ_v for S_v^*—a partitioning of S_v^* into $O(r)$ subsets $S_{v,i}^*$ for some suitably large constant r, each of size between n/r and $n/2r$, and for each subset $S_{v,i}^*$ a simplex containing it—using Matoušek's partition theorem for shallow hyperplanes [84]. The simplicial partitioning Ψ_v has the following property: any hyperplane h for which one of its half-spaces has less than n/r points from S_v^* crosses $O(r^{1-1/\lfloor d/2 \rfloor} + \log r)$ simplices of Ψ_v. We also construct a $(1/r)$-net R_v^* of size $O(r \log r)$ for S_v^*, that

is, a subset of S_v^* such that any halfspace containing at least n/r points of S_v^* must contain at least one point of R_v^*.

The structure now consists of a root node where we store the simplices of Ψ_v and the $(1/r)$-net R_v^*. The root has a subtree for each of the subsets $S_{v,i}^*$ (or rather, the set $S_{v,i}$ of their pre-images), which is constructed recursively.

Gift-wrapping queries. A gift-wrapping query with a line ℓ rotating around a point q can be answered as follows. Suppose we are at some node v of the structure. If $|S_v| \leqslant n/Q^{1+1/\delta}$, then we have the convex hull of S_v explicitly available, so we can answer the query in $O(\log |S_v|)$ time. Otherwise, we find the point $p_i \in R_v$ (the set of point objects that are the pre-images of the points in R_v^*) hit first by ℓ, in a brute-force manner in $O(|R_v|) = O(r \log r)$ time. Let $\ell_q(p_i)$ be the line through q and p_i. Note that all points from R_v lie to the same side of, or on, $\ell_q(p_i)$. Let $\overrightarrow{d}$ be the vector orthogonal to $\ell_q(p_i)$ and pointing in the direction where there are no points from R_v. Then the answer to the query must either be the point p_i, or it must be a point $p_j \in S_v \setminus R_v$ that is more extreme than p_i in the direction $\overrightarrow{d}$. Transform $\overrightarrow{d}$ into a vector $\overrightarrow{d}^*$ in $2(\delta+1)$-dimensional space, as in Lemma 2.7, let h^* be the hyperplane through p_i^* and orthogonal to $\overrightarrow{d}^*$, and let $(h^*)^+$ be the half-space defined by h^* and the vector $\overrightarrow{d}^*$. By Lemma 2.7, any point p_j that is more extreme than p_i in direction $\overrightarrow{d}$ is mapped to a point p_j^* that lies in $(h^*)^+$. Moreover, none of the points in R_v^* lie in $(h^*)^+$, which means that $(h^*)^+$ contains less than n/r points. It follows that no simplex of Ψ_v can lie completely inside $(h^*)^+$. Hence, the query can be answered by recursing only into the subtrees corresponding to intersected simplices, and selecting from the answers found the first point hit.

Extreme-point queries. Extreme-point queries can be answered in a similar way: if we are at a node v with $|S_v| \leqslant n/Q^{1+1/\delta}$, answer the query using the convex hull of S_v. Otherwise, find the point $p_i \in R_v$ that is extreme in the direction $\overrightarrow{d}$, and recurse into subtree corresponding to simplices of Ψ_v that are intersected by h^*, where h^* is the hyperplane through p_i^* and orthogonal to $\overrightarrow{d^*}$.

Convex-hull containment queries. Let $CH(S)$ denote the convex hull of a set S. The query point q lies outside $CH(S)$ if only if there are two half-lines with origin q and tangent to $CH(S)$. Our algorithm is recursive. Suppose we are at some node v of the structure. The algorithm returns true when $q \in CH(S_v)$ and it returns two tangent half-lines for S_v otherwise.

If $|S_v| \leqslant n/Q^{1+1/\delta}$, we have $CH(S_v)$ available. We test whether $q \in CH(S_v)$ and if so return true. Otherwise we return two tangent half-lines for $CH(S_v)$. This takes $O(\log |S_v|)$ time.

Now, assume v is an internal node. If $q \in CH(R_v)$, then $q \in CH(S_v)$ and we return true.

Otherwise, two points $p_i, p_j \in R_v$ are computed such that the lines ℓ_{p_i} and ℓ_{p_j} passing through q and p_i resp. p_j are tangent to $CH(R_v)$. Let $\overrightarrow{d_{p_i}}$ ($\overrightarrow{d_{p_j}}$) be the vector orthogonal to ℓ_{p_i} (ℓ_{p_j}) and pointing in the direction where there are no points from R_v. Let $h^*_{p_i}$ ($h^*_{p_j}$) be the hyperplane through p^*_i (p^*_j) and orthogonal to $\overrightarrow{d_{p_i}}$ ($\overrightarrow{d_{p_j}}$). Using the same reasoning as for gift-wrapping queries, we can argue that we only have to recurse into subtrees of simplices intersected by $h^*_{p_i}$ or $h^*_{p_j}$. If one of these calls returns true, we also return true. Otherwise we collect all the tangent half-lines. If there is no line through q such that all half-lines lie to the same side of the line, we return true. Otherwise, from these we can easily select the two half-lines that are tangent to $CH(S_v)$ and return them.

Theorem 2.8 *Let $S = \{p_1, \ldots, p_n\}$ be a set of moving point objects in the plane such that the position of $p_i(t) = (x_i(t), y_i(t))$, where x_i and y_i are polynomials with degree at most δ. For any Q with $1 \leqslant Q \leqslant n$ and any $\varepsilon > 0$ there is a KDS that handles $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events such that gift-wrapping queries, extreme-point queries, and convex-hull containment queries can be answered in $O(Q \log n)$ time. The KDS uses $O(n \log n)$ storage, and events can be handled in $O(\log^2 n)$ time.*

Proof. The partition-tree part of our KDS is static—it stores the trajectories rather than the current positions of the points—so events only occur in the kinetic convex-hull structures. There are $Q^{1+1/\delta}$ such structures, each of them storing at most $n/Q^{1+1/\delta}$ point objects and processing $O((n/Q^{1+1/\delta})^{2+\varepsilon})$ events [23]. In total, this gives $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events. The bounds on the storage and the time needed to handle an event follow directly from the bounds on the kinetic convex-hull structure [23].

Next we bound the time for a gift-wrapping query; the analysis for extreme-point queries and for convex-hull containment queries is similar. $T(m)$, the query time on a subtree storing m points, satisfies the following recurrence:

$$T(m) = \begin{cases} O(\log m) & \text{if } m \leqslant \frac{n}{Q^{1+1/\delta}} \\ & \\ O(r \log r) + & \\ O(r^{1-1/(\delta+1)} + \log r) \cdot T(2m/r) & \text{otherwise.} \end{cases}$$

For any ε, we can choose r sufficiently large such that the solution of the recurrence is $O(Q^{1+\varepsilon} \log m)$. The factor Q^{ε} in the query time can be avoided by replacing Q by $Q^{1-\varepsilon}$; this gives an extra factor $Q^{\varepsilon(1+1/\delta)})$ in the number of events, which is swallowed by the n^{ε} factor that we already have in the number of events.

□

Remark: Instead of switching to the kinetic convex hull structure of Basch *et al.* when the number of points becomes small, we could also dualize the points in S^*_v and switch to a structure based on cuttings in $2(\delta + 1)$-dimensional space. This would lead to a structure with the same query time and no events to be processed, but with a much higher storage cost.

Using the gift-wrapping KDS we can easily reconstruct the convex hull:

Corollary 2.9 *Let $S = \{p_1, \ldots, p_n\}$ be a set of moving point objects in the plane such that the position of $p_i(t) = (x_i(t), y_i(t))$, where x_i and y_i are polynomials with degree at most δ. For any Q with $1 \leqslant Q \leqslant n$ and any $\varepsilon > 0$ there is a kinetic data structure that handles $O(n^{2+\varepsilon}/Q^{1+1/\delta})$ events, such that we can reconstruct the convex hull of S at any time in $O(hQ \log n)$ time, where h is the number of vertices of the convex hull. The KDS uses $O(n \log n)$ storage, and each event can be handled in $O(\log^2 n)$ time.*

Proof. Maintain the gift-wrapping KDS of Theorem 2.8 on the points, and maintain a kinetic tournament tree [23] on the y-coordinates of the points. Using the kinetic tournament tree, we always have the lowest point of S available, which implies that we can reconstruct the convex hull by $O(h)$ gift-wrapping queries. The number of events in the kinetic tournament tree is $O(n \log n)$, which is subsumed by the number of events in the gift-wrapping KDS. □

Note that the structure can not easily handle flight plan updates–see also the discussion of the end of the conclusion.

2.4 Conclusions

We have studied trade-offs for the kinetic sorting problem, which is to maintain a KDS on a set of points moving on the real line such that one can quickly generate a sorted list of the points, at any given time. We have proved a lower bound for this problem showing the following: with a subquadratic maintenance cost one cannot obtain any significant speed-up on the time needed to generate the sorted list (compared to the trivial $O(n \log n)$ time), even for linear motions.

This negative result gives a strong indication that good trade-offs are not possible for a large number of geometric problems—Voronoi diagrams and Delaunay triangulations, for example, or convex hulls—as the sorting problem can often be reduced to such problems (This is not a formal proof, because our lower-bound model is not suitable for computing convex hulls or Voronoi diagrams). For the convex-hull problem, however, we have shown that good trade-offs are possible if the number of vertices of the convex hull is small. We obtained this result by developing a KDS for gift-wrapping queries, which is of independent interest. Our structure can also answer extreme-point queries and convex-hull containment queries. It would be interesting to see if we can develop a KDS with a similar performance for line-intersection queries: report the intersection points of a query line ℓ with the current convex hull?

Another open problem is to make the KDS less sensitive to changes in the motions of the points. In our structure, a change in motion means we have to delete the point (or rather, its trajectory) from the structure and reinsert the new trajectory. Using the dynamic version

of the structure of Matoušek and Schwarzkopf [85] this might be possible, but it would be nicer if we had a structure where no changes are needed (except for a recomputation of the failure times) when a point changes its motion. Note, however, that with a small change in the definition of our structure we can at least ensure that it will function correctly when a point changes its motion. All we have to do is add at every node v one point from each simplex in Ψ_v to the net R_v. With this change the KDS will always report the correct answer, even if we keep the wrong trajectories in the top part of our structure (we still have to update the kinetic convex-hull structures that we store at the "leaves" of our structures, of course, using the update algorithm of Basch *et al.*). Now we no longer have any guarantees on the query time, however.

Chapter 3

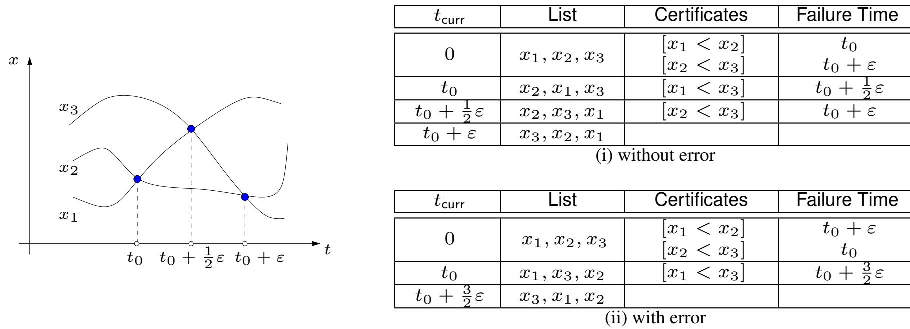
Out-of-order event processing

Abstract. We study the problem of designing kinetic data structures when event times cannot be computed exactly and events may be processed in a wrong order. In traditional KDSs this can lead to major inconsistencies from which the KDS cannot recover. We present more robust KDSs for the maintenance of two fundamental structures, kinetic sorting and tournament trees, which overcome the difficulty by employing a refined event scheduling and processing technique. We prove that the new event scheduling mechanism leads to a KDS that is correct except for finitely many short time intervals. We analyze the maximum delay of events and the maximum error in the structure, and we experimentally compare our approach to the standard event scheduling mechanism.

An extended abstract of this chapter was previously published as: M. A. Abam, P. K. Agarwal, M. de Berg, and H. Yu, Out-of-order event processing in kinetic data structures, In *Proc. European Symposium on Algorithms (ESA)*, pages 624–635, 2006.

3.1 Introduction

Background. In the KDS framework, to be able to process each event at the right time, a global event queue Q is maintained to process the events in the right (chronological) order. This is a priority queue on the events, with the priority of an event being its failure time. Unfortunately, the event scheduling is not as easy as it seems. Suppose that a new certificate arises due to some event. When the failure time of the certificate lies in the past we should not schedule it, and when it lies in the future we should. But what if the event time is equal to the current time t_{curr}? In such a degenerate situation one has to be very careful to avoid an infinite loop. A more serious problem arises when the event times are not computed exactly. This will indeed be the case if the trajectories are polynomials of high degree or more complex curves. As a result, events may be processed in a wrong order, or we may fail to schedule an event because we think it has already taken place. This in turn may not only lead to serious errors in the geometric attribute the KDS is maintaining but also cause the algorithm to crash.



t_{curr}	List	Certificates	Failure Time
0	x_1, x_2, x_3	$[x_1 < x_2]$ $[x_2 < x_3]$	t_0 $t_0 + \varepsilon$
t_0	x_2, x_1, x_3	$[x_1 < x_3]$	$t_0 + \frac{1}{2}\varepsilon$
$t_0 + \frac{1}{2}\varepsilon$	x_2, x_3, x_1	$[x_2 < x_3]$	$t_0 + \varepsilon$
$t_0 + \varepsilon$	x_3, x_2, x_1		

(i) without error

t_{curr}	List	Certificates	Failure Time
0	x_1, x_2, x_3	$[x_1 < x_2]$ $[x_2 < x_3]$	$t_0 + \varepsilon$ t_0
t_0	x_1, x_3, x_2	$[x_1 < x_3]$	$t_0 + \frac{3}{2}\varepsilon$
$t_0 + \frac{3}{2}\varepsilon$	x_3, x_1, x_2		

(ii) with error

Figure 3.1 An example that numerical errors in the event times may cause fatal errors in the KDS. (Left) the trajectories of the points. (Right) the status of the KDS at various times of execution.

As a concrete example, consider the kinetic sorting problem: maintain the sorted order of a set S of points moving on the real line. We store S in a sorted array $A[1..n]$. For each $1 \leqslant i < n$ there is a certificate $\big[A[i] < A[i+1]\big]$. Whenever $A[j] = A[j+1]$ for some j, we have a certificate failure. At such an event we swap $A[j]$ and $A[j+1]$. Furthermore, at most three new certificates arise: $\big[A[j-1] < A[j]\big]$, $\big[A[j] < A[j+1]\big]$, and $\big[A[j+1] < A[j+2]\big]$. We compute the failure time of each of them, based on our knowledge of their current motions, and insert the failure times that are not in the past into the event queue Q. Some certificates may also disappear because the two points involved are no longer neighbors; they have to be deleted from Q. Now suppose that due to errors in the computed failure times the difference between the exact and the computed failure time of each certificate can be as large as ε, for some $\varepsilon > 0$. Consider three moving points x_1, x_2 and x_3 whose trajectories in the tx-plane are depicted in Figure 3.1. Table (i) shows what happens when we can compute the exact failure times. Table (ii) shows what

happens when the computed failure times of the certificates $[x_1 < x_2]$, $[x_1 < x_3]$, and $[x_2 < x_3]$ are $t_0 + \varepsilon$, $t_0 + \frac{3}{2}\varepsilon$, and t_0 respectively: the KDS is not just temporarily incorrect, but gets into an incorrect state from which it never recovers.

This is a serious problem for the applicability of the KDS framework in practice. The goal of this chapter is to address this issue: is it possible to do the event scheduling and processing in such a way that the KDS is more robust under errors in the computation of event times? The KDS may process the events in a wrong order and thus may maintain a wrong geometric attribute from time to time, but we would like the KDS to detect these errors and fix them quickly.

Related work. There is a large body of work on robust computations in geometric algorithms [47, 91, 101], including geometric software libraries [29, 34]. The goal there is to implement various geometric primitives in a robust manner, including *predicates*, which test the sign of an arithmetic expression (e.g., ORIENTATION and INCIRCLE predicates), and *constructions*, which compute the value of an arithmetic expression (e.g., computing the intersection of two lines). There are two broad paradigms. The first approach, exact computation, performs computation with enough precision to ensure predicates can be evaluated correctly. This has been the main paradigm in computational geometry. Many methods have been proposed to remove degeneracies (e.g., simulation of simplicity) and to speedup the computation by adaptively changing the precision (e.g., floating point filters). The second approach focuses on performing computation with finite precision and computing an output as close to the correct one as possible.

Despite much work on robust geometric computation, little has been done on addressing robustness issues in KDSs. One could use exact computation but, as noticed by several researchers [57, 59], in practice a significant portion of the running time of a KDS is spent on computing certificate failure times. Expensive exact root comparisons will only make this worse and, hence, may lead to unacceptable performance in practice. See [58] for a comparison of various exact root computation techniques in the context of kinetic data structures. Guibas and Karavelas [57] described a method to speedup exact root comparisons by grouping the roots into intervals that are refined adaptively. However, like other exact methods, the performance of the algorithm deteriorates when many events are very close to each other.

An alternative is to apply *controlled perturbation* [63] to the KDS. In this method, we perturb the initial positions of the moving objects by some amount δ so that with high probability the roots of all pertinent functions are at least Δ far away from each other. This means one can compare any two roots exactly as long as every root is computed within a precision of $\Delta/2$. While controlled perturbation has been successful on a number of static problems [48, 63], it does not seem to work well on kinetic data structures because the large number of events in the KDS makes the required perturbation bound δ fairly large.

Recently, Milenkovic and Sacks [87] studied the computation of arrangements of x-monotone curves in the plane using a plane sweep algorithm, under the assumption that

intersection points of curves cannot be computed exactly. For infinite curves this boils down to the kinetic sorting problem, because one has to maintain the sorted order of the curves along the sweep line. In fact, our KDS for the kinetic sorting problem is very similar to their algorithm. The difference is in the subroutine to compute intersection points of curves which we assume to have available; this subroutine is stronger than the subroutine they assume—see Section 3.2 for details. This allows us to ensure that we never process more events than the number of actual crossings, whereas Milenkovic and Sacks may process a quadratic number of events in the worst case even when there is only a linear number of crossings. The main difference between our and their papers, however, lies in the different view on the problem: since we are looking at the problem from a KDS perspective, we are especially interested in the delay of events and the error in the output for each snapshot of the motion, something that was not studied in [87]. Moreover, we study other KDS problems as well.

Our results. The main problem we face when event times are not computed exactly is that events may be processed in a wrong order. We present KDSs that are robust against this out-of-order processing, including kinetic sorting and kinetic tournaments. Our algorithms are *quasi-robust* in the sense that the maintained attribute of the moving objects will be correct for most of the time, and when it is incorrect, it will not be far from the correct attribute. For the kinetic sorting problem, we obtain the following results:

- We prove that the KDS can only be incorrect when the current time is close to an event.
- We prove that an event may be processed too late, but not by more than $O(n\varepsilon)$ time. This bound is tight in the worst case.
- We prove bounds on the geometric error of the structure—the maximum distance between the i-th point in the maintained list and the i-th point in the correct list—that depend on the velocities of the points.

We obtain similar results for kinetic tournaments and kinetic range trees. As a by-product of our approach, degeneracy problems (how to deal with multiple events occurring simultaneously) arising in traditional KDS algorithms naturally disappear, because our KDS no longer cares about in which order these simultaneous events are processed.

We have implemented the robust sorting and tournament KDS algorithms and tested them on a number of inputs, including highly degenerate ones. Our sorting algorithm works very well on these inputs: of course it does not get stuck and the final list is always correct (after all, this is what we proved), but the maximum delay of an event is usually much less than the worst-case bound suggests (namely $O(\varepsilon)$ instead of $\Theta(n\varepsilon)$). This is in contrast to the classical KDS, which either falls into an infinite loop or misses many kinetic events along the way and maintains a list that deviates far from the true sorted list both geometrically and combinatorially. Our kinetic tournament algorithm is also robust and reduces the geometric error by orders of magnitude.

3.2 Our model

In this section we describe our model for computing the event times of certificates. In a standard KDS, each certificate c is a predicate, and there is a characteristic function $\chi_c : \mathbb{R} \to \{1, 0, -1\}$ associated with c so that $\chi_c(t) = 1$ if c is true at time t, -1 if c is false at time t. The values of t at which χ_c is switching from 1 to -1 or vice vera are the event times of c, and $\chi_c(t) = 0$ at these event times. In our applications, $\chi_c(t)$ can be derived from the sign of some continuous function $\varphi_c(t)$. For example, if $x(t)$ and $y(t)$ are two points, each moving in $\mathbb{R}^1$, then for the certificate $c := [x < y]$ we have $\chi_c(t) = 1$ if and only if $\mathrm{sign}(\varphi_c, t) > 0$ for $\varphi_c(t) = y(t) - x(t)$. For simplicity, we assume that $\mathrm{sign}(\varphi_c, t) = 0$ for a finite number, s, of values of t.

We assume that the trajectory of each object is explicitly described by a function of time, which means in our applications that the function φ_c is also explicitly described, and that event times can be computed by computing the roots of the function φ_c. These are standard assumptions in traditional KDS's. In order to model the inaccuracy in computing event times, we fix a parameter $\varepsilon > 0$, which will determine the accuracy of the root computation. We assume there is a subroutine, denoted by $\textsc{crop}\big(f(t)\big)$, to compute the roots of a function $f(t)$, whose output is as follows:

(A1) a set of disjoint, open intervals $U_1, \ldots, U_m$, where $|U_i| \leqslant \varepsilon$ for each i, that cover all roots of $f(t)$.

(A2) the sign of $f(t)$ between any two consecutive intervals;

For polynomial functions, Descartes' sign rule [31] and Sturm sequences [69] are standard approaches to implement CROP. We also assume that

(A3) CROP is deterministic: it always returns the same result when run on the same function.

Among the intervals returned by $\textsc{crop}\big(f(t)\big)$, we call an interval whose two endpoints have the same sign a *turbulent interval*, and an interval whose two endpoints have different signs an *event interval*; see Figure 3.2. Let $\mathcal{R}_f$ denote the union of all the turbulent and event intervals. In our applications, we can ignore turbulent intervals (intuitively, we can pretend that the sign of $f(t)$ does not change during a turbulent interval). We will use $\mathcal{I} = \langle I_1, \ldots I_k\rangle$ to denote the set of event intervals, and assume that

(A4) CROP only outputs the set $\mathcal{I}$ of event intervals.

Let λ_j (resp. ρ_j) denote the left (resp. right) endpoint of I_j, i.e., $I_j = (\lambda_j, \rho_j)$. As we will see below, we will always schedule events at the right endpoints of event intervals (intuitively, we can pretend that the sign of $f(t)$ within an event interval is the same as at its left endpoint and that it changes at its right endpoint). Observe that if $f(t)$

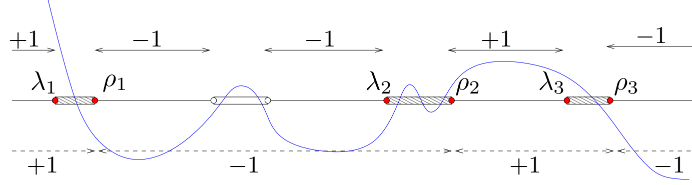


Figure 3.2 A function and the intervals computed by the CROP procedure. Intervals with filled (hollow) endpoints are event (turbulent) intervals; solid arrow lines denote the intervals where the sign of the function is known, and dashed arrow lines denote the signs pretended by the KDS.

does not have any roots, then CROP$\big(f(t)\big)$ does not return any intervals and no events will be scheduled. This is where our subroutine is more powerful than the subroutine of Milenkovic and Sacks [87], and this is why we can ensure that we only handle events if there is a real crossing of trajectories.

We use t_{curr} to denote the current time of the KDS, which is the maximum computed event time over all processed events. We assume that tests as to whether t_{curr} lies inside an event interval computed by CROP are exact. In the actual implementation, this can be achieved by enforcing all interval endpoints (and consequently, t_{curr}) to be rationals and using exact arithmetic to compare between rationals. The pseudo-code for computing the failure time of a certificate c at time t_{curr} is given below.

Algorithm EVENTTIME (c)
1. $\mathcal{I} := \langle I_1 = (\lambda_1, \rho_1), \ldots, I_k = (\lambda_k, \rho_k)\rangle \leftarrow$ CROP (φ_c)
2. $\rho_0 \leftarrow -\infty$; $\rho_{k+1} \leftarrow +\infty$
3. $\mathsf{last} \leftarrow$# intervals in $\mathcal{I}$ to the left of t_{curr}
4. **if** $\chi_c(\rho_{\mathsf{last}}) = -1$
5. **then** return ρ_{last}
6. **else** return $\rho_{\mathsf{last}+1}$

Note that if $\chi_c(\rho_{\mathsf{last}}) = -1$, then the event time returned by EVENTTIME(c) (i.e., ρ_{last}) is in the past. As we will see in the next section, when we handle an event in the past, we do not reset t_{curr}: the time t_{curr} will always be the maximum of the computed event times over all processed events. Finally, the above procedure has the following properties: If it returns a finite value ρ_i, then

(I1) ρ_i is the right endpoint of an event interval;

(I2) the certificate c is valid at λ_i and invalid at ρ_i, i.e., $\chi_c(\lambda_i) = 1$ and $\chi_c(\rho_i) = -1$. In fact, c is valid at all times in $[\rho_{i-1}, \rho_i] \setminus \mathcal{R}_{\chi_c}$, and is invalid at all times in $[\rho_i, \rho_{i+1}] \setminus \mathcal{R}_{\chi_c}$.

Lemma 3.1 *Suppose* EVENTTIME(c) *returns a finite value* ρ_i. *For any* $t \in \mathbb{R}$, *if (i)* $t \in [\rho_{i-1}, \rho_i]$ *and* c *is invalid at time* t, *or (ii)* $t \in [\rho_i, \rho_{i+1}]$ *and* c *is valid at time* t, *then* $\chi_c(\gamma) = 0$ *for some* $\gamma \in (t - \varepsilon, t)$.

Proof. We only prove case (i) as case (ii) is similar. By (A1) and (I2), it is clear that $t \in \mathcal{R}_{\chi_c}$. As such, a turbulent or event interval (λ, ρ) of c contains t. Note that $\lambda \in [\rho_{i-1}, \rho_i]$, and therefore c is valid at time λ by (I2). However, c is invalid at time t by our assumption. This implies that there exists a value $\gamma \in (\lambda, t)$ such that $\chi_c(\gamma) = 0$. Finally, observe that $(\lambda, t) \subseteq (t - \varepsilon, t)$. □

3.3 Kinetic sorting

Let S be a set of n points moving continuously on the real line. The value of a point $x \in S$ is a continuous function of time t, which we denote by $x(t)$. Let $S(t) = \{x(t) : x \in S\}$ denote the configuration of S at time t. For simplicity, we write S and x instead of $S(t)$ and $x(t)$, respectively, provided that no confusion arises. In the kinetic sorting problem, we want to maintain the sorted order of S during the motion.

The algorithm. As in the standard algorithm, we maintain an array A that stores the points in S. The events are stored in a priority queue Q, called global event queue. The certificates are standard as well: the certificate $c := \big[x < y\big]$ belongs to the current certificate set of the KDS if $x = A[k]$ and $y = A[k+1]$ for some $1 \leqslant k \leqslant n-1$. We call these $n-1$ certificates *active*. We need the following notation regarding the failure times.

$t_{\mathsf{cp}}(x, y)$:	the computed failure time[1] of certificate $\big[x < y\big]$
$t_{\mathsf{pr}}(x, y)$:	the time at which the failure of $\big[x < y\big]$ is actually processed
$t_{\mathsf{ex}}(x, y)$:	the exact time at which the certificate $\big[x < y\big]$ fails

For the exact failure time, more formally,

$$t_{\mathsf{ex}}(x, y) := \arg \max_{t < t_{\mathsf{cp}}(x,y)} x(t) = y(t). \tag{3.1}$$

Note that $t_{\mathsf{ex}}(x, y) < t_{\mathsf{cp}}(x, y) \leqslant t_{\mathsf{pr}}(x, y)$. Furthermore, we know by (I1) that $t_{\mathsf{cp}}(x, y)$ is the right endpoint of an event interval of c, and $t_{\mathsf{ex}}(x, y)$ lies inside that event interval by (3.1). As such, $t_{\mathsf{cp}}(x, y) < t_{\mathsf{ex}}(x, y) + \varepsilon$.

[1] This is a slight abuse of notation, because points can swap more than once, so the same certificates can fail multiple times. It will be convenient to treat these certificates as different. Formally we should write $t_{\mathsf{cp}}((x, y), t_{\mathsf{curr}})$ for the failure time of the certificate $\big[x < y\big]$ computed by the KDS at time t_{curr}. Since this is always clear from the context we omit the time parameter.

The new kinetic sorting algorithm is described below. The major difference with the standard algorithm is that we use the algorithm EVENTTIME to compute the failure time of a certificate.

Algorithm KINETICSORTING
1. $t_{\mathsf{curr}} \leftarrow -\infty$; Initialize A and Q.
2. **while** $Q \neq \emptyset$
3. **do** $c : \big[x < y\big] \leftarrow$ DELETEMIN(Q)
4. $t_{\mathsf{curr}} \leftarrow \max\{t_{\mathsf{curr}}, t_{\mathsf{cp}}(x, y)\}$
5. Swap x and y (which are adjacent in A).
6. Remove from Q all certificates that become inactive.
7. $\mathcal{C} \leftarrow$ set of new certificates that become active.
8. **for** each $c : \big[a < b\big] \in \mathcal{C}$
9. **do** $t_{\mathsf{cp}}(a, b) \leftarrow$ EVENTTIME (c)
10. **if** $t_{\mathsf{cp}}(a, b) \neq \infty$
11. **then** Insert $\big[a < b\big]$ into Q, with $t_{\mathsf{cp}}(a, b)$ as failure time.

Note that in lines 10–11, even in the case $t_{\mathsf{cp}}(a, b) < t_{\mathsf{curr}}$ for some certificate $\big[a < b\big] \in \mathcal{C}$ (i.e., the event lies in the past), we still insert this event into the queue because the certificate $\big[a < b\big]$ is not valid at t_{curr} and thus the combinatorial structure of the KDS is not correct. Apparently we missed an event, which we must still handle. As such, unlike the standard algorithm, our algorithm may process events in the past. Note that t_{curr} is not affected when this happens (see line 4).

Basic properties. The status of the KDS *at time* t is defined as the status of the KDS after all events whose processing times are at most t have been processed. In the kinetic sorting problem, the status refers to the maintained array A. We say that a point x *precedes* a point y in the maintained array A if $x = A[k]$ and $y = A[l]$ for some $k < l$. If $k = l - 1$, then x *immediately precedes* y.

Since events may be processed in a wrong order, the above KDS could perhaps get into an infinite loop. However, if a certificate c is processed by the algorithm (line 5) at time t_0 and c becomes active again at time t_0, then EVENTTIME ensures that the failure time of c is in the future. This implies that the algorithm does not get into an infinite loop. We next show the KDS almost always maintains a correctly sorted list in A.

Lemma 3.2 *If x immediately precedes y in A at time t_{curr}, then either (i) $x(t_{\mathsf{curr}}) \leqslant y(t_{\mathsf{curr}})$, which means the order is correct, or (ii) $x(\gamma) = y(\gamma)$ for some $\gamma \in (t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$.*

Proof. Let t^* be the last time less than or equal to t_{curr} at which x becomes a neighbor of y such that x is immediately preceding y. (Note that t^* may be equal to $-\infty$, referring to the time of initialization of the KDS; see lines 1 of KINETICSORTING.) Let $c = \big[x < y\big]$, and

let $t_{\mathsf{cp}}(x,y)$ be the time returned by EVENTTIME(c) at time t^*. Since x and y are always adjacent between time t^* and t_{curr}, either $t_{\mathsf{cp}}(x,y) = \infty$, in which case the certificate failure is not scheduled (line 10), or $t_{\mathsf{curr}} < t_{\mathsf{cp}}(x,y) < \infty$, in which case the certificate failure is scheduled but not yet handled by the KDS. In either case, $t_{\mathsf{cp}}(x,y) > t_{\mathsf{curr}}$. Now assume case (i) is not true, i.e., the certificate c is invalid at time t_{curr}. By Lemma 3.1 (i), there exists a value $\gamma \in (t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$ such that $x(\gamma) - y(\gamma) = 0$, which is case (ii), as desired. □

The following theorem shows when the ordering maintained by the kinetic sorting algorithm is correct.

Theorem 3.3 *(Correctness). The ordering maintained by the kinetic sorting algorithm is correct except during at most μ time intervals of length at most ε, where μ is the number of collisions of points in S over the entire motion.*

Proof. Let $t \in \mathbb{R}$ be a time such that no two points of S collide within time $(t-\varepsilon, t)$. We claim that the ordering maintained by the KDS at time t must be correct. The theorem then follows since there are only μ collisions of points in S.

Suppose at time t there exist two points $x, y \in S$ that are adjacent in A but in incorrect order. By Lemma 3.2 applied to x and y at time t, we have $x(\gamma) = y(\gamma)$ for some $\gamma \in (t-\varepsilon, t)$. But this contradicts with our assumption that no two points collide within the time interval $(t-\varepsilon, t)$. Therefore all adjacent pairs of points in the maintained list A are in correct order, implying that the list A itself must also be correct. □

Delay of events. Theorem 3.3 shows that the ordering may be incorrect only near collision times, but many collisions may "cascade" and thus an event may not be processed for a long time, thereby resulting in a wrong ordering in the KDS for a long time. Specifically, when the failure of a certificate $\left[x < y\right]$ is handled by the KDS, we define its *delay* by $t_{\mathsf{pr}}(x,y) - t_{\mathsf{ex}}(x,y)$. Next we bound the maximum delay of an event. The bound holds when every pair of points swaps at most s times for some parameter $s > 0$.

Lemma 3.4 *Let $c = \left[x < y\right]$ be a certificate that fails at the exact time $t_{\mathsf{ex}}(x,y)$ and is handled by the KDS at time $t_{\mathsf{pr}}(x,y)$. Let τ be such that $t_{\mathsf{ex}}(x,y) \leqslant \tau < t_{\mathsf{pr}}(x,y) - \varepsilon$. Then there is a point $p \in S \setminus \{x\}$ such that $x(t) = p(t)$ for some $t \in (\tau, \tau+\varepsilon]$.*

Proof. Suppose to the contrary that $x(t) \neq p(t)$ for all $p \in S \setminus \{x\}$ during the interval $(\tau, \tau+\varepsilon]$. We first claim that $y(t) < x(t)$ during this interval. Indeed, otherwise we have $y(t) > x(t)$ and hence the certificate c is always valid during the interval $(\tau, \tau+\varepsilon]$. However, by applying Lemma 3.1 (ii) with $t = \tau+\varepsilon$, we know that $\chi_c(\gamma) = 0$ for some $\gamma \in (\tau, \tau+\varepsilon)$, a contradiction. (Note that $t = \tau+\varepsilon$ satisfies the condition of Lemma 3.1 (ii) because $t_{\mathsf{cp}}(x,y) < t < t_{\mathsf{pr}}(x,y)$.)

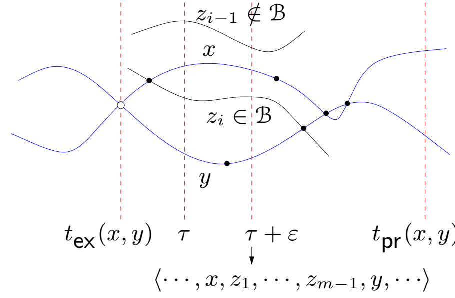


Figure 3.3 Illustration for the proof of Theorem 3.5.

Next, let $A[\tau+\varepsilon]$, the list maintained by the algorithm at time $\tau+\varepsilon$, be $\langle \ldots, x = z_0, z_1, \ldots, z_m = y, \ldots \rangle$. Let $\mathcal{B} \subseteq \{z_1, \ldots, z_m\}$ be the subset of points that are smaller than x during the interval $(\tau, \tau+\varepsilon]$. Since no point collides with x during this interval, $\mathcal{B}$ remains fixed during $(\tau, \tau+\varepsilon]$. Note that $z_m \in \mathcal{B}$. Let $1 \leqslant i \leqslant m$ be the smallest index such that $z_i \in \mathcal{B}$. Then $z_i(t) < x(t) \leqslant z_{i-1}(t)$, for all $t \in (\tau, \tau+\varepsilon]$, and z_{i-1} immediately precedes z_i in $A[\tau+\varepsilon]$, which contradicts Lemma 3.2. This completes the proof of the lemma. □

Theorem 3.5 *(Delay). Suppose that the trajectories of every pair of points in S intersect at most s times. Then an event can be delayed by at most $ns \cdot \varepsilon$ time.*

Proof. Consider a certificate $c = \big[x < y\big]$ that fails at the exact time $t_{\mathsf{ex}}(x,y)$ and is handled by the KDS at time $t_{\mathsf{pr}}(x,y)$. Let t be a time such that $t_{\mathsf{ex}}(x,y) \leqslant t$ and $t+\varepsilon < t_{\mathsf{pr}}(x,y)$. By Lemma 3.4, there is a point $p \in S \setminus \{x\}$ whose trajectory intersects the trajectory of x during $(t, t+\varepsilon]$. Let k be an integer such that $t_{\mathsf{pr}}(x,y) - t_{\mathsf{ex}}(x,y) = k\varepsilon + \delta$ where $\delta < \varepsilon$. We split the interval $[t_{\mathsf{ex}}(x,y), t_{\mathsf{pr}}(x,y)]$ into k intervals, each of width ε, and one interval (the last one) of width δ. Now we can charge each of the first k intervals to an intersection point of the trajectory of x and the trajectory of a point $p \in S$. Since any two trajectories intersect at most s times, k is at most $(n-1)s$, implying that $t_{\mathsf{pr}}(x,y) - t_{\mathsf{ex}}(x,y) \leqslant (n-1)s \cdot \varepsilon + \delta < ns \cdot \varepsilon$. □

The following theorem shows the above bound on the delay is almost tight in the worst case.

Theorem 3.6 *For any n, there is a set S of n points such that the trajectories of any two points intersect at most s times and $t_{\mathsf{pr}}(x,y) - t_{\mathsf{ex}}(x,y) \geq (n-2)s \cdot \varepsilon$ for some pair $x, y \in S$.*

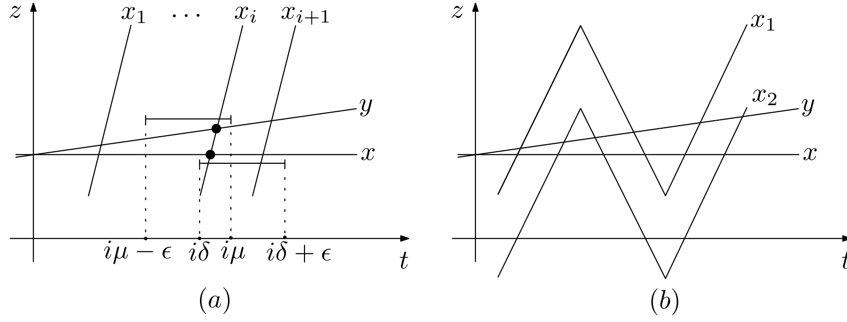


Figure 3.4 The lower-bound example.

Proof. We first describe a lower bound example for linear motions. Suppose that the set S is $\{x, y, x_1, \ldots, x_{n-2}\}$. The trajectories of x and y in the tz-plane are set to be $z = 1$ and $z = at + 1$ for a sufficiently small positive number a. The trajectory of the x_i's in tz-plane is parallel lines such that $i\delta < t_{\mathsf{ex}}(x, x_i) < t_{\mathsf{ex}}(y, x_i) < i\mu$ where δ and μ are two numbers satisfying the following inequalities:

$$\frac{n-1}{n}\varepsilon < \delta < \mu < \left(\frac{n-1}{n} + \frac{1}{n^2}\right)\varepsilon.$$

Assume CROP $(x(t) - y(t)) = ((\mu - \varepsilon)/2, (\mu + \varepsilon)/2)$, CROP $(x(t) - x_i(t)) = (i\delta, i\delta + \varepsilon)$ and CROP $(y(t) - x_i(t)) = (i\mu - \varepsilon, i\mu)$ for any $1 \leqslant i \leqslant n-2$ (see Figure 3.4(a)).

Using induction, we prove the status of the maintained list at time $i\mu$ ($2 \leqslant i \leqslant n-2$) is

$$\langle x_{n-2}, \ldots, x_{i+1}, y, x_i, x_{i-1}, x, x_{i-2}, \ldots x_1 \rangle.$$

Since the maintained list at time $-\infty$ is $\langle x_{n-2}, \ldots, x_1, y, x \rangle$ and the right endpoints of all returned intervals by CROP are greater than zero, the maintained list at time 0 is $\langle x_{n-2}, \ldots, x_1, y, x \rangle$. At time 0, the only certificate failures in the event queue are $t_{\mathsf{cp}}(x, y) = (\mu + \varepsilon)/2$ and $t_{\mathsf{cp}}(y, x_1) = \mu$. Since $\mu < (\mu + \varepsilon)/2$, the status of the KDS at time μ is $\langle x_{n-2}, \ldots, x_2, y, x_1, x \rangle$ and the certificate failures in the event queue are $t_{\mathsf{cp}}(x, x_1) = \delta + \varepsilon$, $t_{\mathsf{cp}}(y, x_2) = 2\mu$. Since we have $2\mu < \delta + \varepsilon$ (later we will prove that $(i+1)\mu < i\delta + \varepsilon$), the status of the KDS at time 2μ is $\langle x_{n-2}, \ldots, x_3, y, x_2, x_1, x \rangle$, which means the case $i = 2$ is clearly true. Now assume the maintained list at time $i\mu$ is $\langle x_{n-2}, \ldots, x_{i+1}, y, x_i, x_{i-1}, x, x_{i-2}, \ldots x_1 \rangle$. We have to show that the maintained list at time $(i+1)\mu$ is $\langle x_{n-2}, \ldots, x_{i+2}, y, x_{i+1}, x_i, x, x_{i-1}, \ldots x_1 \rangle$.

The computed failure times of current certificates in the event queue are $t_{\mathsf{cp}}(x, x_{i-1}) = (i-1)\delta + \varepsilon$, $t_{\mathsf{cp}}(y, x_{i+1}) = (i+1)\mu$. Since $(i-1)\delta + \varepsilon < (i+1)\mu$, the point x_{i-1} swaps with the point x at time $(i-1)\delta + \varepsilon$ and at the same time $t_{\mathsf{cp}}(x, x_{i-1})$ is removed from

the event queue and $t_{\mathsf{cp}}(x, x_i) = i\delta + \varepsilon$ is inserted into the event queue. We know that

$$i\delta + \varepsilon > \frac{n-1}{n} \cdot i\varepsilon + \varepsilon > \left(\frac{n-1}{n} + \frac{1}{n^2}\right)(i+1)\varepsilon > (i+1)\mu.$$

This implies at time $(i+1)\mu$ the points x_{i+1} and y swap and $t_{\mathsf{cp}}(y, x_{i+1})$ is removed from the event queue and instead $t_{\mathsf{cp}}(y, x_{i+2}) = (i+2)\mu$ which is greater than $(i+1)\mu$ is inserted into the event queue. Therefore, the status of the KDS at time $(i+1)\mu$ is

$$\langle x_{n-2}, \ldots, x_{i+2}, y, x_{i+1}, x_i, x, x_{i-1}, \ldots x_1 \rangle.$$

Now consider the time $(n-2)\mu$ at which the maintained list is

$$\langle y, x_{n-2}, x_{n-3}, x, x_{n-4}, \ldots, x_1 \rangle.$$

The only certificate failure scheduled in the KDS is for $[x_{n-3} < x]$, with failure time $(n-3)\delta + \varepsilon$. After processing this certificate failure, the only certificate failure in the event queue is $t_{\mathsf{cp}}(x, x_{n-2}) = (n-2)\delta + \varepsilon$. After processing $[x_{n-2} < x]$, we realize that the certificate $[y < x]$ which fails in the past must be processed. Therefore,

$$t_{\mathsf{pr}}(x, y) - t_{\mathsf{ex}}(x, y) = (n-2)\delta + \varepsilon > (n-2)\varepsilon.$$

We use the above construction as a base component to construct a lower-bound example for the general case where any two points can swap s times. To this end, we glue s base components together such that the slopes of lines alternate between being positive and negative, i.e., the slopes of lines in the first component is positive, in the second component is negative, and so on as depicted in Fig. 3.4(b). Note that in the odd components, certificates $[x_i < y]$ are roughly processed at the right time and certificates $[x_i < x]$ are roughly processed with a delay of ε, but in the even components, certificates $[x < x_i]$ are roughly processed at the right time and certificates $[y < x_i]$ are roughly processed with a delay of ε (indeed we can imagine that x and y are exchanged). The main condition that we need is $(i+1)\mu < i\delta + \varepsilon$ for any $i = 1, \ldots, s(n-2)$. We can satisfy this condition by choosing δ and μ such that

$$\frac{sn-1}{sn}\varepsilon < \delta < \mu < \left(\frac{sn-1}{sn} + \frac{1}{s^2n^2}\right)\varepsilon.$$

Next we discuss what happens to the maintained list when two components are glued together. Because of symmetry, we just consider the status of the KDS around the time at which the first and the second component are glued together. Consider time $(n-2)\mu$ in which the KDS is

$$\langle y, x_{n-2}, x_{n-3}, x, x_{n-4}, \ldots, x_1 \rangle.$$

As we explained above, at time $(n-3)\delta + \varepsilon$, the certificate $[x_{n-3} < x]$ is processed and x and x_{n-2} become adjacent, which means $[x_{n-2} < x]$ must be scheduled. Because two intersections of x and x_{n-2} are at most ε far away from each other, we replace the previous assumption $t_{\mathsf{cp}}(x, x_{n-2}) = (n-2)\delta + \varepsilon$ with the assumption that

the turbulent interval $((n-2)\delta, (n-2)\delta+\varepsilon)$ contains both intersections. Since CROP ignores turbulent intervals, the order of x and x_{n-2} does not change. Moreover, since $t_{\mathsf{cp}}(x_{n-2}, y) = (n-1)\delta + \varepsilon$ (recall that in the even components $[y < x_i]$ is processed with a delay of ε), x_{n-2} and y do not swap before time $(n-1)\delta + \varepsilon$. This implies x and y cannot get adjacent before $(n-1)\delta + \varepsilon$. On the other hand, x_{n-3} and x must swap before this time—note that $t_{\mathsf{cp}}(x_{n-3}, x) = n\mu$. After time $(n-1)\delta + \varepsilon$, the same scenario as the first component happens. Putting everything together we conclude that $t_{\mathsf{pr}}(x, y) - t_{\mathsf{ex}}(x, y) \geq (n-2)s \cdot \varepsilon$ in the above construction. □

Error bounds. We turn our attention to the "error" in the array A. Combinatorially, Lemma 3.2 implies that if there are k event intervals containing t_{curr}, then the array A at time t_{curr} can be decomposed into at most $k+1$ (contiguous) subarrays, each of which is in sorted order. Next we discuss how far the maintained order can be from the correct order geometrically. In particular, we present a bound on the maximum distance between two points that are in the wrong order in the array and on how far away the k-th point in the maintained order—that is, the point $A[k]$—can be from the true point of rank k.

Theorem 3.7 *(Geometric error). Let $\langle y_1, \ldots, y_n \rangle$ and $\langle z_1, \ldots, z_n \rangle$ be the sequence maintained by the algorithm and the correctly sorted sequence at some given time t_{curr}, respectively. Let V_{max} be the maximum velocity of any point in S over the time interval $[t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$. Then for any $1 \leqslant i < j \leqslant n$,*

(i) $y_i(t_{\mathsf{curr}}) - y_j(t_{\mathsf{curr}}) \leqslant (j-i+1)\varepsilon \cdot V_{\mathsf{max}}$, and

(ii) $|y_i(t_{\mathsf{curr}}) - z_i(t_{\mathsf{curr}})| \leqslant n\varepsilon \cdot V_{\mathsf{max}}$.

Proof.

(i) For simplicity we write $t = t_{\mathsf{curr}}$. For any $1 \leqslant k < n$, if y_k and y_{k+1} are in the correct order in the maintained list, then $y_k(t) \leqslant y_{k+1}(t)$. If they are in the incorrect order, then by Lemma 3.2 (ii), there exists a time $\gamma \in (t-\varepsilon, t)$ such that $y_k(\gamma) = y_{k+1}(\gamma)$. Hence,

$$\begin{aligned} y_k(t) - y_{k+1}(t) &= (y_k(t) - y_k(\gamma)) + (y_k(\gamma) - y_{k+1}(\gamma)) \\ &\quad + (y_{k+1}(\gamma) - y_{k+1}(t)) \leqslant 2\varepsilon V_{\mathsf{max}}. \end{aligned}$$

Therefore we always have $y_k(t) - y_{k+1}(t) \leqslant 2\varepsilon V_{\mathsf{max}}$, which immediately implies that $y_i(t) - y_j(t) = \sum_{\ell=i}^{j-1} (y_\ell(t) - y_{\ell+1}(t)) \leqslant 2(j-i)\varepsilon \cdot V_{\mathsf{max}}$ for any $1 \leqslant i < j \leqslant n$. To further prove the promised upper bound, let us consider bounding $y_k(t) - y_{k+2}(t)$. If either $y_k(t) \leqslant y_{k+1}(t)$ or $y_{k+1}(t) \leqslant y_{k+2}(t)$, then we immediately have

$$y_k(t) - y_{k+2}(t) = (y_k(t) - y_{k+1}(t)) + (y_{k+1}(t) - y_{k+2}(t)) \leqslant 2\varepsilon V_{\mathsf{max}}.$$

Now assume $y_k(t) > y_{k+1}(t)$ and $y_{k+1}(t) > y_{k+2}(t)$, which means that the relative order of y_k and y_{k+1}, as well as the relative order of y_{k+1} and y_{k+2} are incorrect in the maintained list. As such, there exist $\gamma_1, \gamma_2 \in (t - \varepsilon, t)$ such that $y_k(\gamma_1) = y_{k+1}(\gamma_1)$ and $y_{k+1}(\gamma_2) = y_{k+2}(\gamma_2)$. It follows that

$$\begin{aligned} y_k(t) - y_{k+2}(t) &= (y_k(t) - y_k(\gamma_1)) + (y_k(\gamma_1) - y_{k+1}(\gamma_1)) \\ &+ (y_{k+1}(\gamma_1) - y_{k+1}(\gamma_2)) + (y_{k+1}(\gamma_2) - y_{k+2}(\gamma_2)) \\ &+ (y_{k+2}(\gamma_2) - y_{k+2}(t)) \\ &\leqslant |t - \gamma_1| \cdot V_{\mathsf{max}} + 0 + |\gamma_1 - \gamma_2| \cdot V_{\mathsf{max}} + 0 + |t - \gamma_2| \cdot V_{\mathsf{max}} \\ &\leqslant 2\varepsilon V_{\mathsf{max}}. \end{aligned}$$

Hence we always have $y_k(t) - y_{k+2}(t) \leqslant 2\varepsilon V_{\mathsf{max}}$. Now, for any $1 \leqslant i < j \leqslant n$, one can prove $y_i(t) - y_j(t) \leqslant (j - i + 1)\varepsilon V_{\mathsf{max}}$ by a simple induction on $j - i$ (the base case $j - i = 1$ has been proved above):

$$\begin{aligned} y_i(t) - y_j(t) &= (y_i(t) - y_{i+2}(t)) + (y_{i+2}(t) - y_j(t)) \\ &\leqslant 2\varepsilon V_{\mathsf{max}} + (j - (i+2) + 1)\varepsilon V_{\mathsf{max}} \\ &\leqslant (j - i + 1)\varepsilon V_{\mathsf{max}}. \end{aligned}$$

(ii) We consier the case $z_i \neq y_i$; otherwise the claim is trivially true. Suppose $z_i = y_j$ for some $j > i$; the other case $j < i$ is symmetric. Also suppose $y_i = z_k$ for some $1 \leqslant k \leqslant n$. We have two cases. If $k > i$, then since $y_j(t_{\mathsf{curr}}) = z_i(t_{\mathsf{curr}}) \leqslant z_k(t_{\mathsf{curr}}) = y_i(t_{\mathsf{curr}})$, we can write

$$|z_i(t_{\mathsf{curr}}) - y_i(t_{\mathsf{curr}})| = y_i(t_{\mathsf{curr}}) - y_j(t_{\mathsf{curr}}) \leqslant n\varepsilon \cdot V_{\mathsf{max}},$$

by (i). Otherwise if $k < i$, there must exist r and ℓ with $r < i < \ell$, such that $z_\ell = y_r$. Then

$$\begin{aligned} |z_i(t_{\mathsf{curr}}) - y_i(t_{\mathsf{curr}})| &= z_i(t_{\mathsf{curr}}) - z_k(t_{\mathsf{curr}}) \leqslant z_\ell(t_{\mathsf{curr}}) - z_k(t_{\mathsf{curr}}) \\ &= y_r(t_{\mathsf{curr}}) - y_i(t_{\mathsf{curr}}) \leqslant n\varepsilon \cdot V_{\mathsf{max}}, \end{aligned}$$

by (i), thus proving the theorem.

□

3.4 Kinetic tournaments

A kinetic tournament [23] is a KDS that maintains the maximum of a set S of moving points in $\mathbb{R}$ by maintaining a tournament tree $\mathcal{T}$ over S. Each interior node u of $\mathcal{T}$ has a certificate of the form $[x < y]$, where $x, y \in S$ are the two points stored at the children of u, and y is also currently stored at u. To handle events, we need a subroutine that compares two points at time t_{curr} in a way that is consistent with EVENTTIME.

Algorithm COMPUTEMAX(x, y)
1. $\mathcal{I} := \langle I_1 = (\lambda_1, \rho_1), \ldots, I_k = (\lambda_k, \rho_k)\rangle \leftarrow$ CROP $(x(t) - y(t))$
2. $\rho_0 \leftarrow -\infty$
3. $\mathsf{last} \leftarrow$number of intervals in $\mathcal{I}$ to the left of t_{curr}
4. **if** $\operatorname{sign}(x(\rho_{\mathsf{last}}) - y(\rho_{\mathsf{last}})) = 1$
5. **then** return x
6. **else** return y

In the algorithm below, the point stored at a node $u \in \mathcal{T}$ is denoted by p_u, and we assume $\operatorname{parent}(\mathit{root}) = $ **nil**.

Algorithm KINETICTOURNAMENT
1. $t_{\mathsf{curr}} \leftarrow -\infty$; Initialize $\mathcal{T}$ and Q.
2. **while** $Q \neq \emptyset$
3. **do** $c : \big[x < y\big] \leftarrow$ DELETEMIN(Q)
4. $t_{\mathsf{curr}} \leftarrow t_{\mathsf{cp}}(x, y)$
5. $u \leftarrow$ the node at which the certificate c fails.
6. **while** $u \neq$ **nil**
7. **do** Let z_1 and z_2 be the points stored at u's children.
8. $p_u \leftarrow$COMPUTEMAX(z_1, z_2); $u \leftarrow \operatorname{parent}(u)$
9. Remove from Q all certificates that become inactive.
10. $\mathcal{C} \leftarrow$ set of new certificates that become active.
11. **for** each $c : \big[a < b\big] \in \mathcal{C}$
12. **do** $t_{\mathsf{cp}}(a, b) \leftarrow$EVENTTIME (c)
13. **if** $t_{\mathsf{cp}}(a, b) \neq \infty$
14. **then** Insert $\big[a < b\big]$ into Q, with $t_{\mathsf{cp}}(a, b)$ as failure time.

The set $\mathcal{C}$ in line 10 consists of certificates that correspond the nodes along the path from the node where the event occurs to the root. In lines 5–8, the algorithm has used COMPUTEMAX to make sure that each certificate $c \in \mathcal{C}$ is valid at the right endpoint of the last event interval of c before time t_{curr}. Since COMPUTEMAX (line 8) and EVENTTIME (line 12) base their decisions on the order at the same time, we obtain the following lemma.

Lemma 3.8 *In line 12, the computed event time $t_{\mathsf{cp}}(a, b)$ is always in the future (i.e., $t_{\mathsf{cp}}(a, b) > t_{\mathsf{curr}}$).*

The lemma implies that we never schedule an event in the past and, in fact, never schedule an event at the current time either. Hence, the algorithm does not get into an infinite loop.

Lemma 3.9 *After an event has been processed at time t_{curr}, the point p_u stored at any internal node u of the tournament is always one of the points stored at its children. Moreover, either p_u is the correct current maximum of the two children, or the trajectories of points stored at the two children intersect during the period $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$.*

Proof. It is obvious that the first part of the lemma is true. The proof of the second part is similar to Lemma 3.2. Assume there is a node u with children u_1 and u_2, and assume without loss of generality that $p_u = p_{u_1}$ while in fact $p_{u_2}(t_{\mathsf{curr}}) > p_{u_1}(t_{\mathsf{curr}})$. Let t^* be the last time at which p_{u_1} and p_{u_2} were compared. Thus COMPUTEMAX(p_{u_1}, p_{u_2}) executed at time t^* returns p_{u_1}. But then, since $p_{u_2}(t_{\mathsf{curr}}) > p_{u_1}(t_{\mathsf{curr}})$, an event must have been scheduled for the certificate $c = [p_{u_2} < p_{u_1}]$, and the failure time t' of this certificate must have satisfied $t' > t^*$ by Lemma 3.8. We cannot have $t' < t_{\mathsf{curr}}$, because that contradicts the definition of t^*. Hence $t' \geqslant t_{\mathsf{curr}}$. Since c is invalid at time t_{curr}, by Lemma 3.1 (i), it follows that that the trajectories of p_{u_1} and p_{u_2} must intersect during the period $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$. □

Following standard KDS terminology, we call an event *external* if the attribute to be maintained changes due to the event; for a kinetic tournament this means an event where the maximum of S changes. Other events are *internal*.

Lemma 3.10 *If there is no external event during the period* $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$*, then the maximum maintained by the algorithm is correct at time* t_{curr}.

Proof. By assumption, the true maximum of S during $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$ is a unique point, x. In particular, x does not cross any other point in S during this time period. Suppose for the sake of contradiction that x is not the maximum maintained by the algorithm at time t_{curr}. Then at time t_{curr}, the algorithm stores x at an internal node v of the tournament tree, and stores another point $y \in S$ in the sibling and the parent u of v. Applying Lemma 3.9 to the node u, we obtain that the trajectories of x and y intersect at some time in $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$, a contradiction. □

The following two results are immediate consequences of Lemma 3.10.

Theorem 3.11 *(Correctness). The maximum maintained by the kinetic tournament is correct except during at most* μ *time intervals of length at most* ε*, where* μ *is the number of external events.*

Theorem 3.12 *(Delay). If a point* $x \in S$ *becomes the true maximum at time* t *(i.e., an external event at time* t*), then either* x *becomes the maintained maximum by time* $t + \varepsilon$ *(i.e., the external event is delayed by at most* ε*), or another external event occurs before time* $t + \varepsilon$ *(i.e., the old external event becomes obsolete).*

We now turn our attention to the geometric error of our KDS—the difference in value between the point stored in the root of the kinetic tournament tree and the true maximum—as a function of the maximum velocity. Interestingly, the geometric error is much smaller than in the sorting KDS, because it now depends on the depth of the tournament tree, which is $\lceil \log n \rceil$. The following theorem makes this precise.

Theorem 3.13 *(Geometric error). Let x denote the point stored in the root of the kinetic tournament tree at some time t_{curr}, and let y denote the point with the maximum value at time t_{curr}. Then $x(t_{\mathsf{curr}}) \geqslant y(t_{\mathsf{curr}}) - (\lceil \log n \rceil + 1)\varepsilon \cdot V_{\mathsf{max}}$, where V_{max} is the maximum velocity of any point in S over the time interval $[t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$.*

Proof. Consider a node v (other than the root) and its parent u. We claim that

$$p_v(t_{\mathsf{curr}}) - p_u(t_{\mathsf{curr}}) \leqslant 2\varepsilon V_{\mathsf{max}}. \tag{3.2}$$

If $p_v(t_{\mathsf{curr}}) \leqslant p_u(t_{\mathsf{curr}})$, (3.2) is trivially true. Otherwise, by Lemma 3.9, the trajectories of p_v and p_u intersect at some time in $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}})$. Arguing as in Theorem 3.7 (i), we can then obtain (3.2). Summing up (3.2) for all consecutive nodes along the path from the node storing the true maximum y to the root, we obtain $y(t_{\mathsf{curr}}) - x(t_{\mathsf{curr}}) \leqslant 2h\varepsilon \cdot V_{\mathsf{max}}$, where $h \leqslant \lceil \log n \rceil$ is the length of the path. The inequality can be further improved to $y(t_{\mathsf{curr}}) - x(t_{\mathsf{curr}}) \leqslant (h+1)\varepsilon \cdot V_{\mathsf{max}}$ by using the same argument as in Theorem 3.7 (i), thus completing the proof. □

3.5 Kinetic range trees

Our robust kinetic sorting algorithm can be applied directly to maintaining the standard kinetic range trees [24] of a set S of moving points in $\mathbb{R}^d$ for orthogonal range searching. By the properties of the robust kinetic sorting algorithm, we immediately know that the robust kinetic range tree is correct except for at most E time intervals of length at most ε, where E is the total number of swaps of the input points along each axis, and that the delay of each event is at most $O(n\varepsilon)$.

We can also prove bounds on the geometric error. For a d-dimensional (axis-aligned) box $R = \Pi_{i=1}^{d}[a_i, b_i]$ and a parameter $\Delta > 0$, let $R_\Delta^- = \Pi_{i=1}^{d}[a_i + \Delta, b_i - \Delta]$ and $R_\Delta^+ = \Pi_{i=1}^{d}[a_i - \Delta, b_i + \Delta]$. We call a subset $Q \subseteq S$ a Δ-*approximation* to $S \cap R$ if

$$S \cap R_\Delta^- \subseteq Q \subseteq S \cap R_\Delta^+.$$

In other words, points at L_∞-distance at most Δ to the boundary of R may or may not be included in Q, but other points are in Q if and only if they are in R. The next theorem shows that the kinetic range tree, when using our robust kinetic sorting algorithm, always returns a Δ-approximation to the true answer of an orthogonal range query, for an appropriate value of Δ. This follows more or less from Theorem 3.7. (The fact that the maintained tree is not necessarily a correct search tree does not impose any difficulty upon performing a standard binary search on the tree.)

Theorem 3.14 *For any time t and any d-dimensional (axis-aligned) box $R \subseteq \mathbb{R}^d$, the subset $Q(t) \subseteq S(t)$ returned by querying R on the maintained kinetic range tree at time t*

is a Δ*-approximation to* $S(t) \cap R$*, where* $\Delta = n\varepsilon V_{\max}$ *and* $V_{\max}$ *is the maximum speed of a point in* S *over the time interval* $[t-\varepsilon, t]$.

Proof. We proceed by induction on d. Let us first consider the one-dimensional case, where a range tree of S is simply a binary search tree of S. Let $\langle y_1(t), y_2(t), \ldots, y_n(t)\rangle$ be the sequence maintained by the algorithm; also let $y_0 = -\infty$ and $y_{n+1} = +\infty$. Suppose for a query range $R = [a, b]$ the maintained tree returns $Q(t) = \langle y_i, y_{i+1}, \ldots, y_j\rangle$. Observe that although the maintained binary search tree is not necessarily correct, we still have $y_{i-1} < a \leqslant y_i$ and $y_j \leqslant b < y_{j+1}$. By Theorem 3.7, for each $i \leqslant \ell \leqslant j$, $y_\ell \geqslant y_i - \Delta \geqslant a - \Delta$ and $y_\ell \leqslant y_j + \Delta \leqslant b + \Delta$. Thus $Q(t) \subseteq S(t) \cap R^+_\Delta$. On the other hand, for each $\ell < i$, $y_\ell \leqslant y_{i-1} + \Delta < a + \Delta$, and for each $\ell > j$, $y_\ell \geqslant y_{j+1} - \Delta > b - \Delta$. This implies $S(t) \cap R^-_\Delta \subseteq Q(t)$. Hence $Q(t)$ is a Δ-approximation to $S(t) \cap [a, b]$.

In $\mathbb{R}^d$, to perform a query $R = \Pi_{i=1}^d [a_i, b_i]$ on the maintained d-dimensional range tree, one first performs the query $[a_1, b_1]$ on the primary range tree, and then performs the query $\Pi_{i=2}^d [a_i, b_i]$ recursively into appropriate secondary range trees. Let $S' \subseteq S$ be the subset of points stored in those queried secondary trees. It follows from the above analysis that

$$S(t) \cap \Big([a_1 + \Delta, b_1 - \Delta] \times \mathbb{R}^{d-1}\Big) \subseteq S'(t) \subseteq S(t) \cap \Big([a_1 - \Delta, b_1 + \Delta] \times \mathbb{R}^{d-1}\Big). \quad (3.3)$$

Furthermore, by the induction hypothesis,

$$S'(t) \cap \Big(\mathbb{R} \times \prod_{i=2}^{d} [a_i + \Delta, b_i - \Delta]\Big) \subseteq Q(t) \subseteq S'(t) \cap \Big(\mathbb{R} \times \prod_{i=2}^{d} [a_i - \Delta, b_i + \Delta]\Big). \quad (3.4)$$

Putting (3.3) and (3.4) together, we obtain $S(t) \cap R^-_\Delta \subseteq Q(t) \subseteq S(t) \cap R^+_\Delta$, as desired. □

3.6 Experiments

We have implemented our robust kinetic sorting and kinetic tournament algorithms to test the effectiveness of our technique for handling out-of-order event processing. The programs are written in C++ and run in the Linux 2.4.20 environment. We also implemented these two algorithms using the traditional KDS event-scheduling approach and compared them with their robust counterparts by testing the errors in the output.

Input data. We used the following synthetic datasets in our experiments, as illustrated in Figure 3.5. The inputs are low-degree motions because we have not yet implemented a full-fledged CROP procedure, and it becomes easier for us to compute delays of the events. Nonetheless, these inputs already cause trouble to traditional KDSs and are sufficient to illustrate the effectiveness of our algorithms.

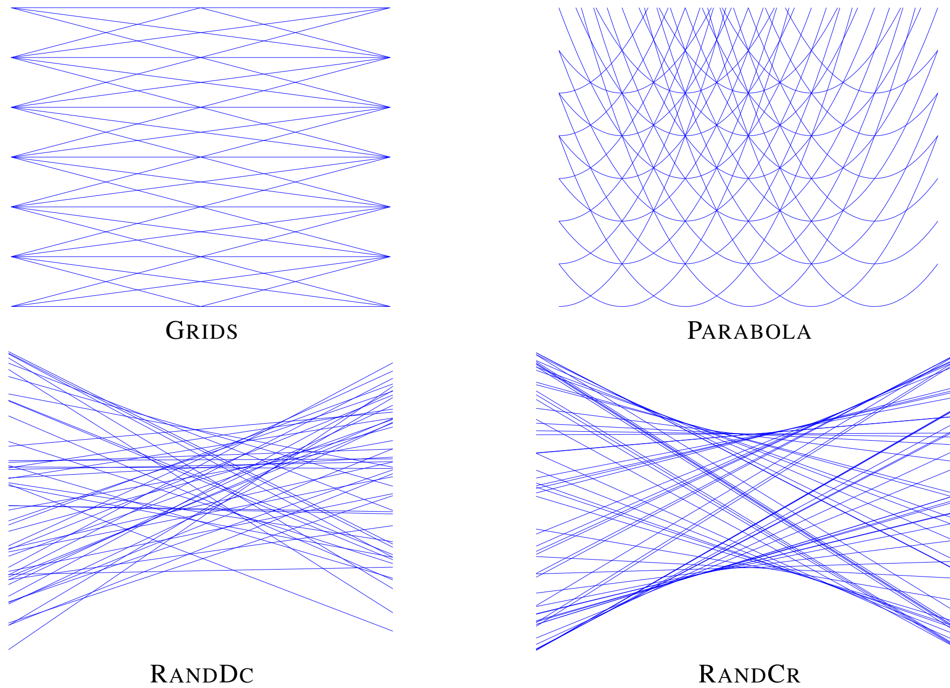


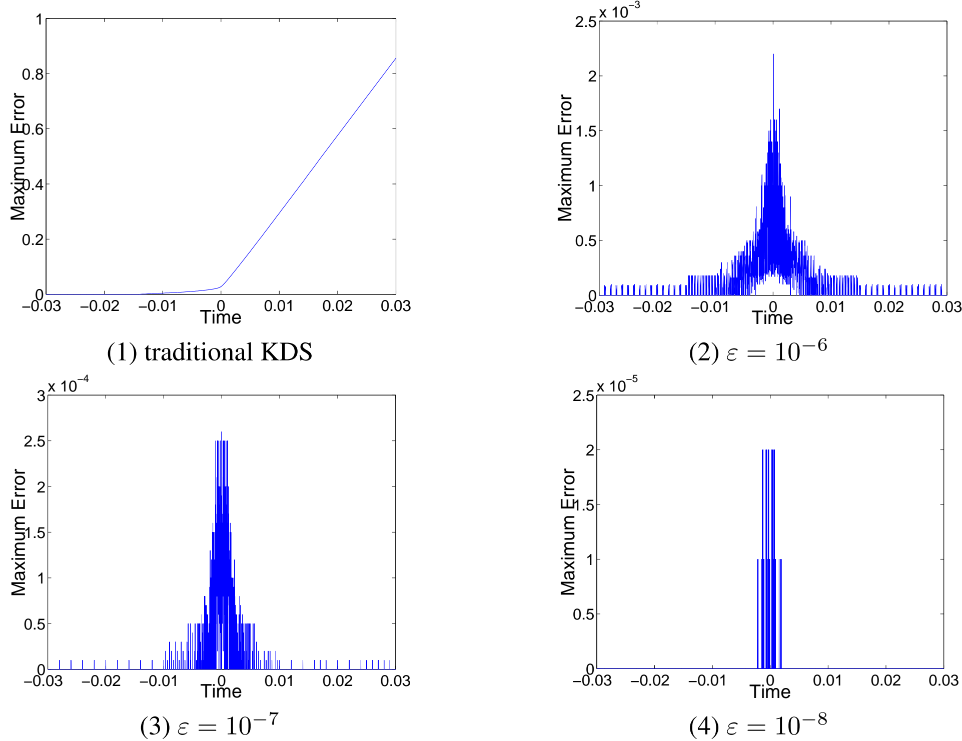
Figure 3.5 Datasets used in the experiments. The figures depict trajectories of the moving points in tx-plane, after an appropriate scaling.

- GRIDS: a set of linear trajectories whose dual points form a uniform grid;
- PARABOLA: a set of congruent parabolic trajectories with apexes sitting on a grid in tx-plane;
- RANDDC: a set of linear trajectories whose dual points are randomly distributed in a disk;
- RANDCR: a set of linear trajectories whose dual points are randomly distributed on a circle.

Kinetic sorting. We tested the kinetic sorting algorithms on the first three types of input data. All experiments were run on inputs of size 900. We measure the error of the sorting KDSs at time t by

$$\mathrm{err}(t) = \max_i |y_i(t) - z_i(t)|,$$

where $\langle y_1, \ldots, y_n \rangle$ and $\langle z_1, \ldots, z_n \rangle$ are the sequence maintained by the KDS and the correctly sorted sequence at time t respectively. In Figures 3.6-3.8 we plot $\mathrm{err}(t)$ as t



(1) traditional KDS (2) $\varepsilon = 10^{-6}$

(3) $\varepsilon = 10^{-7}$ (4) $\varepsilon = 10^{-8}$

Figure 3.6 Maximum error of kinetic sorting on a GRID input of size 900; scales on the vertical axis are different.

varies, by measuring $\mathrm{err}(t)$ every other 10^{-7} seconds. Note the different scales on the vertical axis in these figures.

We first discuss the behavior of the traditional kinetic sorting algorithm, which uses floating point arithmetic. In a few instances, the algorithm went into an infinite loop because of simultaneous events. Although this problem could be fixed in general, a more careful implementation of the traditional KDS is required. As for the geometric error in the maintained structures, the traditional KDS was very fragile: it quickly ran into noticeable errors and was unable to recover from these errors (see Figures 3.6 (1), 3.7 (1), and 3.8 (1)). The reason is that some events that should have been scheduled into the global queue were discarded by the KDS because their computed event times happened to lie in the past because of numerical errors.

We now turn our attention to the geometric error in the structures maintained by our robust kinetic sorting algorithm, under different precisions ε in the CROP procedure. As can be seen, while the traditional KDS quickly ran into serious errors and was never able

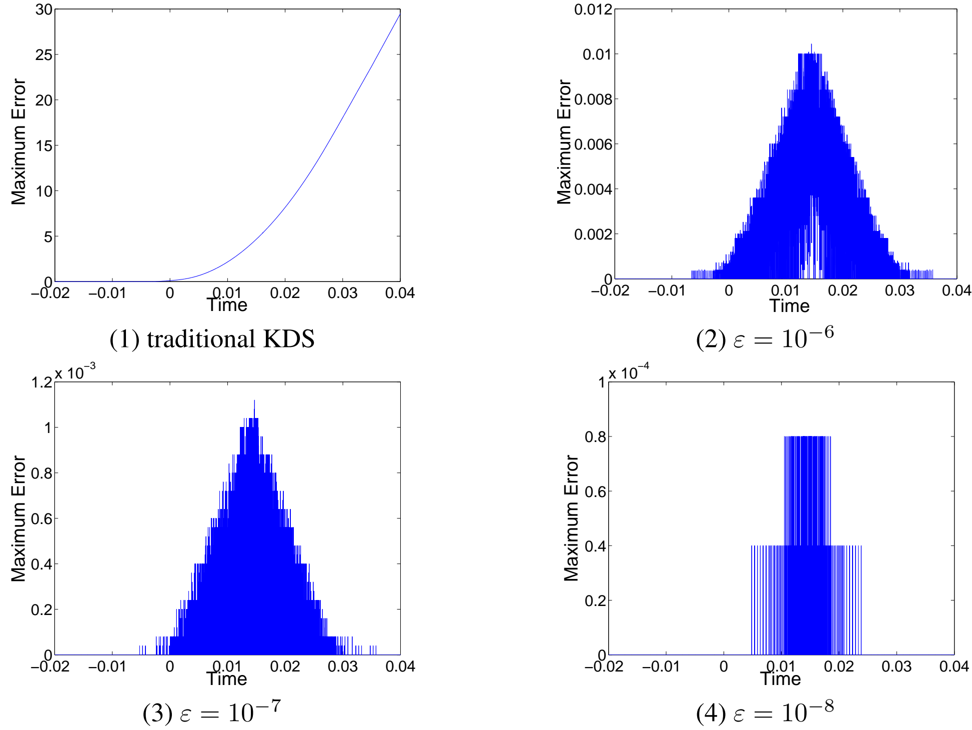
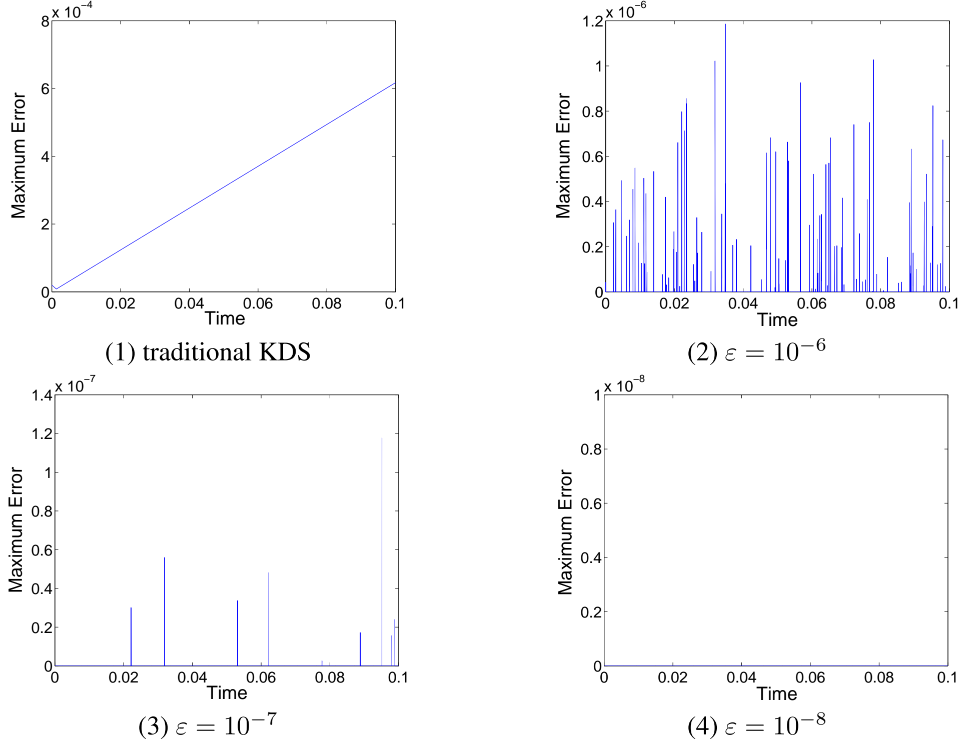


Figure 3.7 Maximum error of kinetic sorting on a PARABOLA input of size 900; scales on the vertical axis are different.

to recover, our robust KDS maintained a rather small error all the time. Observe that the error of the robust KDS reduces as the precision of the CROP procedure increases. We also tested the algorithm on a number of larger inputs, and the error remained roughly the same.

We also studied how long an event could be delayed before it is eventually processed in the robust kinetic sorting algorithm—see Table 3.1. It can be seen that the Root Mean Square (RMS, for short) of the delays are always very small for all inputs. As for the maximum delay, we only observed one instance in the first two types of inputs in which some events are delayed by about 2ε; in all other cases, the maximum delay never exceeds ε, which is far below the rather contrived worst-case bound in Theorem 3.6.

Kinetic tournament. We tested the kinetic tournament algorithms on the RANDCR data as this input tends to have a large number of external events. The geometric error is measured by $\mathrm{err}(t) = |y(t) - z(t)|$, where y and z are the maximum maintained by the



(1) traditional KDS (2) $\varepsilon = 10^{-6}$ (3) $\varepsilon = 10^{-7}$ (4) $\varepsilon = 10^{-8}$

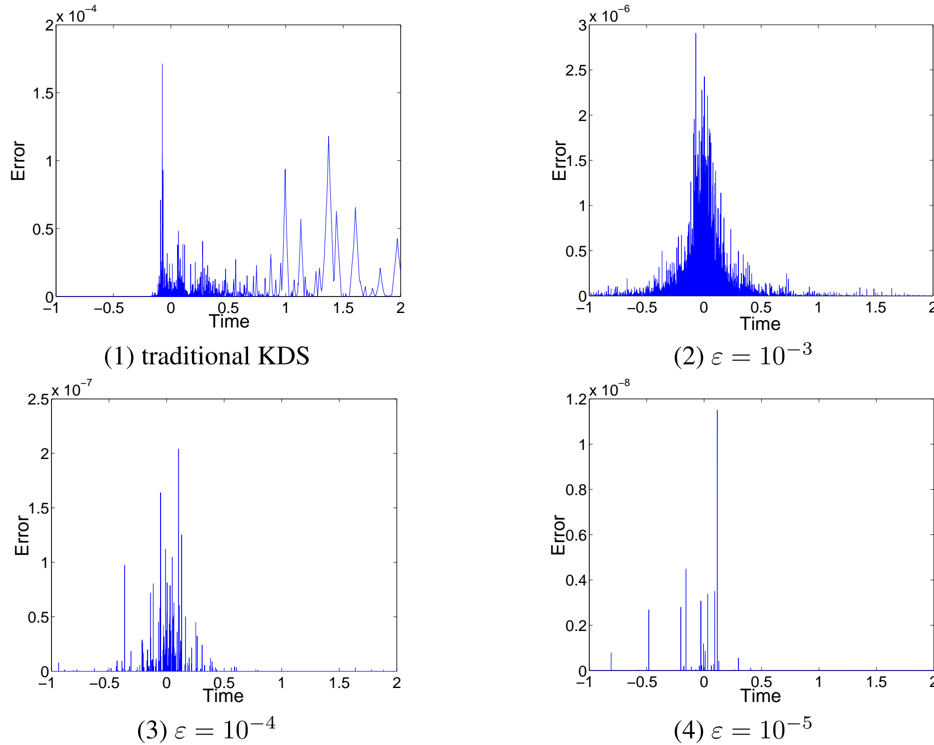
Figure 3.8 Maximum error of kinetic sorting on a RANDDC input of size 900; scales on the vertical axis are different.

Precision of CROP	GRIDS		PARABOLA		RANDDC	
	RMS	Max	RMS	Max	RMS	Max
$\varepsilon = 10^{-6}$	$0.48 \times \varepsilon$	$2.00 \times \varepsilon$	$0.37 \times \varepsilon$	$1.00 \times \varepsilon$	$0.42 \times \varepsilon$	$1.00 \times \varepsilon$
$\varepsilon = 10^{-7}$	$0.43 \times \varepsilon$	$1.00 \times \varepsilon$	$0.37 \times \varepsilon$	$1.00 \times \varepsilon$	$0.42 \times \varepsilon$	$1.00 \times \varepsilon$
$\varepsilon = 10^{-8}$	$0.42 \times \varepsilon$	$1.00 \times \varepsilon$	$0.39 \times \varepsilon$	$1.00 \times \varepsilon$	$0.41 \times \varepsilon$	$1.00 \times \varepsilon$

Table 3.1 Delay of events in kinetic sorting.

KDS and the true maximum at time t respectively. Since kinetic tournaments are less sensitive to simultaneous events than kinetic sorting, we artificially lowered the precision in computing the event times so as to cause noticeable geometric errors in the tested algorithms. Specifically, in the traditional KDS we round the event times to the precision of 10^{-5}, and in the robust KDS we vary the precision ε in CROP from 10^{-3} to 10^{-5}.

We first noticed that the traditional kinetic tournament algorithm did not go into an infinite



(1) traditional KDS (2) $\varepsilon = 10^{-3}$

(3) $\varepsilon = 10^{-4}$ (4) $\varepsilon = 10^{-5}$

Figure 3.9 Geometric error of the kinetic tournament on a RANDCR input of size 10000; scales on the vertical axis are different.

loop; this is because events are always "pushed" up in the tournament tree. However, as for the geometric error, one can see from Figure 3.9 (1) that the KDS maintains a rather inaccurate maximum over time. In contrast, the geometric errors in our robust KDS are smaller by orders of magnitudes, even though the event time computation is less precise than in the traditional KDS.

3.7 Conclusions

In this chapter we studied the problem of designing kinetic data structures that are robust against out-of-order event processing due to numerical errors in computing event times. We showed that the proposed robust kinetic sorting and kinetic tournament algorithms have several nice properties, including guaranteed correctness for all but a finite number of small time intervals, short delays in event processing, and small geometric errors over time. Combining the resulting kinetic range tree and kinetic tournament, we can also

maintain the closest-pair of a set of moving points robustly [24]. It is interesting to see whether similar results can be obtained for other more complex kinetic data structures as well. In particular, so far we have been unable to extend our techniques to kinetic Delaunay triangulations. The main difficulty is that we cannot argue the algorithm does not get into an infinite loop of edge flips. We leave it as an interesting open question for future research.

Chapter 4

Kinetic kd-trees and longest-side kd-trees

Abstract. We propose a simple variant of kd-trees, called rank-based kd-trees, for sets of points in $\mathbb{R}^d$. We show that a rank-based kd-tree, like an ordinary kd-tree, supports range search queries in $O(n^{1-1/d} + k)$ time, where k is the output size. The main advantage of rank-based kd-trees is that they can be efficiently kinetized: the KDS processes $O(n^2)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories, each event can be handled in $O(\log n)$ time, and each point is involved in $O(1)$ certificates.

We also propose a variant of longest-side kd-trees, called rank-based longest-side kd-trees (RBLS kd-trees, for short), for sets of points in $\mathbb{R}^2$. RBLS kd-trees can be kinetized efficiently as well and like longest-side kd-trees, RBLS kd-trees support nearest-neighbor, farthest-neighbor, and approximate range search queries in $O((1/\varepsilon)\log^2 n)$ time. The KDS processes $O(n^3 \log n)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories; each event can be handled in $O(\log^2 n)$ time, and each point is involved in $O(\log n)$ certificates.

An extended abstract of this chapter was previously published as: M. A. Abam and M. de Berg, and B. Speckmann, Kinetic kd-tree and longest-side kd-tree, In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 364–372, 2007.

4.1 Introduction

Background. Range searching is a fundamental problem that has been studied extensively both in the database community as well as in the computational-geometry community. Here, the goal is to construct a data structure for a set of n points such that for any given *range query*, the points lying inside a query range can be reported quickly. Due to the increased availability of motion data in a variety of application areas—air-traffic control, mobile communication and geographic information systems, for instance—it is not surprising a lot of work has been dedicated to developing range searching structures for moving points in 2- or higher-dimensional space.

Related work. There are several papers that describe KDSs for the orthogonal range-searching problem, where the query range is an axis-parallel box. Basch *et al.* [24] kinetized d-dimensional range trees. Their KDS supports range queries in $O(\log^d n + k)$ time and uses $O(n \log^{d-1} n)$ storage. If the points follow constant-degree algebraic trajectories then their KDS processes $O(n^2)$ events; each event can be handled in $O(\log^{d-1} n)$ time. In the plane, Agarwal *et al.* [6] obtained an improved solution: their KDS supports orthogonal range-searching queries in $O(\log n + k)$ time, it uses $O(n \log n / \log\log n)$ storage, and the amortized cost of processing an event is $O(\log^2 n)$.

Although these results are nice from a theoretical perspective, their practical value is limited for several reasons. First of all, they use super-linear storage, which is often undesirable. Second, they can perform only orthogonal range queries; queries with other types of ranges or *nearest-neighbor queries* ("Report the point that is currently closest to a query point") are not supported. Finally, especially the solution by Agarwal *et al.* [6] is rather complicated. Indeed, in the setting where the points do not move, the static counterparts of these structures are usually not used in practice. Instead, simpler structures such as quadtrees, kd-trees, or bounding-volume hierarchies (R-trees, for instance) are used. In this chapter we consider one of these structures, namely the kd-tree.

Kd-trees were initially introduced by Bentley [25]. A kd-tree for a set of points in the plane is obtained recursively as follows. At each node of the tree, the current point set is split into two equal-sized subsets with a line. When the depth of the node is even the splitting line is orthogonal to the x-axis, and when it is odd the splitting line is orthogonal to the y-axis. In d-dimensional space, the orientations of the splitting planes cycle through the d axes in a similar manner. Kd-trees use $O(n)$ storage and support orthogonal range searching queries in $O(n^{1-1/d} + k)$ time, where k is the number of reported points. Maintaining a standard kd-tree kinetically is not efficient. The problem is that a single event—two points swapping their order on x- or y-coordinate—can have a dramatic effect: a new point entering the region corresponding to a node could mean that almost the entire subtree must be re-structured. Hence, a variant of the kd-tree is needed when the points are moving.

Agarwal *et al.* [12] proposed two such variants for moving points in $\mathbb{R}^2$: the δ-pseudo

kd-tree and the δ-overlapping kd-tree. In a δ-pseudo kd-tree each child of a node ν can be associated with at most $(1/2+\delta)n_\nu$ points, where n_ν is the number of points in the subtree of ν. In a δ-overlapping kd-tree the regions corresponding to the children of ν can overlap as long as the overlapping region contains at most δn_ν points. Both kd-trees support orthogonal range queries in time $O(n^{1/2+\varepsilon} + k)$, where k is the number of reported points. Here ε is a positive constant that can be made arbitrarily small by choosing δ appropriately. These KDSs process $O(n^2)$ events if the points follow constant-degree algebraic trajectories. Although it can take up to $O(n)$ time to handle a single event, the amortized cost is $O(\log n)$ time per event. Neither of these two solutions is completely satisfactory: their query time is worse by a factor $O(n^\varepsilon)$ than the query time in standard kd-trees, there is only a good amortized bound on the time to process events, and only a solution for the 2-dimensional case is given. Our goal is to developed a kinetic kd-tree variant that does not have these drawbacks.

Even though a kd-tree can be used to search with any type of range, there are only performance guarantees for orthogonal ranges. *Longest-side kd-trees*, introduced by Dickerson *et al.* [43], are better in this respect. In a longest-side kd-tree, the orientation of the splitting line at a node is not determined by the level of the node, but by the shape of its region: namely, the splitting line is orthogonal to the longest side of the region. Although a longest-side kd-tree does not have performance guarantees for exact range searching, it has very good worst-case performance for ε-approximate range queries, which can be answered in $O(\varepsilon^{1-d}\log^d n + k)$ time. (In an ε-approximate range query, points that are within distance ε·diameter(Q) of the query range Q may also be reported.) Moreover, a longest-side kd-tree can answer ε-approximate nearest-neighbor queries (or: farthest-neighbor queries) in $O(\varepsilon^{1-d}\log^d n)$ time. The second our goal is to develop a kinetic variant of the longest-side kd-tree.

Our results. Our first contribution is a new and simple variant of the standard kd-tree for a set of n points in d-dimensional space. Our *rank-based kd-tree* supports orthogonal range searching in time $O(n^{1-1/d} + k)$ and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized easily and efficiently. The rank-based kd-tree processes $O(n^2)$ events in the worst case if the points follow constant-degree algebraic trajectories[1] and each event can be handled in $O(\log n)$ worst-case time. Moreover, each point is involved only in a constant number of certificates. Thus we improve the both the query time and the event-handling time as compared to the planar kd-tree variants of Agarwal *et al.* [12], and in addition our results work in any fixed dimension.

Our second contribution is the first kinetic variant of the longest-side kd-tree, which we call the *rank-based longest-side kd-tree* (or *RBLS kd-tree*, for short), for a set of n points in the plane. (We have been unable to generalize this result to higher dimensions.) An RBLS kd-tree uses $O(n)$ space and supports approximate nearest-neighbor, approximate

[1]For the bound on the number of events in our rank-based kd-tree, it is sufficient that any pair of points swaps x- or y-order $O(1)$ times. For the bounds on the number of events in the RBLS kd-tree, we need that every two pairs of points define the same x- or y-distance $O(1)$ times.

farthest-neighbor, and approximate range queries in the same time as the original longest-side kd-tree does for stationary points, namely $O((1/\varepsilon)\log^2 n)$ (plus the time needed to report the answers in case of range searching). The kinetic RBLS kd-tree maintains $O(n)$ certificates, processes $O(n^3 \log n)$ events if the points follow constant-degree algebraic trajectories[1], each event can be handled in $O(\log^2 n)$ time, and each point is involved in $O(\log n)$ certificates.

4.2 Rank-based kd-trees

Let $\mathcal{P}$ be a set of n points in $\mathbb{R}^d$ and let us denote the coordinate-axes with $x_1, \ldots, x_d$. To simplify the discussion we assume that no two points share any coordinate, that is, no two points have the same x_1-coordinate, or the same x_2-coordinate, etc. (Of course coordinates will temporarily be equal when two points swap their order, but the description below refers to the time intervals in between such events.) In this section we describe a variant of a kd-tree for $\mathcal{P}$, the *rank-based kd-tree*. A rank-based kd-tree preserves all main properties of a kd-tree and, additionally, it can be kinetized efficiently.

Before we describe the actual rank-based kd-tree for $\mathcal{P}$, we first introduce another tree, namely the *skeleton* of a rank-based kd-tree, denoted by $\mathcal{S}(\mathcal{P})$. Like a standard kd-tree, $\mathcal{S}(\mathcal{P})$ uses axis-orthogonal splitting hyperplanes to divide the set of points associated with a node. As usual, the orientation of the axis-orthogonal splitting hyperplanes is alternated between the coordinate axes, that is, we first split with a hyperplane orthogonal to the x_1-axis, then with a hyperplane orthogonal to the x_2-axis, and so on. Let ν be a node of $\mathcal{S}(\mathcal{P})$. $h(\nu)$ is the splitting hyperplane stored at ν, axis(ν) is the coordinate-axis to which $h(\nu)$ is orthogonal, and $\mathcal{P}(\nu)$ is the set of points stored in the subtree rooted at ν. A node ν is called an x_i*-node* if axis(ν) $= x_i$ and a node ω is referred to as an x_i*-ancestor* of a node ν if ω is an ancestor of ν and axis(ω) $= x_i$. The first x_i-ancestor of a node ν (that is, the x_i-ancestor closest to ν) is the x_i*-parent*(ν) of ν.

A standard kd-tree chooses $h(\nu)$ such that $\mathcal{P}(\nu)$ is divided roughly in half. In contrast, $\mathcal{S}(\mathcal{P})$ chooses $h(\nu)$ based on a range of ranks associated with ν, which can have the effect that the sizes of the children of ν are completely unbalanced. We now explain this construction in detail. We use d arrays $\mathcal{A}_1, \ldots, \mathcal{A}_d$ to store the points of $\mathcal{P}$ in d sorted lists; the array $\mathcal{A}_i[1, n]$ stores the sorted list based on the x_i-coordinate. As mentioned above, we associate a range $[r, r']$ of ranks with each node ν, denoted by range(ν), with $1 \leqslant r \leqslant r' \leqslant n$. Let ν be an x_i-node. If x_i-parent(ν) does not exist, then range(ν) is equal to $[1, n]$. Otherwise, if ν is contained in the left subtree of x_i-parent(ν), then range(ν) is equal to the first half of range(x_i-parent(ν)), and if ν is contained in the right subtree of x_i-parent(ν), then range(ν) is equal to the second half of range(x_i-parent(ν)). If range(ν) $= [r, r']$ then $\mathcal{P}(\nu)$ contains at most $r' - r + 1$ points. We explicitly ignore all nodes (both internal as well as leaf nodes) that do not contain any points, they are not part of $\mathcal{S}(\mathcal{P})$, independent of their range of ranks. A node ν is a leaf of $\mathcal{S}(\mathcal{P})$ if range(ν) $= [j, j]$ for some j. Clearly a leaf contains exactly one point, but

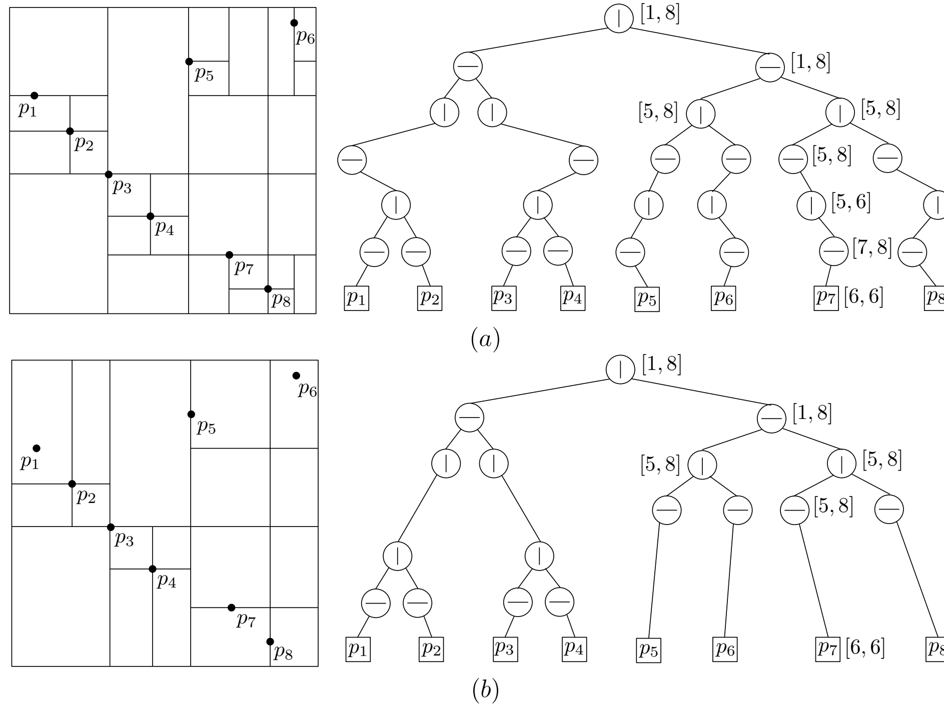


Figure 4.1 (a) The skeleton of a rank-based kd-tree and (b) the rank-based kd-tree itself.

not every node that contains only one point is a leaf. (We could prune these nodes, which always have a range $[j, k]$ with $j < k$, but we chose to keep them in the skeleton for ease of description.) If ν is not a leaf and axis$(\nu) = x_i$ then $h(\nu)$ is defined by the point whose rank in $\mathcal{A}_i$ is equal to the median of range(ν). (This is similar to the approach used in the kinetic BSP of [39].) It is not hard to see that this choice of the splitting plane $h(\nu)$ is equivalent to the following. Let region$(\nu) = [a_1 : b_1] \times \cdots \times [a_d : b_d]$ and suppose for example that ν is an x_1-node. Then, instead of choosing $h(\nu)$ according to the median x_1-coordinate of all points in region(ν), we choose $h(\nu)$ according to the median x_1-coordinate of all points in the slab $[a_1, b_1] \times [-\infty : \infty] \times \cdots \times [-\infty : \infty]$.

We construct $\mathcal{S}(\mathcal{P})$ incrementally by inserting the points of $\mathcal{P}$ one by one. (Even though we proceed incrementally, we still use the rank of each point with respect to the whole point set, not with respect to the points inserted so far.) Let p be the point that we are currently inserting into the tree and let ν be the last node visited by p; initially $\nu =$ root$(\mathcal{S}(\mathcal{P}))$. Depending on which side of $h(\nu)$ contains p we select the appropriate child ω of ν to be visited next. If ω does not exist, then we create it and compute range(ω) as described above. We recurse with $\nu = \omega$ until range$(\nu) = [j, j]$ for some j. We always

reach such a node after $d \log n$ steps, because the length of range(ν) is a half of the length of range$(x_i$-parent$(\nu))$ and depth(ν) = depth$(x_i$-parent$(\nu)) + d$ for an x_i-node ν. Figure 4.1(a) illustrates $\mathcal{S}(\mathcal{P})$ for a set of eight points. Since each leaf of $\mathcal{S}(\mathcal{P})$ contains exactly one point of $\mathcal{P}$ and the depth of each leaf is $d \log n$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$.

Lemma 4.1 *The depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ and the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$ for any fixed dimension d. $\mathcal{S}(\mathcal{P})$ can be constructed in $O(n \log n)$ time.*

A node $\nu \in \mathcal{S}(\mathcal{P})$ is *active* if and only if both its children exist, that is, both its children contain points. A node ν is *useful* if it is either active, or a leaf, or its first $d-1$ ancestors contain an active node. Otherwise a node is *useless*. We derive the rank-based kd-tree for $\mathcal{P}$ from the skeleton by pruning all useless nodes from $\mathcal{S}(\mathcal{P})$. The parent of a node ν in the rank-based kd-tree is the first unpruned ancestor of ν in $\mathcal{S}(\mathcal{P})$. Roughly speaking, in the pruning phase every long path whose nodes have only one child each is shrunk to a path whose length is less than d. The rank-based kd-tree has exactly n leaves and each contains exactly one point of $\mathcal{P}$. Moreover, every node ν in the rank-based kd-tree is either active or it has an active ancestor among its first $d-1$ ancestors. The rank-based kd-tree derived from Figure 4.1(a) is illustrated in Figure 4.1(b).

Lemma 4.2

(i) *A rank-based kd-tree on a set of n points in $\mathbb{R}^d$ has depth $O(\log n)$ and size $O(n)$.*

(ii) *Let ν be an x_i-node in a rank-based kd-tree. In the subtree rooted at a child of ν, there are at most 2^{d-1} x_i-nodes ω such that x_i-parent$(\omega) = \nu$.*

(iii) *Let ν be an x_i-node in a rank-based kd-tree. On every path starting at ν and ending at a descendant of ν and containing at least $2d-1$ nodes, there is an x_i-node ω such that x_i-parent$(\omega) = \nu$.*

Proof.

(i) A rank-based kd-tree is at most as deep as its skeleton $\mathcal{S}(\mathcal{P})$. Since the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ by Lemma 4.1, the depth of a rank-based kd-tree is also $O(\log n)$. To prove the second claim, we charge every node that has only one child to its first active ancestor. Recall that each active node has two children. We charge at most $2(d-1)$ nodes to each active node, because after pruning there is no path in the rank-based kd-tree whose length is at least d and in which all nodes have one child. Therefore, to bound the size of the rank-based kd-tree it is sufficient to bound the number of active nodes. Let $\mathcal{T}$ be a tree containing all active nodes and all leaves of the rank-based kd-tree. A node ν is the parent of a node ω in $\mathcal{T}$ if and only if ν is the first active ancestor of ω in the rank-based kd-tree. Obviously, $\mathcal{T}$ is a binary tree with n leaves where each internal node has two children. Hence, the size of $\mathcal{T}$ is $O(n)$ and consequently the size of the rank-based kd-tree is $O(n)$.

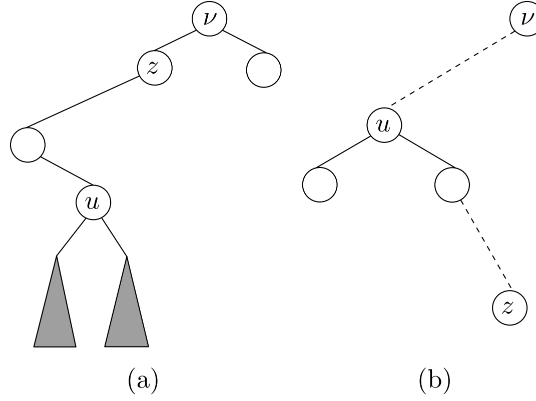


Figure 4.2 Illustration for the proof of Lemma 4.2.

(ii) To simplify notation, let ω' denote the node in $\mathcal{S}(\mathcal{P})$ that corresponds to a node ω in the rank-based kd-tree. Let z be a child of ν and and let u be the first active node in the subtree rooted at z as depicted in Fig. 4.2(a), that is, u is the highest active node in the subtree rooted at z. Note that the definition of active node ensures that u is unique, and note that u can be z. Now assume x_i-parent$(\omega) = \nu$ where ω is an x_i-node in the subtree rooted at z. If ω is not a node in the subtree rooted at u, then there is just one node ω in the subtree rooted at z satisfying x_i-parent$(\omega) = \nu$, since every node between z and u has only one child. This means that we are done. Otherwise, if ω is a node in the subtree rooted at u, then ω' must be in the subtree rooted at u' of $\mathcal{S}(\mathcal{P})$. Let s' be the first x_i-node on the path from u' to ω'. Because one of any d consecutive nodes in $\mathcal{S}(\mathcal{P})$ uses a hyperplane orthogonal to the x_i-axis as a splitting plane, depth$(s') \leqslant$ depth$(u') + d - 1$. Since u' is active and depth$(s') \leqslant$ depth$(u') + d - 1$, the node s' must appear as a node, s, in the rank-based kd-tree. This and the assumption that x_i-parent$(\omega) = \nu$ imply $\omega = s$ which means depth$(\omega') \leqslant$ depth$(u') + d - 1$. Hence the number of nodes ω is at most 2^{d-1}.

(iii) Let u be the first active node on the path starting at ν and ending at a descendant z of ν and containing at least $2d-1$ nodes as depicted in Fig. 4.2(b). Because there is no path in the rank-based kd-tree that contains d nodes such that every node in the path has only one child, depth$(u) \leqslant$ depth$(\nu) + d - 1$ which implies depth$(z) \geqslant$ depth$(u) + d - 1$—note that on the path from ν to z there are $2d - 1$ nodes. Let ω' be the first x_i-node in the path starting at u' and ending at z' in $\mathcal{S}(\mathcal{P})$. Because one of any d consecutive nodes in $\mathcal{S}(\mathcal{P})$ uses a hyperplane orthogonal to the x_i-axis to split points, and depth$(z') \geqslant$ depth$(u') + d - 1$, the node ω' exists. The node ω' must appear as a node, ω, in the kd-tree, because either $\omega' = u'$ or among the first $d - 1$ ancestor of ω' there is an active ancestor, namely u'. Putting it all together

we can conclude that depth$(\omega) \leqslant$ depth$(\nu) + 2d - 2$ which implies the claim.

□

The region associated with a node ν, denoted by region(ν), is the maximal volume bounded by the splitting hyperplanes stored at the ancestors of ν. More precisely, the region associated with the root of a rank-based kd-tree is simply the whole region, and the region corresponding to the right child of a node ν is the maximal subregion of region(ν) on the right side of $h(\nu)$ and the region corresponds to the left child of ν is the rest of region(ν) (for an appropriate definition of left and right in d dimensions). A point p is contained in $\mathcal{P}(\nu)$ if and only if p lies in region(ν). Like a kd-tree, a rank-based kd-tree can be used to report all points inside a given orthogonal range search query—the reporting algorithm is exactly the same. At first sight, the fact that the splits in our rank-based kd-tree can be very unbalanced may seem to have a big, negative impact on the query time. Fortunately this is not the case. To prove this, we next bound the number of cells intersected by an axis-parallel plane h. As for normal kd-trees, this is immediately gives a bound on the total query time.

Lemma 4.3 *Let h be a hyperplane orthogonal to the x_i-axis for some i. The number of nodes in a rank-based kd-tree whose regions are intersected by h is $O(n^{1-1/d})$.*

Proof. Imagine a dummy node μ with axis$(\mu) = x_i$ as the parent of the root. We charge every node ν whose region is intersected by h to x_i-parent(ν). Thanks to μ, x_i-parent(ν) exists for every node of the tree and hence every node is indeed charged to an x_i-node. Lemma 4.2(iii) implies depth$(\nu) \leqslant$ depth$(x_i$-parent$(\nu)) + 2d - 2$ which implies that at most 2^{2d-2} nodes are charged to each x_i-node. Therefore it is sufficient to bound the number of x_i-nodes whose regions are intersected by h.

Let $\mathcal{T}$ be the tree containing all x_i-nodes in the rank-based kd-tree and let $\mathcal{T}'$ be the tree containing all x_i-nodes in the skeleton $\mathcal{S}(\mathcal{P})$. A node ν is the parent of a node ω in $\mathcal{T}$ if and only if x_i-parent$(\omega) = \nu$ in the rank-based kd-tree; the equivalent definition holds for $\mathcal{T}'$. According to Lemma 4.2(ii), every node ν in $\mathcal{T}$ has at most 2^d children and each side of $h(\nu)$ contains the regions corresponding to at most 2^{d-1} children of ν. Note that the dummy node has at most 2^{d-1} children in total. Let $\mathcal{T}^*$ be yet another tree containing all nodes in $\mathcal{T}$ whose regions are intersected by h. Since h is parallel to $h(\nu)$ for every node ν of $\mathcal{T}$, it can intersect only the regions that lie to one side of $h(\nu)$. Hence every node of $\mathcal{T}^*$ has at most 2^{d-1} children. The idea behind the proof is to consider a top part of $\mathcal{T}^*$ consisting of $n^{1-1/d}$ nodes of $\mathcal{T}^*$, and then argue that all subtree below this top part together contain $n^{1-1/d}$ nodes as well. Next we make this idea precise.

Let TOP$(\mathcal{T}^*)$ be a tree containing all nodes of $\mathcal{T}^*$ whose depths in $\mathcal{T}^*$ are at most $\lfloor (1/d) \log n \rfloor$, and let $\nu_1, \ldots, \nu_c$ be the leaves of TOP$(\mathcal{T}^*)$ whose depthes are exactly $\lfloor (1/d) \log n \rfloor$. Clearly c is at most $(2^{d-1})^{(1/d) \log n} = n^{1-1/d}$ and hence the size of TOP$(\mathcal{T}^*)$ is at most $2n^{1-1/d}$. Let $\nu_1', \ldots, \nu_c'$ be the nodes corresponding to $\nu_1, \ldots, \nu_c$ in $\mathcal{T}'$. Furthermore, let $u_1', \ldots, u_m'$ be the distinct nodes in $\mathcal{T}'$ at depth $\lfloor (1/d) \log n \rfloor$ such that every u_k' has at least one node ν_j' as descendant and every ν_j' has a node u_k' as

an ancestor—note that due to pruning the depth of ν'_j can be larger than $\lfloor (1/d) \log n \rfloor$. Because the nodes ν'_j are disjoint, we have $\sum_1^c |\mathcal{P}(\nu'_j)| \leqslant \sum_1^m |\mathcal{P}(u'_k)|$.

Let U_k be the set of splitting hyperplanes stored in the ancestors of u'_k in $\mathcal{T}'$. Recall that all nodes u'_k are x_i-nodes whose regions are intersected by h. Furthermore, all nodes u'_k have the same depth in $\mathcal{T}'$. Together this implies that $U_k = U_l$ for all $1 \leqslant k, l \leqslant m$ because their x_i-ranges must be the same. Let h_1 be the last hyperplane in U_k on the left side of region(u'_1) and let h_2 be the first hyperplane in U_k on the right side of region(u'_1). Because $U_k = U_l$ for all $1 \leqslant k, l \leqslant m$, all regions u'_k are bounded by h_1 and h_2. We know that range(u'_k) contains $n/2^{(1/d)\log n} = n^{1-1/d}$ ranks, hence there are at most $n^{1-1/d}$ points inside the region bounded by h_1 and h_2. Since the nodes u'_k are disjoint and the region bounded by h_1 and h_2 contains $n^{1-1/d}$ points, we have $\sum_1^m |\mathcal{P}(u'_k)| \leqslant n^{1-1/d}$ which implies $\sum_1^c |\mathcal{P}(\nu_j)| = \sum_1^c |\mathcal{P}(\nu'_j)| \leqslant n^{1-1/d}$.

Finally, let $f(n)$ denote the number of x_i-nodes whose regions are intersected by h. We have $f(n) = |\text{TOP}(\mathcal{T}^*)| + \sum_1^c f(|\mathcal{P}(\nu_j)|)$. Since $f(|\mathcal{P}(\nu_j)|) \leqslant |\mathcal{P}(\nu_j)|$, $\sum_1^c |\mathcal{P}(\nu_j)| \leqslant n^{1-1/d}$, and $|\text{TOP}(\mathcal{T}^*)| \leqslant 2n^{1-1/d}$, we can conclude that $f(n) = O(n^{1-1/d})$. □

The following theorem summarizes our results.

Theorem 4.4 *A rank-based kd-tree for a set $\mathcal{P}$ of n points in d dimensions uses $O(n)$ storage and can be built in $O(n \log n)$ time. An orthogonal range search query on a rank-based kd-tree takes $O(n^{1-1/d} + k)$ time where k is the number of reported points.*

The KDS. We now describe how to kinetize a rank-based kd-tree for a set of continuously moving points $\mathcal{P}$. The combinatorial structure of a rank-based kd-tree depends only on the ranks of the points in the arrays $\mathcal{A}_i$, that is, it does not change as long as the order of the points in the arrays $\mathcal{A}_i$ remains the same. Hence it suffices to maintain a certificate for each pair p and q of consecutive points in every array $\mathcal{A}_i$, which fails when p and q change their order. Now assume that a certificate, involving two points p and q and the x_i-axis, fails at time t. To handle the event, we simply delete p and q and re-insert them in their new order. (During the deletion and re-insertion there is no need to change the ranks of the other points.) These deletions and insertions do not change anything for the other points, because their ranks are not influenced by the swap and the deletion and re-insertion of p and q. Hence the rank-based kd-tree remains unchanged except for a small part that involves p and q. A detailed description of this “small part” can be found below.

Deletion. Let ν be the first active ancestor of the leaf μ containing p—see Figure 4.3(a). The leaf μ and all nodes on the path from μ to ν must be deleted, since they do not contain any points anymore (they only contained p and p is now deleted). Furthermore, ν stops being active. Let ω be the first active descendent of ν if it exists and otherwise let ω be the leaf whose ancestor is ν. There are at most d nodes on the path from ν to ω. Since ν is not active anymore, any of the nodes on this path might become useless and hence have to be deleted.

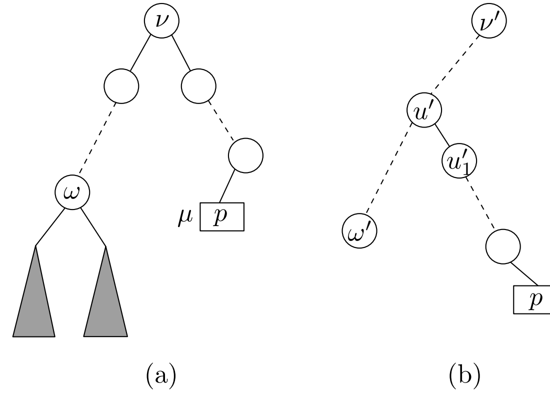


Figure 4.3 Deleting and inserting point p.

Insertion. Let ν be the highest node in the rank-based kd-tree such that its region contains p and the region corresponding to its only child ω does not contain p—note that p cannot reach a leaf when we re-insert p, because the range of a leaf is $[j, j]$ for some j and there cannot be two points in this range. Let ν' and ω' be the nodes in $\mathcal{S}(P)$ corresponding to ν and ω. Let u' be the lowest node on the path from ν' to ω' whose region contains both region(ω') and p as illustrated in Figure 4.3(b)—note that we do not maintain $\mathcal{S}(\mathcal{P})$ explicitly but with the information maintained in ν and ω the path between ν' and ω' can be constructed temporarily. Because u' will become an active node, it must be added to the rank-based kd-tree and also every node on the path from u' to ω' must be added to the rank-based kd-tree if they are useful. From u', the point p follows a new path $u'_1, \ldots, u'_k$ which is created during the insertion. All first $d-1$ nodes in the list $u'_1, \ldots, u'_k$ and the leaf u'_k must be added to the rank-based kd-tree—note that range$(u'_k) = [j, j]$ for some j.

Theorem 4.5 *A kinetic rank-based kd-tree for a set $\mathcal{P}$ of n moving points in d dimensions uses $O(n)$ storage and processes $O(n^2)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories. Each event can be handled in $O(\log n)$ time and each point is involved in $O(1)$ certificates.*

4.3 Rank-based longest-side kd-trees

Longest-side kd-trees are a variant of kd-trees that choose the orientation of the splitting hyperplane for a node ν according to the shape of the region associated with ν, always splitting the longest side first. Dickerson *et al.* [43] showed that a longest-side kd-tree can be used to answer the following queries quickly:

$(1+\varepsilon)$-nearest neighbor query: For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, and $\varepsilon > 0$, this query returns a point $p \in \mathcal{P}$ such that $d(p,q) \leqslant (1+\varepsilon)d(p^*,q)$, where p^* is the true nearest neighbor to q and $d(\cdot,\cdot)$ denotes the Euclidean distance.

$(1-\varepsilon)$-farthest neighbor query: For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, and $\varepsilon > 0$, this query returns a point $p \in \mathcal{P}$ such that $d(p,q) \geqslant (1-\varepsilon)d(p^*,q)$, where p^* is the true farthest neighbor to q.

ε-approximate range search query: For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query region Q with diameter D_Q, and $\varepsilon > 0$, this query returns (or counts) a set $\mathcal{P}'$ such that $\mathcal{P} \cap Q \subset \mathcal{P}' \subset \mathcal{P}$ and for every point $p \in \mathcal{P}'$, $d(p,Q) \leqslant \varepsilon D_Q$.

The main property of a longest-side kd-tree—which is used to bound the query time—is that the number of disjoint regions associated with its nodes and intersecting at least two opposite sides of a hypercube $\mathcal{C}$ is bounded by $O(\log^{d-1} n)$. It seems difficult to directly kinetize a longest-side kd-tree. Hence, using similar ideas as in the previous section, we introduce a simple variation of 2-dimensional longest-side kd-trees, so called *ranked-based longest-side kd-trees* (RBLS kd-trees, for short). An RBLS kd-tree does not only preserve all main properties of a longest-side kd-tree but it can be kinetized easily and efficiently. As in the previous section we first describe another tree, namely the skeleton of an RBLS kd-tree denoted by $\mathcal{S}(\mathcal{P})$. We then show how to extract an RBLS kd-tree from the skeleton $\mathcal{S}(\mathcal{P})$ by pruning.

We recursively construct $\mathcal{S}(\mathcal{P})$ as follows. We again use two arrays $\mathcal{A}_1$ and $\mathcal{A}_2$ to store the points of $\mathcal{P}$ in two sorted lists; the array $\mathcal{A}_i[1,n]$ stores the sorted list based on the x_i-coordinate. Let the points in $\mathcal{P}$ be inside a box, which is the region associated with the root, and let ν be a node whose subtree must be constructed; initially $\nu = \text{root}(\mathcal{S}(\mathcal{P}))$. If $\mathcal{P}(\nu)$ contains only one point, then the subtree is just a single leaf, i.e, ν is a leaf of $\mathcal{S}(\mathcal{P})$. (Note that this is slightly different from the previous section.) If $\mathcal{P}(\nu)$ contains more than one point, then we have to determine the proper splitting line. Let the longest side of region(ν) be parallel to the x_i-axis. We set axis(ν) to be x_i. If x_i-parent(ν) does not exist, then we set range(ν) $= [1,n]$. Otherwise, if ν is contained in the left subtree of x_i-parent(ν), then range(ν) is equal to the first half of range(x_i-parent(ν)), and if ν is contained in the right subtree of x_i-parent(ν), then range(ν) is equal to the second half of range(x_i-parent(ν)). The splitting line of ν, denoted by $l(\nu)$, is orthogonal to axis(ν) and specified by the point whose rank in $\mathcal{A}_i$ is the median of range(ν). If there is a point of $\mathcal{P}(\nu)$ on the left side of $l(\nu)$ (on the right side of $l(\nu)$ or on $l(\nu)$), a node is created as the left child (the right child) of ν. The points of $\mathcal{P}(\nu)$ which are on the left side of $l(\nu)$ are associated with the left child of ν, the remainder is associated with the right child of ν. The region of the right child is the maximal subregion of region(ν) on the right side of $l(\nu)$ and the region of the left child is the rest of region(ν).

Lemma 4.6 *The depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$, and $\mathcal{S}(\mathcal{P})$ can be constructed in $O(n \log n)$ time.*

Proof. Assume for contradiction that the depth of a leaf ν is at least $2\log n + 1$. Now consider the path from the root to ν. Because there are only two distinct axes, there are at least $\log n + 1$ nodes on this path whose axes are the same, for example x_i. Let $\nu_1, \ldots, \nu_k$ be these nodes. Since $|\text{range}(\nu_{j+1})| \leqslant \lceil (1/2)|\text{range}(\nu_j)| \rceil$ $(j = 1, \ldots, k-1)$ and $k > \log n$, ν_k must be empty, which is a contradiction. Hence the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$.

Since each leaf contains exactly one point and the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$. Furthermore it is easy to see that it takes $O(|\mathcal{P}(\nu)|)$ time to split the points at a node ν. Hence we spend $O(n)$ time at each level of $\mathcal{S}(\mathcal{P})$ during construction, for a total construction time of $O(n \log n)$. □

The following lemma shows that RBLS kd-trees preserve the main property of longest-side kd-trees, which is used to bound the query time.

Lemma 4.7 *Let $\mathcal{C}$ be any square, and let N be any set of nodes whose regions are pairwise disjoint and such that these regions all intersect two opposite sides of $\mathcal{C}$. Then $|N| = O(\log n)$.*

Proof. Dickerson *et al.* [43] showed that a longest-side kd-tree on a set of points in $\mathbb{R}^2$ has this property. Their proof uses only two properties of a longest side kd-tree: (i) the depth of a longest-side kd-tree is $O(\log n)$ and (ii) the longest side of a region is split first. Since an RBLS kd-tree has these two properties, their proof simply applies. □

As in the previous section, we obtain our structure by pruning useless nodes from $\mathcal{S}(\mathcal{P})$. It will be convenient to alter the definition of useful nodes slightly, as follows. A node ν is useful if ν is a leaf, or an active node, or $l(\nu)$ defines one of the sides of the boundary of region(ω) where ω is an active descendant of ν. Otherwise ν is useless. An RBLS kd-tree is obtained from $\mathcal{S}(\mathcal{P})$ by pruning useless nodes. The parent of a node ν in the RBLS kd-tree is the first unpruned ancestor of ν in $\mathcal{S}(\mathcal{P})$. The following lemma shows that an RBLS kd-tree has linear size and that it preserves the main property of a longest-side kd-tree.

Theorem 4.8

(i) An RBLS longest-side kd-tree on a set of n points in $\mathbb{R}^2$ has depth $O(\log n)$ and size $O(n)$.

(ii) The number of nodes in an RBLS longest-side kd-tree whose regions are disjoint and that intersect at least two opposite sides of a square $\mathcal{C}$ is $O(\log n)$.

Proof.

(i) An RBLS kd-tree is at most as deep as its skeleton $\mathcal{S}(\mathcal{P})$. Since the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ by Lemma 4.6, the depth of an RBLS kd-tree is also at most $O(\log n)$. To prove the second claim, we first show that there is no path containing five nodes such that every node on the path has only one child. Assume for contradiction that

there is such a path from ν to one of its descendants ω. Because there are only two distinct axes, there must be three nodes u_1, u_2, and u_3 on this path using the same axis. Clearly at most two of $l(u_1)$, $l(u_2)$, and $l(u_3)$ can define one of the sides of the boundary of any region associated with a descendant of ω. Therefore, at least one of u_1, u_2, and u_3 must be useless, which is a contradiction. We now charge every node that has only one child to its first active ancestor. Because there is no path containing five nodes such that every node on the path has only one child, we charge at most eight nodes to each active node. Since the number of active nodes is linear, the size of an RBLS longest-side kd-tree is $O(n)$.

(ii) Let L be a set of nodes in an RBLS kd-tree whose regions are disjoint and that intersect at least two opposite sides of a square $\mathcal{C}$. We define a set L' of nodes as follows. Consider a node $\nu \in L$. If ν is active then we add ν to L'. If ν is not active, then we consider the first active ancestor u of ν. We add the child w of u to L' that is on the path from u to ν (note that w could be ν). The regions in L' are disjoint and we have $|L| = |L'|$. Since the region associated with a node is a subregion of the region associated with its ancestor, the regions associated with the nodes in L' intersect at least two opposite sides of $\mathcal{C}$. Let ν' be the corresponding node to ν in $\mathcal{S}(\mathcal{P})$. The definition of a useful node implies region(ν) = region(ν') for every active node ν—note that this may be false for other nodes. Thus, if $\nu \in L'$ is active, then region(ν) = region(ν') and if ν is a child of an active node ω, then region(ν) = region(u') where u' is the child of ω' that is on the path from ω' to ν'. Thus, for every node ν in L', there is a node ω' in $\mathcal{S}(\mathcal{P})$ such that region(ν) = region(ω'). This observation together with Lemma 4.7 shows that $|L'| = O(\log n)$ which implies $|L| = O(\log n)$.

□

Using an RBLS kd-tree, similar algorithms to the algorithms of Dickerson *et al.* [43] can be used to answer $(1+\varepsilon)$-nearest neighbor, $(1-\varepsilon)$-farthest neighbor and ε-approximate range search queries.

Theorem 4.9 *An RBLS kd-tree for a set of n points in the plane supports $(1+\varepsilon)$-nearest or $(1-\varepsilon)$-farthest neighbor queries in $O((1/\varepsilon)\log^2 n)$ time. Moreover, for any constant-complexity convex region and any constant-complexity non-convex region a counting (or reporting) ε-approximate range search query can be performed in time $O((1/\varepsilon)\log^2 n)$ and $O((1/\varepsilon^2)\log^2 n)$, respectively (plus the output size in the reporting case).*

The KDS. We now describe how to kinetize a RBLS kd-tree for a set of continuously moving points $\mathcal{P}$. Clearly the combinatorial structure of an RBLS kd-tree changes only when one of the following two events occurs.

Ordering event: Two points change their ordering on one of the coordinate-axes.

Longest-side event: A side of a region starts to be the longest side of that region.

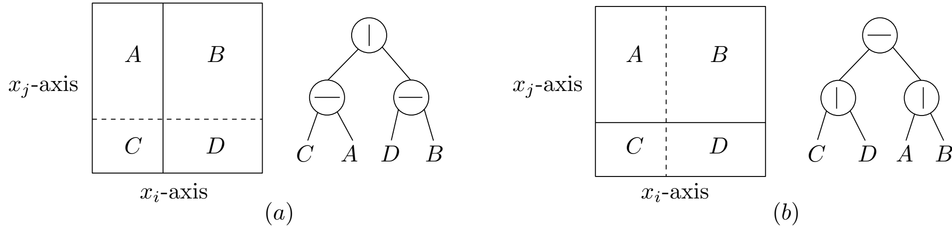


Figure 4.4 The status of the RBLS kd-tree before handling a longest-side event and after handling the event.

We first describe how to detect these events, then we explain how to handle them. Ordering events can be easily detected. We maintain a certificate for each pair p and q of consecutive points in the two arrays $\mathcal{A}_1$ and $\mathcal{A}_2$, which fails when p and q change their order.

Longest-side events are a bit tricky to detect efficiently. An easy way would be to maintain a certificate $s_1(\nu) < s_2(\nu)$ (or $s_2(\nu) < s_1(\nu)$) for each node ν in $\mathcal{S}(\mathcal{P})$ where $s_i(\nu)$ denotes the length of the x_i-side of region(ν). Let $x_i(p)$ denote the x_i-coordinate of p. We have $s_i(\nu) = x_i(p) - x_i(q)$ where p and q are two points specifying two splitting lines in the x_i-ancestors of ν in $\mathcal{S}(\mathcal{P})$. More precisely, the splitting lines defined by p and q are associated with the first *left ancestor* and the first *right ancestor* of ν in $\mathcal{S}(\mathcal{P})$, that is, the first nodes u and w such that ν is a left child of u and a right child of w. The problem with this approach lies in the fact that $x_i(p) - x_i(q)$ can be the side length of a linear number of regions and hence our KDS would not be local. It would also not be responsive, because if two points change their ordering we might have to update a linear number of longest-side certificates.

We avoid these problems by not maintaining a separate longest-side certificate for every region of the RBLS kd-tree. Instead, we identify all pairs of points that can define either the vertical or the horizontal side length of a region. We add all these pairs to one single list, the so-called *side-length list* which is sorted on the length of the sides. A longest-side event can happen only when two adjacent elements in the side-length list define the same length. (More precisely, they also have to define both a vertical and a horizontal side—nothing happens if two vertical sides have the same length. In fact, even when a vertical side and a horizontal side get the same length, it is possible that nothing happens, because they need not be sides of the same region.) So we have to maintain a certificate for each pair of consecutive elements in the side-length list. It remains to explain which sides precisely appear in the side-length list. To determine this, we construct two one-dimensional rank-based kd-trees $\mathcal{T}_i$ on the x_i-coordinates of the points in $\mathcal{P}$. Since all splitting lines for the nodes of $\mathcal{T}_i$ are orthogonal to the x_i-axis, $\mathcal{T}_i$ is in fact a balanced binary search tree. Let ν be a node in $\mathcal{T}_i$ and let ν_r and ν_ℓ be the first right and the first left ancestors of ν in $\mathcal{T}_i$. If p and q are the two points used in ν_r and ν_ℓ as splitting points, then

$x_i(p) - x_i(q)$ appears in the side-length list. Since the number of nodes in $\mathcal{T}_i$ is $O(n)$ and a node can be either the first left ancestor or the first right ancestor of at most $O(\log n)$ nodes, the number of elements in the side-length list is $O(n)$ and each point is involved in $O(\log n)$ elements of the side-length list. Moreover, all sides of all regions in $\mathcal{S}(\mathcal{P})$ exist in the side-length list.

Ordering event. When handling an ordering event that involves two points p and q and the x_i-axis, we have to update $\mathcal{A}_i$, the side-length list and the RBLS kd-tree. We update the array $\mathcal{A}_i$ by swapping p and q and updating the at most three certificates in which p and q are involved. We update the side-length list by replacing p by q and vice versa and computing the failure times of all certificates affected by these replacements. To quickly find in which elements of the side-length list a point p is involved we maintain for each rank i a list of elements of the side-length list in which rank i is involved. Since the number of elements in the side-length list is $O(n)$ and two ranks are involved in each element, this additional information uses $O(n)$ space. Since each rank is involved in $O(\log n)$ elements of the side-length list, updating the side-length list takes $O(\log n)$ time and inserting the failures times of the new certificates into the event queue takes $O(\log^2 n)$. To update the RBLS kd-tree, we first delete p and q from the RBLS kd-tree and then we re-insert them in their new order.

Deletion. Let ν be the lowest active node whose region contains p. The leaf containing p is a child of ν. This leaf must be removed. Let ω be the first active ancestor of ν. All nodes on the path from ω to ν must be checked whether they are useless. If so, they must be removed from the RBLS kd-tree.

Insertion. Let ν be the highest node in the RBLS kd-tree whose region contains p and such that the region corresponding to its only child ω does not contain p. Let ν' and ω' be the nodes in $\mathcal{S}(P)$ corresponding to ν and ω. Let u' be the lowest node on the path from ν' to ω' whose region contains both region(ω') and p as illustrated in Figure 4.3(b)— note that we do not explicitly maintain $\mathcal{S}(\mathcal{P})$ but the path between ν' and ω' can be constructed temporarily in $O(\log n)$ time. Because u' will become active, it must be added as a node, u, to the RBLS kd-tree and also every node on the path from ν' to u' must be added to the RBLS kd-tree if they are useful. The point p is maintained in a leaf whose parent is u.

Longest-side event. When handling a longest-side event that occurs at time t we first update the side-length list and the certificates involved in the event. Then we update the RBLS kd-tree as follows. Let p, q, p', and q' be the points involved in the event, more precisely, let $x_i(p(t)) - x_i(q(t)) = x_j(p'(t)) - x_j(q'(t))$. If $i = j$, then there is nothing to do, because the certificate failure can not correspond to a real longest-side event. Otherwise, we need to determine which, if any, of the regions of $\mathcal{S}(\mathcal{P})$ corresponds to the event. Because two sides of the region are given, we can follow a path from the

root to some node while temporally constructing each node from $\mathcal{S}(\mathcal{P})$ on the path which does not appear in the RBLS kd-tree. If there is no region with the two given sides, then we delete the temporary nodes and stop handling the event.

Otherwise there is exactly one region in $\mathcal{S}(\mathcal{P})$ that is specified by the two sides that triggered the event. (Note that this is only true in two dimensions, in higher dimensions the boundary of many regions can be defined by two sides—this is the only problem when attempting to extend these results to higher dimensions.) Let ν be the node that is associated with the event region. We add the two children ν_r and ν_ℓ of ν in $\mathcal{S}(\mathcal{P})$ to the RBLS kd-tree provided that they do not already exist in the RBLS kd-tree. Let the x_i-side of region(ν) be bigger than the x_j-side of region(ν) at the point in time just before t, denoted by t^-. At time t^-, $l(\nu)$ must be orthogonal to the x_i-axis and $l(\nu_\ell)$ and $l(\nu_r)$ must be orthogonal to the x_j-axis as illustrated in Figure 4.4(a)—note that region(ν) is a square at time t. Moreover, $l(\nu_\ell) = l(\nu_r)$, because the median of all points between the two x_i-sides of region(ν) is chosen to specify $l(\nu_\ell)$ and $l(\nu_r)$. Let A, B, C, and D be the four regions defined by $l(\nu)$, $l(\nu_\ell)$ and $l(\nu_r)$ as illustrated in Figure 4.4(a). We now split region(ν) with a line that is orthogonal to the x_j-axis and region(ν_r) and region(ν_ℓ) with a line that is orthogonal to the x_i-axis. Clearly $l(\nu)$ at time t is equal to $l(\nu_\ell)$ and $l(\nu_r)$ at time t^- and $l(\nu_\ell)$ and $l(\nu_r)$ at time t are equal to $l(\nu)$ at time t^-. The four subregion A, B, C, and D do not change and we only have to put them in the correct positions in the RBLS kd-tree as illustrated in Figure 4.4(b). Finally every node on the path from the root to ν as well as ν_r and ν_ℓ must be checked whether they are useless. If so, they must be removed from the RBLS kd-tree.

The number of events. Assume that the points in $\mathcal{P}$ follow constant-degree algebraic trajectories. Clearly the number of ordering events is $O(n^2)$. To count the number of longest-side events, we charge a longest-side event in which two sides s_1 and s_2 are involved to the side (either s_1 or s_2) that appeared in the side-length list later. At any point in time there are $O(n)$ elements in the side-length list and elements are only added or deleted whenever a ordering event occurs. During each ordering event, $O(\log n)$ elements can be added to the side-length list. All longest-side events that involve one of these "new" elements and one of the "old" elements are charged to one of the new elements, hence a total of $O(n \log n)$ events is charged to the new elements that are created during one ordering event. Since there are $O(n^2)$ ordering events, the number of longest-side events is $O(n^3 \log n)$. (This bound subsumes events that involve two new elements or two of the initial elements of the side-length list.)

Theorem 4.10 *A kinetic RBLS kd-tree for a set $\mathcal{P}$ of n moving points in $\mathbb{R}^2$ uses $O(n)$ storage and processes $O(n^3 \log n)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories. Each event can be handled in $O(\log^2 n)$ time and each point is involved in $O(\log n)$ certificates.*

4.4 Conclusions

We presented a variant of kd-tress, called rank-based kd-trees, for sets of points in $\mathbb{R}^d$. We showed that our rank-based kd-tree supports orthogonal range searching in $O(n^{1-1/d}+k)$ time and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized easily and efficiently. In the dynamic setting, either inserting or deleting a point affects the ranks of points which may cause a dramatic change in the rank-based kd-tree. A challenging problem is how to adapt the rank-based kd-tree to the insertion and deletion of points such that the query time does not change asymptotically.

We also proposed a variant of longest-side kd-trees, called rank-based longest-side kd-trees, for sets of points in $\mathbb{R}^2$. We showed RBLS kd-trees can be kinetized efficiently as well and like longest-side kd-trees, RBLS kd-trees support nearest-neighbor, farthest-neighbor, and approximate range search queries in $O((1/\varepsilon)\log^2 n)$ time. Unfortunately we have been unable to generalize this result to higher dimension. We leave it as an interesting open problem for future research.

Chapter 5

Kinetic collision detection for convex fat objects

Abstract. We design compact and responsive kinetic data structures for detecting collisions between n convex fat objects in 3-dimensional space that can have arbitrary sizes. Our main results are:

(i) If the objects are 3-dimensional balls that roll on a plane, then we can detect collisions with a KDS of size $O(n \log n)$ that can handle events in $O(\log^2 n)$ time. This structure processes $O(n^2)$ events in the worst case, assuming that the objects follow constant-degree algebraic trajectories.

(ii) If the objects are convex fat 3-dimensional objects of constant complexity that are free-flying in $\mathbb{R}^3$, then we can detect collisions with a KDS of $O(n \log^6 n)$ size that can handle events in $O(\log^7 n)$ time. This structure processes $O(n^2)$ events in the worst case, assuming that the objects follow constant-degree algebraic trajectories. If the objects have similar sizes then the size of the KDS becomes $O(n)$ and events can be handled in $O(\log n)$ time.

An extended abstract of this chapter was previously published as: M. A. Abam and M. de Berg, S-H. Poon, and B. Speckmann, Kinetic collision detection for convex fat objects, In *Proc. European Symposium on Algorithms (ESA)*, pages 4–15, 2006. The full paper will be published in *Algorithmica*.

5.1 Introduction

Background. Collision detection is a basic problem arising in all areas of computer science involving objects in motion—motion planning, animated figure articulation, computer-simulated environments, or virtual prototyping, to name a few. Very often the problem of detecting collisions is broken down into two phases: a *broad phase* and a *narrow phase*. The broad phase determines pairs of objects that might possibly collide, frequently using (hierarchies of) bounding volumes to speed up the process. The narrow phase then uses specialized techniques to test each candidate pair, often by tracking closest features of the objects in question, a process that can be sped up significantly by exploiting spatial and temporal coherence. See [83] for a detailed overview of algorithms for such collision and proximity queries.

Related work. One of the first papers on kinetic collision detection was published by Basch *et al.* [22], who designed a KDS for collision detection between two simple polygons in the plane. Their work was extended to an arbitrary number of polygons by Agarwal *et al.* [10]. Kirkpatrick *et al.* [74] and Kirkpatrick and Speckmann [75] also described KDSs for kinetic collision detection between multiple polygons in the plane. These solutions all maintain a decomposition of the free space between the polygons into “easy” pieces (usually pseudo-triangles). Unfortunately it seems quite hard to define a suitable decomposition of the free space for objects in 3D, let alone maintain it while the objects move—the main problem being, that all standard decomposition schemes in 3D can have quadratic complexity. Hence, even though collision detection is the obvious application for kinetic data structures, there has hardly been any work on kinetic collision detection in 3D.

There are only a few papers that deal directly with (specialized versions of) kinetic 3D collision detection. Guibas *et al.* [60], extending work by Erickson et al. [45] in the plane, show how to certify the separation of two convex polyhedra moving rigidly in 3D using certain outer hierarchies. Basch *et al.* [24] describe a structure for collision detection among multiple convex *fat* objects that have almost the same size. The structure of Basch *et al.* uses $O(n \log^2 n)$ storage and processes $O(n^2)$ events and events can be processed in $O(\log^3 n)$ time. Coming and Staadt [33] kinetize the sweep-and-prune approach to find candidate pairs of objects that might collide. Their method has a quadratic worst-case bound and they give only experimental evidence for its performance. If all objects are spheres of similar sizes Kim *et al.* [72] present an event-driven approach that subdivides space into cells and processes events whenever a sphere enters or leaves a cell. This approach was later extended [73] to accommodate spheres with unknown trajectories but still similar sizes. There is only experimental evidence for the performance of this method. Finally, Guibas *et al.* [60] use the power diagram of a set of arbitrary balls in 3D to kinetically maintain the closest pair among them. The worst-case complexity of this structure is quadratic and it might undergo more than cubically many changes.

Our results. Our main goal is to develop KDSs for 3D collision detection that have a *near-linear number of certificates* for *multiple* convex fat objects of *varying sizes*. As discussed above, none of the existing solutions achieves all these goals simultaneously. Our KDSs can be viewed as structures that perform the broad phase of the global collision-detection approach sketched above; one still has to detect collisions between the candidate pairs of objects produced by the KDS. Assuming the objects have constant complexity, this can trivially be done in constant time per pair; how to do this for complex objects is beyond the scope of this chapter. Thus the challenge is to get a near-linear number of certificates, so that the number of candidate pairs is reduced from quadratic to near-linear.

We start in Section 5.2 with the special case of n balls of arbitrary sizes rolling on a plane. Here we present an elegant and simple KDS that uses $O(n \log n)$ storage and processes $O(n^2)$ events in the worst case if the objects follow constant-degree algebraic[1] trajectories. Processing an event takes $O(\log^2 n)$ time.

In Section 5.3 we turn our attention to free-flying convex fat objects. Note that we do not assume the objects to be polyhedral. We first study fat objects that have similar sizes. We give an almost trivial KDS that has $O(n)$ size and processes $O(n^2)$ events; handling an event takes $O(\log n)$ time. This improves both the storage and the event-handling time of the KDS of Basch *et al.* [24] by several logarithmic factors. Next we consider the much more difficult general case, where the fat objects can have vastly different sizes. Here we present a KDS that uses $O(n \log^6 n)$ storage and processes $O(n^2)$ events; handling an event takes $O(\log^7 n)$ time. This is the first collision-detection KDS for multiple objects in 3D that has a near-linear number of certificates and does not require the objects to have similar sizes. Even though our KDS for this case uses $O(n \log^6 n)$ storage, it maintains only a linear number of candidate pairs of objects to test for collisions; the additional storage is used in various supporting data structures. Our structure is based on the following idea: we put a number of points—we call them guards—around each object in such a way that if two objects collide, one must contain a guard from the other. Because the objects are fat, we can show that a constant number of guards per object suffices. The idea of reducing problems on fat objects to problems on suitably chosen points has been used before—see e.g. [37, 40]. In our context, however, it is far from straightforward to apply since detecting collisions between objects and guards is nearly as difficult as detecting collisions between the objects themselves. Nevertheless, using several additional ideas, we show how to make this approach work.

[1]In fact, the bound on the number of events holds in a more general setting: we maintain lists of certain x- and y-coordinates—for instance the coordinates of the tangency points of the disks with the plane on which they roll—whose values change according to the motions of the objects. The number of events is bounded by the number of changes (swaps) in these sorted lists. The $O(n^2)$ bound thus holds if we assume that any pair of coordinates swaps $O(1)$ times (which is for example the case if the motions are constant-degree algebraic). A similar remark holds for the other KDSs that we develop.

5.2 Balls rolling on a plane

Assume that we are given a set $\mathcal{B}$ of n 3-dimensional balls which are rolling on a 2-dimensional plane T, that is, the balls in $\mathcal{B}$ move continuously while remaining tangent to T—see Figure 5.1(Left). In this section we describe a responsive and compact KDS that detects collisions between the balls in $\mathcal{B}$.

The basic idea behind our KDS is to construct a *collision tree* recursively as follows:

- If $|\mathcal{B}| = 1$, then there are obviously no collisions and the collision tree is just a single leaf.
- If $|\mathcal{B}| > 1$, then we partition $\mathcal{B}$ into two subsets, $\mathcal{B}_S$ and $\mathcal{B}_L$. The subset $\mathcal{B}_S$ contains the $\lfloor n/2 \rfloor$ smallest balls and the subset $\mathcal{B}_L$ contains the $\lceil n/2 \rceil$ largest balls from $\mathcal{B}$, where ties are broken arbitrarily. The collision tree now consists of a root node that has an associated structure to detect collisions between any ball from $\mathcal{B}_S$ and any ball from $\mathcal{B}_L$, and two subtrees that are collision trees for the sets $\mathcal{B}_S$ and $\mathcal{B}_L$, respectively.

To detect all collisions between the balls in $\mathcal{B}$ it suffices to detect collisions between the two subsets maintained at every node of the collision tree. Let $\mathcal{B}_S$ and $\mathcal{B}_L$ denote the two subsets maintained at a particular node. The remainder of this section focusses on detecting collisions between the balls in $\mathcal{B}_S$ and those in $\mathcal{B}_L$. In particular, we describe a KDS of size $O(|\mathcal{B}_S| + |\mathcal{B}_L|)$ that can handle events in $O(\log n)$ time—see Lemma 5.5. The structure processes $O((|\mathcal{B}_S| + |\mathcal{B}_L|)^2)$ events in the worst case, assuming that the balls follow constant-degree algebraic trajectories. Since the same event can occur simultaneously at $O(\log n)$ nodes of the collision tree, we obtain the following theorem:

Theorem 5.1 *For any set $\mathcal{B}$ of n 3-dimensional balls that roll on a plane, there is a KDS for collision detection that uses $O(n \log n)$ space and processes $O(n^2)$ events in the worst case, assuming that the balls follow constant-degree algebraic trajectories. Each event can be handled in $O(\log^2 n)$ time.*

5.2.1 Detecting collisions between small and large balls

As mentioned above, we can restrict ourselves to detecting collisions between balls from two disjoint sets $\mathcal{B}_S$ and $\mathcal{B}_L$ where the balls in $\mathcal{B}_L$ are at least as large as the balls in $\mathcal{B}_S$. Recall that all balls are rolling on a plane T. Our basic strategy is the following: we associate a region D_i on T with each $B_i \in \mathcal{B}_L$ such that if the point of tangency of a ball $B_j \in \mathcal{B}_S$ and T is not contained in D_i, then B_j cannot collide with B_i. The regions associated with the balls in $\mathcal{B}_L$ need to have two important properties: (i) each point in T is contained in a constant number of regions and (ii) we can efficiently detect whenever a region starts or stops to contain a tangency point when the balls in $\mathcal{B}_L$ and $\mathcal{B}_S$ move.

We first deal with the first requirement, that is, we consider $\mathcal{B}_L$ to be static. For a ball B_i let r_i denote its radius and let t_i be the point of tangency of B_i and T.

The threshold disk. We define the distance of a point q in the plane T to a ball B_i as follows. Imagine that we place a ball $B(q)$ of initial radius 0 at point q. We then inflate $B(q)$ while keeping it tangent to T at q, until it collides with B_i. We define the distance of q and B_i, which we denote by $\mathrm{dist}(q, B_i)$, to be the radius of $B(q)$. More precisely, $\mathrm{dist}(q, B_i)$ is the radius of the unique ball that is tangent to T at q and tangent to B_i. It is easy to show that $\mathrm{dist}(q, B_i) = d(q, t_i)^2/4r_i$ where $d(q, t_i)$ denotes the Euclidean distance between q and t_i.

Since we have to detect collisions only with balls from $\mathcal{B}_S$ and the balls in $\mathcal{B}_L$ are at least as large as those in $\mathcal{B}_S$, we can stop inflating when $B(q)$ is as large as the smallest ball in $\mathcal{B}_L$. Based on this, we define the threshold disk D_i of a ball $B_i \in \mathcal{B}_L$ as follows: a point $q \in T$ belongs to D_i if and only if $\mathrm{dist}(q, B_i) \leqslant r_{\min}$ where $r_{\min}$ is the radius of the smallest ball in $\mathcal{B}_L$—see Figure 5.1(Right). Because $\mathrm{dist}(q, B_i) = d(q, t_i)^2/4r_i$, D_i is a disk whose radius is $2\sqrt{r_i \cdot r_{\min}}$ and whose center is t_i.

Clearly a ball $B_j \in \mathcal{B}_S$ cannot collide with a ball $B_i \in \mathcal{B}_L$ as long as t_j is outside D_i—see Figure 5.2(Left). In the following, we prove that a point $q \in T$ can be contained in at most a constant number of threshold disks. We start by proving a more general result, which we will need later when we replace the threshold disks by threshold boxes. For a given constant $c \geqslant 0$, let $c \cdot D_i$ denote the disk with radius $c \cdot \mathrm{radius}(D_i)$ and center t_i.

Lemma 5.2 *The number of threshold disks D_j that are at least as large as a given threshold disk D_i and for which $c \cdot D_i \cap c \cdot D_j \neq \emptyset$, is at most $(8\,c^2 + 2\,c + 1)^2 + 1$.*

Proof. Let $\mathcal{D}(i)$ be the set of all threshold disks D_j that are at least as large as D_i and for which $c \cdot D_i \cap c \cdot D_j \neq \emptyset$. First we prove that there are no two balls B_j and B_k such that $r_k \geqslant r_j > 16\,c^2\,r_i$ and $D_j, D_k \in \mathcal{D}(i)$. Assume, for contradiction, that there are two balls B_j and B_k with this property. Since B_j and B_k are disjoint, we have $d(t_j, t_k) \geqslant ((r_i + r_j)^2 - (r_i - r_j)^2)^{1/2} = 2\sqrt{r_j \cdot r_k} > 8\,c\sqrt{r_k \cdot r_i}$. We also know that

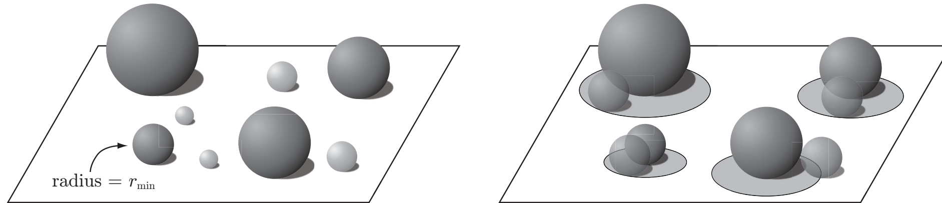


Figure 5.1 (Left) Balls rolling on a plane—balls in $\mathcal{B}_S$ ($\mathcal{B}_L$) are light gray (dark gray). (Right) The radius $r_{\min}$ of the smallest ball in $\mathcal{B}_L$ defines the threshold disks.

$d(t_j, t_k) \leqslant d(t_j, t_i) + d(t_i, t_k) \leqslant 8\,c\sqrt{r_k \cdot r_i}$ which is a contradiction. Hence, there is at most one ball B_j such that $r_j > 16\,c^2\,r_i$ and $D_j \in \mathcal{D}(i)$.

It remains to show that the number of balls B_j whose radii are not greater than $16\,c^2\,r_i$ and for which $D_j \in \mathcal{D}(i)$ is at most $(8\,c^2 + 2\,c + 1)^2$. Let B_j be one of these balls and let x be a point in $c \cdot D_j \cap c \cdot D_i$. Since

$$d(t_i, t_j) \leqslant d(t_i, x) + d(t_j, x) \leqslant 2\,c\sqrt{r_i \cdot r_{\min}} + 2\,c\sqrt{r_j \cdot r_{\min}} \leqslant (2\,c + 8\,c^2)\,r_i\ ,$$

t_j must lie in a disk whose center is t_i and whose radius is $(2\,c + 8\,c^2)\,r_i$. We also know that $d(t_j, t_k) \geqslant 2\sqrt{r_j \cdot r_k} \geqslant 2\,r_i$ for any two such balls B_j and B_k. Thus the set $\mathcal{D}'(i)$ of disks centered at t_j with radius r_i for all $D_j \in \mathcal{D}(i)$ are disjoint. Note that any disk in $\mathcal{D}'(i)$ lies inside the disk centered at t_i with radius $((2\,c + 8\,c^2) + 1)\,r_i$. Thus $|\mathcal{D}'(i)| \leqslant (\pi(2\,c + 8\,c^2 + 1)^2 r_i^2)/(\pi r_i^2) = (2\,c + 8\,c^2 + 1)^2$ which implies $|\mathcal{D}(i)| \leqslant (2\,c + 8\,c^2 + 1)^2 + 1$. □

Lemma 5.3 *Each point $q \in T$ is contained in at most a constant number of threshold disks.*

Proof. Let D_i be the smallest threshold disk containing q. Lemma 5.2 with $c = 1$ implies that the number of disks not smaller than D_i and intersecting D_i is constant. Hence the number of threshold disks containing q is constant. □

The threshold box. The threshold disks have the important property that each point in T is contained in a constant number of disks. But unfortunately, as the balls in $\mathcal{B}_L$ and $\mathcal{B}_S$ move, it is difficult to detect efficiently whenever a tangency point enters or leaves a threshold disk. Hence we replace each threshold disk by its axis-aligned bounding box—see Figure 5.2(Right). The bounding box of a threshold disk D_i associated with a $B_i \in \mathcal{B}_L$ is called a *threshold box* and is denoted by $\mathrm{TB}(B_i)$. The following lemma states that the threshold boxes retain the crucial property of the threshold disks, namely,

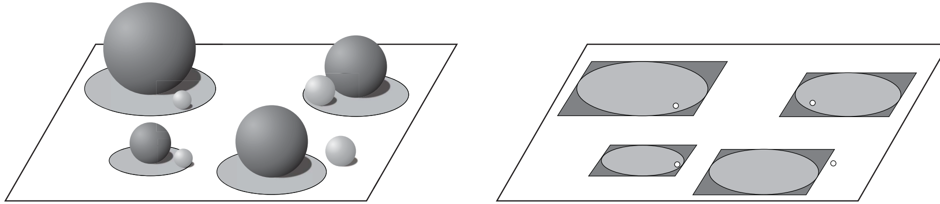


Figure 5.2 (Left) Detecting collisions with the threshold disks. (Right) Replacing threshold disks with threshold boxes.

that each point $q \in T$ is contained in at most a constant number of threshold boxes. It follows fairly easily from Lemma 5.2.

Lemma 5.4 *Each point $q \in T$ is contained in at most a constant number of threshold boxes.*

Proof. Instead of considering the threshold boxes directly, we consider the disks defined by the circumcircles $D(\mathrm{TB}(B_j))$ of each threshold box $\mathrm{TB}(B_j)$ with $B_j \in \mathcal{B}_L$. We have $\mathrm{radius}(D(\mathrm{TB}(B_j))) = \sqrt{2} \cdot \mathrm{radius}(D_j)$ for all $B_j \in \mathcal{B}_L$. Let $\mathrm{TB}(B_i)$ be the smallest box containing q. Lemma 5.2 with $c = \sqrt{2}$ implies that the number of circumcircle disks that are at least as large as $D(\mathrm{TB}(B_i))$ and that intersect $D(\mathrm{TB}(B_i))$ is constant. Hence the number of threshold boxes that are not smaller than $\mathrm{TB}(B_i)$ and intersect $\mathrm{TB}(B_i)$ is constant, and so is the number of threshold boxes containing q. □

Kinetic maintenance. Recall that to detect collisions between $\mathcal{B}_S$ and $\mathcal{B}_L$, for each ball $B_j \in \mathcal{B}_S$ we determine which threshold boxes of balls in $\mathcal{B}_L$ contain the tangency point t_j. Note that according to Lemma 5.4, t_j is contained in a constant number of threshold boxes. For each $B_j \in \mathcal{B}_S$ we maintain the set of threshold boxes that contain t_j and certificates that guarantee disjointness of B_j and the balls from $\mathcal{B}_L$ whose threshold boxes contain t_j.

To maintain our structure we only need to detect when a tangency point t_j enters or leaves a threshold box. To do so, we maintain two sorted lists: one storing the x-coordinates of the tangency points of $\mathcal{B}_S$ and the minimum and maximum x-coordinates of the threshold boxes associated with the balls in $\mathcal{B}_L$, the other storing the y-coordinates of the tangency points of $\mathcal{B}_S$ and the minimum and maximum y-coordinates of the threshold boxes. If the objects follow constant-degree algebraic trajectories, the number of events processed by our structure—that is, the number of swaps in these sorted lists—is quadratic in the size of $\mathcal{B}_S$ and $\mathcal{B}_L$. Moreover, each such event can be processed in $O(\log n)$ time: $O(1)$ time to swap the points, and $O(\log n)$ time to update the event queue.

Lemma 5.5 *Let $\mathcal{B}_S$ and $\mathcal{B}_L$ be two disjoint sets of balls that roll on a plane where the balls in $\mathcal{B}_L$ are at least as large as the balls in $\mathcal{B}_S$. There is a KDS for collision detection between the balls of $\mathcal{B}_S$ and those of $\mathcal{B}_L$ that uses $O(|\mathcal{B}_S|+|\mathcal{B}_L|)$ space, and that processes $O((|\mathcal{B}_S| + |\mathcal{B}_L|)^2)$ events if the balls follow constant-degree algebraic trajectories. Each event can be handled in $O(\log n)$ time.*

Remark. Recall that we have a collision-detection KDS as in Lemma 5.5 for every node of a collision tree, as described at the beginning of this section. Each such collision-detection KDS generates events, but we do not maintain these events in separate event queues. Instead we maintain a global event queue and we insert the failure time of each certificate into the global event queue. It is easy to see that at any time there are $O(n \log n)$ certificates in the global event queue. Hence, the asymptotic complexity of inserting (deleting) certificates into (from) the global event queue remains $O(\log n)$.

Remark. In the above KDS, we maintain two sorted lists for every node of the collision tree. Thus an event may happen in $O(\log n)$ nodes on a path from the root to a leaf simultaneously. This forces the KDS to insert (delete) $O(\log n)$ certificates into (from) the event queue which takes $O(\log^2 n)$ time. Another possibility is to maintain two global sorted lists based on x- and y-coordinates, instead of having two sorted lists for each node. This way an event can create or delete a constant number of certificates, which implies the response time is $O(\log n)$. Since every ball is associated with $O(\log n)$ threshold boxes, the size of the global sorted list is $O(n \log n)$, which means the number of events is $O(n^2 \log^2 n)$. Hence, the decrease in response time comes at the cost of an increase in the number of events to be processed.

5.3 Free-flying fat objects in 3-space

We now turn our attention to collision detection for a set $\mathcal{K}$ of n free-flying objects in 3-space. We will show how to obtain a compact and responsive KDS when $\mathcal{K}$ consists of convex, constant-complexity fat objects. Note that we do not require the objects to be polyhedral.

We will use the following definition of fatness [71]. An object K is called *ρ-fat*, for some $\rho \geqslant 1$, if there are two concentric balls $B^-(K)$ and $B^+(K)$ such that $B^-(K) \subset K \subset B^+(K)$ and

$$\mathrm{radius}(B^+(K)) / \mathrm{radius}(B^-(K)) \leqslant \rho\,.$$

Since we are dealing with convex objects, this definition is equivalent up to constant factors to other definitions of fatness that have been used [41]. We call $\mathrm{radius}(B^-(K))$ and $\mathrm{radius}(B^+(K))$ the *inner radius* and *outer radius* of K, respectively, and we call the common center of $B^-(K)$ and $B^+(K)$ the *center* of K. We say that an object K is *larger* than another object K' if the inner radius of K is larger than the inner radius of K'.

Unfortunately the approach of the previous section does not work for free-flying objects, not even if we are dealing with balls. The problem is that the radius of the threshold ball of a ball B_i will now be $r_i + r_{\min}$ instead of $2\sqrt{r_i \cdot r_{\min}}$ and this invalidates the proof of Lemma 5.2 for $c > 1$ and thus invalidates Lemma 5.4.

5.3.1 Similarly sized objects

We first consider the case where the objects have similar sizes. More precisely, let σ be the *scale factor* of the scene, that is, the ratio between the sizes of the largest and the smallest inner ball:

$$\sigma = \frac{\max_{K \in \mathcal{K}} \mathrm{radius}(B^-(K))}{\min_{K \in \mathcal{K}} \mathrm{radius}(B^-(K))}$$

It follows from the results of Zhou and Suri [104] that the number of pairs of intersecting bounding boxes of the objects in $\mathcal{K}$ is at most $O(\rho\sqrt{\rho^3\sigma^3 n}) = O(\rho^2\sigma\sqrt{\rho\sigma}n)$. (A

similar but slightly weaker result also follows directly from results in Van der Stappen's thesis [96].) Hence, if σ is a constant, we can simply maintain the set of pairs of intersecting bounding boxes, and for each such pair add a certificate to test for disjointness of the corresponding objects.

To maintain the pairs of intersecting bounding boxes, we maintain three sorted lists: one on the minimum and maximum x-coordinates of the boxes, one on the minimum and maximum y-coordinates of the boxes, and one on the minimum and maximum z-coordinates of the boxes. Whenever there is a swap in one of these lists, two boxes may intersect or become apart. If two boxes intersect, we add a certificate for the corresponding objects. If they become apart, we remove the corresponding certificate. This leads to the following theorem.

Theorem 5.6 *For any set $\mathcal{K}$ of n convex, constant-complexity ρ-fat objects with scale factor σ, there is a KDS for collision detection that uses $O(\rho^2\sigma\sqrt{\rho\sigma}n)$ storage and processes $O(n^2)$ events in the worst case, assuming that the objects follow constant-degree algebraic trajectories. Each event can be handled in $O(\log n)$ time.*

5.3.2 Arbitrarily sized objects

When the sizes of the objects vary greatly, then there can be a quadratic number of intersecting bounding boxes even when the objects are fat. Hence, a more sophisticated approach is needed. Our global strategy for this case is as follows. We place a number of so-called *guarding points*—or *guards*, for short—around each object $K \in \mathcal{K}$. The guards for K are defined in a local reference frame for K, so they follow the motion of K. More precisely, they follow the motion of a fixed reference point of K. We choose the guards in such a way that when two objects collide, the larger object must contain at least one guard from the smaller object. This reduces the collision-detection problem to maintaining for each guard which object contains it. The next lemma states that we can always find a small guarding set because the objects are fat.

Lemma 5.7 *For any ρ-fat object K, there is a set $G(K)$ of $O(\rho^6)$ guarding points such that any ρ-fat object K' that collides with K and is at least as large as K contains a point from $G(K)$.*

Proof. Let $r := \mathrm{radius}(B^-(K))$. Let C be the cube whose center coincides with the center of K, and whose side length is $2(\rho+1)r$. Draw a regular grid in C whose cells have side length $2r/(\sqrt{3}(\rho+1))$—see Figure 5.3(i) for a (2-dimensional) illustration. The grid points together form the set $G(K)$. Clearly $|G(K)| = O(\rho^6)$. It remains to argue that $G(K)$ is a guarding set.

Let K' be an object colliding with K and at least as large as K, and let p be a point where K and K' touch. Because K' is at least as large as K, the ball $B^-(K') \subset K'$ has radius r' at least r. Consider the line ℓ through p and $\mathrm{center}(B^-(K'))$. Let d be the distance

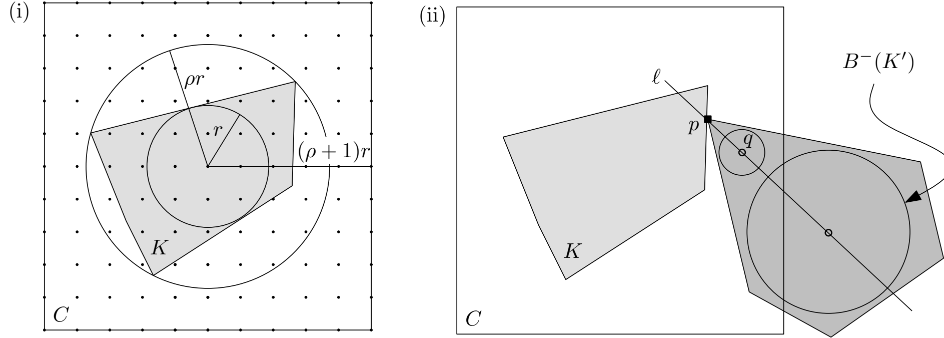


Figure 5.3 Illustrations for the proof of Lemma 5.7.

between p and $\mathrm{center}(B^-(K'))$, and let q be the point in between p and $\mathrm{center}(B^-(K'))$ at distance $\frac{r/(\rho+1)}{r'} \cdot d$ from p. Finally, let $B(q)$ be the ball centered at q and with radius $r/(\rho+1)$—see Figure 5.3(ii). Observe that $B(q)$ can be obtained by scaling $B^-(K')$ with respect to p by a factor of $\frac{r/(\rho+1)}{r'}$. Since K' is convex, $p \in K'$, and $B^-(K') \subset K'$, this implies $B(q) \subset K'$. We claim that $B(q) \subset C$. Since $B(q)$ has radius $r/(\rho+1)$, this means it must contain at least one point of $G(K)$, which will prove the lemma.

It remains to prove the claim that $B(q) \subset C$. To this end note that d is at most $\rho \cdot r'$, because K' is ρ-fat. This implies that the distance of p to any point in $B(q)$ is at most

$$\frac{r/(\rho+1)}{r'} \cdot d \;+\; \frac{r}{\rho+1} \;\leqslant\; r.$$

On the other hand, the distance of p to the boundary of C is at least r. Hence, $B(q) \subset C$, as claimed. □

Our KDS for collision detection thus works as follows. For each object $K \in \mathcal{K}$ we compute a set $G(K)$ of guards according to Lemma 5.7. Our goal is now to maintain for each $g \in G(K)$ the object $K(g)$ containing g (if such an object exists). Let $\mathrm{Cand}(K) := \{K(g) : g \in G(K)\}$; the set $\mathrm{Cand}(K)$ contains the candidates with which we check for collisions. More precisely, for each object $K(g) \in \mathrm{Cand}(K)$, our KDS has a certificate testing for the disjointness of K and $K(g)$.

Unfortunately, it seems difficult to maintain the set $\mathrm{Cand}(K)$ directly. This would require us to detect when an object K' starts to contain a guard g, which is difficult to do efficiently. Hence, we replace the objects by their bounding boxes. Because the bounding boxes are axis-aligned, it will be easier to check whether any of them starts (or stops) to contain a guard of some other object. This introduces a new problem, however; a guard can be contained in many bounding boxes—see Figure 5.4. Clearly, we cannot afford to maintain for each guard g all the bounding boxes that contain it. Next we describe how to deal with this problem.

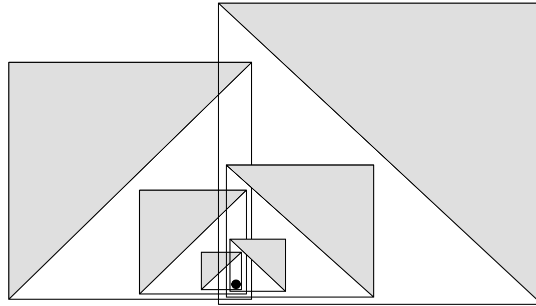


Figure 5.4 A guard can be contained in many bounding boxes.

Consider a guard g. As noted earlier, there can be many disjoint objects whose bounding boxes contain g. When this happens, however, the objects must become larger and larger, as shown in Figure 5.4, with the larger objects being "behind" the smaller ones. Thus the objects that are closest to g in a some direction are the candidates for containing g. Hence, the idea is to maintain for g not all objects whose bounding boxes contain g, but only the closest objects around g.

To make this idea work, we first partition the space around g into cones, as follows. Let U be the unit cube, centered at the origin. Draw a grid on each face of U, such that the grid cells have edge length $1/(2\sqrt{6}\rho)$. Triangulate each grid cell. We have now partitioned the surface of U into $O(\rho^2)$ triangles. Each triangle defines, together with the origin, an (infinite) cone γ by taking the union of all rays emanating from the origin and passing through the triangle. Since the grid cells have edge length $1/(2\sqrt{6}\rho)$ their diagonals have length $1/(4\sqrt{3}\rho)$, which implies the following.

Lemma 5.8 *Let ℓ_1 and ℓ_2 be two rays originating from the apex of a cone γ and being inside the cone. Then the angle between ℓ_1 and ℓ_2 is at most $\arctan(1/(2\sqrt{3}\rho))$.*

The set of cones for a guard g is obtained by translating these cones such that their apices—the origin in the construction—coincide with g. We denote this set by $\Gamma(g)$.

Note that the motion of each cone is purely translational: even when an object K rotates, its guards just follow the path of the reference point of K and so the cones for that guard only translate. This means that any cone will always be a translated copy of one of the "standard" cones defined for U. From now on, whenever we speak of a cone we refer to a cone constructed for a guard, as described above.

Since it seems to be difficult to efficiently maintain the closest object to g, the apex of a cone γ, we maintain the object whose center's orthogonal projection onto a specific side of γ is the closest one to g. More precisely, for each cone we defined for U we choose one of its edges as its *representative edge*. This also gives us a representative edge for each cone constructed for any guard g. From now on, whenever we are discussing a cone γ

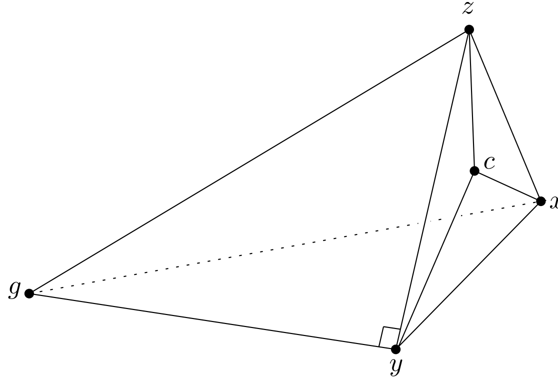


Figure 5.5 The intersection of cone γ and the plane orthogonal to the representative edge of γ.

with apex g and we are talking about the object closest to g, we refer to the object whose center's orthogonal projection onto γ's representative edge is closest to g.

The next lemma implies that we can indeed restrict our attention to the closest object to g among those objects whose bounding boxes contain g. For an object K, let $\mathrm{bb}(K)$ denote its (axis-aligned) bounding box.

Lemma 5.9 *Let $\mathcal{K}(\gamma)$ be the set of all objects K whose center lies in a cone γ and such that $\mathrm{bb}(K)$ contains the apex g of the cone. Suppose that one of these objects, $K(g)$, contains g. Then $K(g)$ must be the closest object to g in $\mathcal{K}(\gamma)$. Moreover, suppose the objects in $\mathcal{K}(\gamma)$ move in such a way that their centers remain inside γ. Then the order of the orthogonal projections of their centers onto the representative edge of γ remains unchanged.*

Proof. Let r and c be the inner radius and the center of an object $K \in \mathcal{K}(\gamma)$, respectively. Consider a plane passing through c and being orthogonal to the representative edge of γ. The intersection of γ and this plane is a triangle xyz—see Fig. 5.5. We claim (and will prove later) that $d(c,x), d(c,y), d(c,z) \leqslant r$. Because K is convex, this implies that the triangle xyz is inside the object K. Hence, the objects whose centers are inside the cone cannot exchange order during the motion as long as their centers remain inside the cone.

Now let $K = K(g)$. Then the tetrahedron $gxyz$ is inside $K(g)$ which means the centers of the other objects must lie outside $gxyz$. This implies $K(g)$ is the closest object to g.

It remains to prove the claim that $d(c,x), d(c,y), d(c,z) \leqslant r$. Assume (the extension of) gy is the representative edge of γ. Since $\mathrm{bb}(K)$ contains g and $\angle gyc$ is a right angle, we have

$$d(g,y) \leqslant d(g,c) \leqslant \sqrt{3}\rho r.$$

Moreover, we have

$$\begin{aligned} d(c,y) = d(g,y)\cdot\tan(\angle cgy) &\leqslant d(g,y)\cdot\tan(\arctan(1/(2\sqrt{3}\rho))) \\ &\leqslant (\sqrt{3}\rho r)\cdot(1/(2\sqrt{3}\rho) \leqslant r/2. \end{aligned}$$

Similarly, $d(z,y) \leqslant r/2$ and $d(x,y) \leqslant r/2$. It follows that

$$d(c,z) \leqslant d(c,y) + d(y,z) \leqslant r$$

and

$$d(c,x) \leqslant d(c,y) + d(y,x) \leqslant r.$$

□

To summarize, our KDS works as follows. For each object $K \in \mathcal{K}$ we compute a set $G(K)$ of guards according to Lemma 5.7. For each guard g we construct a collection $\Gamma(g)$ of infinite cones with apex g. For each cone $\gamma \in \Gamma(g)$ we maintain the closest object whose center is inside γ and whose bounding box contains g, and we have a certificate testing for disjointness for this object with the object for which g is a guard. Next we describe a KDS that maintains all this information efficiently.

Details of the KDS.

Let $G(\mathcal{K}) := \{G(K) : K \in \mathcal{K}\}$ denote the set of all guards over all objects, let $\Gamma(\mathcal{K}) := \{\Gamma(g) : g \in G(\mathcal{K})\}$ denote the collection of all cones, and let $\mathrm{bb}(\mathcal{K})$ denote the set of bounding boxes of the objects in $\mathcal{K}$.

Detecting events. We wish to maintain for each $\gamma \in \Gamma(g)$ the closest object $\mathcal{K}^*(\gamma)$ to g whose center is inside γ and whose bounding box contains g. By Lemma 5.9 this object can change only when one of the following two events happens:

Box event: a bounding box starts or stops to contain a guard.

Center event: a center moves into or out of a cone.

To detect box events, we maintain three sorted lists. The first list is sorted on x-coordinate and contains the guards in $G(\mathcal{K})$ as well as the bounding boxes, where each bounding box occurs twice (according to its maximum and minimum x-coordinates). We have similar lists sorted on y- and z-coordinates.

To detect center events, we observe that each cone is a translate of one of the $O(\rho^2)$ cones defined for the unit cube. The cones are generated by six triangulated grids, one on each facet of the unit cube, so the facets of the cones have only $O(\rho)$ distinct orientations. Hence, we can detect center events using $O(\rho)$ sorted lists. Each sorted list corresponds to

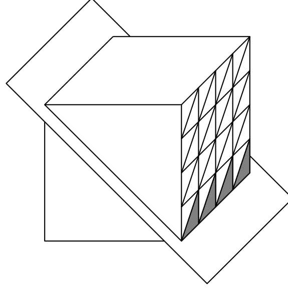


Figure 5.6 The cones corresponding to the shaded triangles all share a common plane defining one of their facets.

a possible orientation of a cone facet, and stores the object centers and the cones that have a facet in the given orientation. More precisely, instead of storing the cones themselves, we store the planes containing the facets of the cones. Notice that a plane bounds up to $O(\rho)$ cones—see Figure 5.6.

Lemma 5.10 *The box and center events can be detected with a KDS that uses $O(\rho^7 n)$ storage and that processes $O(\rho^{13} n^2)$ events in total, assuming the objects follow constant-degree algebraic trajectories. At each event processed by this KDS, we spend $O(\log \rho)$ time to test whether the event corresponds to an actual box or center event.*

Proof. Recall that we have $O(\rho^6)$ guards per object, and $O(\rho^2)$ cones per guard.

For the box events we have three sorted lists, each storing $O(n)$ boxes and $O(\rho^6 n)$ guards. Hence, their total size is $O(\rho^6 n)$ and the total number of events is $O(\rho^{12} n^2)$. Whenever we have a swap in one of these lists, we just check in $O(1)$ time whether it corresponds to a guard entering or leaving a box.

For the center events we have $O(\rho)$ sorted lists. For each guard, the cones are defined by triangulated grids on the facets of a unit cube centered at the guard. This grid is induced by $O(\rho)$ lines on the facets. Hence, as remarked earlier, the $O(\rho^2)$ cones are generated by $O(\rho)$ planes—one plane for each grid line and one for each diagonal line inducing the triangulation. Since we have $O(\rho^6)$ guards per object, we have in total $O(\rho^7)$ planes per object. Each guard contributes one plane to each list. Hence, we have $O(\rho)$ lists, each containing $O(\rho^6 n)$ planes. This means these lists together use $O(\rho^7 n)$ storage and have $O(\rho^{13} n^2)$ events. Whenever we have an event in one of these lists we check whether it corresponds to a center crossing a plane. If so, we must find out which of the $O(\rho)$ cones bounded by that plane, if any, are involved. Note that there can be two: the center could enter one cone and leave another cone. Finding out the cones involved can easily be done in $O(\log \rho)$ time. □

Handling events. When we have detected a center event, we may have to update the object $\mathcal{K}^*(\gamma)$ of at most two cones. Next we describe how to handle the event involving an object K and some cone γ defined for a guard g.

When $\mathrm{bb}(K)$ starts to contain g, or when the center of K moves into γ, things are easy: If $\mathcal{K}^*(\gamma)$ does not yet exist, K becomes the closest object to g and so we set $\mathcal{K}^*(\gamma) := K$; otherwise, we check whether K is closer to g than the current $\mathcal{K}^*(\gamma)$ and, if so, we set $\mathcal{K}^*(\gamma) := K$.

Handling the case where $\mathrm{bb}(K)$ stops to contain g, or when the center of K moves out of γ, is more difficult. For this we need a supporting data structure that can answer the following query:

> Given a cone γ with apex g, report the closest object to g whose center is in γ and whose bounding box contains g.

Recall that the set of cones can be partitioned into $O(\rho^2)$ subsets, where the cones in each subset are translates of some "standard" cone. We construct a data structure for each subset separately. Because the facets of the cones in a subset have only three distinct orientations, we can find all centers inside a query cone in with a three-level range tree. Finding the bounding boxes containing the apex of the query cone can be done with a three-level segment tree, and filtering out the closest object requires a sorted list on the orthogonal projections of the object centers onto the representative edge of the cone. Hence, our total data structure will be have seven levels. Answering a query can be done in $O(\log^6 n)$ time—the query time is not $O(\log^7 n)$ because in the last level we only need to report the closest object—and the amount of storage is $O(n \log^6 n)$. To kinetize the structure, we use the kinetic variants of range trees [24] and segment trees [39] and sorted lists (which are trivial to maintain). The number of events processed to maintain our seven-level structure is $O(n^2)$ and each event can be handled in $O(\log^7 n)$ time.

Lemma 5.11 *When a center or box event occurs, we can update the closest object $\mathcal{K}^*(\gamma)$ in $O(\log^6 n)$ time, using a supporting KDS that uses $O(\rho^2 n \log^6 n)$ storage. The supporting KDS processes $O(\rho^2 n^2)$ events in the worst case, assuming the objects follow constant-degree algebraic trajectories, and the response time is $O(\log^7 n)$.*

This leads to our main result.

Theorem 5.12 *For any set $\mathcal{K}$ of n convex, constant-complexity ρ-fat objects, there is a KDS for collision detection that uses $O(\rho^2 n \log^6 n + \rho^7 n)$ storage and that processes $O(\rho^{13} n^2)$ events in the worst case, assuming the objects follow constant-degree algebraic trajectories. Each event can be handled in $O(\log \rho + \log^7 n)$ time.*

Our KDS is compact and responsive, but unfortunately it is not local: a large object K with many small objects around can be involved in many certificates, because it may contain guards for each of the small objects. However, we can show that the locality of

our KDS depends on the ratio of the size of the biggest object and the smallest object in $\mathcal{K}$.

Theorem 5.13 *Each object in the KDS of Theorem 5.12 is involved in $O(\rho^8 + \rho^3\sigma^3)$ certificates, where σ is the ratio of the largest inner radius to the smallest inner radius of the objects in $\mathcal{K}$.*

Proof. Consider an object K. There are two kinds of certificates in which K is involved:

Order certificates: these certificates arise from the sorted lists which we maintain.

Collision certificates: these certificates certify disjointness for all candidate pairs that include K.

Since the number of sorted lists in the KDS is $O(\rho)$ and there are $O(\rho^6)$ guards for each object, the number of order certificates involving K is $O(\rho^7)$. It remains to count the number of collision certificates. The number of such certificates involving K and a larger object K' is $O(\rho^8)$, because the guarding set of K has $O(\rho^6)$ size and for each guarding point we maintain $O(\rho^2)$ objects—one per cone defined for the guard. Now we have to count the number of objects K' smaller than K such that $\mathrm{bb}(K)$ contains at least one guard of K'. Since the volume of K is less than $O(\rho^3\sigma^3)$ times the volume of K', a simple packing argument shows that the number of such objects K' is $O(\rho^3\sigma^3)$—note that if $\mathrm{bb}(K)$ contain more than one guarding point of K', we just maintain one collision certificate between K and K'. Therefore, the total number of certificates involving K is $O(\rho^8 + \rho^3\sigma^3)$. □

5.4 Conclusions

We presented the first KDSs for collision detection between multiple convex fat 3D objects that use a near-linear number of certificates and do not require the objects to have similar sizes. We believe that this is an important step forward in the theoretical investigation of KDSs for 3D collision detection. Our KDS for balls rolling on a plane is simple, and may perform well in practice. Our general KDS for free-flying objects of varying sizes, however, is complicated and the dependency on the fatness parameter ρ is large. Thus our result should be seen as a proof that good bounds are possible in theory—whether a simple and practical solution exists that achieves similar worst-case bounds is still open.

As remarked above, our structures are not local: a single object can be involved in a linear number of certificates. Unfortunately, this seems very hard (if not impossible) to avoid if there is a single large object that is closely surrounded by many tiny objects. Thus we do not expect to see a local KDS that can deal with arbitrarily sized objects. (We have shown though that a local KDS is possible for convex fat objects when their sizes are similar.)

Chapter 6

Streaming algorithms for line simplification

Abstract. We study the following variant of the well-known line-simplification problem: we are getting a possibly infinite sequence of points $p_0, p_1, p_2, \ldots$ in the plane defining a polygonal path, and as we receive the points we wish to maintain a simplification of the path seen so far. We study this problem in a streaming setting, where we only have a limited amount of storage so that we cannot store all the points. We analyze the competitive ratio of our algorithms, allowing resource augmentation: we let our algorithm maintain a simplification with $2k$ (internal) points, and compare the error of our simplification to the error of the optimal simplification with k points. We obtain the algorithms with $O(1)$ competitive ratio for three cases: convex paths where the error is measured using the Hausdorff distance (or Fréchet distance), xy-monotone paths where the error is measured using the Hausdorff distance (or Fréchet distance), and general paths where the error is measured using the Fréchet distance. In the first case the algorithm needs $O(k)$ additional storage, and in the latter two cases the algorithm needs $O(k^2)$ additional storage.

An extended abstract of this chapter was previously published as: M. A. Abam and M. de Berg, P. Hachenberger, and A. Zarei, Streaming algorithms for line simplification, In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 175–183, 2007.

6.1 Introduction

Motivation. Suppose we are tracking one, or maybe many, moving objects. Each object is equipped with a device that is continuously transmitting its position. Thus we are receiving a stream of data points that describes the path along which the object moves. The goal is to maintain this path for each object. We are interested in the scenario where we are tracking the objects over a very long period of time, as happens for instance when studying the migratory patterns of animals. In this situation it may be undesirable or even impossible to store the complete stream of data points. Instead we have to maintain an approximation of the input path. This leads us to the following problem: we are receiving a (possibly infinite) stream $p_0, p_1, p_2, \ldots$ of points in the plane, and we wish to maintain a simplification (of the part of the path seen so far) that is as close to the original path as possible, while using not more than a given (fixed) amount of available storage.

Related work. The problem described above is a streaming version of line simplification, one of the basic problems in GIS. In a line simplification problem one is given a polygonal path $P := p_0, p_1, \ldots, p_n$ in the plane, and the goal is to find a path $Q := q_0, q_1, \ldots, q_k$ with fewer vertices that approximates the path P well. In fact, this problem arises whenever we want to perform data reduction on a polygonal shape in the plane, and so it plays a role not only in GIS but also in areas like image processing and computer graphics. Line simplification has been studied extensively both in these application areas as well as in computational geometry. We study line simplification in a streaming setting, where we only have a limited amount of storage so that we cannot store all the points. A similar streaming model for geometric algorithms has been used by e.g. Agarwal and Yu [19], and Zarrabi-Zadeh and Chan [103].

The line-simplification problem has many variants. For example, we can require the sequence of vertices of Q to be a subsequence of P (with $q_0 = p_0$ and $q_k = p_n$)—this is sometimes called the *restricted version*—or we can allow arbitrary points as vertices. Here, as in most other papers, we consider the restricted version, and we limit our discussion to this version from now on; some results on the unrestricted version can be found in [53, 56, 62]. In the restricted version, each link $q_l q_{l+1}$ of the simplification corresponds to a shortcut $p_i p_j$ (with $j > i$) of the original path, and the error of the link is defined as the distance between $p_i p_j$ and the subpath $p_i, \ldots, p_j$. To measure the distance between $p_i p_j$ and $p_i, \ldots, p_j$ one often uses the Hausdorff distance, but the Fréchet distance can be used as well—see below for definitions. The error of the simplification Q is now defined as the maximum error of any of its links. Once the error measure has been defined, we can consider two types of optimization problems: the min-k and the min-δ problem. In the min-k problem, one is given the path P and a maximum error δ, and the goal is to find a simplification Q with as few vertices as possible whose error is at most δ. In the min-δ problem, one is given the path P and a maximum number of vertices k, and the goal is to find a simplification with the smallest possible error that uses at most k vertices.

The oldest and most popular algorithm for line simplification under the Hausdorff distance

is the Douglas-Peucker algorithm [44]. A basic implementation of this algorithm runs in $O(n^2)$ time, but more careful implementations run in $O(n \log n)$ time [65] or even $O(n \log^* n)$ time [66]. However, the Douglas-Peucker algorithm is only a heuristic and it is not guaranteed to be optimal (in terms of the number of vertices used, or the error of the resulting simplification). Imai and Iri [68] showed how to solve both versions of the problem optimally in $O(n^2 \log n)$ time by modeling it as a shortest-path problem on directed acyclic graphs. The running time of their method was improved to quadratic or near quadratic by Chin and Chan [30], and Melkman and O'Rouke [86]. Finally, Agarwal and Varadarajan [17] improved the running time to $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$, for the L_1-metric and the so-called uniform metric—here $d(p,q) = |p_y - q_y|$ if $p_x = q_x$ and $d(p,q) = \infty$ otherwise—by implicitly representing the graph.

The line-simplification problem was first studied for the Fréchet distance by Godau [52]. Alt and Godau [20] proposed an algorithm to compute the Fréchet distance between two polygonal paths in quadratic time; combined with the approach of Imai and Iri [68] this can be used to compute an optimal solution to the min-δ or the min-k problem for the Fréchet distance.

Since solving the line-simplification problem exactly is costly—the best known algorithm for the Hausdorff distance (under the L_2 metric) and for the Fréchet distance take quadratic time or more—Agarwal *et al.* [15] consider approximation algorithms. In particular, they consider the min-k problem for both the Hausdorff distance for x-monotone paths (in the plane) and the Fréchet distance for general paths (in d-dimensional space). They give near-linear time algorithms that compute a simplification whose error is at most δ and whose number of vertices is at most the minimum number of vertices of a simplification of error at most $\delta/2$. Their algorithms are greedy and iterative. Because the algorithms are iterative they can be used in an on-line setting, where the points are given one by one and the simplification must be updated at each step. However, since they solve the min-k problem, they cannot be used in a streaming setting, because the complexity of the produced simplification for an input path of n points can be $\Theta(n)$. (Note that an iterative greedy approach can be used in the min-k problem—try to go as far as possible with each link, while staying within the error bound δ—but that for the min-δ problem this does not work.) Moreover, their algorithm for the Hausdorff distance does not work when the normal Euclidean distance is used in the definition of Hausdorff distance, but only when the uniform distance is used. [1] The other existing algorithms for line simplification cannot be used in a streaming setting either.

Definitions, notation, and problem statement. To be able to state the problem we wish to solve and the results we obtain more precisely, we first introduce some terminology and definitions. Let $p_0, p_1, \ldots$ be the given stream of input points. We use $P(n)$ to denote

[1] The technical problem is the following. Consider the shortcut $p_i p_j$. For each vertex p_l, with $i < l < j$, place a disk D_l centered at p_l and of radius δ. Then they claim that $p_i p_j$ has an error of at most δ if and only if $p_i p_j$ intersects the disks $D_{i+1}, \ldots, D_{j-1}$ *in order*. This is incorrect, as $p_i p_j$ has error at most δ even if it intersects the disks in a different order, and such a situation can arise even if the input path is monotone. Thus the greedy iterative approach they use does not work.

the path defined by the points $p_0, p_1, \ldots, p_n$—that is, the path connecting those points in order—and for any two points p, q on the path we use $P(p, q)$ to denote the subpath from p to q. For two vertices p_i, p_j we use $P(i, j)$ as a shorthand for $P(p_i, p_j)$. A segment $p_i p_j$ with $i < j$ is called a *link* or sometimes a *shortcut*. Thus $P(n)$ consists of the links $p_{i-1}p_i$ for $0 < i \leqslant n$. We assume a function $error$ is given that assigns a non-negative error to each link $p_i p_j$. An *ℓ-simplification* of $P(n)$ is a polygonal path $Q := q_0, q_1, \ldots, q_k, q_{k+1}$ where $k \leqslant \ell$ and $q_0 = p_0$ and $q_{k+1} = p_n$, and $q_1, \ldots, q_k$ is a subsequence of $p_1, \ldots, p_{n-1}$. The error of a simplification Q for a given function $error$, denoted $error(Q)$, is defined as the maximum error of any of its links. We will consider two specific error functions for our simplifications, one based on the Hausdorff distance, and one based on the Fréchet distance, as defined next. For two objects o_1 and o_2, we use $d(o_1, o_2)$ to denote the Euclidean distance between o_1 and o_2. (For two points p_i and p_j, we sometimes also use $|p_i p_j|$ to denote the Euclidean distance between p_i and p_j, which is equal to the length of the segment $p_i p_j$.)

- In the Hausdorff error function $error_{\mathrm{H}}$, the error of the link $p_i p_j$ is $d_H(p_i p_j, P(i, j))$, the Hausdorff distance of the subpath $P(i, j)$ to the segment $p_i p_j$:

$$error_{\mathrm{H}}(p_i p_j) := d_H(p_i p_j, P(i, j)),$$

  where $d_H(p_i p_j, P(i, j)) = \max_{i<l<j} d(p_l, p_i p_j)$.

- The Fréchet distance between two paths A and B, which we denote by $d_F(A, B)$, is defined as follows. Consider a man with a dog on a leash, with the man standing at the start point of A and the dog standing at the start point of B. Imagine that the man walks to the end of A and the dog walks to the end of B. During the walk they can stop every now and then, but they are not allowed to go back along their paths. Now the Fréchet distance between A and B is the minimum length of the leash needed for this walk, over all possible such walks. More formally, $d_F(A, B)$ is defined as follows. Let A and B be specified by functions $A : [0, 1] \to \mathbb{R}^2$ and $B : [0, 1] \to \mathbb{R}^2$. Any non-decreasing continuous function $\alpha : [0, 1] \to [0, 1]$ with $\alpha(0) = 0$ and $\alpha(1) = 1$ defines a re-parametrization A_α of A by setting $A_\alpha(t) = A(\alpha(t))$. Similarly, any non-decreasing continuous function $\beta : [0, 1] \to [0, 1]$ with $\beta(0) = 0$ and $\beta(1) = 1$ defines a re-parametrization B_β of B. The Fréchet distance $d_F(A, B)$ between two paths A and B is now defined as

$$d_F(A, B) := \inf_{\alpha, \beta} \max_{0 \leqslant t \leqslant 1} d(A_\alpha(t), B_\beta(t))$$

  where the infimum is taken over all re-parametrizations A_α of A and B_β of B. In the Fréchet error function $error_{\mathrm{F}}$, the error of the link $p_i p_j$ is the Fréchet distance of the subpath $P(i, j)$ to the segment $p_i p_j$:

$$error_{\mathrm{F}}(p_i p_j) := d_F(P(i, j), p_i p_j).$$

Now consider an algorithm $\mathcal{A} := \mathcal{A}(\ell)$ that maintains an ℓ-simplification for the input stream $p_0, p_1, \ldots$, for some given ℓ. Let $Q_{\mathcal{A}}(n)$ denote the simplification that $\mathcal{A}$ produces for the path $P(n)$. Let $Opt(\ell)$ denote an optimal off-line algorithm that produces

an ℓ-simplification. Thus $error(Q_{Opt(\ell)}(n))$ is the minimum possible error of any ℓ-simplification of $P(n)$. We define the quality of $\mathcal{A}$ using the *competitive ratio*, as is standard for on-line algorithms. We also allow *resource augmentation*. More precisely, we allow $\mathcal{A}$ to use a $2k$-simplification, but we compare the error of this simplification to $Q_{Opt(k)}(n)$. (This is similar to Agarwal *et al.* [15] who compare the quality of their solution to the min-k problem for a given maximum error δ to the optimal value for maximum error $\delta/2$.) Thus we define the competitive ratio of an algorithm $\mathcal{A}(2k)$ as

$$\text{competitive ratio of } \mathcal{A}(2k) := \max_{n \geqslant 0} \frac{error(Q_{\mathcal{A}(2k)}(n))}{error(Q_{Opt(k)}(n))},$$

where $\frac{error(Q_{\mathcal{A}(2k)}(n))}{error(Q_{Opt(k)}(n))}$ is defined as 1 if $error(Q_{\mathcal{A}(2k)}(n)) = error(Q_{Opt(k)}(n)) = 0$. We say that an algorithm is *c-competitive* if its competitive ratio is at most c.

Our results. We present and analyze a simple general streaming algorithm for line simplification. Our analysis shows that the algorithm has good competitive ratio under two conditions: the error function that is used is *monotone*—see Section 6.2 for a definition—and there is an oracle that can approximate the error of any candidate link considered by the algorithm. We then continue to show that the Hausdorff error function is monotone on convex paths and on xy-monotone paths. (It is not monotone on general paths.) The Fréchet error function is monotone on general paths. Finally, we show how to implement the error oracles for these three settings. Putting everything together leads to the following results.

(i) For convex paths and the Hausdorff error function (or the Fréchet error function) we obtain a 3-competitive streaming algorithm using $O(k)$ additional storage that processes an input point in $O(\log k)$ time.

(ii) For xy-monotone paths and the Hausdorff error function (or the Fréchet error function) we can, for any fixed $\varepsilon > 0$, obtain a $(4 + \varepsilon)$-competitive streaming algorithm that uses $O(k^2/\sqrt{\varepsilon})$ additional storage and processes each input point in $O(k \log(1/\varepsilon))$ amortized time.

(iii) For general paths and the Fréchet error function we can, for any fixed $\varepsilon > 0$, obtain a $(4\sqrt{2}+\varepsilon)$-competitive streaming algorithm that uses $O(k^2/\sqrt{\varepsilon})$ additional storage and processes each input point in $O(k \log(1/\varepsilon))$ amortized time.

Finally, we give a negative result in Section 6.5. We show that for the Hausdorff error function it is not possible to have a streaming algorithm that maintains a path with less than $2k$ points whose competitive ratio (with respect to $Opt(k)$) is bounded, unless the algorithm uses $\Omega(n/k)$ additional storage.

6.2 A general algorithm

In this section we describe a general strategy for maintaining an ℓ-simplification of an input stream $p_0, p_1, \ldots$ of points in the plane, and we will show that it has a good competitive ratio under two conditions: the error function is *monotone* (as defined below), and we have an *error oracle* at our disposal that computes or approximates the error of a link. We denote the error computed by the oracle for a link p_ip_j by $error^*(p_ip_j)$. In later sections we will prove that the Hausdorff error metric is monotone on convex or xy-monotone paths and that the Fréchet error function is monotone on general paths, and we will show how to implement the oracles for these settings.

Our algorithm is quite simple. Suppose we have already handled the points $p_0, \ldots, p_n$. (We assume $n > \ell + 1$; until that moment we can simply use all points and have zero error.) Let $Q := q_0, q_1, \ldots, q_\ell, q_{\ell+1}$ be the current simplification. Our algorithm will maintain a priority queue $\mathcal{Q}$ that stores the points q_i with $1 \leqslant i \leqslant \ell$, where the priority of a point is the error (as computed by the oracle) of the link $q_{i-1}q_{i+1}$. In other words, the priority of q_i is (an approximation of) the error that is incurred when q_i is removed from the simplification. Now the next point p_{n+1} is handled as follows:

1. Set $q_{\ell+2} := p_{n+1}$, thus obtaining an $(\ell+1)$-simplification of $P(n+1)$.
2. Compute $error^*(q_\ell q_{\ell+2})$ and insert $q_{\ell+1}$ into $\mathcal{Q}$ with this error as priority.
3. Extract the point q_s with minimum priority from $\mathcal{Q}$; remove q_s from the simplification.
4. Update the priorities of q_{s-1} and q_{s+1} in $\mathcal{Q}$.

Next we analyze the competitive ratio of our algorithm.

We say that a link p_ip_j *encloses* a link p_lp_m if $i \leqslant l \leqslant m \leqslant j$, and we say that $error$ is a *c-monotone error function* for a path $P(n)$ if for any two links p_ip_j and p_lp_m such that p_ip_j encloses p_lp_m we have

$$error(p_lp_m) \leqslant c \cdot error(p_ip_j).$$

In other words, an error function is c-monotone if the error of a link cannot be worse than c times the error of any link that encloses it.

Furthermore, we denote an error oracle as an *e-approximate error oracle* if

$$error(p_ip_j) \leqslant error^*(p_ip_j) \leqslant e \cdot error(p_ip_j)$$

for any link p_ip_j for which the oracle is called by the algorithm above.

Theorem 6.1 *Suppose that we use a c-monotone error function and that we have an e-approximate error oracle at our disposal. Then the algorithm described above with $\ell = 2k$*

is ce-competitive with respect to $Opt(k)$. The time the algorithm needs to update the simplification Q upon the arrival of a new point is $O(\log k)$ plus the time spent by the error oracle. Besides the storage needed for the simplification Q, the algorithm uses $O(k)$ storage plus the storage needed by the error oracle.

Proof. Consider an arbitrary $n \geqslant 0$, and let $Q(n)$ denote the $2k$-simplification produced by our algorithm. Since the error of $Q(n)$ is the maximum error of any of its links, we just need to show that $error(\sigma) \leqslant c \cdot e \cdot error(Q_{Opt(k)}(n))$ for any link σ in $Q(n)$. Let $m \leqslant n$ be such that σ appears in the simplification when we receive point p_m. If $m \leqslant 2k+2$, then $error(\sigma) = 0$ and we are done. Otherwise, let $Q(m-1) := q_0, \ldots, q_{2k+1}$ be the $2k$-simplification of $P(m-1)$. Upon the arrival of $p_m = q_{2k+2}$ we insert $q_{2k+1} = p_{m-1}$ into $\mathcal{Q}$. A simple counting argument shows that at least one of the shortcuts $q_{t-1}q_{t+1}$ for $1 \leqslant t \leqslant 2k+1$, let's call it σ', must be enclosed by one of the at most $k+1$ links in $Q_{Opt(k)}(n)$. Since σ is the link with the smallest priority among all links in $\mathcal{Q}$ at that time, its approximated error is smaller than that of σ'. Therefore,

$$\begin{aligned} error(Q_{Opt(k)}(n)) &\geqslant \tfrac{1}{c}\, error(\sigma') &\geqslant \tfrac{1}{c \cdot e}\, error^*(\sigma') \\ &\geqslant \tfrac{1}{c \cdot e}\, error^*(\sigma) &\geqslant \tfrac{1}{c \cdot e}\, error(\sigma). \end{aligned}$$

We conclude that our algorithm is ce-competitive with respect to $Opt(k)$.

Besides the time and storage needed by the error oracle, the algorithm only needs $O(k)$ space to store the priority queue and $O(\log k)$ for each update of the priority queue. □

6.3 The Hausdorff error function

The algorithm presented above has good competitive ratio if the error function being used is monotone and can be approximated well. In this section we show that these properties hold for the Hausdorff error function on convex and xy-monotone paths. (A path is convex if by connecting the last point to the first point on the path we obtain a convex polygon. A path is xy-monotone if any horizontal or vertical line intersects it in at most one point.) Note that for these two case the Hausdorff distance between a link p_ip_j and the subpath $P(i,j)$ is identical to the Frechet distance between them. Thus the results from this section hold for the Frechet distance as well. They improve on the result that will be given in the next section for the Fréchet distance on general curves. During this section all results stated for the Hausdorff distance also hold for the Fréchet distance.

The following lemma gives results on the monotonicity of various types of paths under the Hausdorff error function.

Lemma 6.2 *The Hausdorff error function is 1-monotone on convex paths and 2-monotone on xy-monotone paths. Moreover, there is no constant c such that the Hausdorff error function is c-monotone on y-monotone paths.*

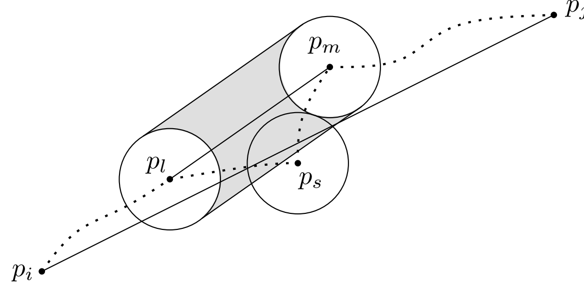


Figure 6.1 The Hausdorff error function is 2-monotone on any xy-monotone path.

Proof. It is easy to see that the Hausdorff error function is 1-monotone on convex paths. It is also not difficult, given any constant c, to give an example of an y-monotone path such that the Hausdorff error function is not c-monotone—a zigzag with four vertices such that the first and third are very close together and the second and fourth are very close together will do.

So now consider an xy-monotone path $p_0, \dots, p_n$. Let $p_i p_j$ and $p_l p_m$ be two links such that $p_i p_j$ encloses $p_l p_m$, and let p_s be a point on the subpath $P(l, m)$ such that $d(p_s, p_l p_m) = error_{\mathrm{H}}(p_l p_m)$. Consider the circles C_l, C_m and C_s of radius $error_{\mathrm{H}}(p_i p_j)$ centered at points p_l, p_m and p_s—see Figure 6.1. Since the distance of the link $p_i p_j$ to the points p_l, p_s, and p_m is at most $error_{\mathrm{H}}(p_i p_j)$, it must intersect these circles. Let p'_s, p'_l, and p'_m be the orthogonal projections of p_s, p_l, and p_m onto the link $p_i p_j$. Clearly, p'_s, p'_l and p'_m are inside C_s, C_l and C_m, respectively. Since $P(i, j)$ is xy-monotone, p'_s lies between p'_l and p'_m, which implies

$$d(p'_s, p_l p_m) \leqslant \max(d(p'_l, p_l), d(p'_m, p_m)) \leqslant error_{\mathrm{H}}(p_i p_j).$$

Therefore,

$$\begin{aligned} error_{\mathrm{H}}(p_l p_m) &= error_{\mathrm{H}}(p_s, p_l p_m) \\ &\leqslant d(p_s, p'_s) + d(p'_s, p_l p_m) \\ &\leqslant 2\, error_{\mathrm{H}}(p_i p_j). \end{aligned}$$

Note that the link $p_i p_j$ can be tangent to C_s, C_l, and C_m, which shows that the monotonicity factor 2 is tight. □

The next step is to implement the error oracles for convex paths and for xy-monotone paths. We start with the case of convex paths.

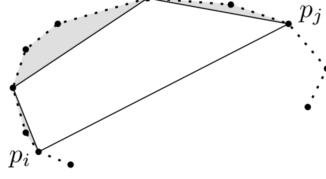


Figure 6.2 The areas maintained by the error oracle for convex paths.

6.3.1 The error oracle for convex paths

The idea of the error oracle is to maintain an approximation of the area enclosed by p_ip_j and the path $P(i,j)$ for each link p_ip_j. Let $area(i,j)$ denote this area. If the two angles $\angle p_{i+1}p_ip_j$ and $\angle p_{j-1}p_jp_i$ are at most 90 degrees, we can deduce an approximation of $error_{\mathrm{H}}(p_ip_j)$ from $area(i,j)$ and $|p_ip_j|$. Indeed, if $d_H(p_ip_j, P(i,j)) = d$, then the maximum area enclosed by p_ip_j and $P(i,j)$ is achieved by a rectangle with base p_ip_j and height d, and the minimum area is achieved by a triangle with base p_ip_j and height d. Hence,

$$\begin{aligned} d_H(p_ip_j, P(i,j)) &\leqslant 2 \cdot area(i,j)/|p_ip_j| \\ &\leqslant 2 \cdot d_H(p_ip_j, P(i,j)), \end{aligned}$$

and so $2 \cdot area(i,j)/|p_ip_j|$ can be used as a 2-approximate error oracle. Unfortunately this approach does not work if $\angle p_{i+1}p_ip_j$ and/or $\angle p_{j-1}p_jp_i$ are bigger than 90 degrees. We therefore proceed as follows.

For each shortcut p_ip_j used in the current approximation, partition the path $P(i,j)$ into at most five pieces by splitting it at each vertex that is extreme in x- or y-direction. (If, say, there is more than one leftmost vertex on the path, we cut at the first such vertex.) The information we maintain for p_ip_j is the set of cut points as well as area enclosed by each such piece $P(l,m)$ and the corresponding shortcut p_lp_m—see Figure 6.2. Notice that if $P(i,j)$ does not contain an extreme point we simply maintain $area(i,j)$, as before.

As $d_H(p_ip_j, P(i,j))$ is the maximum of $d_H(p_ip_j, P(l,m))$ over all pieces $P(l,m)$ into which $P(i,j)$ is cut, it is sufficient to approximate $d_H(p_ip_j, P(l,m))$ for each piece. Note that both $\angle p_{l+1}p_lp_m$ and $\angle p_{m-1}p_mp_l$ are at most 90 degrees. We approximate $d_H(p_ip_j, P(l,m))$ by

$$(2 \cdot area(l,m)/|p_lp_m|) + d_H(p_ip_j, p_lp_m).$$

We claim this gives us a 3-approximation. We have

$$\begin{aligned} d_H(p_ip_j, P(l,m)) &\leqslant d_H(p_lp_m, P(l,m)) + d_H(p_ip_j, p_lp_m) \\ &\leqslant \tfrac{2 \cdot area(l,m)}{|p_lp_m|} + d_H(p_ip_j, p_lp_m). \end{aligned}$$

On the other hand,

$$
\begin{aligned}
3 \cdot d_H(p_i p_j, P(l,m)) &\geqslant 3 \cdot \max(d_H(p_l p_m, P(l,m)), d_H(p_i p_j, p_l p_m)) \\
&\geqslant \tfrac{2 \cdot area(l,m)}{|p_l p_m|} + d_H(p_i p_j, p_l p_m),
\end{aligned}
$$

so $(2 \cdot area(l,m)/|p_l p_m|) + d_H(p_i p_j, p_l p_m)$ is a 3-approximation of $d_H(p_i p_j, P(l,m))$.

What remains is to show that we can maintain this information as more points are received and the simplification changes. First consider step 2 of the algorithm, where we need to compute $error^*(q_\ell q_{\ell+2})$. Since we have the information described above available for $q_\ell q_{\ell+1}$, and $q_{\ell+1}$ and $q_{\ell+2}$ are consecutive points of the original path P, we can compute the necessary information for $q_\ell q_{\ell+2}$ in $O(1)$ time. Similarly, in step 4 we can update the information in $O(1)$ time. We omit the easy details.

Lemma 6.3 *There is a 3-approximate error oracle for the Hausdorff error function on convex paths that uses $O(k)$ storage and can be updated in $O(1)$ time.*

Putting everything together we obtain the following theorem.

Theorem 6.4 *There is a streaming algorithm that maintains a $2k$-simplification for convex planar paths under the Hausdorff error function (or the Fréchet error function) and that is 3-competitive with respect to $Opt(k)$. The algorithm uses $O(k)$ additional storage and each point is processed in $O(\log k)$ time.*

6.3.2 The error oracle for xy-monotone paths

We use the notion of width for approximating $error_{\mathrm{H}}$ of an xy-monotone path. The *width* of a set of points with respect to a given direction $\overrightarrow{d}$ is the minimum distance of two lines being parallel to $\overrightarrow{d}$ that enclose the point set. Let $w(i,j)$ be the width of the points in subpath $P(i,j)$ with respect to the direction $\overrightarrow{p_i p_j}$. Since $P(i,j)$ is xy-monotone, it is contained inside the axis-parallel rectangle defined by p_i and p_j. Therefore,

$$
w(i,j)/2 \leqslant error_H(p_i p_j) \leqslant w(i,j)
$$

and $w(i,j)$ can be used as a 2-approximate error oracle for $error_{\mathrm{H}}(p_i p_j)$.

Agarwal and Yu [19] have described a streaming algorithm for maintaining a core-set that can be used to approximate the width of a set in any direction. More precisely, given a data stream $p_0, p_1, \ldots$, they maintain an ε-core-set of size $O(1/\sqrt{\varepsilon})$ in $O(\log(1/\varepsilon))$ amortized time per insertion. The width in a given direction can be efficiently computed from the core-set if we additionally maintain the convex hull of the core-set using the dynamic data structure by Brodal and Jacob [27]. This data structure uses linear space and can be updated in logarithmic time. Also it supports queries for the extreme point in a given direction in logarithmic time. Thus we can compute the extreme points that define the width in a given direction in $O(\log(1/\varepsilon))$ time. The core-set gives us an $(2+\varepsilon)$-approximate error oracle.

Lemma 6.5 *There is a $(2+\varepsilon)$-approximate error oracle for the Hausdorff error function on xy-monotone paths that uses $O(k^2/\sqrt{\varepsilon})$ storage and has $O(k \log(1/\varepsilon))$ amortized update time.*

Proof. Although our algorithm only needs the approximate errors of the links $q_{i-1}q_{i+1}$ to decide which point q_s is erased next, we must maintain a core-set for each link that might be needed at some later time in our simplification. These are the links q_iq_j, with $0 \leqslant i < j-1 < 2k+1$. So we need to maintain a core-set for each of these $O(k^2)$ links. Considering a new point $q_{2k+2} = p_{n+1}$, we must create $O(k)$ new core-sets, one for each of the links q_ip_{n+1}, with $0 \leqslant i \leqslant 2k$. We create such core-sets for the links q_ip_{n+1}, by copying the core-sets q_iq_{2k+1} and inserting point p_{n+1} to them using the algorithm by Agarwal and Yu. When some point q_s is removed from the simplification in Step 3 of our algorithm and the link $q_{s-1}q_{s+1}$ is added, the core-sets for all links that start or end at q_s have become meaningless and are therefore deleted.

In total, $O(k^2/\sqrt{\varepsilon})$ storage is needed for the $O(k^2)$ core-sets. The update of the oracle involves creation of $O(k)$ core-sets by duplicating current ones and therefore needs $O(k \log(1/\varepsilon))$ time. The new point is added to these core-sets in $O(k \log(1/\varepsilon))$ amortized time. Thus, the time the oracle needs to process a new point is $O(k \log(1/\varepsilon))$. □

Putting everything together we obtain the following theorem.

Theorem 6.6 *There is a streaming algorithm that maintains a $2k$-simplification for xy-monotone planar paths under the Hausdorff error function (or the Fréchet error function) and that is $(4+\varepsilon)$-competitive with respect to $Opt(k)$. The algorithm uses $O(k^2/\sqrt{\varepsilon})$ additional storage and each point is processed in $O(k \log(1/\varepsilon))$ amortized time.*

6.4 The Fréchet error function

We now turn our attention to the Fréchet error function. We will show that we can obtain an $O(1)$-competitive algorithm for arbitrary paths. The first property we need is that the Fréchet error function is monotone. This has in fact already been proven by Agarwal *et al.* [15].

Lemma 6.7 **[15]** *The Fréchet error function is 2-monotone on arbitrary paths.*

6.4.1 The error oracle

Next we turn our attention to the implementation of the error oracle for the Fréchet error function. We use two parameters to approximate $error_{\mathrm{F}}(p_ip_j)$. The first one is $w(i,j)$, the width of the points of $P(i,j)$ in the direction of p_ip_j, which we also used to approximate the Hausdorff error in the case of xy-monotone paths. The other parameter is

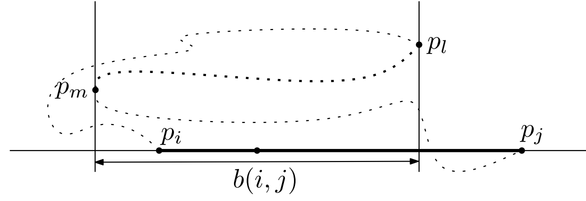


Figure 6.3 Relation between Fréchet distance and back-paths.

the length of the largest back-path in the direction of p_ip_j, which is defined as follows. Assume without loss of generality that p_ip_j is horizontal with p_j to the right of p_i. For two points p_l, p_m on the path $P(i,j)$ with $l < m$ we define $P(l,m)$ to be a *back-path on* $P(i,j)$ if $(p_m)_x < (p_l)_x$. In other words $P(l,m)$ is a back-path if, relative to the direction $\overrightarrow{p_ip_j}$, we go back when we move from p_l to p_m—see Figure 6.3. The *length* of a back-path $P(l,m)$ on $P(i,j)$ is defined to be the length of the projection of p_lp_m onto a line parallel to p_ip_j, which is equal to $(p_l)_x - (p_m)_x$ since we assumed p_ip_j is horizontal. We define $b(i,j)$ to be the maximum length of any back-path on $P(i,j)$.

Lemma 6.8 *The Fréchet error of a shortcut p_ip_j satisfies the following inequalities:*

$$\max(\frac{w(i,j)}{2}, \frac{b(i,j)}{2}) \leqslant \mathit{error}_{\mathrm{F}}(p_ip_j) \leqslant 2\sqrt{2}\max(\frac{w(i,j)}{2}, \frac{b(i,j)}{2})$$

Proof. As above we will without loss of generality assume that p_ip_j is horizontal with p_j to the right of p_i.

We observe that $\mathit{error}_{\mathrm{F}}(p_ip_j) \geqslant \mathit{error}_{\mathrm{H}}(p_ip_j) \geqslant w(i,j)/2$. Next we will show that $b(i,j)/2 \leqslant \mathit{error}_{\mathrm{F}}(p_ip_j)$. Consider a back-path $P(l,m)$ on $P(i,j)$ determining $b(i,j)$, as shown in Figure 6.3. Let r be the point on the line through p_ip_j midway between p_l and p_m, that is, the point on the line through p_ip_j with x-coordinate $((p_l)_x + (p_m)_x)/2$. Note that r does not necessarily lie on p_ip_j. The Fréchet distance between p_ip_j and $P(i,j)$ is determined by some optimal pair of parametrizations of p_ip_j and $P(i,j)$ that identifies each point p of $P(i,j)$ with a point $\overline{p}$ on p_ip_j in such a way that if p comes before q along $P(i,j)$ then $\overline{p}$ does not come later than $\overline{q}$ along p_ip_j. Now consider the images $\overline{p_l}$ and $\overline{p_m}$. If $\overline{p_l}$ lies to the left of r then $|p_l\overline{p_l}| > b(i,j)/2$. If, on the other hand, $\overline{p_l}$ lies on or to the right of r then $\overline{p_m}$ lies on or to the right of r as well, and we have $|p_m\overline{p_m}| > b(i,j)/2$. We conclude that $\max(w(i,j)/2, b(i,j)/2) \leqslant \mathit{error}_{\mathrm{F}}(p_ip_j)$, which proves the first part of the lemma.

For the second part we need to show that $\mathit{error}_{\mathrm{F}}(p_ip_j) \leqslant \sqrt{2}\max(w(i,j), b(i,j))$. It is convenient to think about the Fréchet distance in terms of the man-dog metaphor. In these terms, we have to find a walking schedule where the man walks along p_ip_j and the dog walks along $P(i,j)$ such that they never go back along their paths and their distance is never more than $\sqrt{2}\max(w(i,j), b(i,j))$. We can find such a walk as follows. Denote the position of the man by p_{man} and the position of the dog by p_{dog}. Initially $p_{\mathrm{man}} =$

$p_{\rm dog} = p_i$. Let ℓ be the vertical line through p_i. Of all the intersection points of ℓ with $P(i,j)$, let p be one furthest along $P(i,j)$. (If ℓ does not intersect $P(i,j)$ except at p_i, then $p = p_i$.) We let the dog walk along $P(p_i p)$, while the man waits at p_i. Let q be an arbitrary point on $P(p_i p)$. Then there must be points p_l, p_m with $l < m$ such that $(p_l)_x \geqslant (p_i)_x$ and $(p_m)_x \leqslant (q)_x$. Hence, we have $|(q)_x - (p_i)_x| \leqslant (p_l)_x - (p_m)_x \leqslant b(i,j)$. Furthermore, $|(q)_y - (p_i)_y| \leqslant w(i,j)$. Hence, during this first phase we have $|p_{\rm man} p_{\rm dog}| \leqslant \sqrt{2}\max(w(i,j), b(i,j))$.

We continue the walk as follows. Sweep ℓ to the right. Initially ℓ will intersect $P(pp_j)$ in only one point. As long as this is the case, we set $p_{\rm man} = \ell \cap p_i p_j$ and we set $p_{\rm dog} = \ell \cap P(pp_j)$. During this part we clearly have $|p_{\rm man} p_{\rm dog}| \leqslant w(i,j)$. At some point ℓ may intersect $P(p_{\rm dog}, p_j)$ in one (or more) point(s) other than $p_{\rm dog}$. When this happens we take the intersection point p that is furthest along $P(p_{\rm dog}, p_j)$, and let the dog proceed to p while the man waits at his current position. By the previous argument, $|p_{\rm man} p_{\rm dog}| \leqslant \sqrt{2}\max(w(i,j), b(i,j))$ during this phase. Then we continue to sweep ℓ to the right again, letting $p_{\rm man} = \ell \cap p_i p_j$ and $p_{\rm dog} = \ell \cap P(pp_j)$. The process ends when the sweep line reaches p_j. We have thus found a walking schedule with $|p_{\rm man} p_{\rm dog}| \leqslant \sqrt{2}\max(w(i,j), b(i,j))$ at all times, finishing the proof of the lemma. □

According to the above lemma, in order to approximate $error_{\rm F}(p_i p_j)$ it suffices to approximate $\max(w(i,j), b(i,j))$. In the previous section we already described how to approximate $w(i,j)$, when we were studying the Hausdorff error function for xy-monotone paths. Next we describe a method for approximating $b(i,j)$, and show how to combine these two methods to build the oracle for $error_{\rm F}(p_i p_j)$. (Note that if there are no back-paths, then the Fréchet error is equal to the Hausdorff error, so the case of xy-monotone paths for Hausdorff error is a special case of our current setting.)

In the algorithm as presented in Section 6.2 we need to maintain (an approximation of) the error of each shortcut $q_l q_{l+2}$ in the current simplification. For this we need to know the maximum length of a back-path on the path from q_l to q_{l+2}. The operations we must do are to add a point $q_{\ell+2} = p_{n+1}$ at the end of the simplification, and to remove a point q_s from the simplification. To this end we maintain the following information. For the moment let's assume that all we need is the maximum length of the back-path with respect to the positive x-direction. Then we maintain for each link $p_i p_j$ of the simplification the following values:

(i) $b(i,j)$, the maximum length of a back-path (w.r.t. the positive x-direction) on $P(i,j)$;

(ii) $xmax(i,j)$, the maximum x-coordinate of any point on $P(i,j)$;

(iii) $xmin(i,j)$, the minimum x-coordinate of any point on $P(i,j)$.

Now consider a shortcut $q_l q_{l+2}$. Let $q_l = p_i$, $q_{l+1} = p_t$ and $q_{l+2} = p_j$. Then $b(i,j)$, the maximum length of a back-path on $P(q_l, q_{l+2}) = P(i,j)$, is given by

$$\max\ (\ b(i,t),\ b(t,j),\ xmax(i,t) - xmin(t,j)\ ).$$

Adding a point $q_{\ell+2}$ is easy, because we only have to compute the above three values for $q_{\ell+1}q_{\ell+2}$, which is trivial since $q_{\ell+1}$ and $q_{\ell+2}$ are consecutive points on the original path. Removing a point q_s can also be done in $O(1)$ time (let $q_{s-1} = p_i$ and $q_{s+1} = p_j$): above we have shown how to compute $b(i,j)$ from the available information for $q_{s-1}q_s$ and $q_s q_{s+1}$, and computing $xmax(i,j)$ and $xmin(i,j)$ is even easier.

Thus we can maintain the maximum length of a back-path. There is one catch, however: the procedure given above maintains the maximum length of a back-path *with respect to a fixed direction* (the positive x-direction). But in fact we need to know for each $q_i q_{i+2}$ the maximum length of a back-path with respect to the direction $\overrightarrow{q_i q_{i+2}}$. These directions are different for each of the links and, moreover, we do not know them in advance. To overcome this problem we define $2\pi/\alpha$ equally spaced canonical directions, for a suitable $\alpha > 0$, and we maintain, for every link $p_i p_j$, the information described above for each direction. Now suppose we need to know the maximum length of a back-path for $p_i p_j$ with respect to the direction $\overrightarrow{p_i p_j}$. Then we will use $b_{\overrightarrow{d}}(p_i p_j)$, the maximum length of a back-path with respect to $\overrightarrow{d}$ instead, where $\overrightarrow{d}$ is the canonical direction closest to $\overrightarrow{p_i p_j}$ in clockwise order. In general, using $\overrightarrow{d}$ may not give a good approximation of the maximum length of a back-path in direction $\overrightarrow{p_i p_j}$, even when α is small. However, the approximation is only bad when $w(i,j)$ is relatively large, which means that the Fréchet distance can still be approximated well. This is made precise in the following lemmas.

Lemma 6.9 *Let w be the width of $P(i,j)$ in direction $\overrightarrow{p_i p_j}$, let b be the maximum length of a back-path on $P(i,j)$ in direction $\overrightarrow{p_i p_j}$, and let b^* be the maximum length of a back-path on $P(i,j)$ in direction $\overrightarrow{d}$, where $\overrightarrow{d}$ is the canonical direction closest to $\overrightarrow{p_i p_j}$ in clockwise order. Then we have:* $b^* - \tan(\alpha) \cdot w \ \leqslant\ b \ \leqslant\ b^* + \tan(\alpha) \cdot (b^* + w)$.

Proof. We first show that $b \leqslant b^* + \tan(\alpha) \cdot (b^* + w)$. Let the sub-path $P(l,m)$ have the maximum back-path length in the direction $\overrightarrow{p_i p_j}$. Consider two half-lines originating from p_m and being parallel to $\overrightarrow{p_i p_j}$ and $\overrightarrow{d}$. Let β denote the angle between these two half-lines. Because $\overrightarrow{d}$ is the canonical direction closest to $\overrightarrow{p_i p_j}$ in clockwise order, clearly $\beta \leqslant \alpha$. Let p and q be the orthogonal projections of p_l onto the lines through p_m in direction $\overrightarrow{p_i p_j}$ and $\overrightarrow{d}$, respectively. We distinguish four cases, depending on the relation of direction $\overrightarrow{p_m p_l}$ to $\overrightarrow{p_i p_j}$ and $\overrightarrow{d}$. The direction $\overrightarrow{p_m p_l}$ can be counterclockwise to $\overrightarrow{p_i p_j}$, between $\overrightarrow{p_i p_j}$ and $\overrightarrow{d}$, or clockwise to $\overrightarrow{d}$. If $\overrightarrow{p_m p_l}$ is counterclockwise to $\overrightarrow{p_i p_j}$, we also distinguish whether the angle between $\overrightarrow{p_m p_l}$ and $\overrightarrow{p_i p_j}$ is less or more than $90 - \beta$ degrees. Note, that since $p_l p_m$ is a back-path, the angle between $\overrightarrow{p_m p_l}$ and $\overrightarrow{p_i p_j}$ cannot be larger than 90 degrees. All four cases are illustrated in Figure 6.4. The corresponding proof is as follows.

(a) $\overrightarrow{p_m p_l}$ is between 90 and $90 - \beta$ degrees counterclockwise to $\overrightarrow{p_i p_j}$.

$$
\begin{aligned}
b = |p_m q| &\leqslant |p_l q| \tan(\beta) \\
&\leqslant w \tan(\alpha) \\
&\leqslant b^* + \tan(\alpha) \cdot (b^* + w)
\end{aligned}
$$

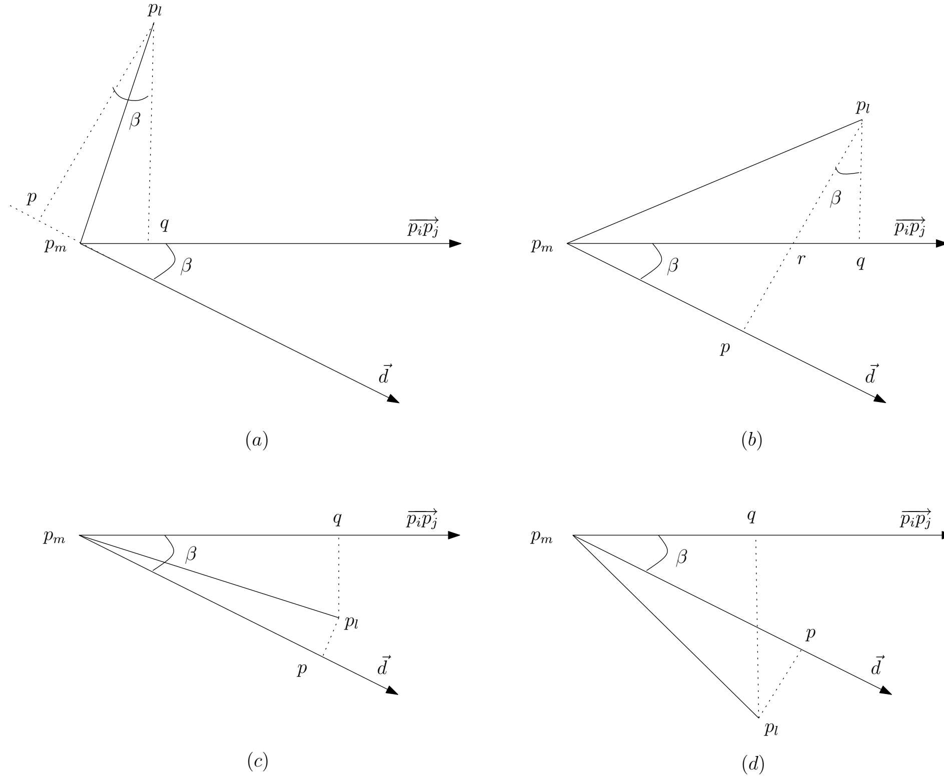


Figure 6.4 Illustration for the proof of Lemma 6.9.

(b) $\overrightarrow{p_m p_l}$ is less than $90 - \beta$ degrees counterclockwise to $\overrightarrow{p_i p_j}$.

$$\begin{aligned} b = |p_m q| &\leqslant |p_m p| + |pr| + |rq| \\ &\leqslant |p_m p| + |p_m p| \tan(\beta) + |p_l q| \tan(\beta) \\ &\leqslant b^* + \tan(\alpha) \cdot (b^* + w) \end{aligned}$$

(c) $\overrightarrow{p_m p_l}$ is between $\overrightarrow{p_i p_j}$ and $\overrightarrow{d}$.

$$\begin{aligned} b = |p_m q| &\leqslant |p_m p_l| \\ &\leqslant |p_m p| + |p p_l| \\ &\leqslant |p_m p| + |p_m p| \tan(\beta) \\ &\leqslant b^* + \tan(\alpha) \cdot (b^* + w) \end{aligned}$$

(d) $\overrightarrow{p_m p_l}$ is clockwise to $\overrightarrow{d}$ by at most $90 - \beta$ degrees.

$$\begin{aligned} b = |p_m q| &\leqslant |p_m p| \\ &\leqslant b^* \\ &\leqslant b^* + \tan(\alpha)\cdot(b^* + w) \end{aligned}$$

The same elementary arguments can be used to show that $b^* - \tan(\alpha)\cdot w \leqslant b$. □

The final oracle is now defined as follows. Let w^* be the approximation of the width of $P(i,j)$ in direction $\overrightarrow{p_i p_j}$ as given by Agarwal and Yu's ε-core-set method, and let b^* be the maximum length of a back-path on $P(i,j)$ in direction $\overrightarrow{d}$, where $\overrightarrow{d}$ is the canonical direction closest to $\overrightarrow{p_i p_j}$ in clockwise order. Then we set

$$error_{\mathrm{F}}{}^*(p_i p_j) \;:=\; \sqrt{2}\cdot\max(w^*, b^* + \tan(\alpha)\cdot(b^* + w^*)).$$

Combing Lemma 6.8 with the observations above, we can prove the following lemma.

Lemma 6.10 $error_{\mathrm{F}}(p_i p_j) \leqslant error_{\mathrm{F}}{}^*(p_i p_j) \leqslant 2\sqrt{2}(1+\varepsilon)(1+4\tan(\alpha))\cdot error_{\mathrm{F}}(p_i p_j)$

Proof. Let w be the width of $P(i,j)$ in direction $\overrightarrow{p_i p_j}$, let b be the maximum length of a back-path on $P(i,j)$ in direction $\overrightarrow{p_i p_j}$. Because w^* is the width of an ε-core-set, we have $w \leqslant w^* \leqslant (1+\varepsilon)w$. Using Lemma 6.8 we get

$$\begin{aligned} error_{\mathrm{F}}(p_i p_j) &\leqslant 2\sqrt{2}\cdot\max(\tfrac{w}{2},\tfrac{b}{2}) \\ &\leqslant \sqrt{2}\cdot\max(w^*, b^* + \tan(\alpha)\cdot(b^* + w)) \\ &\leqslant \sqrt{2}\cdot\max(w^*, b^* + \tan(\alpha)\cdot(b^* + w^*)) \\ &= error_{\mathrm{F}}{}^*(p_i p_j). \end{aligned}$$

On the other hand

$$\begin{aligned} error_{\mathrm{F}}{}^*(p_i p_j) &= \sqrt{2}\cdot\max(w^*, b^* + \tan(\alpha)\cdot(b^* + w^*)) \\ &\leqslant \sqrt{2}\cdot\max((1+\varepsilon)w, b + \tan(\alpha)w + \\ &\ \tan(\alpha)\cdot(b + \tan(\alpha)w + (1+\varepsilon)w) \\ &\leqslant \sqrt{2}(1+\varepsilon)\cdot\max(w, b \;+\; b\tan(\alpha) + 3w\tan(\alpha)) \\ &\leqslant \sqrt{2}(1+\varepsilon)(1+4\tan(\alpha))\cdot\max(w,b) \\ &\leqslant 2\sqrt{2}(1+\varepsilon)(1+4\tan(\alpha))\cdot\max(\tfrac{w}{2},\tfrac{b}{2}) \\ &\leqslant 2\sqrt{2}(1+\varepsilon)(1+4\tan(\alpha))\cdot error_{\mathrm{F}}(p_i p_j) \end{aligned}$$

□

Taking ε and α sufficiently small, we get our final result.

Theorem 6.11 *There is a streaming algorithm that maintains a $2k$-simplification for general planar paths under the Fréchet error function and that is $(4\sqrt{2}+\varepsilon)$-competitive with respect to $Opt(k)$. The algorithm uses $O(k^2/\sqrt{\varepsilon})$ additional storage and each point is processed in $O(k\log(1/\varepsilon))$ amortized time.*

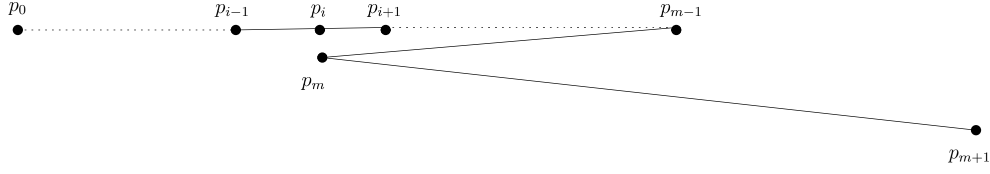


Figure 6.5 Base path component for Theorem 6.12.

6.5 The Hausdorff error function for general paths

In this section we show that for the Hausdorff error function it is not possible to have a streaming algorithm that maintains a path with less than $2k$ points whose competitive ratio (with respect to $Opt(k)$) is bounded, unless the algorithm uses $\Omega(n/k)$ additional storage. In fact, this even holds when the input path is known to be y-monotone.

Theorem 6.12 *Let $\mathcal{A}$ be a streaming algorithm that maintains a $(2k-1)$-simplification for a path $P(n)$, and that is able to store at most $m-1$ of the input points, where $2k+2 \leqslant m \leqslant n/k$. For any $c > 0$ and $n \geqslant km+1$, there is a y-monotone path $p_0, p_1, \ldots, p_n$ such that $error_{\mathrm{H}}(Q_{\mathcal{A}(2k-1)}(n)) > c \cdot error_{\mathrm{H}}(Q_{Opt(k)}(n))$.*

Proof. Figure 6.5 shows the basic component of the path having the following properties.

(i) Points $p_0, \cdots, p_{m-1}$ are collinear,

(ii) $|p_i p_{i+1}| > c \cdot d_H(p_{m-1}, p_i p_{m+1})$ for all $0 \leqslant i \leqslant m-2$,

(iii) $d_H(p_{m-1}, p_{i-1} p_{m+1}) > c \cdot d_H(p_{m-1}, p_i p_{m+1})$ for all $1 \leqslant i \leqslant m-1$

To obtain a configuration with the above properties, we take a horizontal line ℓ and a point p_{m-1} on ℓ. We put p_{m+1} below ℓ and arbitrarily far from p_{m-1} to the right of p_{m-1} such that its distance to ℓ is greater than $(c+\varepsilon)^{m-1}$ where $\varepsilon > 0$ is an arbitrarily small number. We put p_i ($i = 0, \cdots, m-2$) on ℓ to the left of p_{m-1} such that $p_i p_{m+1}$ is tangent to the circle whose radius is $(c+\varepsilon)^{m-i-1}$ and whose center is p_{m-1}. The point p_i always exists, because $d_H(p_{m+1}, \ell) > (c+\varepsilon)^{m-i-1}$.

Let $\mathcal{A}$ be a simplification algorithm being able to store at most $m-1$ points. Upon the arrival of p_{m-1}, the algorithm $\mathcal{A}$ is required to delete one of the past points, because it cannot store m points. Let p_i, with $1 \leqslant i \leqslant m-2$, be the deleted point, and let the next point, p_m, be slightly below p_i (i.e. $|p_i p_m| \approx 0$). Up to here, by choosing p_m in $Q_{\mathcal{A}(1)}(m)$, we have $error_{\mathrm{H}}(Q_{\mathcal{A}(1)}(m)) = error_{\mathrm{H}}(Q_{Opt(1)}(m)) = 0$. Now consider the next point p_{m+1}, which lies on its position according to our construction. Obviously, $Q_{Opt(1)}(m+1)$ is p_0, p_i, p_{m+1}, and its Hausdorff error is $d_H(p_{m-1}, p_i p_{m+1})$. Since $\mathcal{A}$ has missed the point p_i, $Q_{\mathcal{A}(1)}(m+1)$ is p_0, p_j, p_{m+1} for some $j \neq i$. There are three possibilities for j:

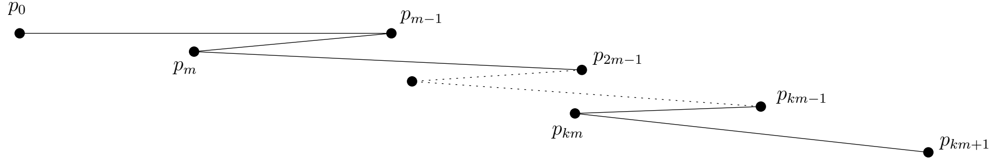


Figure 6.6 A path that cannot be simplified within a bounded competitive ratio.

1. $0 \leqslant j < i$: using property (iii) we have:

$$\begin{aligned} error_{\mathrm{H}}(Q_{\mathcal{A}(1)}(m+1)) &= d_H(p_{m-1}, p_j p_{m+1}) \\ &> c \cdot d_H(p_{m-1}, p_i p_{m+1}) \\ &= c \cdot error_{\mathrm{H}}(Q_{Opt(1)}(m+1)) \end{aligned}$$

2. $i < j \leqslant m-1$: using property (ii) we have:

$$\begin{aligned} error_{\mathrm{H}}(Q_{\mathcal{A}(1)}(m+1)) &\geqslant d_H(p_m, p_j p_{m+1}) \approx |p_i p_j| \\ &> c \cdot d_H(p_{m-1}, p_i p_{m+1}) \\ &= c \cdot error_{\mathrm{H}}(Q_{Opt(1)}(m+1)) \end{aligned}$$

3. $j = m$: using property (ii) we have:

$$\begin{aligned} error_{\mathrm{H}}(Q_{\mathcal{A}(1)}(m+1)) &= d_H(p_{m-1}, p_0 p_m) \approx |p_i p_{m-1}| \\ &> c \cdot d_H(p_{m-1}, p_i p_{m+1}) \\ &= c \cdot error_{\mathrm{H}}(Q_{Opt(1)}(m+1)) \end{aligned}$$

Therefore, in order to be within a bounded competitive ratio, $\mathcal{A}$ must store at least the two points p_{m-1} and p_m, which leads to a 2-simplification.

We concatenate k of these components in such a way that for any two consecutive components, the first two points of the latter lie on the last two points of the former as illustrated in Figure 6.6. Other than the first and the last points, it is straightforward to show that $\mathcal{A}$ has to store 2 points of each component to be within a bounded competitive ratio. This implies $error_{\mathrm{H}}(Q_{\mathcal{A}(2k-1)}) > c \cdot error_{\mathrm{H}}(Q_{Opt(k)}(n))$. □

6.6 Conclusions

We presented the first line-simplification algorithms in the streaming model, where we want to maintain a simplification of a path described by a (possibly infinite) stream of input points, while having only a limited amount of storage available. We obtained algorithms with $O(1)$ competitive ratio for convex planar paths and xy-monotone planar paths

under the Hausdorff error function (or the Fréchet error function), and for general planar paths under the Fréchet distance. Our results imply linear-time approximation when k is a constant (where the approximation factor is with respect to the optimal solution using half the number of links).

Our algorithms all use resource augmentation: they maintain a $2k$-simplification but we compare the error of our simplification to the error of an optimal k-simplification. One obvious question is whether we can do with less, or maybe no, resource augmentation. We have shown that this is not the case for general planar paths under the Hausdorff error function, but note that we have not been able to give any $O(1)$-competitive algorithm for this case, not even with resource augmentation. Thus there is a significant gap between our positive and our negative results.

Another aspect where improvement may be possible is the implementation of the error oracles, which need $O(k^2)$ storage for xy-monotone paths under the Hausdorff error function and for general paths under the Fréchet distance. For instance, if we can maintain core-sets in a streaming setting such that one can also merge two core-sets, then this will reduce the dependency on k in the storage from quadratic to linear. (Note that we need to be able to do an unbounded number of merges.)

Our general approach extends to higher dimensions (but the approximation factors and running times will change).

Chapter 7

Concluding remarks

In this thesis, we concentrated on the algorithmic study of moving objects. We mainly focused on the kinetic-data-structure framework introduced by Basch *et al.* [23], which is a common model for designing and analyzing algorithms and data structures for moving objects within computational geometry. The kinetic-data-structure framework is based on a scenario where the trajectories of moving objects are continuous and explicitly known in advance (at least in the near future). We also studied another reasonable scenario, where the trajectory of a object is not given explicitly and instead a (possible infinite) *stream* of points describing consecutive locations of the moving object is received as an input.

Below we summarize our contributions in this domain, and discuss some directions for further research. (Some other specific open problems were already mentioned in the previous chapters.)

One interesting research direction that we explored in Chapter 2 and that needs more investigation is the following: instead of requiring that the KDS explicitly maintain the attribute of interest, we wish to view KDSs as query structures, and study trade-offs between maintenance cost and query time. For the kinetic sorting problem, we proved a lower bound for this problem showing the following: with a subquadratic maintenance cost one cannot obtain any significant speed-up on the time needed to generate the sorted list (compared to the trivial $O(n \log n)$ time), even for linear motions. This negative result gives a strong indication that good trade-offs are not possible for a large number of geometric problems—Voronoi diagrams and Delaunay triangulations, for example, or convex hulls—as the sorting problem can often be reduced to such problems. But still there is hope to find a good trade-off when the goal is not maintaining a uniquely defined attribute such as the convex hull, but we want to answer some queries such as ”Which are the points currently inside a query rectangle?”, for instance, or ”What is the nearest point to the given query point?”. An example of such a good trade-off is given by Agarwal *et al.* [6]. They designed a KDS for orthogonal range queries, which uses a super-linear storage and allows a trade-off between the total number of events and the query time: they can achieve

$O(Q)$ query time by processing $O(n^{2+\varepsilon}/Q^2)$ events. Unfortunately, their KDS heavily uses its knowledge of the motions and only works for linear motions, which are rather restrictive. A simple method to obtain a trade-off for this problem which does not use any knowledge of the motions and works for any algebraic motions is to partition the point set into Q subsets of size n/Q each, and maintain a kinetic range tree for each subset. Then, we achieve $O(Q)$ query time by processing $O(n^2/Q)$ events. This method is a rather naive way of obtaining a trade-off. What would be the best trade-off we can get for algebraic motions? The same question can be asked for other query structures such as segment trees and kd-trees using a linear storage. As it was mentioned in the introduction, the performance of a KDS is measured according to four criteria. Obtaining good trade-offs between those criteria is another interesting research direction. For example, one may want to obtain a trade-off between the number of candidate pairs for collision detections (compactness) and the number of events (efficiency).

Although the KDS framework is a beautiful and successful framework in theory, it has not been established whether it is effective in practice. One important issue that threatens the applicability of the KDS framework in practice is how to cope with the situation where event times cannot be computed exactly and events may be processed in a wrong order. In Chapter 3, we introduced a new event scheduling mechanism for the kinetic sorting and the kinetic tournament to deal with the out-of-order events. But we do not know whether we can extend our technique to other existing KDSs and in particular, kinetic collision detections. In our event scheduling mechanism, some events may be processed late but the delay in processing events may cause that we miss some collisions, which usually is unsatisfactory. Then, it seems that in collision detection we are obliged to compute the exact order of events. There are elegant and efficient KDSs for collision detections—our KDS for multiple convex fat objects of varying sizes (Chapter 5), for instance—but only a few attempts [33, 60] have been done to experimentally compare actual implementations of KDSs with existing packages providing time-step approaches.

Due to the extensive research interest in KDSs over the pas few years, KDSs have been developed for a variety of structures. But still there are some structures that have not been investigated. For instance, although there are some papers [11, 14, 32, 39] dealing with kinetic binary space partitions, no research has been dedicated to design a kinetic BSP for fat objects. It would be interesting if we could apply our technique in Chapter 4 to De Berg's algorithm [37] which produces a linear-size BSP for static fat objects. Kinetizing quadtrees and BAR trees are other examples which need more investigation.

In a practical setting, objects may not follow algebraic trajectories or the explicit descriptions of their trajectories are not known in advance. For example, in Chapter 6, we considered a model of motions in which, instead of getting an explicit description of the trajectory, we are getting a (possible infinite) stream of points describing consecutive locations of the moving object. We showed how to maintain an approximation of the trajectory of an object, if only limited storage is available. However, it would be also be interesting to study other problems in this model. For example, assume we are given a pattern path P of size m and we are getting a stream of points describing consecutive

locations of the moving object. The goal is to find the maximum sub-path of P matching the received path so far under assumption that a limited amount of memory is available. As an application, when studying the migratory patterns of animals, one may want to know how much the migratory patterns of animals are close to a specific pattern. Another natural problem in this model is to consider a different error measure than the Fréchet (or Hausdorff) distance, namely one that also takes into account the time at which the object is at a certain position.

References

[1] M.A. Abam, P.K. Agarwal, M. de Berg, and H. Yu. Out-of-order event processing in kinetic data structures. In *Proc. European Symposium on Algorithms (ESA)*, pages 624–635, 2006.

[2] M.A. Abam and M. de Berg. Kinetic sorting and kinetic convex hulls. *Computational Geometry: Theory and Applications*, 37:16–26, 2007.

[3] M.A. Abam, M. de Berg, P. Hachenberger, and A. Zarei. Streaming algorithms for line simplification. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 175–183, 2007.

[4] M.A. Abam, M. de Berg, S.-H. Poon, and B. Speckmann. Kinetic collision detection for convex fat objects. In *Proc. European Symposium on Algorithms (ESA)*, pages 4–15, 2006.

[5] M.A. Abam, M. de Berg, and B. Speckmann. Kinetic kd-trees and longest-side kd-trees. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 364–372, 2007.

[6] P.K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66(1):207–243, 2003.

[7] P.K. Agarwal, L. Arge, J. Erickson, and H. Yu. Efficient tradeoff schemes in data structures for querying moving objects. In *Proc. European Symposium on Algorithms (ESA)*, pages 4–15, 2004.

[8] P.K. Agarwal, L. Arge, and J. Vahrenhold. Time responsive external data structures for moving points. In *Proc. Workshop on Algorithms and Data Structures*, pages 50–61, 2001.

[9] P.K. Agarwal, J. Basch, M. de Berg, L.J. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. *Discrete & Computational Geometry*, 24:721–733, 2000.

[10] P.K. Agarwal, J. Basch, L.J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tilings for kinetic collision detection. *International Journal of Robotics Research*, 21:179–197, 2002.

[11] P.K. Agarwal, J. Erickson, and L.J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 107–116, 1998.

[12] P.K. Agarwal, J. Gao, and L.J. Guibas. Kinetic medians and kd-trees. In *Proc. European Symposium on Algorithms (ESA)*, pages 5–16, 2002.

[13] P.K. Agarwal, L.J. Guibas, J. Hershberger, and E. Veach. Maintaining the extent of a moving point set. *Discrete & Computational Geometry*, 26:353–374, 2001.

[14] P.K. Agarwal, L.J. Guibas, T.M. Murali, and J.S. Vitter. Cylindrical static and kinetic binary space partitions. *Computational Geometry: Theory and Applications*, 16(2):103–127, 2000.

[15] P.K. Agarwal, S. Har-Peled, N.H. Mustafa, and Y. Wang. Near-linear time approximation algorithms for curve simplification. *Algorithmica*, 42:203–219, 2005.

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

[17] P.K. Agarwal and K.R. Varadarajan. Efficient algorithms for approximating polygonal chains. *Discrete & Computational Geometry*, 23(2):273–291, 2000.

[18] P.K. Agarwal, Yusu Wang, and Hai Yu. A 2d kinetic triangulation with near-quadratic topological changes. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 180–189, 2004.

[19] P.K. Agarwal and Hai Yu. A space-optimal data-stream algorithm for coresets in the plane. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 1–10, 2007.

[20] H. Alt and M. Godau. Computing the frechet distance between two polygonal curves. *International Journal of Computational Geometry and Applications*, 5:75–91, 1995.

[21] A. Bar-Noy, I. Kessler, and M. Sidi. Mobile users: To update or not to update? *ACM/Baltzer J. Wireless Networks*, 1(2):175–195, 1995.

[22] J. Basch, J. Erickson, L.J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection for two simple polygons. *Computational Geometry: Theory and Applications*, 27(3):211–235, 2004.

[23] J. Basch, L.J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

[24] J. Basch, L.J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 344–351, 1997.

[25] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[26] S. Brakatsoulas, D. Pfoser, and N. Tryfona. Modeling, storing, and mining moving object databases. In *Proc. International Database Engineering and Applications Symposium (IDEAS)*, pages 68–77, 2004.

[27] G.S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002.

[28] S. Cameron. Collision detection by four–dimensional intersectin testing. *IEEE Transaction on Robotics and Automation*, 6(3):291–302, 1990.

[29] The CGAL Library. `http://www.cgal.org/`.

[30] W.S. Chan and F. Chin. Approximation of polygonal curves with minimum number of line segments. In *Proc. International Symposium on Algorithms and Computation (ISAAC)*, pages 378–387, 1992.

[31] G. Collins and A. Akritas. Polynomial real root isolation using descarte's rule of signs. In *Proc. ACM Symposium on Symbolic and Algebraic Computation*, pages 272–275, 1976.

[32] J.L.D. Comba. *Kinetic vertical decomposition trees*. PhD thesis, Department Of Computer Science, Stanford University, 2000.

[33] D. Coming and O. Staadt. Kinetic sweep and prune for collision detection. In *Proc. Workshop on Virtual Reality Interactions and Physical Simulations*, pages 81–90, 2005.

[34] The Core Library. `http://www.cs.nyu.edu/exact/`.

[35] R. Culley and K. Kempf. A collision detection algorithm based on velocity and distance bounds. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1064–1069, 1986.

[36] S.R. Das, R. Castaneda, J. Yan, and R. Sengupta. Comparative performance evaluation of routing protocols for mobile, ad hoc networks. In *Proc. International Conference on Computer Communications and Networks (IC3N)*, pages 153–161, 1998.

[37] M. de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28:353–366, 2000.

[38] M. de Berg. Kinetic dictionaries: How to shoot a moving target. In *Proc. European Symposium on Algorithms (ESA)*, pages 172–183, 2003.

[39] M. de Berg, J. Comba, and L.J. Guibas. A segment-tree based kinetic BSP. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 134–140, 2001.

[40] M. de Berg, H. David, M. Katz, M. Overmars, F. van der Stappen, and J. Vleugels. Guarding scenes against invasive hypercubes. *Computational Geometry: Theory and Applications*, 26:99–117, 2003.

[41] M. de Berg, M. Katz, F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. *Algorithmica*, 34:81–97, 2002.

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

[43] M. Dickerson, C.A. Duncan, and M.T. Goodrich. K-d trees are better when cut on the longest side. In *Proc. European Symposium on Algorithms (ESA)*, pages 179–190, 2000.

[44] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canad. Cartog.*, 10:112–122, 1973.

[45] J. Erickson, L.J. Guibas, J. Stolfi, and L. Zhang. Separation-sensitive collision detection for convex objects. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 327–336, 1999.

[46] A. Foisy and V. Hayward. A safe swept volume method for collision detection. In *Proc. International Symposium of Robotics Research*, pages 61–68, 1993.

[47] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81–128. Information Geometers Ltd., 1993.

[48] S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt. Controlled perturbation for Delaunay triangulations. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1047–1056, 2005.

[49] J. Gao, L.J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. *Discrete & Computational Geometry*, 30(1):45–63, 2003.

[50] E.G. Gilbert and C.-P. Foo. Computing the distance between general convex objects in three-dimensional space. In *Proc. IEEE International Conference on Robotics and Automation*, pages 53–61, 1990.

[51] E.G. Gilbert and S.M. Hong. A new algorithm for detecting the collision of moving objects. In *Proc. IEEE International Conference on Robotics and Automation*, pages 8–14, 1989.

[52] M. Godau. A natural metric for curves: computing the distance for polygonal chains and approximation algorithms. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 127–136, 1991.

[53] M.T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric: (preliminary version). In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 322–331, 1994.

[54] L.J. Guibas. Kinetic data structure: a state of art report. In *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209, 1998.

[55] L.J. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. *Discrete & Computational Geometry*, 25:591–610, 2001.

[56] L.J. Guibas, J.E. Hershberger, J.S.B. Mitchell, and J.S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *International Journal of Computational Geometry and Applications*, 3:383–415, 1993.

[57] L.J. Guibas and M. Karavelas. Interval methods for kinetic simulation. In *Proc. ACM Symposium Computational Geometry (SCG)*, pages 255–264, 1999.

[58] L.J. Guibas, M. Karavelas, and D. Russel. A computational framework for handling motion. In *Proc. Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.

[59] L.J. Guibas and D. Russel. An empirical comparison of techniques for updating Delaunay triangulations. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 170–179, 2004.

[60] L.J. Guibas, F. Xie, and L. Zhang. Kinetic collision detection: Algorithms and experiments. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 2903–2910, 2001.

[61] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[62] S.L. Hakimi and E.F. Schmeichel. Fitting polygonal functions to a set of points in the plane. *CVGIP: Graph. Models Image Process.*, 53:132–136, 1991.

[63] D. Halperin and C. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Computational Geometry: Theory and Applications*, 10:273–287, 1998.

[64] M. Herman. Fast, three-dimensional, collision-free motion planning. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1056–1063, 1986.

[65] J. Hershberger and J. Snoeyink. An $o(n \log n)$ implementation of the douglas-peucker algorithm for line simplification. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 383–384, 1994.

[66] J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulae and in $o(nlog * n)$ time. In *Proc. Workshop on Algorithms and Data Structures*, pages 93–103, 1997.

[67] J. Hershberger and S. Suri. Kinetic connectivity of rectangles. In *Proc. ACM Symposium Computational Geometry (SCG)*, pages 237–246, 1999.

[68] H. Imai and M. Iri. Polygonal approximations of a curve-formulations and algorithms. In *Computational Morphology*, pages 71–86, 1988.

[69] N. Jacobson. *Basic Algebra I*. W.H. Freeman, New York, 2nd edition, 1985.

[70] D. Johnson. Routing in ad hoc networks of mobile hosts. In *Proc. Workshop on Mobile Computing Systems and Applications*, pages 158–163, 1994.

[71] M.J. Katz. 3-D vertical ray shooting and 2-D point enclosure, range searching, and arc shooting amidst convex fat objects. *Computational Geometry: Theory and Applications*, 8:299–316, 1998.

[72] D. Kim, L.J. Guibas, and S. Shin. Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):230–242, 1998.

[73] H.K. Kim, L. Guibas, , and S.Y. Shin. Efficient collision detection among moving spheres with unknown trajectories. *Algorithmica*, 43:195–210, 2005.

[74] D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic collision detection for simple polygons. *International Journal of Computational Geometry and Applications*, 12(1&2):3–27, 2002.

[75] D. Kirkpatrick and B. Speckmann. Kinetic maintenance of context-sensitive hierarchical representations of disjoint simple polygons. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 179–188, 2002.

[76] Y.-B. Ko and N.H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. *Wirel. Netw.*, 6(4):307–321, 2000.

[77] G. Kollios, D. Gunopulos, and V. Tsotras. Nearest neighbor queries in a mobile environment. In *Proc. Intl. Workshop on Spatiotemporal Database Managment*, pages 119–134, 1999.

[78] G. Kollios, D. Gunopulos, and V. Tsotras. On indexing mobile objects. In *Proc. ACM Symposium on Principles of Database Systems*, pages 261–272, 1999.

[79] Y. Kunita, M. Inami, T. Maeda, and S. Tachi. Real-time rendering system of moving objects. In *Proc. IEEE Workshop on Multi-View Modeling & Analysis of Visual Scenes*, pages 81–88, 1999.

[80] D. Lam, D. Cox, and J. Widom. Teletraffic modeling for personal communications services. *IEEE Communications Magazine*, 35(2).

[81] K.K. Leung, W.A. Massey, and W. Whitt. Traffic models for wireless communication networks. *IEEE Journal on Selected Areas in Communications*, 12(8):1353–1364, 1994.

[82] B. Liang and Z.J. Haas. Predictive distance-based mobility management for PCS networks. In *INFOCOM (3)*, pages 1377–1384, 1999.

[83] M.C. Lin and D. Manocha. Collision and proximity queries. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 787–807. CRC Press, 2nd edition, 2004.

[84] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Apllication*, 2:169–186, 1992.

[85] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete & Computational Geometry*, 10:215–232, 1993.

[86] A. Melkman and J. ORourke. On polygonal chain approximation. In *Computational Morphology*, pages 87–95, 1988.

[87] V. Milenkovic and E. Sacks. An approximate arrangement algorithm for semi-algebraic curves. In *Proc. ACM Symposium on Computational Geometry (SCG)*, pages 237–246, 2006.

[88] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Proc. Conference on Computer Graphics and Interactive Techniques*, pages 289–298, 1988.

[89] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *The VLDB Journal*, pages 395–406, 2000.

[90] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. Indexing the positions of continuously moving objects. In *Proc. SIGMOD Conference*, pages 331–342, 2000.

[91] S. Schirra. Robustness and precision issues in geometric computation. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science, 2000.

[92] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *Proc. ACM Symposium on Computational Geometry*, pages 51–60, 1995.

[93] A.P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 269–280, 1995.

[94] A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. International Conference on Data Engineering (ICDE)*, pages 422–432, 1997.

[95] C.-K Toh. A novel distributed routing protocol to support ad-hoc mobile computing. In *Proc. IEEE Annu. Internat. Phoenix Conf. Comput. Comm.*, pages 480–486, 1996.

[96] A.F. van der Stappen. *Motion planning amidst fat obstacles*. PhD thesis, Utrecht University, Utrecht, the Netherlands, 1994.

[97] E.W. Weinstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 1999.

[98] O. Wolfson, A.P. Sistla, J. Zhou B. Xu, and S. Chamberlain. Databases for moving objects tracking. In *Proc. SIGMOD Conference*, pages 547–549, 1999.

[99] O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[100] O. Wolfson, A.P. Sistla, B. Xu, J. Zhou, S. Chamberlain, Y. Yesha, and N. Rishe. Tracking moving objects using database technology in domino. In *Proc. Next Generation Information Technologies and Systems (NGITS)*, pages 112–119, 1999.

[101] C. Yap. Robust geometric computation. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2nd edition, 2004.

[102] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4):13, 2006.

[103] H. Zarrabi-Zadeh and T. Chan. A simple streaming algorithm for minimum enclosing balls. In *Proc. Canadian Conference on Computational Geometry (CCCG)*, pages 139–142, 2006.

[104] Y. Zhou and S. Suri. Analysis of a bounding box heuristic for object intersection. *Journal of the ACM*, 46(6):833–857, 1999.

[105] M.M. Zonoozi and P. Dassanayake. User mobility modeling and characterization of mobility patterns. *IEEE Journal on Selected Areas in Communications*, 15(7):1239–1252, 1997.

Summary

New Data Structures and Algorithms for Mobile Data

Recent advances in sensing and tracking technology have led researchers to investigate the problem of designing and analyzing algorithms and data structures for *moving objects*. One important issue in this area of research is which assumptions are made about the motions of the objects. The most common model is one where motions are assumed to be continuous and explicitly known in advance (or at least in the near future), usually as polynomial functions of time. The kinetic-data-structure framework introduced by Basch *et al.* is based on this model. It has become the common model for dealing with moving objects in computational geometry.

A *kinetic data structure* (*KDS*) maintains a discrete attribute of a set of moving objects—the convex hull, for instance, or the closest pair. The basic idea is that although all objects move continuously there are only certain discrete moments in time when the combinatorial structure of the attribute—the ordered set of convex-hull vertices, or the pair that is closest—changes. A KDS contains a set of *certificates* that constitutes a proof that the maintained structure is correct. These certificates are inserted in a priority queue based on their time of expiration. The KDS then performs an event-driven simulation of the motion of the objects, updating the structure whenever an *event* happens, that is, when a certificate fails.

In some applications, continuous tracking of a geometric attribute may be more than is needed; the attribute is only needed at certain times. This leads us to view a KDS as a *query structure*: we want to maintain a set S of moving objects in such a way that we can reconstruct the attribute of interest efficiently whenever this is called for. This makes it possible to reduce the maintenance cost, as it is no longer necessary to update the KDS whenever the attribute changes. On the other hand, a reduction in maintenance cost will have an impact on the query time, that is, the time needed to reconstruct the attribute. Thus there is a trade-off between maintenance cost and query time. In Chapter 2, we show a lower bound for the kinetic sorting problem showing the following: with a subquadratic maintenance cost one cannot obtain any significant speed-up on the time needed to generate the sorted list (compared to the trivial $O(n \log n)$ time), even for linear motions.

This negative result gives a strong indication that good trade-offs are not possible for a large number of geometric problems—Voronoi diagrams and Delaunay triangulations, for example, or convex hulls—as the sorting problem can often be reduced to such problems.

KDSs form a beautiful framework from a theoretical point of view, but whether or not they perform well in practice is unclear. A serious problem for the applicability of the KDS framework in practice is how to cope with the situation where event times cannot be computed exactly and events may be processed in a wrong order. We addresses this problem in Chapter 3. We present KDSs that are robust against the out-of-order processing, including kinetic sorting and kinetic tournaments. Our algorithms are *quasi-robust* in the sense that the maintained attribute of the moving objects will be correct for most of the time, and when it is incorrect, it will not be far from the correct attribute.

The aim of the KDS framework is not only maintaining a uniquely defined geometric attribute but also maintaining a *query data structure* in order to quickly answer queries involving objects in motion such as "Which are the points currently inside a query rectangle?", or "What is currently the nearest point to a given query point?". In Chapter 4, we study the kinetic maintenance of kd-trees which are practical data structures to quickly report all points inside any given region. We present a new and simple variant of the standard kd-tree, called *rank-based kd-trees*, for a set of n points in d-dimensional space. Our rank-based kd-tree supports orthogonal range searching in time $O(n^{1-1/d}+k)$ and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized easily and efficiently. We obtain the similar results for longest-side kd-trees.

Collision detection is a basic problem arising in all areas of geometric modeling involving objects in motion—motion planning, computer-simulated environments, or virtual prototyping, to name a few. Kinetic methods are naturally applicable to this problem. Although most applications of collision detection are more concerned with three dimensions than two dimensions, so far KDSs have been mostly developed for two-dimensional settings. In Chapter 5, we develop KDSs for 3D collision detection that have a *near-linear number of certificates* for *multiple* convex fat objects of *varying sizes* and for a special case of *balls* rolling on a plane.

In a practical setting, the object motion may not known exactly or the explicit description of the motion may be unknown in advance. For instance, suppose we are tracking one, or maybe many, moving objects. Each object is equipped with a device that is transmitting its position at certain times. Then, we just have access to some sample points of the object path instead of the whole path, and an explicit motion description is unavailable. Thus, we are just receiving a stream of data points that describes the path along which the object moves. This model is the subject of Chapter 6. Here, we present the first general algorithm for maintaining a simplification of the trajectory of a moving object in this model, without using too much storage. We analyze the competitive ratio of our algorithms, allowing resource augmentation: we let our algorithm maintain a simplification with $2k$ (internal) points, and compare the error of our simplification to the error of the optimal simplification with k points.

Curriculum Vitae

Mohammad Ali Abam was born on the 5th of January 1977 in Tehran, Iran. In 1995, he graduated from the Roshd High School in Mathematics and Physics. He received his Bachelor and Master degrees, both in computer engineering from Sharif University of Technology, Tehran, Iran in 1999 and 2001, respectively. Since January 2004, he has been a Ph.D. student within the Computer Science Department of the Technische Universiteit Eindhoven (TU/e).

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition*

and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies,*

and Business Applications. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical En-

gineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementa-*

tion and Composition. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery Universiteit van Amsterdam, CWI*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. de Graaf. *Model-Driven Evolution of Software Architectures, Universiteit van Delft*. Faculty of Electrical Engineering, Mathematics, and Computer Science Delft University of Technology. 2007-16

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-17