Three Approaches to Building Time-Windowed Geometric Data Structures

by

Simon David Pratt

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

Abstract

Given a set of geometric objects (points or line segments) each associated with a time value, we wish to determine whether a given property is true for a subset of those objects whose time values fall within a query time window. We call such problems *time-windowed decision problems*. We present algorithms to preprocess for the time-windowed closest pair decision problem in $\mathcal{O}(n)$ expected time, for the time-windowed 2D diameter decision problem in $\mathcal{O}(n \log n)$ time, the time-windowed 2D convex hull area decision problem in $\mathcal{O}(n\alpha(n) \log n)$ time (where α is the inverse Ackermann function), and the time-windowed 3D diameter decision and orthogonal segment intersection detection problems in $\mathcal{O}(n \operatorname{polylog} n)$ time.

Our first approach is to reduce the closest pair decision problem to 2D dominance range emptiness using grids to compute candidate satisfying pairs. We extend this approach to find the closest pair of points by reducing the problem to 2D dominance range minimum, which we further reduce to 2D point location.

Our second approach is to reduce time-windowed decision problems to a generalized range successor problem, which we solve using a novel way to search range trees.

Our third approach is to use dynamic data structures directly, taking advantage of a new observation that the total number of combinatorial changes to a planar convex hull is near linear for any *FIFO* update sequence, in which deletions occur in the same order as insertions.

Acknowledgements

This thesis is based primarily on 2 conference publications: [CP15] from CCCG 2015, and [CP16] from SoCG 2016. These publications are joint work with my supervisor, Timothy M. Chan, without whom this work would not have been possible. I also wish to thank Haim Kaplan and Micha Sharir for suggesting the use of Tamir's observation [Tam88] in [CP16], which leads to an $\alpha(n)$ factor improvement to an earlier version of Lemma 17, where α is the inverse Ackermann function. I would like to thank my readers, Therese Biedl and J. Ian Munro for their valuable feedback. Additionally, I would like to thank my undergraduate supervisor, Michiel H. M. Smid, without whom I would not have chosen to study computational geometry in the first place.

Finally, I would like to thank my partner, Elsa; my parents, David and Tricha; my brother, Russell; and all of my friends and family whose support made this possible. In particular, I would like to thank Peter Simonyi for his help with typesetting.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Geographic Information System (GIS) data gathered in the field by handheld *Global Positioning System* (GPS) units often consists not only of latitude, longitude, and altitude but also of *time*. A curious GIS professional might naturally ask questions about the geometry of this data based on windows of time.

In a *time-windowed* geometric problem, we wish to process a set of objects (points or segments), where each object is associated with a time value, such that given a query interval of time called a *time window*, we can quickly answer a geometric problem on the subset of objects whose time values fall within that window. For brevity, we say objects whose time values are within a query time window are themselves within the time window.

Essentially, we would like to perform a range reporting query in the time dimension, and then on the points returned by this range query we would like to answer some geometric query such as finding the closest pair. If the geometric query were simply another range reporting query in $\mathbb{R}^d$, then we could simply treat the time dimension as another spatial dimension and answer one range reporting query in $d+1$ dimensions (including time).

For example, consider the time-windowed closest pair problem: preprocess a set S of n points in $\mathbb{R}^d$ such that, given a query time window $[t_1, t_2]$, we can quickly find the pair of points $p, q \in S$ such that $t_1 \leq t(p) \leq t(q) \leq t_2$ and the distance between p and q is minimal among all points within the query time window (see Figure 1.1 for an example in $\mathbb{R}^1$).

Time-windowed geometric problems have been the subject of many recent papers and are motivated both by the aforementioned GIS data as well as timestamped social network data. A 2014 paper by Bannister *et al.* [BDG+14] examined time-windowed versions of convex hull, approximate spherical range searching, and approximate nearest neighbor

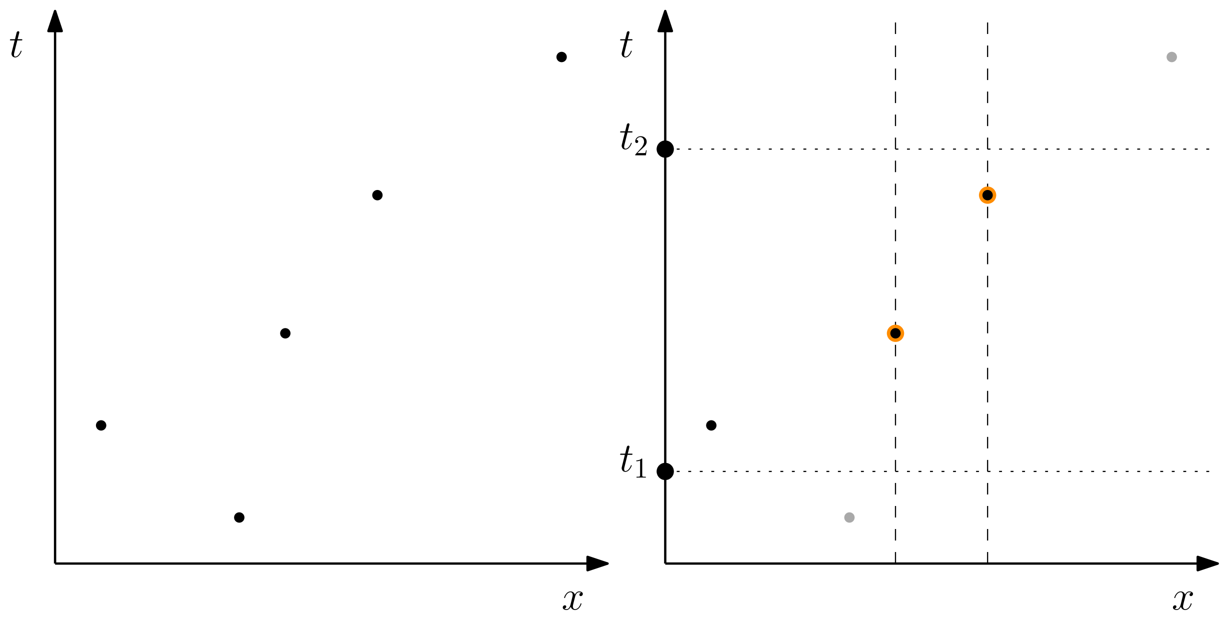

Figure 1.1: *Left*: a set of points in $\mathbb{R}^1$, whose spatial coordinates have been plotted in the horizontal x dimension, and whose temporal coordinates have been plotted in the vertical t dimension. *Right*: the same set of points and a time window $[t_1, t_2]$, whose bounds are indicated by dotted horizontal lines which excludes a pair of points which is lowlighted. The closest pair within the query window is **highlighted**, and the distance between the closest pair is indicated by vertical dashed lines.

queries. At SoCG 2015, Bokal *et al.* [BCE15] presented more results on a variety of other time-windowed problems.

1.1 Problem Statements

We consider the following time-windowed problems:

1. *Closest Pair Decision*: Given a set of n time-labeled points in $\mathbb{R}^d$, we want to determine if there exist two points closer than unit distance apart among all points within a query time window.

2. *Closest Pair*: Given a set of n time-labeled points in $\mathbb{R}^d$, we want to determine the closest pair of points within a query time window.

3. *2D and 3D Diameter Decision*: Given a set of n time-labeled points in $\mathbb{R}^2$ or $\mathbb{R}^3$, determine if there exist two points greater than unit distance apart, whose time values are within a query time window.

4. *2D Orthogonal Segment Intersection Detection*: Given a set of n orthogonal (horizontal or vertical) time-labeled line segments in $\mathbb{R}^2$, we want to determine if there are any intersections between segments whose time values are within a query time window.

5. *2D Convex Hull Area Decision*: Given a set of n time-labeled points in $\mathbb{R}^2$, determine whether the convex hull of points within a query time window has greater than unit area.

6. *2D Width Decision*: Given a set of n time-labeled points in $\mathbb{R}^2$, we want to determine whether the points within a query time window have greater than unit width.

In the following section, we will formally define the convex hull and width of a set of points as part of our discussion of related work.

1.2 Related Work

It is worthwhile to begin by considering the same problems in the standard (non-time-windowed) setting in order to gain some perspective. Since a time-window can include all points, the sum of preprocessing and query time for a problem in the time-windowed setting cannot be less than the time of solving the problem in the standard setting.

Given a set P of n points in $\mathbb{R}^2$, the *convex hull* of that set of points is the minimal convex area that encloses P. The convex hull can be computed using the Jarvis march technique in $\mathcal{O}(nh)$ time [Jar73], where h is the size of the convex hull; or in $\mathcal{O}(n)$ time (after sorting) using Graham's scan [Gra72]; or in $\mathcal{O}(n \log h)$ time using either the Kirkpatrick-Seidel algorithm [KS86] or Chan's algorithm [Cha96]. Yao proved that computing the convex hull requires $\Omega(n \log n)$ time in the quadratic decision tree model, under the assumption that a fixed fraction of the points are on the convex hull [Yao81].

Given a set P of n points, the *closest pair* in that set is a pair of points $p, q \in P$ such that the distance between p and q is minimal among all pairs of points in the input. In any constant dimension, closest pair can be solved in $\mathcal{O}(n \log n)$ time deterministically [SH75, BS76] or $\mathcal{O}(n)$ expected time using randomization [Rab76].

Similarly, the *diameter* of a set of n points is the pair of points whose distance is maximal. Preparata and Shamos give an $\Omega(n \log n)$ lower bound along with an optimal $\Theta(n \log n)$ algorithm [PS85]. Observe that the diameter of a point set is equal to the diameter of that set's convex hull. This means that if the set of points is in $\mathbb{R}^2$, then we can compute the convex hull of the set of points in $\mathcal{O}(n \log h)$ time, and then simulate rotating calipers around the convex hull to find the diameter in $\mathcal{O}(h)$ time [Sha78]. We can use the same approach to find the *width* of a set of points, which is the minimal distance between two parallel lines that are on either side of all points. One or the other of such parallel lines would necessarily be tangent to the convex hull, otherwise we could rotate the lines until one side is tangent, and this would reduce the distance between them while still being on either side of the points.

The problem of finding the diameter of a set P of n points in $\mathbb{R}^3$ is related to finding the intersection of unit balls centered at the points of P. Clarkson and Shor gave optimal randomized algorithms for both problems which run in $\mathcal{O}(n \log n)$ [CS89]. Amato *et al.* matched the same time bound with a deterministic algorithm [AGR94].

Bannister *et al.* [BDG+14] examined time-windowed convex hull and approximate proximity queries. Their approach for answering convex hull queries is to build a balanced binary search tree with the time-labeled points stored at the leaves in time order. Each internal node x is associated with a canonical subset of the time-labeled points at the leaves of the subtree rooted at x. At each such internal node, they store the convex hull of all points in its canonical subset. This approach achieves the following results, where ω is the number of points whose time values are within the query time window:

1. *2D gift wrapping*: Given a set of n time-labeled points in $\mathbb{R}^2$ and a query point q on the convex hull of a query time window, locate the clockwise or counterclockwise adjacent point to q on the convex hull of the query time window. Their approach obtains $\mathcal{O}\left(\log^2 \omega\right)$ time.

2. *2D convex hull*: Given a set of n time-labeled points in $\mathbb{R}^2$, compute the convex hull of points within a query time window. Their approach follows immediately from their gift wrapping result, and obtains $\mathcal{O}\left(h \log^2 \omega\right)$ time, where h is the number of points on the convex hull.

3. *2D tangent*: Given a set of n time-labeled points in $\mathbb{R}^2$, find the two tangents of the convex hull of points within a query time window passing through a query point q outside of the convex hull. Their approach obtains $\mathcal{O}\left(\log^2 \omega\right)$ time.

4. *2D linear programming*: Given a set of n time-labeled points in $\mathbb{R}^2$, find the furthest point in a query direction on the convex hull of all points within a query time window. Their approach obtains $\mathcal{O}(\log \omega)$ time.

5. *2D line stabbing*: Given a set of n time-labeled points in $\mathbb{R}^2$, find all edges of the convex hull of all points within a query time window, such that all returned edges are stabbed by a query line. Their approach obtains $\mathcal{O}\big(\log^2 \omega\big)$ time.

Their approach to answering approximate proximity queries is to build a similar balanced binary search tree with the time-labeled points stored at the leaves in time-order, and store the Z-order (also known as Morton or shuffle order) [BET99] of the canonical subset of each internal node. This approach achieves the following results:

1. *Approximate spherical range searching*: Given a set of n time-labeled points in $\mathbb{R}^d$, and a query point q and radius r, return all points within a query time window whose distance to q is at most r, and such that no returned point has greater than $(1+\varepsilon)r$ distance from q. Their approach obtains $\mathcal{O}(\log \omega + k)$ time, where k is the number of points reported in the output, for fixed $d \geq 2$.

2. *Approximate nearest neighbor*: Given a set S of n time-labeled points in $\mathbb{R}^d$, and a query point q, return a point $p \in S$ whose distance to q is at most $(1+\varepsilon)r$ where r is the distance from q to its nearest neighbor in S. Their approach obtains $\mathcal{O}(\log \omega)$ time, for fixed $d \geq 2$.

3. *Proximity graph construction*: They can construct the Delaunay triangulation, minimum spanning tree, nearest neighbor graph, and Gabriel graph each in $\mathcal{O}(\omega)$ time.

Recently, Bokal *et al.* [BCE15] presented an approach to time-windowed decision problems on *hereditary* properties. If a property $\mathcal{P}$ is hereditary and set S has $\mathcal{P}$ then any superset $S' \supseteq S$ also has it.[1] Examples of hereditary properties include: the set of points has greater than unit diameter, or the convex hull of a set of points has greater than unit area.

Bokal *et al.* observe that it suffices to find for each start time t the minimal end-time t' such that all objects within the window $[t, t']$ have $\mathcal{P}$. We can store the maximal end-time for the input time for each point of the input, then answer a query by table lookup after

[1] The definition in [BCE15] considers subsets instead of supersets, but is equivalent after complementation.

finding the predecessor time for our query. In fact, in Chapter 2 we reduce such queries to rank/select queries in $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ bits of space. Thus, since answering a query after preprocessing is trivial, Bokal *et al.* focus only on bounding the preprocessing time.

They achieve the following geometric results:

1. *2D diameter decision*: Given a set of n time-labeled points in $\mathbb{R}^2$, determine if there exist two points greater than unit distance apart, whose time values are within a query time window. Their approach obtains $\mathcal{O}\big(n \log^2 n\big)$ preprocessing time.

2. *2D convex hull area decision*: Given a set of n time-labeled points in $\mathbb{R}^2$, determine whether the convex hull of points within a query time window has greater than unit area. Their approach obtains $\mathcal{O}(n \log n \log \log n)$ preprocessing time.

3. *2D monotone paths*: Given a set of n points in $\mathbb{R}^2$, determine if the points within a query time window form a monotone path in some (subpath-dependent) direction. Their approach obtains $\mathcal{O}(n)$ preprocessing time.

They also show that their approach works for graph planarity. Given a graph whose edge set contains n time-labeled edges, determine if the graph formed by the edges within a query time window is *planar*, meaning that graph can be drawn with no crossing edges. Their approach obtains $\mathcal{O}(n \log n)$ preprocessing time.

Bokal *et al.*'s main approach computes the values of a binary $n \times n$ upper triangular matrix where entry i, j has value 1 if and only if the set of objects within the window $[i, j]$ has $\mathcal{P}$. The first step is to greedily decompose the matrix into disjoint rectangles along the diagonal. The second step is to compute the values within each rectangle. First they compute the maximal end-time for the row which divides height of the rectangle in half. This splits the rectangle into 4 sub-rectangles, above and below the median row and left and right of its maximal end-time. The entries in the top-right have value 0, the entries in the bottom-left have value 1, and they recurse on top-left and bottom-right. Efficiency comes from using a bounded-size *sketch* of uncomputed regions during recursion. A sketch is similar to a coreset in that it approximates a subset of objects, and its exact nature depends on the problem. Their results for both the 2D diameter and the 2D convex area decision problems are obtained via this approach.

Chan gave a fully dynamic data structure for planar width with $\mathcal{O}(\sqrt{n}\,\mathrm{polylog}(n))$ update time [Cha01b]. Using a naive approach, we can solve the 2D time-windowed width decision problem using this structure in $\mathcal{O}\big(n^{3/2}\,\mathrm{polylog}(n)\big)$ preprocessing time.

1.3 Assumptions

We assume that time values are given as integers from 1 to n, for otherwise we can preprocess the input and replace time values with their rank during preprocessing. The query time only increases by $\mathcal{O}(1)$ predecessor searches on the query time values.

1.4 Summary of Contributions

We achieve the following results:

1. *Closest pair decision*: We give the first nontrivial result for this problem, obtaining $\mathcal{O}(n)$ expected preprocessing time in the word-RAM model.

2. *Closest pair*: We give the first nontrivial result for this problem, answering queries in $\mathcal{O}(\log\log n)$ time with $\mathcal{O}(n\log n)$ words of space and $\mathcal{O}(n\log n\log\log n)$ preprocessing time in the word-RAM model.

3. *2D and 3D diameter decision*: We improve Bokal *et al.*'s preprocessing time bound in 2D from $\mathcal{O}(n\log^2 n)$ to $\mathcal{O}(n\log n)$. Thus, we obtain the first optimal algorithm for the problem in the algebraic decision-tree model [PS85]. Furthermore, we obtain the first nontrivial result in 3D with $\mathcal{O}(n\log^2 n)$ preprocessing time.

4. *2D orthogonal segment intersection detection*: We give the first nontrivial result for this problem, obtaining $\mathcal{O}(n\log n\log\log n)$ preprocessing time.

5. *2D convex hull area decision*: We improve Bokal *et al.*'s preprocessing time bound from $\mathcal{O}(n\log n\log\log n)$ to $\mathcal{O}(n\alpha(n)\log n)$ (where α is the inverse Ackermann function). This is quite close to optimal, except for a very tiny $\alpha(n)$ factor.

6. *2D width decision*: We give the first nontrivial result for this problem, obtaining $\mathcal{O}(n\log^7 n)$ preprocessing time.

1.5 Thesis Organization

In Chapter 2, we discuss a further refinement of the closest pair decision problem, and extended the result to solve the closest pair problem.

In Chapter 3, we discuss time-windowed problems we call "pairwise interaction problems," which are problems for properties that deal with pairs. These include: diameter decision, orthgonal segment intersection detection.

In Chapter 4, we discuss time-windowed convex hull decision problems. These are time-windowed decision problems which are related to computing the convex hull. These include: convex hull area decision, and width decision.

In Appendix A, we discuss the techniques and results by others which are not central to our results but upon which our results depend, typically used as black boxes.

Chapter 2

Time-Windowed Closest Pair

In this chapter, we consider the following time-windowed problems:

Closest Pair Decision Given a set of n time-labeled points in $\mathbb{R}^d$, we want to determine if there exist two points closer than unit distance apart.

Closest Pair Given a set of n time-labeled points in $\mathbb{R}^d$, we want to determine the closest pair of points within a query time window.

The results presented in this chapter are joint work with Timothy M. Chan and appeared in the Proceedings of the 27th Canadian Conference on Computational Geometry (CCCG 2015) [CP15].

2.1 Preliminaries

In this chapter, we work in the w-bit *word-RAM model*, which models the computer as a sequence of w-bit memory locations indexed by a w-bit integer. In this model, we assume that $w \geq \log n$ and standard operations on words take constant time.

2.1.1 Grids and Quadtrees

In the following chapter, we will use the idea of a *grid* which groups the points into *cells*. Our intuitive notion of a grid evokes the image of grid paper on which has been drawn

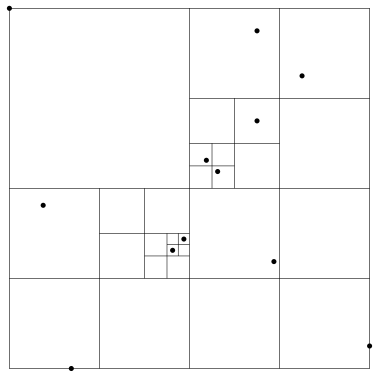

Figure 2.1: A quadtree divides the bounding box of a set of points into 4 cells, then recursively builds a quadtree on the points in each cell until each cell contains only a single point.

evenly spaced parallel lines both vertically and horizontally. This divides the paper into *cells* which are square areas of white space bounded on all four sides by blue lines.

We can formally define a grid and its cells by borrowing some terminology from modular arithmetic. Given $x \in \mathbb{R}$ and $r > 0$, let $x \text{ div } r = \lfloor x/r \rfloor$. Similarly, given a point $p \in \mathbb{R}^d$ with coordinates $(p_1, p_2, \ldots, p_d)$, $p \text{ div } r = (\lfloor p_1/r \rfloor, \lfloor p_2/r \rfloor, \ldots, \lfloor p_d/r \rfloor)$. We say that two points p, q are in the same *grid cell* of *side length* r if $p \text{ div } r = q \text{ div } r$.

The use of grids in computational geometry has been well-studied [GBT84, HP11], and we will use it to solve the time-windowed closest pair decision problem. We will then extend this solution to the exact problem by using a related structure called a *quadtree*.

Let P be a set of n points in $\mathbb{R}^d$, and B be a hypercube (which we call a *quadtree cell*) containing those points. To build the quadtree for P, we divide B into 2^d congruent child hypercubes. For each of these child hypercubes which contain more than a single point, we recursively build a quadtree for that box (see Figure 2.1). For more details on quadtrees, see [FB74, HP11].

The following lemma, similar to Lemma 4 from Arya *et al.* [AMN+98], bounds the number of points in a hypercube by a constant with respect to the hypercube's side length and the distance between the closest pair within that hypercube.

Lemma 1 (Packing). *If a point set has closest-pair distance at least r and lies in a d-dimensional hypercube with side length at most $b \cdot r$, then there are fewer than $c_0(b+1)^d$ points in that point set, where c_0 is some constant that depends only on d. We call c_0 the* packing constant*.*

Proof. Since there are no two points closer than r from each other, we know there exists an empty ball of radius $r/2$ around each point. The maximum number of points that we can pack into a hypercube with side length br is equal to the number of such balls that we can pack into a hypercube with side length $(b+1)r$, whose volume is $(b+1)^d r^d$. The volume of a d-dimensional ball with radius $r/2$ is given by the formula:

$$\frac{\pi^{d/2} r^d}{2^d \cdot \Gamma\left(\frac{d}{2}+1\right)}$$

where Γ is Euler's gamma function. Thus we can pack the following number of such balls into such a hypercube:

$$\begin{aligned} \frac{(b+1)^d r^d}{\frac{\pi^{d/2} r^d}{2^d \cdot \Gamma\left(\frac{d}{2}+1\right)}} &= \frac{2^d \cdot \Gamma\left(\frac{d}{2}+1\right) \cdot (b+1)^d r^d}{\pi^{d/2} r^d} \\ &= \frac{2^d \cdot \Gamma\left(\frac{d}{2}+1\right)}{\pi^{d/2}} \cdot (b+1)^d. \end{aligned}$$

If the dimension d is fixed, then define the constant $c_0 = 2^d \cdot \Gamma\left(\frac{d}{2}+1\right)/\pi^{d/2}$. Therefore, the number of balls we can pack into such a hypercube is at most $c_0 \cdot (b+1)^d$. □

The following lemma and its proof are adapted from Arya *et al.* [AMN+98] (Section 2.3: Midpoint Algorithm), though the concept itself is much older.

Lemma 2 (Centroid)**.** *Given a point set P containing n points, there exists a quadtree cell B, which we call a* centroid cell*, such that $|P \cap B| \leq \alpha n$ and $|P \setminus B| \leq \alpha n$ for some constant $\alpha < 1$ that depends only on d.*

Proof. We will prove this in $\mathbb{R}^2$ with $\alpha = 3/4$, but it generalizes to $\mathbb{R}^d$ with $\alpha = \frac{2^d-1}{2^d}$. Divide the bounding box of the n input points into 4 disjoint, congruent cells of equal area. Consider the cell containing the most points. This cell can contain no fewer than $n/4$ points. If it has fewer than $3n/4$ points, then this is the centroid and we are done. Otherwise, it must contain more than $3n/4$ points and we recurse on it. At some level, the cell with the most points must contain between $n/4$ and $3n/4$ points. □

Recursive application of this lemma gives a data structure on a set of points P, called a *balanced quadtree* [Ber93], defined as a binary tree where the root stores B, the left subtree is the balanced quadtree for $P \cap B$, and the right subtree is the balanced quadtree for $P \setminus B$ where B is a centroid cell of P.

We have the following lemma due to Chan [Cha98] (Observation 3.2 and Lemma 3.3), which says that if we draw a constant number of grids over our points, each shifted by some amount, then we can guarantee that any pair of points must be in the same cell in at least one such grid. Since quadtree cells are related to grid cells, this also implies that the closest pair will be in the same quadtree cell if we build a constant number of quadtrees.

Lemma 3 (Shifting). *Suppose d is even. Let $v^{(j)} = (\lfloor j2^w/(d+1)\rfloor, \ldots, \lfloor j2^w/(d+1)\rfloor) \in \mathbb{R}^d$, where w is the number of bits in a word. For any points p and q and $r' = 2^\ell$ such that $||p-q||_\infty \leq r'$, there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ and $q + v^{(j)}$ belong to the same grid cell with side length $c_1 r'$, where c_1 is the smallest power of 2 bigger than or equal to $2d+2$. We call c_1 the* shifting constant.

For completeness sake, the proof is included in Section A.1 of Appendix A. While the preceding lemma requires d to be even, for odd values of d we can use $d+1$.

2.2 Closest Pair Decision Problem

Before we solve the time-windowed closest pair problem, it helps to consider the decision problem version, in which we are additionally given a fixed distance r and we want to preprocess P into a data structure which can efficiently determine, for any query time window, if there exists a pair of active points pq such that the distance between p and q is at most r. We call such a pair a *satisfying pair*.

The main idea of our approach is to use a constant number of shifted grids, which by Lemma 3 ensures that any two points will appear in the same cell together in at least one such shifted grid. For each point, we consider a constant number of its time-order predecessors and successors within the same cell, which by Lemma 1 we know must include a satisfying pair if one exists. From there, we reduce the problem to a standard 2-dimensional dominance range searching problem.

2.2.1 Computing Candidate Satisfying Pairs

We begin by bucketing the points of P into grid cells with side length $c_1 r'$, where c_1 is the shifting constant from Lemma 3 and r' is the smallest power of 2 bigger than or equal to

r. The number of grid cells is generally unbounded with respect to n, but at most n such cells can contain points. We would like to assign to each cell a unique label $1 \leq \ell \leq n$, and we can do so by building a hash table whose keys are the values of $p \operatorname{div} c_1 r'$, which allows us to determine which cells are non-empty, after which we simply need to choose some label for each such cell. This takes $\mathcal{O}(n)$ expected time. For each point p we create a tuple $(\ell, t(p), p)$, where $t(p)$ is the time value of point p.

For each cell, we build a time-ordered array of the points within that cell. This is done by running radix sort on the tuples created in the previous step, sorting first by grid cell label, and then by time. Since the grid cell labels and time values are both at most n, the radix sort takes $\mathcal{O}(n)$ time.

For each such point p, we consider its $c_0(c_1+1)^d$ predecessors and the same number of successors in the time-ordered array, where c_0 is the packing constant from Lemma 1. Let q be such a predecessor or successor. If the distance between p and q is at most r, then p and q form a *candidate satisfying pair.*

We do the preceding steps $d+1$ times, where each time the cells are shifted by $v^{(j)}$ for $j \in \{0, 1, \ldots, d\}$ as defined in Lemma 3. We union together the results to build the full set of candidate satisfying pairs.

Lemma 4. *There are $\mathcal{O}(n)$ candidate satisfying pairs.*

Proof. Over the $d+1$ shifts, the total is upper-bounded by $(d+1) \cdot c_0(c_1+1)^d n = \mathcal{O}(n)$. □

Lemma 5. *If a time window contains a satisfying pair, then the time window must contain a candidate satisfying pair.*

Proof. Let pq be a satisfying pair for the window which is closest in terms of time order. From Lemma 3, there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ (which we will call p') and $q + v^{(j)}$ (which we will call q') are in the same grid cell of side length $c_1 r'$. Since p' and q' are active, all points between them in time order must also be active. No two points strictly between p' and q' can have distance smaller than r, for otherwise we would have a satisfying pair that is closer than pq in terms of time order. By Lemma 1 there are less than $c_0(c_1+1)^d$ points strictly between p' and q'. Therefore $p'q'$ must be among the candidate satisfying pairs. □

2.2.2 Reduction to 2D Dominance Range Emptiness

Now that the number of pairs we need to consider is reduced to $\mathcal{O}(n)$, we would like to store these pairs in a data structure to support efficient querying. Specifically, given a

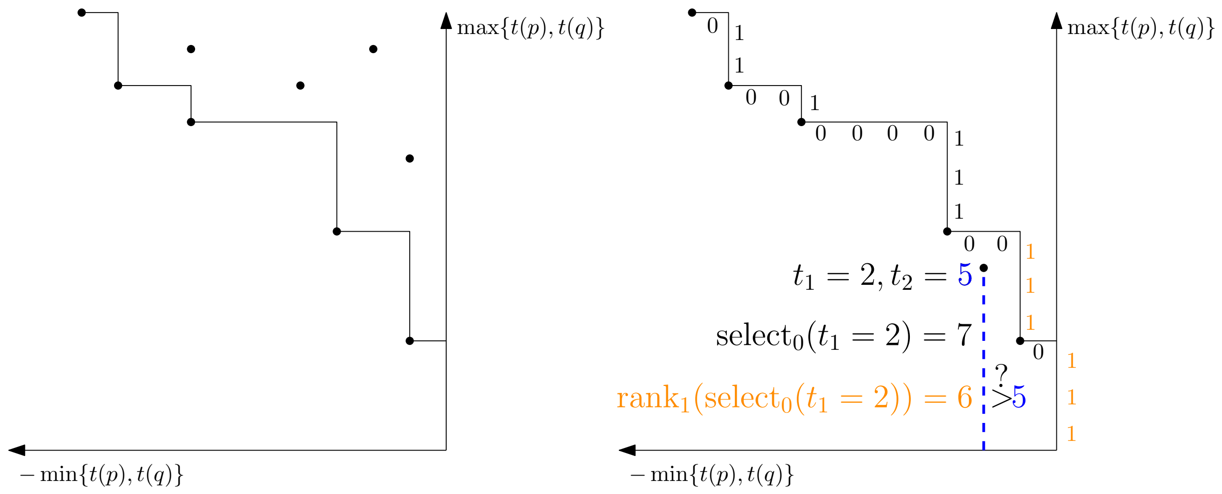

Figure 2.2: *Left*: the *staircase* of all candidate satisfying pairs. *Right*: the staircase can be stored in a succinct rank/select data structure which can be used to answer dominance range emptiness queries.

query window $[t_1, t_2]$ we wish to determine if there exists a candidate satisfying pair pq such that $t_1 \leq \min\{t(p), t(q)\}$ and $t_2 \geq \max\{t(p), t(q)\}$.

Consider each candidate satisfying pair as a point in 2 dimensions with coordinates $(-\min\{t(p), t(q)\}, \max\{t(p), t(q)\})$. Our query problem is equivalent to determining whether the quadrant $(-\infty, -t_1] \times (-\infty, t_2]$ contains any of these points. This is exactly the *2D dominance range emptiness* problem (the interested reader may refer to Section A.8 of Appendix A for more details). This problem can be solved by computing the *minima* of the 2D point set and testing whether the query point is above or below the *staircase* formed by the minima. Computing the minima of $\mathcal{O}(n)$ points takes $\mathcal{O}(n)$ time by a standard sweep-line algorithm, assuming that the x-coordinates (which are time values) have been pre-sorted. Since the x-coordinates are in $\{0, \dots, n-1\}$, pre-sorting takes $\mathcal{O}(n)$ time.

We can use an array to store the y-value of the staircase at every x-value; this requires $\mathcal{O}(n)$ words of space. With this, a query can then be answered in $\mathcal{O}(1)$ time.

2.2.3 Reduction to Rank/Select

Given a sequence of n bits, we can build a data structure using $n + o(n)$ bits of space, which supports reporting the number of 1s in the first i bits of the sequence (which we call $\text{rank}_1(i)$), and finding the jth 0 in the sequence (which we call $\text{select}_0(j)$) in $\mathcal{O}(1)$

time [Cla98]. The interested reader may refer to Section A.2 of Appendix A for more details.

We can use this data structure to reduce the space while keeping the same time bounds. Consider the staircase as a monotone chain (after negating the x-coordinates) through the $n \times n$ grid from the origin to $(n-1, n-1)$. This grid is effectively a plot with start time on the x-axis, and end time on the y-axis. We can encode a monotone chain as a sequence of $2n$ bits. Starting at the origin, whenever the chain moves vertically from end time i to $i+1$, we store a 1 bit. Similarly, whenever the chain moves horizontally, we store a 0 bit. The answer to the query is yes if and only if $t_2 \geq \mathrm{rank}_1(\mathrm{select}_0(t_1))$ (see Figure 2.2).

We have thus proven the following result:

Theorem 6. *The decision problem version of the time-windowed closest pair problem in any fixed dimension can be solved in $\mathcal{O}(1)$ time using $2n + o(n)$ bits of space and $\mathcal{O}(n)$ expected preprocessing time in the word-RAM model.*

2.3 Closest Pair

To solve the original time-windowed closest pair problem, the main new idea is to replace shifted grids with shifted balanced quadtrees. For each point outside of the centroid cell, we consider a constant number of its time-order predecessors and successors within the centroid cell. We then recurse separately on the points inside and outside of the centroid cell. This divide-and-conquer approach gives us $\mathcal{O}(n \log n)$ candidate pairs. From there, we reduce the problem to a 2-dimensional dominance range minimum problem.

2.3.1 Computing Candidate Pairs

We describe our algorithm to generate candidate pairs recursively. We first compute the centroid cell B of the given point set P. Define the set of neighbors $N(p)$ of a point p as its $c_0(2c_1+1)^d$ time-order predecessors and successors within the centroid cell B, where c_0 is the packing constant from Lemma 1 and c_1 is the shifting constant from Lemma 3. For each point p outside of the centroid, we consider each pair pq for $q \in N(p)$ as a *candidate pair*. We then recurse on $P \cap B$ and $P \setminus B$.

We run the preceding algorithm $d+1$ times, where each time the quadtrees are shifted by $v^{(j)}$ for $j \in \{0, 1, \ldots, d\}$ as defined in Lemma 3. We union together the results to build the full set of candidate pairs.

Lemma 7. *There are $\mathcal{O}(n \log n)$ candidate pairs.*

Proof. For each fixed shift, the number of candidate pairs is given by the recurrence $P(n) \leq P(n_1) + P(n_2) + c_0(2c_1 + 1)^d n$, where n_1 and n_2 are the number of points inside and outside of the centroid respectively.

Since $n_1 + n_2 = n$ and $n_1, n_2 \leq \alpha n$, the recurrence solves to $P(n) = \mathcal{O}(n \log n)$. □

Lemma 8. *The closest pair for any time window must be among the candidate pairs.*

Proof. Let pq be the closest pair in the window, with distance r. From Lemma 3, there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ (which we will call p') and $q + v^{(j)}$ (which we will call q') are in the same quadtree cell of side length $c_1 r'$ where r' is the smallest power of 2 greater than r.

There are 3 cases. Either p', q' are both inside or outside of the centroid cell B, or one is inside and the other is outside of B. The first 2 cases can be handled by induction. Now we are in case 3, so suppose q' is inside the centroid. (The case where p' is inside the centroid is symmetric.)

From Lemma 1, there are no more than $c_0(2c_1 + 1)^d$ active points in the centroid cell B, since B has side length at most $2c_1 r$. Since p and q are active during the time window, all points between them in time order must also be active. Therefore, there are fewer than $c_0(2c_1 + 1)^d$ points between p and q in time order, so $q \in N(p)$. □

2.3.2 Reduction to 2D Dominance Range Minimum

Now that the number of pairs we need to consider is reduced to $\mathcal{O}(n \log n)$, we would like to store these pairs in a data structure to support efficient querying. Specifically, given a query window $[t_1, t_2]$ we wish to find a candidate pair pq such that $t_1 \leq \min\{t(p), t(q)\}$ and $t_2 \geq \max\{t(p), t(q)\}$ while minimizing the distance $d(p, q)$.

Consider each candidate pair as a weighted point in 2 dimensions with coordinates $(-\min\{t(p), t(q)\}, \max\{t(p), t(q)\})$ and weight $d(p, q)$. Our query problem is equivalent to finding a point in the quadrant $(-\infty, -t_1] \times (-\infty, t_2]$ with the minimum weight. This is exactly the *2D dominance range minimum* problem, which we can solve by using standard techniques (the interested reader may refer to Section A.9 of Appendix A for more details). Namely, we first lift the 2D weighted points to 3D where the weights become z-coordinates. We compute the *staircase polyhedron* of the 3D point set, defined as the region of all points that are not dominated by any input point. Then a query can be answered by finding

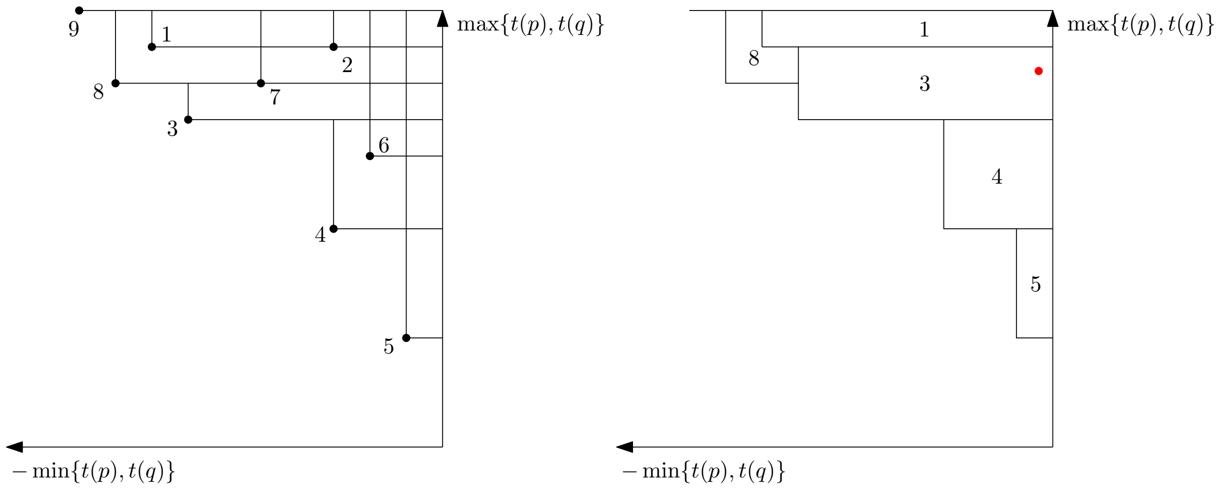

Figure 2.3: The problem of finding the closest pair reduces to dominance range minimum. *Left*: The candidate pairs with coordinates $(-\min\{t(p), t(q)\}, \max\{t(p), t(q)\})$ and weights $d(p, q)$ subdivide the plane into regions in which every point is dominated by a minimum candidate point which forms the closest pair p, q. *Right*: A time interval t_1, t_2 forms a query point with coordinates (t_1, t_2), which becomes a point location query. The regions are labeled with their closest pair p, q, here shown only as the distance between them $d(p, q)$.

the highest point on the staircase polyhedron at a given x- and y-coordinate. Computing the staircase polyhedron is related to the standard problem of computing the *minima* of the 3D point set and can be done by a standard plane sweep algorithm (the interested reader may refer to Section A.4 of Appendix A for more details). For a set of N points in 3D, the plane sweep algorithm takes $\mathcal{O}(N \log\log N)$ time using van Emde Boas trees, assuming that the x- and y-coordinates have been pre-sorted (the z-coordinates need not be pre-sorted). Since the x-coordinates are in $\{0, \ldots, n-1\}$, pre-sorting can be done in $\mathcal{O}(N + n)$ time using radix sort.

Finding the highest point of the staircase polyhedron (a monotone polyhedron in 3D) at a query x- and y-coordinate reduces to point location in a 2D subdivision of $\mathcal{O}(N)$ size, after projecting the faces onto the xy-plane (see Figure 2.3). We can use Chan's *planar orthogonal point location* structure [Cha13] as a black box to answer queries in $\mathcal{O}(\log\log N)$ time using $\mathcal{O}(N)$ space and $\mathcal{O}(N)$ preprocessing time (the interested reader may refer to Section A.7 of Appendix A for more details).

Setting $N = \mathcal{O}(n \log n)$ gives our main result:

Theorem 9. *Time-windowed closest pair queries in any fixed dimension can be answered in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(n\log n)$ words of space and $\mathcal{O}(n\log n\log\log n)$ preprocessing time in the word-RAM model.*

2.3.3 A Lower Bound on the Number of Candidate Pairs

As a final remark, we point out that any approach which stores all candidate pairs must use $\Omega(n\log n)$ space by proving the following observation.

Observation 10. *There exists a set of n points, where each point is associated with a time value, such that there are $\Omega(n\log n)$ distinct closest pairs over all possible time windows.*

Proof. Our construction works in one dimension. Suppose n is a power of 2. The base case $n = 2$ is trivial. To construct a set S of n points on a line, we first recursively construct a set S_1 of $n/2$ points, and duplicate S_1 to create S_2. We increase the labels of points in S_2 by $n/2$ and we shift the points along the line by δ for a sufficiently small $\delta > 0$ (less than half of the closest pair distance in S_1). Since the time labels of S_1 and S_2 are disjoint, any closest pair between points in S remains a closest pair for some time window. Symmetrically, we have the same closest pairs between points in S_2. In addition, for each time value $i \in \{1,\ldots,n/2\}$, the pair of points with time values i and $i+n/2$ is a closest pair for the time window $[i, i+n/2]$, because the pair has the smallest possible distance δ, and any other pair with distance δ has time values of the form j and $j+n/2$, which can't both lie inside $[i, i+n/2]$. This gives $n/2$ additional closest pairs. Therefore, the number of distinct closest pairs is given by the recurrence $C(n) \geq 2C(n/2) + n/2$, which solves to $C(n) = \Omega(n\log n)$.

(Note that this construction can alternatively be described without recursion using *bit-reversal permutations*, which for a fixed number of bits b are the numbers 0 to $2^b - 1$, with the order of their bits reversed. For example, for 2 bits, the numbers are $0, 1, 2, 3$, whose binary representations are $00, 01, 10, 11$. When we reverse the bits in this representation, we get $00, 10, 01, 11$, which corresponds to the sequence $0, 2, 1, 3$.) □

Chapter 3

Time-Windowed Pairwise Interaction Problems

In this chapter, we consider the following time-windowed problems:

2D and 3D Diameter Decision Given a set of n time-labeled points in $\mathbb{R}^2$ or $\mathbb{R}^3$, determine if there exist two points greater than unit distance apart, whose time values are within a query time window.

2D Orthogonal Segment Intersection Detection Given a set of n orthogonal (horizontal or vertical) time-labeled line segments in $\mathbb{R}^2$, determine if there are any intersections between segments whose time values are within a query time window.

The results presented in this chapter are joint work with Timothy M. Chan and appear in the Proceedings of the 32nd International Symposium on Computational Geometry (SoCG 2016) [CP16].

Recall from the introduction that the decision problems we consider in this chapter are on hereditary properties. Since for each time i, we can store the minimal j such that the time interval $[i, j]$ has the property $\mathcal{P}$. Therefore we focus only on bounding the preprocessing time.

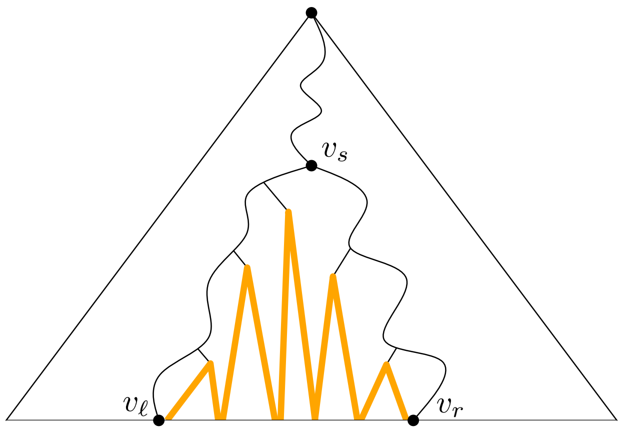

Figure 3.1: A range tree showing the path followed by a query for an interval $[\ell, r]$ which splits at v_s, and the **subtrees** which fall within the query range between leaves v_ℓ and v_r.

3.1 Preliminaries

3.1.1 Range Trees

A balanced binary search tree $\mathcal{T}$ is a *range tree* [Ben79, PS85] if each inner node v is labeled with the greatest element in the tree rooted at its left child. See Figure 3.1. This naturally decomposes any range $[l, r]$ of values into $\mathcal{O}(\log n)$ canonical subtrees by performing a binary search for both l and r until we reach their lowest common ancestor node v_s at which the search splits. The search continues leftwards to leaves v_l and v_r with values l and r respectively. Every right subtree on the path from v_s to v_l, and every left subtree on the path from v_s to v_l are within the range $[l, r]$. Further, a range tree can easily generalize to higher dimensions by storing at each node another range tree on another dimension of the data. Each extra dimension adds an $\mathcal{O}(\log n)$ factor to the query time. The space required to store a range tree on a set of points in $\mathbb{R}^d$ is $\mathcal{O}\big(n \log^{d-1} n\big)$.

3.1.2 Fractional Cascading

Fractional cascading is a data structure technique which allows us to perform iterated binary search on several sorted lists in $\mathcal{O}(\log n)$ total time [CG86a, CG86b].

In general, if we are searching through sorted list $A[k]$ followed by $A[k+1]$, we can speed up the process by fractional cascading. We do so by creating a list $B[k]$ which includes all the elements of $A[k]$ plus some fraction of the elements in $A[k+1]$, along with pointers into $B[k+1]$ created similarly. We call this process *fractionally cascading* from $A[k]$ to $A[k+1]$. Instead of searching $A[k]$, we search $B[k]$ and use the extra pointers to begin our next search in $B[k+1]$. In order to maintain the $\mathcal{O}(\log n)$ time bound and $\mathcal{O}(n)$ space bound, then each list can only be fractionally cascaded into a constant number of other lists, and only a constant number of other lists can be fractionally cascaded into each list. If we think of each list as a vertex in a graph, and fractionally cascading from list k to list $k+1$ creates a directed edge from the vertex representing k to the vertex representing $k+1$, then each vertex must have constant in and out-degree.

3.1.3 Range Successor

Given a set S of n values, the *successor* to a query value q is the smallest value $p \in S$ such that $q \leq p$. The *range successor* problem is to preprocess a set S of n values in order to efficiently find the successor of a query value within a given query range. This problem can be solved with $\mathcal{O}(n \log\log n)$ words of space and $\mathcal{O}(\log\log n)$ query time [Zho16].

3.2 Time-Windowed Range Successor

In this chapter, we focus on time-windowed decision problems for properties that deal with pairs. More precisely, given a symmetric relation $\mathcal{R} = S \times S$, we consider the property $\mathcal{P}$ that there exist $p, q \in S$ such that $(p, q) \in \mathcal{R}$. We call such properties *pairwise interaction properties*. If $(p, q) \in \mathcal{R}$, we say that p *interacts with* q; we also say that p is *in q's range*.

Examples of such problems include: diameter decision, for which two points interact if they are farther apart than unit distance; segment intersection detection, in which two segments interact with each other if they intersect; and closest pair decision, in which two points interact if they are nearer than unit distance. Note that such properties are *hereditary*.

Our approach is to reduce the time-windowed problem to the following data structure problem:

Definition 11. Let each object $s \in S$ have weight $w(s)$. In the *generalized range successor problem*, we want to preprocess S to find the *successor* of a query object q among the objects in q's range, that is, the object $p \in S$ that interacts with q with the smallest $w(p) > w(q)$.

The above is a generalization of the original 1D range successor problem where the objects are points in 1D and the objects' ranges are intervals.

To see how the above data structure problem can be used to solve the time-windowed pairwise interaction problem, we simply find the successor p_q of every $q \in S$ in the generalized range successor problem with weights equal to time values.

Observation 12. *A query window $[t_1, t_2]$ contains an interacting pair if and only if $[t_1, t_2]$ contains $[t(q), t(p_q)]$ for some $q \in S$.*

Proof. The "if" direction is trivial. For the "only if" direction, let p, q be an interacting pair in $[t_1, t_2]$ and assume that $t(q) \leq t(p)$ without loss of generality. Then $t(q) \leq t(p_q) \leq t(p)$ by definition of the successor p_q, and the claim follows. $\square$

Thus, the answer to a query window $[t_1, t_2]$ is yes if and only if the point $(t_1, -t_2)$ is dominated by some point $(t(p), -t(p_q))$. The time-windowed problem can then be solved by precomputing the *maxima* of the 2D point set $\{(t(p), -t(p_q)) \mid q \in S\}$, which takes linear time by a standard plane sweep after pre-sorting (recall that time values have been initially reduced to integers in $\{1, \ldots, n\}$ and can be trivially sorted in linear time). The running time is then dominated by the cost of computing the successors p_q for all $q \in S$. If we can solve the generalized range successor problem in $P(n)$ preprocessing time and $Q(n)$ query time, we can solve the corresponding time-windowed problem in $\mathcal{O}(P(n) + nQ(n))$ preprocessing time.

In the rest of this section, we can thus focus on solving the generalized range successor problem.

One approach is to first consider the decision version of the problem: deciding whether there exists an object p that lies in q's range and has weight $w(p)$ in the interval $[\ell, r]$ for $\ell = w(q)$ and a given value r. This problem can be solved using standard multi-level data structuring techniques: The primary structure is a 1D range tree on the weights (i.e., time values). See Figure 3.1. This naturally decomposes any interval $[\ell, r]$ of weights into $\mathcal{O}(\log n)$ canonical subtrees by performing a binary search for both ℓ and r until we reach their lowest common ancestor node v_s at which the search splits. The search continues leftward and rightward to leaves v_ℓ and v_r with values ℓ and r respectively. Every right subtree on the path from v_s to v_ℓ, and every left subtree on the path from v_s to v_r are within the interval $[\ell, r]$. At each node v, we store the subset S_v of all objects within its interval in a *secondary structure* for the original range searching problem—deciding whether there exists an object in S_v that lies in a query object q's range. A query for the decision problem can then be answered by making $\mathcal{O}(\log n)$ queries to the secondary structures. This increases the query time by a logarithmic factor.

Finally, we can reduce the generalized range successor problem to its decision problem by a binary search over all time values r. This increases the query time by a second logarithmic factor.

3.3 Avoiding Binary Search

In this section, we describe a still better algorithm that solves the generalized range successor problem without going through the decision problem, thereby removing one of the extra logarithmic factors caused by the binary search.

We first find the leaf node v storing q in $\mathcal{O}(\log n)$ time. To answer a successor query for q, we proceed in two phases. (See Figure 3.2.)

- In the first (i.e., "up") phase, we walk upward from v towards the root. Each time our search follows a parent pointer from a left child, we query the secondary structure at the right child to see if there exists an object stored at the right child that is in q's range. If no, we continue upward. If yes, the answer is in the subtree at the right child and we proceed to the second phase starting at this node.
- In the second (i.e., "down") phase, we walk downward from the current node to a leaf. Each time our search descends from a node, we query the secondary structure at the left child to see if there exists an object stored at the left child that is in q's range. If no, the answer is in the right subtree and we descend right. Otherwise, we descend left.

This algorithm makes $\mathcal{O}(\log n)$ queries in the secondary structures. We next apply this algorithm to specific time-windowed pairwise interaction problems.

3.4 2D Diameter Decision

For the application to 2D diameter decision, our set of objects S is composed of points in $\mathbb{R}^2$, and p, q interact if and only if $d(p, q) > 1$. In other words, q's range is the complement of a unit disk.

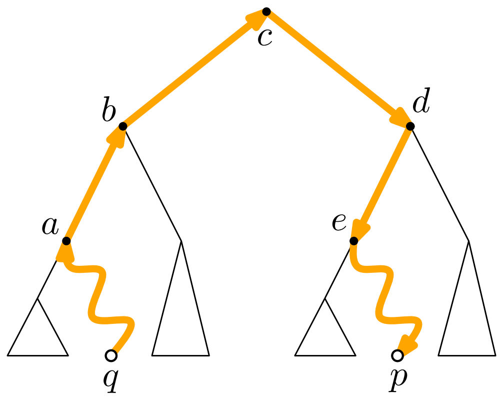

Figure 3.2: A search path finding the successor p of a point q is shown in **bold**. The search is divided into an “up” phase followed by a “down” phase.

The secondary structure at a node v needs to handle the following type of query: decide whether a query point q has greater than unit distance from some point in S_v, that is, decide whether q lies outside the intersection D_v of all unit disks centered at the points of S_v (see Figure 3.3). We store the disks whose boundaries form the boundary of the unit-disk intersection D_v, along with the sorted list X_v of the x-coordinates of the vertices of D_v.

Since we can merge two unit-disk intersections in linear time, we can build the secondary structures at all nodes of the range tree bottom-up in $P(n) = \mathcal{O}(n \log n)$ time (the interested reader may refer to Section A.11 of Appendix A for more details).

We can determine if a query point q is inside or outside of a unit-disk intersection by first performing binary search for the x-coordinate of q in X_v, which identifies a disk D whose boundary bounds the intersection, then checking whether $q \in D$. This takes $\mathcal{O}(\log n)$ time. Since the algorithm in Section 3.3 requires $\mathcal{O}(\log n)$ queries in the secondary structures, the overall query time is $Q(n) = \mathcal{O}\big(\log^2 n\big)$.

We can use *fractional cascading* (see Section 3.1.2 for more details) to speed up the algorithm further. Recall that the algorithm in Section 3.3 is divided into two phases.

- For the “up” phase, we first move the list X_v of each right child v to its parent. We pass a fraction of the elements of the list at each node to the lists at both children during preprocessing. This way, we can determine where the x-coordinate of q is in the list of the parent from where it is in the list of the child in $\mathcal{O}(1)$ time. We can then answer all $\mathcal{O}(\log n)$ queries in the secondary structures during the “up” phase in $\mathcal{O}(\log n)$ overall time, after an initial binary search at the leaf in $\mathcal{O}(\log n)$ time.

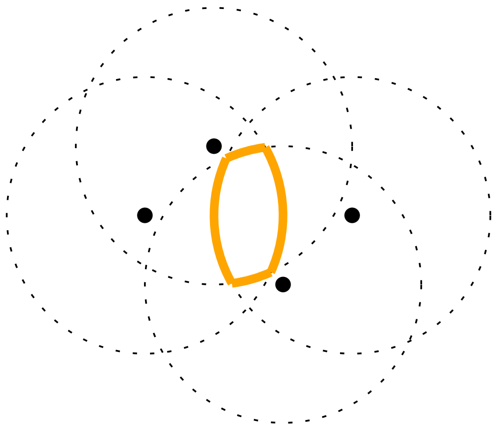
Figure 3.3: The boundaries of unit disks centered at four points in $\mathbb{R}^2$. The boundary of the unit-disk intersection is shown in **bold**.

- For the "down" phase, we pass a fraction of the elements of the list at each node to the list at its parent during preprocessing. This way, we can determine where the x-coordinate of q is in the list of a child from where it is in the list of its parent in $\mathcal{O}(1)$ time. We can then answer all $\mathcal{O}(\log n)$ queries in the secondary structures during the "down" phase in $\mathcal{O}(\log n)$ overall time, after an initial binary search in $\mathcal{O}(\log n)$ time.

We conclude that a generalized range successor query in this setting can be answered in $Q(n) = \mathcal{O}(\log n)$ time. This gives us the following result.

Theorem 13. *We can preprocess for the time-windowed 2D diameter decision problem in $\mathcal{O}(n \log n)$ time.*

3.5 3D Diameter Decision

In 3D, a query in the secondary structure at node v becomes deciding whether the query point q lies outside the intersection D_v of unit balls centered at the points of S_v. We can compute the intersection of unit-balls in $\mathcal{O}(n \log n)$ time [AGR94] (the interested reader may refer to Section A.12 of Appendix A for more details).

We begin by dividing D_v into two parts, by finding its extreme points in both x and y dimensions, which are contained in a unique plane. The boundary above this plane, with respect to z is the *upper* boundary of D_v, and the boundary below is the *lower* boundary.

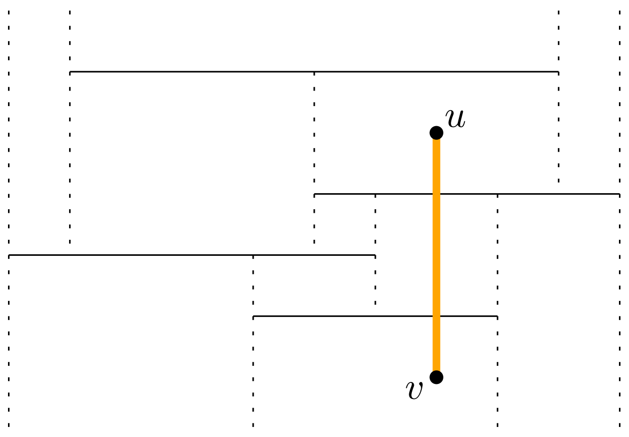

Figure 3.4: A set of horizontal segments is shown in solid lines, with their vertical decomposition shown in dotted lines. A query vertical segment is shown in **bold**, whose endpoints are in different cells.

We store the xy-projection of the upper and lower (with respect to z-coordinate) boundary of D_v in a planar point location structure. We can build the secondary structures at all nodes of the range tree in $\mathcal{O}(n \log n)$ time per level, and thus $P(n) = \mathcal{O}(n \log^2 n)$ total time. (Unlike in 2D, it is not clear if we could speed up the building time by linear-time merging.)

Given point q, a query in a secondary structure reduces to planar point location for the xy-projection of q and takes $\mathcal{O}(\log n)$ time (the interested reader may refer to Section A.6 of Appendix A for more details). Since the algorithm in Section 3.3 requires $\mathcal{O}(\log n)$ queries in the secondary structures, the overall query time is $Q(n) = \mathcal{O}(\log^2 n)$. (Unlike in 2D, we cannot apply fractional cascading to speed up the algorithm.) This gives us the following result.

Theorem 14. *We can preprocess for the time-windowed 3D diameter decision problem in* $\mathcal{O}(n \log^2 n)$ *time.*

3.6 Orthogonal Segment Intersection Detection

For the application to 2D orthogonal segment intersection detection, our set of objects S is composed of vertical and horizontal line segments in $\mathbb{R}^2$, and p, q interact if and only if they intersect (see Figure 3.4).

Without loss of generality, we assume that all x- and y-coordinates are given as integers from 1 to $\mathcal{O}(n)$, for otherwise we can replace coordinate values with their rank during preprocessing. The query time only increases by $\mathcal{O}(1)$ predecessor searches on the coordinate values, costing no more than $\mathcal{O}(\log n)$ time.

The secondary structure at a node v now needs to handle the following type of query: decide whether a query segment q intersects some segment in S_v. Without loss of generality, assume that q is vertical and the segments in S_v are horizontal. We store the vertical decomposition VD_v (also called the trapezoidal decomposition) in a planar point location structure (the interested reader may refer to Section A.5 of Appendix A for more details).

Since we can compute the vertical decomposition VD_v in $\mathcal{O}(|S_v| \log\log |S_v|)$ time by a standard plane sweep with van Emde Boas trees [Cha13], we can build the secondary structures at all nodes of the range tree in $P(n) = \mathcal{O}(n \log n \log\log n)$ time.

Given vertical segment q, a query in the secondary structure at node v requires testing whether both endpoints of q lie in the same cell in VD_v, which reduces to two planar point location queries. Since the subdivision is orthogonal, we can apply Chan's orthogonal point location structure, which achieves $\mathcal{O}(\log\log U)$ query time when coordinates are integers from $\{1, \ldots, U\}$—recall that coordinate values have been initially reduced to integers bounded by $U = \mathcal{O}(n)$. Since the algorithm in Section 3.3 requires $\mathcal{O}(\log n)$ queries in the secondary structures, the overall query time is $Q(n) = \mathcal{O}(\log n \log\log n)$. This gives us the following result.

Theorem 15. *We can preprocess for the time-windowed 2D orthogonal intersection detection problem in $\mathcal{O}(n \log n \log\log n)$ time.*

Remark. An open problem is to remove the extra $\log\log n$ factor. Perhaps the techniques from [CLP11] for the 4D offline dominance searching problem may be relevant.

Chapter 4

Time-Windowed Convex Hull Decision Problems

In this chapter, we consider the following time-windowed problems:

2D Convex Hull Area Decision Given a set of n time-labeled points in $\mathbb{R}^2$, determine whether the convex hull of points within a query time window has greater than unit area.

2D Width Decision Given a set of n time-labeled points in $\mathbb{R}^2$, we want to determine whether the points within a query time window have greater than unit width.

The results presented in this chapter are joint work with Timothy M. Chan and appear in the Proceedings of the 32nd International Symposium on Computational Geometry (SoCG 2016) [CP16].

As in Chapter 3, the decision problems we consider in this chapter are on hereditary properties and therefore we focus only on bounding the preprocessing time.

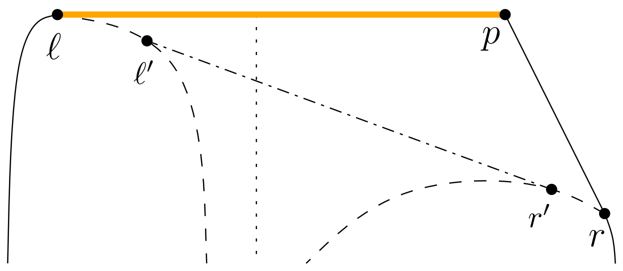

Figure 4.1: Newly added point p to the right of the dotted median line causes a new bridge to vertex ℓ. Vertex r is the point of tangency of the line through p tangent to the right side of the right upper hull. The previous bridge between ℓ' and r' is shown as a dash dotted line. Computing the new **bold** bridge requires walking from r' to r. If point p were instead being deleted, computing the new bridge would require walking from ℓ to ℓ' and r to r'.

4.1 Preliminaries

4.1.1 Hull-Trees

A *hull-tree* [OvL81, Cha85, HS92] is a binary tree whose root node stores the (upper or lower) hull edge (which we call the *bridge*) which crosses the median vertical line. A node's left and right children are the hull trees on all points left and right of the median, respectively (see Figure 4.1). The original paper by Overmars and van Leeuwen details how to insert, delete, and query in $\mathcal{O}(\log^2 n)$ time per operation [OvL81].

Hershberger and Suri show how to build a modified hull-tree in $\mathcal{O}(n \log n)$ time which supports deletion in amortized $\mathcal{O}(\log n)$ time [HS92]. They do so by building the tree on all points S, but keeping track of the points P that have not yet been deleted. They maintain a subtree $T(P)$ by storing at each node v of the hull-tree a list $\text{chain}(v)$ that stores all points on the convex hull of all $p \in P$ in the subtree rooted at v, but aren't on the convex hull of subtrees of ancestors of v. In other words, this stores the convex hull of the entire (not yet deleted) point set at the root, and each child stores the chain of its convex hull which is under the bridge edge which is stored at the root. Additionally, they store $\tan(v)$, a pair of pointers into $\text{chain}(v)$ that point to the end points of the bridge edge at v. These modifications take $\mathcal{O}(n)$ extra pointers, therefore they do not increase the space usage of the tree asymptotically.

The following observation is due to Arie Tamir [Tam88, page 394, final paragraph]:

Observation 16 (Tamir)**.** *There are $\mathcal{O}(n \log n)$ distinct edges created or destroyed in the upper hull of a set of points in $\mathbb{R}^2$ over an arbitrary sequence of n insertions or deletions.*

Proof. Consider the vertical line at the median x-coordinate. At any time, there is just one hull edge that crosses the vertical line (called the *bridge*), and thus the number of possible bridges over time is $\mathcal{O}(n)$. Thus, the number of distinct edges that appear on the upper hull over time satisfies the recurrence

$$E(n) \;=\; 2E(n/2) + \mathcal{O}(n)\,,$$

implying that $E(n) = \mathcal{O}(n \log n)$. □

4.2 FIFO Update Sequence Approach

In the previous section, we have presented an approach to building data structures to solve time-windowed pairwise interaction problems, but the 2D width and the 2D convex hull area decision problems, for instance, cannot be expressed in terms of a pairwise interaction property.

As mentioned in the introduction, both problems are on hereditary properties. The most obvious approach to solve a problem on a hereditary property is to use a dynamic data structure directly, inserting each object in order until $\mathcal{P}$ is satisfied, then deleting each object in the same order until $\mathcal{P}$ is no longer satisfied, and repeating. By storing for each $i \in \{1, \ldots n\}$ the smallest j for which $\{s_i, \ldots, s_j\}$ satisfies $\mathcal{P}$, we can answer queries for the time-windowed problem. However, this approach does not seem to yield efficient solutions in some settings. For example, for the 2D width decision problem, we would need a fully dynamic data structure for 2D width decision, but the best result to date has near $\sqrt{n}$ update time [Cha01b]. Agarwal and Sharir [AS91] gave a dynamic data structure for 2D width decision with polylogarithmic update time but only for *offline* update sequences; their data structure does not seem to work in our application when we do not know a priori in what order the deletions are intermixed with the insertions.

Both the 2D width and 2D convex hull area problem are about the convex hull. There exist sequences of n updates to the convex hull in the plane which cause $\mathcal{O}(n^2)$ many structural changes. For example, consider the case of inserting a point which causes $\mathcal{O}(n)$ points to be no longer on the convex hull, then deleting and re-inserting the same point n times. However, in our application points are deleted in the same order as they are inserted, so this particular example cannot occur.

Restricted update sequences on the dynamic convex hull have been studied before. The insertion-only case was studied by Preparata [Pre79]. The deletion-only case was studied by Chazelle [Cha85], and Hershberger and Suri [HS91]. Random update sequences were studied by Mulmuley [Mul91] and Schwarzkopf [Sch91].

We call an update sequence in which objects are inserted and deleted in the same order a *first-in-first-out (FIFO) update sequence*, which to the best of our knowledge has not been studied before. We prove a combinatorial lemma, stating that for such sequences the number of structural changes to the convex hull is always near linear.

Lemma 17. *The number of structural changes to the upper hull of a set of points in $\mathbb{R}^2$ over n FIFO updates is at most $\mathcal{O}(n \log n)$.*[1]

Lemma 17 follows immediately by combining Observation 16 with another observation about FIFO update sequences:

Observation 18. *For a FIFO update sequence, once an edge uv has been removed from the upper hull by the insertion of a point w, uv can never again be an edge of the upper hull.*

Proof. Since w is above the line through uv, we know that uv cannot be an edge of the upper hull while w is alive. But since w was inserted after u and v, by the FIFO property w must be deleted after u and v, and therefore uv can never again be an upper hull edge. □

4.2.1 2D Width Decision

We can immediately apply Lemma 17 to solve the time-windowed 2D width decision problem, by using Eppstein's dynamic 2D width data structure [Epp00] as a black box. Eppstein's algorithm maintains the width in time $\mathcal{O}(k \cdot f(n) \cdot \log n)$ where k is the number of structural changes to the convex hull, and $f(n)$ is the time to solve the dynamic 3D convex hull problem (more precisely, answer gift-wrapping queries and perform updates for a 3D point set), which takes $\mathcal{O}\left(\log^5 n\right)$ amortized time [Cha10, KMR+16] (the interested reader may refer to Section A.13 of Appendix A for more information).

This proves the following result.

Theorem 19. *We can preprocess for the time-windowed 2D width decision problem in $\mathcal{O}\left(n \log^7 n\right)$ time.*[2]

[1] John Hershberger [Her16] has subsequently discovered a simple proof that the number of changes to the convex hull over n FIFO updates is bounded by $\mathcal{O}(n)$ which this footnote is too small to contain.

[2] [Her16] reduces this to $\mathcal{O}\left(n \log^6 n\right)$ time.

4.2.2 2D Convex Hull Area Decision

For the time-windowed 2D convex hull area decision problem, we can now directly apply known fully dynamic convex hull data structures [OvL81, Cha01a, BJ02], most of which can be modified to maintain the area. For example, Brodal and Jacob's (extremely complicated) data structure [BJ02] can maintain the convex hull and its area in $\mathcal{O}(k \cdot \log n)$ amortized time, where k is the number of structural changes to the convex hull. This would imply an $\mathcal{O}\big(n \log^2 n\big)$-time algorithm, which is worse than Bokal *et al.*'s result [BCE15].

We show how to reduce this to $\mathcal{O}(n\alpha(n) \log n)$ by directly adapting a simpler known dynamic convex hull data structure, namely Overmars and van Leeuwen's *hull tree* [OvL81], and carefully analyzing it for FIFO sequences.

Lemma 20. *We can maintain the convex hull and its area for a 2D set of n points under FIFO updates in $\mathcal{O}(n\alpha(n) \log n)$ total time.*

Proof. It suffices to maintain the upper hull and the area above it inside a sufficiently large bounding box, since we can similarly maintain the lower hull and the area below it, and subtract the areas from the bounding box.

Here, we assume that the x-coordinates of all n points are known in advance, which is true in our application (the assumption can be removed by extra steps to balance the hull tree, for example, via tree rotations [OvL81]). At each node, we store the area above the upper hull. Pointer structures can be set up to let us traverse the upper hull at any node of the tree [OvL81, HS92].

The following definitions will be helpful: Consider the upper hull at a tree node. When we insert a point u which causes a polygonal chain $v_1 v_2 \cdots v_k$ to disappear from the upper hull, we say that u *kills* v_i, and that (u, v_i) forms a *killing pair*, for each $i = 1, 2, \ldots, k$. (For technical reasons, we allow $i = 1$ and $i = k$ in the definition, counterintuitively.) Symmetrically, if we delete a point u which causes a polygonal chain $v_1 v_2 \cdots v_k$ to appear in the upper hull, we say that u *revives* v_i, and that (u, v_i) forms a *revival pair*, for each $i = 1, 2, \ldots, k$.

Deletion. Consider the deletion of a point p at a node of the hull tree. Without loss of generality, suppose that p is to the right of the median. We first recursively delete p in the right subtree. If p is not an endpoint of the bridge of the node, then we are done. Suppose that p was an endpoint of the bridge of the node. We need to compute a new bridge. Overmars and van Leeuwen [OvL81] originally proposed a binary search, but we

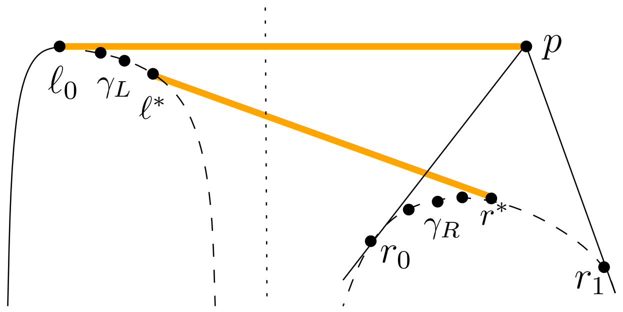

Figure 4.2: Deletion of p causes the old bridge (ℓ_0, p) to change to the new bridge (ℓ^*, r^*), both shown in **bold**. Computing the new bridge requires walking from ℓ_0 to ℓ^* and r_0 to r^*. If point p is instead being inserted, computing the new bridge requires walking from ℓ^* to ℓ_0. Any pair (p, ℓ) with $\ell \in \gamma_L$ or pair (p, r) with $r \in \gamma_R$ is a revival or killing pair at some node of the hull tree.

will use a linear search instead (inspired by the variants of hull trees by Chazelle [Cha85] and Hershberger and Suri [HS92] for deletion-only sequences). Specifically, let ℓ_0 be the left endpoint of the old bridge, and let r_0 and r_1 be the predecessor and successor of p in the old right upper hull respectively. (See Figure 4.2.) A simple rightward linear search from ℓ_0 and r_0 can find the new bridge (ℓ^*, r^*) in $\mathcal{O}(|\gamma_L| + |\gamma_R|)$ time, where γ_L denotes the subchain from ℓ_0 to ℓ^* in the left upper hull, and γ_R denotes the subchain from r_0 to r^* in the right upper hull. (Note that ℓ^* must be right of ℓ_0, and r^* must be right of r_0.) The change in area at the current node can be computed in $\mathcal{O}(|\gamma_L| + |\gamma_R|)$ time (it is the area of the polygon with vertices $\ell_0\gamma_L\ell^*r^*p$, plus the change in area at the right child, minus the area of the polygon with vertices $r_0\gamma_R r^* p$).

To account for the $\mathcal{O}(|\gamma_L|)$ cost, observe that for each $\ell \in \gamma_L$, (p, ℓ) is a revival pair for the upper hull at the current node. We charge one unit to each such pair (p, ℓ). Note that each pair is charged at most once during the entire algorithm.

To account for the $\mathcal{O}(|\gamma_R|)$ cost, observe that for each $r \in \gamma_R$, (p, r) is a revival pair for the upper hull at the right child. We charge one unit to each such pair (p, r). If (p, r) is charged, then r lies strictly below the upper hull at the current node and cannot be charged again at an ancestor. Thus, each pair is charged at most once this way.

Insertion. Consider the insertion of a point p at a node of the hull tree. Without loss of generality, suppose that p is to the right of the median. We first recursively insert p in the right subtree. We need to compute the new bridge (if it changes). We can just mimick the deletion algorithm in reverse. In fact, the details are a little simpler: a linear search from ℓ^* can find ℓ_0, the left endpoint of the new bridge (see Figure 4.2) in $\mathcal{O}(|\gamma_L|)$ time. The change in the area can again be computed in $\mathcal{O}(|\gamma_L| + |\gamma_R|)$ time. We can account for the cost again by charging, this time, to killing instead of revival pairs.

Total time. The total cost over all updates is proportional to the number of charges, which is bounded by $\overline{K}(n)$, the worst-case number of distinct pairs (u, v) such that (u, v) is a killing/revival pair for the upper hull of at least one node of the hull tree, over all sets of n points. In the next subsection, we prove that $\overline{K}(n) = \mathcal{O}(n\alpha(n) \log n)$ (Lemma 23), which would then imply an $\mathcal{O}(n\alpha(n) \log n)$ time bound. □

Theorem 21. *We can preprocess for the time-windowed 2D convex hull area decision problem in* $\mathcal{O}(n\alpha(n) \log n)$ *time.*[3]

4.2.3 Bounding the Number of Killing/Revival Pairs

One ingredient is still missing: a proof that $\overline{K}(n) = \mathcal{O}(n\alpha(n) \log n)$. Naively we could bound the number of killing/revival pairs by the number of structural changes to the upper hull at each node, and applying Lemma 17 would give us the recurrence $\overline{K}(n) = 2\overline{K}(n/2) + \mathcal{O}(n \log n)$, implying a weaker bound $\overline{K}(n) = \mathcal{O}\big(n \log^2 n\big)$.

We propose a different combinatorial argument to bound $\overline{K}(n)$. Our approach contains a nice application of *Davenport-Schinzel (DS) sequences* [SA95]. Recall that a DS sequence *of order 3* is a sequence Σ of characters $s_1, s_2, \ldots$ from an alphabet A such that no two consecutive characters are the same and for any two characters $a, b \in A$, the alternating sequence a, b, a, b, a of length 5 does not appear as a subsequence anywhere in Σ, whether contiguous or not. Hart and Sharir [HS86, SA95] proved that an order-3 DS sequence over an alphabet of size n has length at most $\mathcal{O}(n\alpha(n))$ (the interested reader may refer to Section A.10 of Appendix A for more details).

DS sequences occur often in computational geometry, such as in bounding the combinatorial complexity of the lower envelope of line segments. Surprisingly in our case, we do not relate our problem to lower envelopes or other substructures in arrangements. Rather, we relate directly to DS sequences.

It suffices to bound the number of killing pairs, since revival pairs are symmetric, by reversing time. To this end, we first concentrate on a special kind of killing pairs: for the upper hull at a fixed node of the hull tree, a *bridge killing pair* is a killing pair (u, v) where u and v lie on opposite sides of the median vertical line.

[3] [Her16] reduces this to $\mathcal{O}(n \log n)$ time.

Lemma 22. *For the upper hull at a fixed node, the number of bridge killing pairs is at most $\mathcal{O}(n\alpha(n))$ for any FIFO update sequence.*

Proof. By symmetry, it suffices to count killing pairs (u, v) where u is to the right and v is to the left of the median vertical line. Take the sequence of all such killing pairs $(u_1, v_1), \ldots, (u_n, v_m)$ ordered by time of u_i, where in case of ties, simultaneous killings are ordered in decreasing x-order of v_i. Define the sequence Σ to be $v_1, \ldots, v_m$. We claim that Σ cannot have any alternating subsequence of length 5.

Assume that Σ contains a length-5 alternating subsequence of killed points, either $\ldots, a, \ldots, b, \ldots, a, \ldots, b, \ldots, a, \ldots$ or $\ldots, b, \ldots, a, \ldots, b, \ldots, a, \ldots, b, \ldots$. Without loss of generality, assume that a is to the left of b. In either of the above cases, the subsequence $\ldots, b, \ldots, a, \ldots, b, \ldots, a, \ldots$ of length 4 occurs in Σ.

Consider the time in this length-4 subsequence when a is killed ($b\underline{a}ba$) and let the vertex which kills it be u (see Figure 4.3). At this time, b must exist, because it will be killed later; furthermore, b is on or below the current upper hull and t othe right of a, and so lies below the line segment $\overline{au}$. Now, fast forward to the time in the subsequence when b is next killed ($ba\underline{b}a$). Note that this must be at a different time, because if u kills a and b at the same time, b would be placed before a in Σ by our tie-breaking rule. At this new time, a and u must both exist, because a will be killed later again, and u was inserted after b and will be deleted after b by definition of FIFO sequences. But b is below $\overline{au}$ and cannot appear on the upper hull and cannot be killed at this time: a contradiction. This completes the proof of the claim.

The sequence Σ may still have identical consecutive characters, but a repeated pair can occur only "between" two different insertion events and there are at most n insertion events. After removal of $\mathcal{O}(n)$ repeated characters, Σ thus becomes a DS sequence of order 3 and by the known upper bound has length at most $\mathcal{O}(n\alpha(n))$ [SA95]. The lemma follows. □

Lemma 23. *The number of distinct killing/revival pairs over all nodes in the hull tree satisfies $\overline{K}(n) = \mathcal{O}(n\alpha(n)\log n)$ for any FIFO update sequence.*

Proof. By Lemma 22, there are $\mathcal{O}(n\alpha(n))$ bridge killing pairs for the upper hull at the root node. The remaining killing pairs are killing pairs at nodes of the left subtree and nodes of the right subtree. We thus obtain the recurrence

$$K(n) \;=\; 2K(n/2) + \mathcal{O}(n\alpha(n))\,,$$

implying that $K(n) = \mathcal{O}(n\alpha(n)\log n)$. □

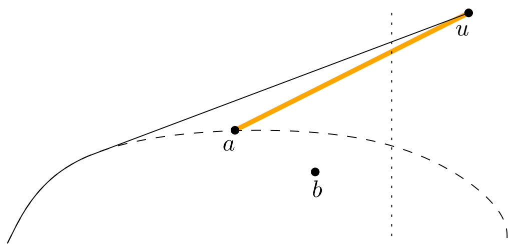

Figure 4.3: u kills a. The dotted line is the median vertical line and the dashed line was part of the convex hull before u was inserted. The point b must be below the **bold** line segment $\overline{au}$.

Chapter 5

Conclusion

We have presented three approaches to building data structures which answer queries related to time-windowed geometric problems. The first uses grids and quadtrees to reduce the problem space before reduction to point location. The second approach is to solve the related problem of time-windowed range successor. Finally, the third approach is to reduce to operations on a dynamic data structure. These approaches yield a few optimal solutions to the problems considered herein, but largely only lay the foundations of the topic of time-windowed geometry.

5.1 Future Work

The following time-windowed problems considered in this thesis are open for improvement:

1. *Closest pair*: Can we maintain the $\mathcal{O}(\log\log n)$ query time while improving the preprocessing time from $\mathcal{O}(n\log n\log\log n)$ to $\mathcal{O}(n\log n)$ and/or improving the space from $\mathcal{O}(n\log n)$ to $\mathcal{O}(n)$?
2. *3D diameter decision*: Can we improve the preprocessing time from $\mathcal{O}\big(n\log^2 n\big)$ to $\mathcal{O}(n\log n)$?
3. *2D orthogonal segment intersection detection*: Can we improve the preprocessing time from $\mathcal{O}(n\log n\log\log n)$ to $\mathcal{O}(n\log n)$?
4. *2D width decision*: Can we further reduce the preprocessing time from $\mathcal{O}\big(n\log^6 n\big)$ to $\mathcal{O}\big(n\log^5 n\big)$?

In addition to the possible improvements listed above, there are also the exact versions of many of the decision problems considered herein. These include:

1. *2D and 3D diameter*: Preprocess a set of n time-labeled points in $\mathbb{R}^2$ or $\mathbb{R}^3$ in order to quickly determine the pair of points p, q within a query time window such that the distance between p and q is maximal among all points within the query time window.
2. *2D orthogonal intersection reporting*: Preprocess a set of n orthogonal (horizontal or vertical) time-labeled line segments in $\mathbb{R}^2$, in order to quickly report the intersecting line segments within a query time window.
3. *2D convex hull area*: Preprocess a set of n time-labeled points in $\mathbb{R}^2$ in order to quickly compute the area of the convex hull of all points within a query time window.
4. *2D width*: Preprocess a set of n time-labeled points in $\mathbb{R}^2$ in order to quickly compute the width of the all points within a query time window.

Many of these problems are related to range search, which has interesting variations such as counting. Related counting problems include:

1. *close/far pair counting*: Preprocess a set of n time-labeled points in $\mathbb{R}^d$ in order to quickly determine the number of pairs of points closer than or farther than a query distance r apart.
2. *2D orthogonal intersection counting*: Preprocess a set of n orthogonal (horizontal or vertical) time-labeled line segments in $\mathbb{R}^2$, in order to quickly report the number of intersecting line segments within a query time window.
3. *2D convex hull counting*: Preprocess a set of n time-labeled points in $\mathbb{R}^2$ in order to quickly compute the number of points on the convex hull of all points within a query time window.

We have considered closest pair and diameter, but we could instead report the k-closest pairs or k-farthest pairs.

Many of the problems considered herein are limited to 2D and 3D, and the techniques used to solve those problems don't extend trivially to higher dimensions, leaving another avenue of extension.

We have only considered discrete points in time, but a natural extension would be to consider continuous time which seems intuitively very similar to the *kinetic* setting, in which objects are continually moving.

Finally, we have only considered *static* time-windowed problems. In other words, we have only considered problems in which all of the input is available to us before any queries are given. If we consider *dynamic* versions of these problems, we need to build a structure which supports efficient updates. It is natural to consider the points being added in time order, in other words the ith geometric object inserted has time i. If we support insertions at arbitrary times, this is somewhat similar to *retroactive* data structures [DIL07].

References

[Afs14] Peyman Afshani. Fast computation of output-sensitive maxima in a word ram. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1414–1423, 2014.

[AGR94] Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 683–694, 1994.

[AM95] Pankaj K. Agarwal and Jirı Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

[AMN$^+$98] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[AS91] Pankaj K. Agarwal and Micha Sharir. Off-line dynamic maintenance of the width of a planar point set. *Computational Geometry: Theory and Applications*, 1:65–78, 1991.

[BCE15] Drago Bokal, Sergio Cabello, and David Eppstein. Finding all maximal subsequences with hereditary properties. In *Proceedings of the 31st International Symposium on Computational Geometry (SoCG)*, pages 240–254, 2015.

[BDG$^+$14] Michael J. Bannister, William E. Devanny, Michael T. Goodrich, Joseph A. Simons, and Lowell Trott. Windows into geometric events: Data structures for time-windowed querying of temporal point sets. In *Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG)*, pages 11–19, 2014.

[Ben79] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[Ber93] Marshall Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, 1993.

[BET99] Marshall Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry & Applications*, 9(06):517–532, 1999.

[BJ02] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 617–626, 2002.

[BS76] Jon Louis Bentley and Michael I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing (STOC)*, pages 220–230, 1976.

[CG86a] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[CG86b] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1(1-4):163–191, 1986.

[Cha85] Bernard Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31(4):509–517, 1985.

[Cha96] Timothy M Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, 1996.

[Cha98] Timothy M. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20(3):359–373, 1998.

[Cha01a] Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *Journal of the ACM*, 48(1):1–12, 2001.

[Cha01b] Timothy M. Chan. A fully dynamic algorithm for planar width. In *Proceedings of the 17th Annual Symposium on Computational Geometry (SoCG)*, pages 172–176, 2001.

[Cha02] Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[Cha10] Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *Journal of the ACM*, 57(3):16:1–16:15, 2010.

[Cha13] Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Transactions on Algorithms*, 9(3):22, 2013.

[Cla98] David Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1998.

[CLP11] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th Annual Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2009.

[CM96] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[CP15] Timothy M. Chan and Simon Pratt. Time-windowed closest pair. In *Proceedings of the 27th Canadian Conference on Computational Geometry (CCCG)*, pages 141–144, 2015.

[CP16] Timothy M. Chan and Simon Pratt. Two approaches to building time-windowed geometric data structures. In *Proceedings of the 32nd International Symposium on Computational Geometry (SoCG)*, pages 28:1–28:15, 2016.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(5):387–421, 1989.

[CT15] Timothy M. Chan and Konstantinos Tsakalidis. Optimal deterministic algorithms for 2-d and 3-d shallow cuttings. In *31st International Symposium on Computational Geometry (SoCG)*, volume 34, pages 719–732, 2015.

[dBCvKO00] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[DIL07] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(2):13, 2007.

[DS65] Harold Davenport and Andrzej Schinzel. A combinatorial problem connected with differential equations. *American Journal of Mathematics*, pages 684–694, 1965.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC)*, pages 109–121, 1986.

[EGS86] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[Epp00] David Eppstein. Incremental and decremental maintenance of planar width. *Journal of Algorithms*, 37(2):570–577, 2000.

[EZ14] Hicham El-Zein. On the succinct representation of equivalence classes. Master's thesis, University of Waterloo, 2014.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[Gra72] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[Her16] John Hershberger. Personal communication, 2016.

[HP11] Sariel Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Society, 2011.

[HS86] Sergiu Hart and Micha Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6(2):151–178, 1986.

[HS91] John Hershberger and Subhash Suri. Offline maintenance of planar configurations. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 32–41, 1991.

[HS92] John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics*, 32(2):249–267, 1992.

[Jar73] Ray A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[KM95] Samir Khuller and Yossi Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995.

[KMR$^+$16] Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications. *CoRR*, abs/1604.03654, 2016.

[KS86] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[Mul90] Ketan Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3):253–280, 1990.

[Mul91] Ketan Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 180–196, 1991.

[Mun96] J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.

[Pre79] Franco P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402–405, July 1979.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[Rab76] Michael O. Rabin. Probabilistic algorithms. In Traub [Tra76], pages 21–39.

[Ruž09] Milan Ružić. Making deterministic signatures quickly. *ACM Transactions on Algorithms*, 5(3):26, 2009.

[SA95] Micha Sharir and Pankaj K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, 1995.

[Sch91] Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 197–206, 1991.

[SH75] Michael I. Shamos and Dan Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 151–162, 1975.

[Sha78] Michael I. Shamos. *Computational geometry.* PhD thesis, Yale University, 1978.

[Sno97] Jack Snoeyink. Handbook of discrete and computational geometry. chapter Point Location, pages 559–574. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[ST86] Neil Sarnak and Robert E Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[Tam88] Arie Tamir. Improved complexity bounds for center location problems on networks by using dynamic data structures. *SIAM Journal on Discrete Mathematics*, 1(3):377–396, 1988.

[Tra76] Joseph F. Traub, editor. *Algorithms and Complexity: New Directions and Recent Results.* Academic Press, New York, 1976.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81 – 84, 1983.

[Yao81] Andrew Chi-Chih Yao. A lower bound to finding convex hulls. *J. ACM*, 28(4):780–787, October 1981.

[Zho16] Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Information Processing Letters*, 116(2):171–174, 2016.

Appendix A

Background

In this appendix, we discuss at a high level the results upon which our own results depend but whose details are not centrally important. Typically, these results are used as black boxes.

A.1 Grid/Quadtree Shifting

We reproduce Observation 3.2 as well as Lemma 3.3 and its proof from [Cha98], adapted for the word-RAM model as in [Cha02].

First some terminology, given $x \in \mathbb{R}$ and $r > 0$, let $x \bmod r = x - \lfloor x/r \rfloor r$. We say that a point $q = (q_1, q_2, \ldots, q_d)$ is *α-central in its r-grid cell* if and only if, for each $i = 1, \ldots, d$, we have $\alpha r \le q_i \bmod r < (1-\alpha) r$ or equivalently, $(q_i + \alpha r) \bmod r \ge 2\alpha r$.

We decompose Lemma 3 into a simpler lemma and the following observation:

Observation 24. *Let $p, q \in \mathbb{R}^d$. If q is α-central in its r-grid cell and $||p - q||_\infty \le \alpha r$, then p and q belong to the same r-grid cell.*

Then it suffices to prove the following:

Lemma 25. *Suppose d is even. Let $v^{(j)} = (\lfloor j2^w/(d+1) \rfloor, \ldots, \lfloor j2^w/(d+1) \rfloor) \in \mathbb{R}^d$. For any point $p \in \mathbb{R}^d$ and $r = 2^\ell$ ($\ell \in \mathbb{N}$), there exists $j \in \{0, 1, \ldots, d\}$ such that $p + v^{(j)}$ is $(1/(2d+2))$-central in its r-grid cell.*

Proof. Suppose, on the contrary, that $p + v^{(j)}$ is not $(1/(2d + 2))$-central for any $j = 0, 1, \ldots, d$. Then, for each j, there is an index $i(j) \in \{1, \ldots, d\}$ with

$$\left(p_{i(j)} + \left\lfloor \frac{j2^w}{d+1} \right\rfloor + \frac{r}{2d+2}\right) \bmod r < \frac{r}{d+1}$$

or equivalently, by multiplying both sides by $(d + 1)2^{-\ell}$,

$$((d+1)2^{-\ell} p_{i(j)} + \lfloor 2^{-\ell} j \rfloor + 1/2) \bmod (d+1) < 1.$$

By the pigeonhole principle, there exist two distinct indices $j, j' \in \{0, 1, \ldots, d\}$ with $i(j) = i(j')$. Letting $z = (d + 1)2^{-\ell} p_{i(j)} + \frac{1}{2}$, we have $(z + 2^{-\ell} j) \bmod (d + 1) < 1$ as well as $(z + 2^{-\ell} j') \bmod (d + 1) < 1$. This is possible only if $2^{-\ell} j \equiv 2^{-\ell} j' (\bmod (d + 1))$. Since $2^{-\ell}$ and $d + 1$ are relatively prime, then we must have $j = j'$: a contradiction! □

A.2 Succinct Rank/Select

The rank/select problem is to preprocess a sequence of n bits in order to quickly answer the following queries:

1. $rank_1(j)$ returns the number of 1s in the first j positions of the sequence, and
2. $select_0(i)$ returns the position of the i^{th} 0 in the sequence.

The obvious approaches give a time-space trade-off. You could simply store the $\mathcal{O}(n)$ bits and answer queries in $\mathcal{O}(n)$ time, or you could preprocess for these queries, storing two arrays: one which stores $rank_1(j)$ for each j from 1 to n, and the other array storing $select_0(i)$ for each i from 1 to n. These arrays could be queried in $\mathcal{O}(1)$ time, but occupy $\mathcal{O}(n \log n)$ bits of space.

In fact, Clark and Munro [CM96, Cla98] gave a solution to the rank/select problem in $n + o(n)$ bits of space which supports the above queries in $\mathcal{O}(1)$ time. For a good discussion of the details, see [EZ14].

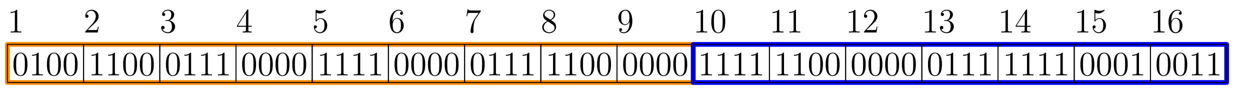

Figure A.1: A bit vector of length 64 grouped into 16 sub-blocks each composed of $\lceil \frac{1}{2} \log 64 \rceil = 4$ bits, a **block** of size $\lceil \log^2 64 \rceil = 36$, and an **incomplete block** of size 28.

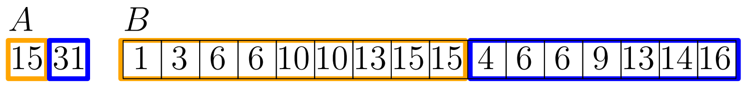

Figure A.2: Arrays A and B for the bit vector in Figure A.1.

A.2.1 Rank

At a high-level, the idea is to combine the obvious approaches. Let the input bits be $b_1, b_2, \ldots, b_n$. Group these bits into blocks of size $\lceil \log^2 n \rceil$. For each block i store the number of 1s in blocks 1 to i in our first array, A.

Next, group the bits into sub-blocks of size $\lceil \frac{1}{2} \log n \rceil$. For each sub-block i, let j be the block containing sub-block i. In array B, store the number of bits from the beginning of block j to the last bit of sub-block i.

Finally, we will create a lookup table to answer rank queries for every possible position on every possible value of a sub-block.

For example, given the sequence of bits in Figure A.1, for which we build the structures A and B in Figure A.2 and the rank lookup table in Table A.1. Say we wish to find the rank for position 43. First, we query A and find the number of bits in all preceding blocks, in this case 15. Next we query B and find the number of bits in all preceding sub-blocks within the same block as our query, in this case 4. Finally, we get all the bits in the sub-block containing our query, in this case 1100, and use the rank lookup table to find the rank at the 3rd position, in this case: 2. We add the result of these three queries together to get $15 + 4 + 2 = 21$.

	1	2	3	4
0000	0	0	0	0
0001	0	0	0	1
0010	0	0	1	1
0011	0	0	1	2
0100	0	1	1	1
0101	0	1	1	2
0110	0	1	2	2
0111	0	1	2	3
1000	1	1	1	1
1001	1	1	1	2
1010	1	1	2	2
1011	1	1	2	3
1100	1	2	2	2
1101	1	2	2	3
1110	1	2	3	3
1111	1	2	3	4

Table A.1: Rank lookup table for 4 bits.

Analysis

Array or table lookups take $\mathcal{O}(1)$ time. If we assume $w \geq \log n$ then retrieving $\lceil \frac{1}{2} \log n \rceil$ bits from the input also takes $\mathcal{O}(1)$ time. Since these are the only operations we use in this algorithm, the total query time is $\mathcal{O}(1)$.

In addition to the n bits of the input, we store A, B, and our lookup table. A is an array with $n/\lceil \log^2 n \rceil$ elements, each of size $\log n$, which takes $\mathcal{O}(n/\log n)$ total space. B is an array with $n/\lceil \frac{1}{2} \log n \rceil$ elements. Each element of B can be from 0 to $\lceil \log^2 n \rceil$, and therefore we can store it in $\log \log n$ bits, so all of B takes $\mathcal{O}(n \log \log n/\log n)$ bits. Finally, our rank lookup table has $\lceil \frac{1}{2} \log n \rceil$ columns and $2^{\lceil \frac{1}{2} \log n \rceil} = \mathcal{O}(\sqrt{n})$ rows, and each value can be from 0 to $\lceil \frac{1}{2} \log n \rceil$ and therefore takes $\mathcal{O}(\log \log n)$ bits. Therefore, the entire table takes a total space of $\mathcal{O}(\sqrt{n} \log n \log \log n)$. Thus, the total space to store the input bits, arrays A and B, and the lookup table is:

$$n + \mathcal{O}(n/\log n) + \mathcal{O}(n \log \log n/\log n) + \mathcal{O}\big(\sqrt{n} \log n \log \log n\big) = n + o(n)\,.$$

A.2.2 Select

To support select requires the careful application of the same technique of grouping the bits into blocks and sub-blocks to build arrays and lookup tables.

Lemma 26. *We can build a data structure in $n + o(n)$ bits of space that can answer* rank_1 *and* select_0 *queries on a sequence of n bits in $\mathcal{O}(1)$ time.*

A.3 Predecessor Search and van Emde Boas Trees

The *predecessor search* problem is to preprocess a set S of integers from a bounded universe $\{1, \ldots, U\}$ in order to quickly determine the largest $s \in S$ smaller than a query integer q. We say s is q's predecessor.

A van Emde Boas (vEB) tree solves the predecessor search problem by recursively storing $\sqrt{U}$ vEB trees, each of which keeps track of $\sqrt{U}$ of the integers in the universe, simply storing whether or not each integer is in the set. An auxiliary structure is also maintained which stores whether or not each of the $\sqrt{U}$ children trees are empty. This auxiliary structure is therefore also a vEB tree over $\sqrt{U}$ values. Each tree stores the minimum and maximum values among its range. These $\sqrt{U} + 1$ vEB trees can be stored in $\mathcal{O}(U)$ space and support update and query in $\mathcal{O}(\log \log U)$ time [CLRS09, Chapter 20]. In fact, we can even reduce the space of this structure to $\mathcal{O}(n)$ [Wil83, Ruž09].

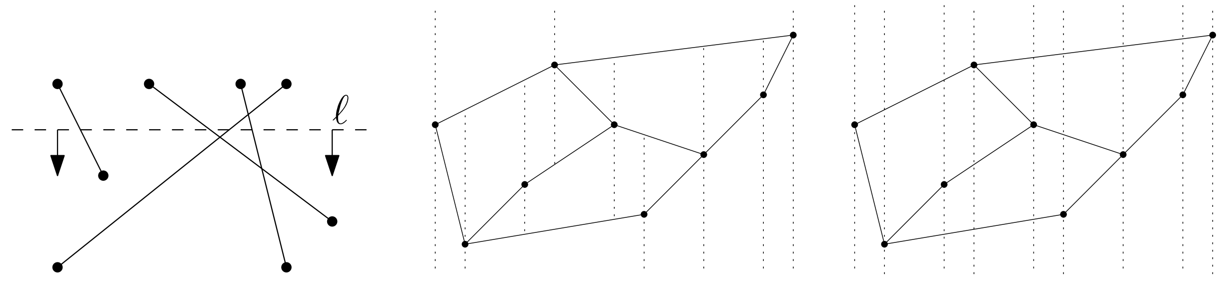

Figure A.3: *Left*: A *sweep line* sweeping downwards to compute line segment intersections. *Middle*: The *vertical* or *trapezoidal decomposition* of a set of points. *Right*: The *slabs* of the same set of points.

A.4 Plane Sweep

Given a set of n line segments in the plane, find all intersections. To compute this, we can imagine a line sweeping across our line segments (see Figure A.3). We call such a line a *sweep line* and such an algorithm is called a *plane sweep algorithm* [dBCvKO00, Chapter 2: Line Segment Intersection]. Rather than continuously moving this sweep line across the segments, it suffices to consider the line at each endpoint. If the relative order of the intersections of the segments with the sweep line changes from one endpoint to the next, then an intersection must have occurred. After sorting the input by the order in which the sweep line is to proceed (say, from top to bottom), a plane sweep algorithm takes $\mathcal{O}(n+k)$ time to perform, where k is the number of elements in the output.

A.5 Vertical/Trapezoidal Decomposition and Slabs

Given a connected planar straight-line graph (PSLG), the *vertical* or *trapezoidal decomposition* of that graph is obtained by shooting a ray up and down from every vertex, each ray ends wherever it hits an edge (see Figure A.3). This decomposes every face into triangles and trapezoids, and every face is bounded above by at most one edge, and similarly below. We can compute this decomposition in $\mathcal{O}(n \log n)$ time with $\mathcal{O}(n)$ space [PS85]. This decomposition can be used to solve problems such as planar point location (see Section A.6).

A similar idea is to decompose such a PSLG into a set of vertical *slabs*, simply by extending the rays from each vertex to infinity (see Figure A.3). The slabs can be constructed in $\mathcal{O}(n)$ time by a simple plane sweep algorithm as in Section A.4.

A.6 Planar Point Location

The *planar point location* problem is to preprocess a subdivision of the plane such that given a query point q, we can quickly find the region of the subdivision that contains q. We can solve this problem in $\mathcal{O}(n \log n)$ preprocessing time with $\mathcal{O}(n)$ space and $\mathcal{O}(\log n)$ query time using any of the following techniques: triangulation refinement, separating chains and fraction cascading, persistent search trees, or randomized incremental construction [Sno97]. Note that in the case of the randomized incremental construction, the query time is in expectation.

One simple way of solving this problem is to use *persistence*. Usually, data structures are *ephemeral* in that when they are updated, the version of the data structure before the update is in some sense lost. A data structure which allows its previous versions to be queried (but not updated) is said to be *partially persistent*. Pointer-based data structures (such as a list or a binary search tree) can be made partially persistent in $\mathcal{O}(1)$ amortized extra space per update and an amortized extra constant factor overhead to update/query time [DSST86].

To use persistence to solve planar point location we combine it with the slab decomposition from Section A.5. First compute the slab decomposition, and then sweep left-to-right across the slabs, for each slab building a version of a persistent binary search tree. Given a query point $q = (x(q), y(q))$, simply perform binary search on the x coordinates of the input vertices to find the version of the tree containing $x(q)$, then query that version of the binary tree to find the edge above $y(q)$ [ST86]. This gives us the following lemma:

Lemma 27. *We can preprocess for the planar point location problem in $\mathcal{O}(n \log n)$ time, building a data structure with $\mathcal{O}(n)$ words of space, and use this structure to answer queries in $\mathcal{O}(\log n)$ time.*

A.7 Orthogonal Planar Point Location

A special case of planar point location occurs when the subdivisions are orthogonal, in other words if the edges bounding every region are either horizontal or vertical. If, additionally, the coordinates are integers in a bounded universe $\{1, \ldots, U\}$, we can use a kind of van-Emde-Boas-style recursion to solve orthogonal planar point location in the word-RAM model in $\mathcal{O}(\log \log U)$ query time with $\mathcal{O}(n)$ space [Cha13].

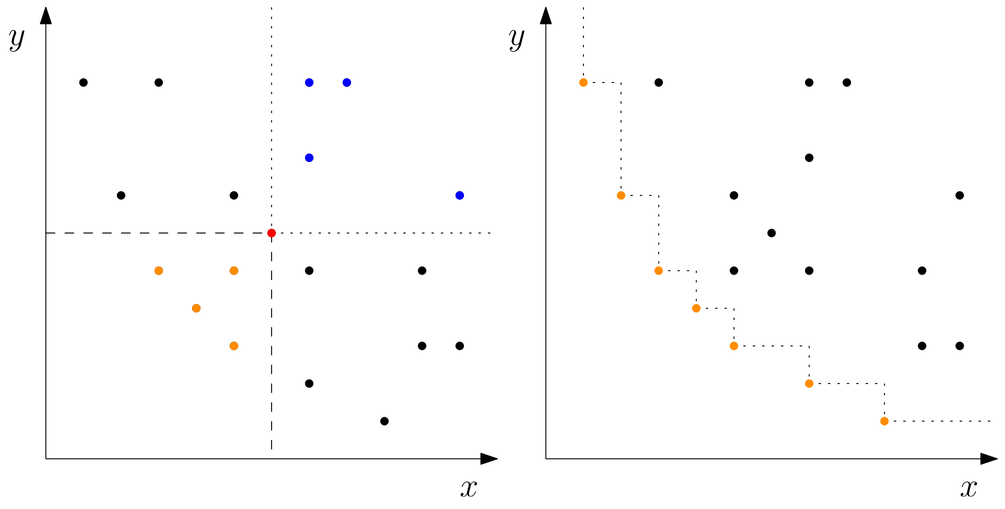

Figure A.4: *Left*: A set of points in 2D. The dashed line (or dotted line) segments bound the region dominated by (or dominating) a point, showing which points are **dominated by** or **dominate** this point. *Right*: The *staircase* defined by the **2D minima** of the same set of points.

A.8 2D Minima/Maxima and Dominance Range Emptiness

Given points $p, q \in \mathbb{R}^2$, p *dominates* q if $x(p) > x(q), y(p) > y(q)$. The *2D minima* of a set of points P is the subset $P' \subseteq P$ such that each point $p \in P'$ dominates no points. Similarly, the *2D maxima* of P is $P' \subseteq P$ such that each $p \in P'$ is dominated by no points.

The set of all points dominated by a point p is the quadrant of the plane whose upper-right corner is p and that extends downwards and leftwards to infinity. The union of the quadrants of the 2D maxima of a set of points forms a region bounded by a monotone chain called a *staircase*. These maxima, and the staircase they form, can be computed by a standard plane sweep of the points [PS85]. Symmetrically, the set of all points which dominate a point p is the quadrant of the plane whose lower-left corner is p, and that extends upwards and rightwards to infinity. The union of the quadrants of the 2D minima of a set of points also forms a region bounded by a staircase, and can be computed by a plane sweep.

The *2D dominance range emptiness* problem is to preprocess a set P of n points in $\mathbb{R}^2$ in order to quickly determine whether a query range $(-\infty, x] \times (-\infty, y]$ is empty. Notice that this is equivalent to determining if the point $q = (x, y)$ dominates any point $p \in P$, hence the name of the problem.

This problem can be solved by computing the staircase of P, then determining if q is above or below. If q is above the staircase, then there must be at least one point $p \in P$ that q dominates, and therefore the range is not empty.

A.9 3D Minima/Maxima and 2D Dominance Range Minimum

In 3D, the set of points dominating a point p is an octant (or in general, an orthant) with p at its lower-left-near corner. Symmetrically, the set of points which are dominated by a point p is an octant with p at its upper-right-far corner. As in 2D, the minima (maxima) of a set of points in 3D are the points which dominate no points (are not dominated by any points) [PS85, Afs14]. Computing the *minima* of the 3D point set can be done by a standard plane sweep algorithm.

The *2D dominance range minimum* problem is to preprocess a set P of n points in $\mathbb{R}^2$ such that each point p is associated with a weight $w(p)$, in order to quickly determine the point with minimum weight in a query range $(-\infty, x] \times (-\infty, y]$.

We can solve this problem by considering all $p \in P$ as points in 3D with $z(p) = w(p)$. Then we can compute the 3D minima of P, then project these onto the plane to build an orthogonal subdivision of the plane in which each region is associated with a point in P. We can answer a query $q = (x, y)$ by finding q in this subdivision using orthogonal planar point location (see Section A.7).

A.10 Davenport-Schinzel Sequences

Davenport-Schinzel (DS) sequences [DS65] are defined as follows:

Definition 28. A DS sequence of order k is a sequence Σ of characters $s_1, s_2, \ldots$ from an alphabet A such that no two consecutive characters are the same and for any two characters $a, b \in A$, the alternating sequence $a, b, a, b, \ldots$ of length $k + 2$ does not appear as a subsequence anywhere in Σ, whether contiguous or not.

In particular, a DS sequence *of order 3* is a sequence Σ of characters $s_1, s_2, \ldots$ from an alphabet A such that no two consecutive characters are the same and for any two characters $a, b \in A$, the alternating sequence a, b, a, b, a (whose length is 5) does not appear as a subsequence anywhere in Σ, whether contiguous or not.

Hart and Sharir [HS86, SA95] proved that an order-3 DS sequence over an alphabet of size n has length at most $\mathcal{O}(n\alpha(n))$. In fact, there is a matching lower bound.

A.11 Unit-Disk Intersections

A *disk* with center $c \in \mathbb{R}^2$ and radius r is the set of all points in $\mathbb{R}^2$ whose distance to c is at most r. We say that a disk with unit radius is a *unit disk*. The intersection of two overlapping unit disks d_1, d_2 is bounded by two arcs, one from the boundary of d_1 entirely within d_2, and the other from the boundary of d_2 entirely within d_1. In general, a *unit-disk intersection* is the intersection of a set D of n disks, $\bigcap\limits_{d \in D} d$. We wish to prove the following:

Lemma 29. *The intersection of n unit disks in $\mathbb{R}^2$ has linear combinatorial complexity and can be constructed in $\mathcal{O}(n \log n)$ time.*

First, let I be the intersection of disks with arbitrary radii. The sequence of disks around the boundary of I is a DS sequence of order 2 since having a sequence of disks a, b, a, b would imply that either a or b is not convex. When the disks all have the same radius, then each disk appears at most once. Thus, the intersection of n unit disks has linear combinatorial complexity.

It remains to show that we can construct this intersection in $\mathcal{O}(n \log n)$ time. We will do so by proving that unit-disk intersections can be merged in linear time proportional to the total number of disks. Therefore, to compute the intersection of n unit disks is analogous to merge sort, with the same recurrence giving its time bound: $T(n) = 2T(n/2) + \mathcal{O}(n)$. The basic idea of this merge algorithm is the same as the algorithm in [PS85, Chapter 7, Section 7.2.1], which is to use the slab decomposition (see Section A.5), and compute the intersection within each slab.

Lemma 30. *Merging two unit-disk intersections with a total of n disks in $\mathbb{R}^2$ can be performed in $\mathcal{O}(n)$ time.*

A.12 Unit-Ball Intersections

The 3D equivalent of a disk is a *ball*, defined as all points in $\mathbb{R}^3$ at most some radius r from a center point c. If a ball has unit radius, then it is called a *unit ball*. The intersection of n unit balls still has linear combinatorial complexity and can be constructed in $\mathcal{O}(n \log n)$ time by Clarkson and Shor's randomized algorithm [CS89] or Amato *et al.*'s deterministic algorithm [AGR94]. This gives us the following lemma:

Lemma 31. *The intersection of n unit balls in $\mathbb{R}^3$ has linear combinatorial complexity and can be constructed in $\mathcal{O}(n \log n)$ time.*

A.13 Dynamic 2D Width

The *width* of a set of points is the narrowest section of its convex hull. More precisely, the width is bounded by an edge e and a vertex v on the convex hull, such that the distance from v to the line containing e is minimal. The *dynamic 2D width* problem is to maintain a data structure that supports *insertion* of a point in $\mathbb{R}^2$, *deletion* of a point already in the structure, and *querying* for the width of all inserted points that have not been deleted. Chan gives a structure to solve this problem with $\mathcal{O}(\sqrt{n}\,\mathrm{polylog}(n))$ update time [Cha01b].

For our purposes it would be most useful to bound the update time in terms of k, the number of structural changes to the convex hull caused by an update. Eppstein gives a structure that solves this problem with $\mathcal{O}(k \cdot f(n) \cdot \log n)$ update time, where $f(n)$ is the time to solve the dynamic 3D convex hull problem (more precisely, answer gift-wrapping queries and perform updates for a 3D point set) [Epp00].

Eppstein originally used Agarwal and Matoušek as a black box to solve the dynamic 3D convex hull problem in $f(n) = \mathcal{O}(n^\epsilon)$ time [AM95], but this has since been improved by Chan to $f(n) = \mathcal{O}\big(\log^6 n\big)$ expected time [Cha10], and derandomized by Chan and Tsakalidis [CT15]. Finally, Kaplan *et al.* improved this to $f(n) = \mathcal{O}\big(\log^5 n\big)$ amortized time [KMR+16].