Partial Enclosure Range Searching

by

Gregory Bint

A thesis submitted to
the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Carleton University
Ottawa, Ontario

Abstract

This thesis examines a new type of range searching problem which we have called *Partial Enclosure Range Searching*. In this problem, given a set of geometric objects and a query region, our goal is to identify those objects which intersect the query region by at least a fixed proportion of their original size.

We consider several variations of this problem with different types of geometric objects and query regions. When querying line segments, this thesis presents methods for transforming the problem into one which can be solved using standard range searching techniques. When querying convex or monotone polygons, this thesis presents methods for calculating the area of the intersection between the polygon and a query rectangle, which also has uses outside of determining the partial enclosure property.

Acknowledgements

There are a great many people I would like to thank for helping me achieve success at Carleton.

My supervisors, Anil Maheshwari and Michiel Smid, have had their doors open to me for my entire tenure at Carleton, ever since my first summer in the Computational Geometry lab way back at the beginning of my undergraduate degree. They have endured countless drop-in questions, and provided many impromptu lectures on the walk to lunch any time I needed help navigating a problem.

Much credit is also due to Subhas Nandy of the Indian Statistical Institute, Kolkata, who got me started with this thesis work. Thank you for all your hospitality in India, and of course, for all the chai.

The Computational Geometry lab has been an amazing environment to work and learn in. The entire faculty of the CG lab, Prosenjit Bose, Vida Dujmovic, Anil Maheshwari, Pat Morin, and Michiel Smid, put in countless hours supporting each and every student of the lab. I would like to thank Jean-Lou De Carufel for teaching me the art of inequalities, and my fellow students Nima Hoda and Simon Pratt for teaching me as much about our coursework as the lectures did.

I gratefully acknowledge Carleton University, the School of Computer Science,

and the National Sciences and Engineering Research Council of Canada for their generous funding support.

Finally, to my amazing wife, Kathryn Bint. I would never have survived my degree without your love and unerring support through uncountably many busy evenings and weekends.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Range searching is one of the most common types of problems which arise in everyday computer use. With a range search, we are given a set of objects and asked to identify those which satisfy some bounded criteria. Range searches can take many different forms: searching for email received between two dates, looking for restaurants near your present location, or identifying what a video game player should see on their screen in any one frame; these are all examples of range searches.

In computational geometry, range searching takes on a more abstract quality. Typically we are given an environment containing a set of geometric objects such as points, lines, circles, or boxes. A query is itself another well-defined geometric object, and our goal is to identify all elements of the environment contained within the query region. When addressing a range searching problem, we want to develop a method for preprocessing the input environment so that we can answer any query as efficiently as possible. Given how common and flexible range searching is to such a wide range of practical computer science, it is not surprising to find that a great deal of research has been expended in this area.

In this thesis, we address the notion of *Partial Enclosure Range Searching (PERS)*, which, to the best of our knowledge, has not been explored previously. In this setting, our goal is to identify, for a given query region, all objects which satisfy the *Partial Enclosure Property*, which specifies that an object must intersect a query region by at least some fixed proportion of the object's own size (e.g., length, area, volume) in order to be selected.

This chapter is organized as follows. Section 1.1 begins by describing our motivation for this problem. In Section 1.2 we describe the specific variations of the PERS problem that we will address in this thesis. In Section 1.3, we discuss related problem domains, and contrast them to our own. Section 1.4 outlines the contributions made by this thesis. We conclude the introduction by outlining the organization of the remainder of the thesis in Section 1.5.

1.1 Motivation

This problem was inspired by the author's use of Microsoft OneNote. Using a digital pen, OneNote can be used much like a paper notebook, allowing the user to add handwriting, diagrams, equations, and any other such thing to a page. Unlike a paper notebook, OneNote also allows the user to select previously drawn objects in order to translate, scale, copy and otherwise manipulate them. Figure 1.1 shows some handwritten notes, and a diagram which has been partially selected.

Looking carefully at the figure, we can see that even though the horizontal line segments of the diagram have not been entirely enclosed by the selection tool, they nevertheless appear as part of the set of selected items. This behaviour

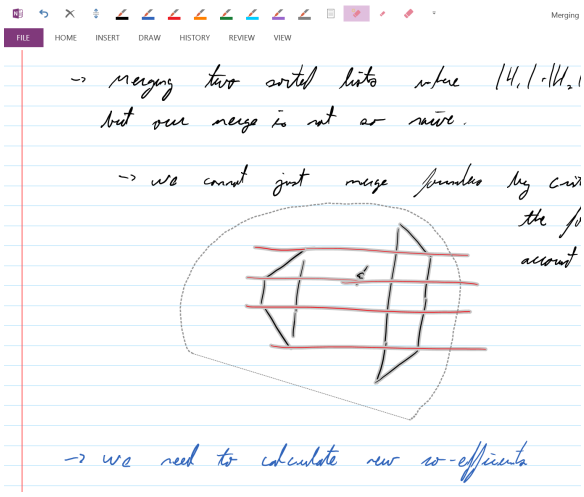

Figure 1.1: An example of practical partial enclosure range searching in Microsoft OneNote. The line segments in the middle are selected even though they are not entirely enclosed.

of selecting partially enclosed objects is described in a patent filed by Microsoft Corporation[16]. From the patent:

> [T]here is a need for a selection tool that will allow a user to conveniently select one or more graphical objects in their entirety, without requiring an inconvenient amount of precision from the user.

With the rising popularity of touch and pen-enabled devices, this need is likely to increase. As we can see from the figure, OneNote already includes an implementation of a partial selection tool like the one described by the patent. Although the details of the implementation are proprietary, it becomes apparent while using the software that it suffers from poor performance as more items are selected. It is from this observation that the work in this thesis was inspired. Although the problems that we will examine take place in simpler settings than the patent describes, we

will nevertheless develop an understanding of the major challenges of this problem domain, as well as some techniques for addressing them.

1.2 Problem Statements

This thesis proposes algorithms for several different partial enclosure range search settings. Each problem addresses a different type of geometric object to be queried, or a different type of query region.

Line Segments and Query Rectangles. Line segments and rectangles are one of the simplest settings in which we can perform partial enclosure range searches. Given an axis-parallel query rectangle, we address methods which consider only axis-parallel segments, or which consider arbitrarily-oriented ones.

Line Segments and Query Slabs. Relaxing the axis-parallel query regions of the preceding problem, we consider an arbitrarily-oriented slab. As the query slab can have any orientation, intersections with a horizontal segment will involve both the height of the segment and the slope of the slab. We also consider a query defined as the intersection of two slabs.

Convex Polygons and Query Rectangles. Moving away from line segments, we consider problems of partial enclosure with respect to area. For any such problem, deciding on the partial enclosure property is straight-forward once we know the area of the input object and the area enclosed by a query region. We consider how to determine the proportion of a convex polygon enclosed by a query rectangle.

Monotone Polygons and Query Rectangles. This more complex shape does not generally decompose into big convex regions, so a different method of preprocessing will be needed. We consider methods for calculating the area of the polygon to one side of a query line. With such a method in hand, determining the area of a rectangle is a matter of executing several queries and combining their results.

1.3 Related Work

In this section, we review some existing variations of the range searching problem. As a whole, due to its applicability to such a wide assortment of problems, range searching has received a great deal of research attention. Aside from the solutions to specific problems that this research has found, it has also resulted in a vast toolbox of observations and techniques, many of which apply to our own problem.

Range searching structures are typically constructed by subdividing the input objects, or the space in which the input objects exist, into several regions which express useful properties. Ideally, we are able to choose these regions so that a query area will entirely contain, or not contain, most of these regions, while intersecting only a small number of them. On those intersected regions, we continue the query on the recursively finer subdivisions stored there.[18]

This recursive decomposition into regions which is employed by many techniques has an extremely beneficial side-effect. At each region in this hierarchy, we can associate any other structure of our choosing. These so-called multi-level structures can be used to answer complex queries by using the outer structure to find objects satisfying one query parameter, and then continuing with the associated

structures to find objects which satisfy additional properties. We discuss examples of this type of multi-level query in detail in Chapter 2.

The remainder of this section gives a brief introduction to some specific range searching problems, and history of some methods for answering each. We refer the reader to an excellent survey on range searching by Agarwal and Erickson[1] for a more detailed introduction.

Orthogonal Range Searching

In its general form, an orthogonal range searching problem starts with a set S of points in $\mathbb{R}^d$, where d is a small, fixed constant. A query is a d-dimensional, axis-parallel box, and our goal is to identify the subset of our input points which are contained within this box.

Quadtrees, first described by Finkel and Bentley[13] (see also [10, Chapter 14]), are one of the very first structures for performing range searches. When we are considering points in the plane, a quadtree works by dividing the space containing the points into squares, and then recursively splitting any region which contains more than one point into four smaller, but equally-sized squares.

Because this recursion is based on splitting space rather than the points within the space, the recursion depth, and thus the depth of the quadtree, is not related to the number of points. Instead, the size of the initial square from which we start splitting and the distance between the closest pair of points are the important factors. For an initial square with a side length of s and a closest pair with distance c, the recursion depth is $O\big(\log \frac{s}{c}\big)$. The construction and query times are related to this recursion depth, making the structure somewhat slow, but it *does* work!

Quadtrees can be easily extended to work in higher dimensions. In $\mathbb{R}^3$ quadtrees are called *Octrees*, and the general method can be extended to even higher dimensions. For points and an initial box in $\mathbb{R}^d$, decomposition proceeds along the same basic rules, but using progressively smaller d-boxes instead of just squares.

KD-Trees[2] were the first major improvement over quadtrees. Unlike a quadtree which partitions space, a KD-Tree directly partitions the pointset in a recursive manner. For points in the plane, each level of the tree corresponds to a particular axis. Suppose that we start with the x-axis at the root level. We consider the median point m with respect to the x-coordinates of all the points, and then all remaining points are partitioned as left or right of this vertical *splitting line* through m.

On the next level of the recursion we alternate to the y-axis, select the median point with respect to y-coordinates, and partition around the horizontal splitting line through that point. The process continues, alternating the axis on each level, until we create leaves containing single points. This process creates a balanced tree, where each path through the tree alternates between axes. Each node in the tree represents a region of the plane bounded by its own splitting line and the splitting lines of its ancestors.

Querying a KD-Tree by a rectangular region involves traversing the tree while only visiting those nodes whose region is intersected by the query region. While the preprocessing requirements of a KD-Tree are quite good, at $O(n)$ space and $O(n \log n)$ preprocessing time, the query time is $O(\sqrt{n})$.

A final note about KD-trees, this recursive construction can be generalized to higher dimensions in a straight-forward manner, by simply cycling through each of the d axes in order. The preprocessing requirements are unchanged asymptotically,

since we still create just a single binary tree with n leaves. The query algorithm is also similar to the $\mathbb{R}^2$ case, but with a query time of $O\big(n^{1-1/d}\big)$.

The next improvement in orthogonal range searching came in the form of a polylogarithmic method known as *Range Trees*. Range trees were discovered by Bentley[3], who was involved in the last two structures, but were also simultaneously developed by other independent researchers. This method is constructed on binary search trees and can query any open or closed query box in $O\big(\log^{d-1} n\big)$ time with only $O\big(\log^{d-1} n\big)$ time and space pre-processing. We discuss this structure in more detail in Section 2.1.

Finally, a noteworthy special case occurs when our points are in the plane and our query is a three-sided rectangle, i.e., when it is open to one side. In this case, we can use a method developed by McCreight[19] which gives $O(\log n + k)$ query time, where k is the number of points found, while using only $O(n)$ space and $O(n \log n)$ preprocessing time.

The real power of orthogonal range searching is in how other problems can be reduced to instances of it by encoding query keys as coordinates of a high-dimensional point. For example, in the geometric sense, we can search for rectangles inside of rectangles by mapping the coordinates of an input rectangle in $\mathcal{R}^d$ to a point in $\mathcal{R}^{2d}$, and performing an orthogonal range search in that space. Rectangle-in-rectangle queries are important as rectangles make easy approximations of more complicated geometric objects that we may want to look for.

Outside of purely geometric applications, many more general types of range searching problems can be similarly reduced to instances of orthogonal range searching. For example, given some integer representation of a date, we can search for

emails received within a date range by encoding the date stamp of each email into a point, and then searching for points between the two integer representations of our query range. Extending this idea to more complex queries is just a matter of mapping each key we wish to search on to further components of a higher-dimensional point. In this way, multi-dimensional orthogonal range searching can be used to perform multi-key searching.[23]

Half-plane Range Searching

In general, when performing half-plane queries, we need to make a choice between using a structure which has low space or one with a fast (polylogarithmic) query time. We introduce a structure suitable for each side of this trade-off.

If we want to save space, we can use a *Ham Sandwich Cut Tree*[12, 11]. To construct this tree, we divide all of our points into four quadrants of equal size, which we accomplish with an algorithm by Megiddo[20] in $O(n)$ time. We continue construction by repeating this process on each quadrant recursively, and building a tree out of the results of each step. The final tree has size $O(n)$ and is constructed in $O(n \log n)$ time.

We can query this tree with the line representing the boundary of a query half-plane. The quadrants at any step are the result of two intersecting lines. Their intersection point must be on or to one side of any query line, and thus, the query line will intersect at most three quadrants. The non-intersected quadrants must then be either entirely inside the query half-plane, or entirely outside of it. The query then recurses on the intersected quadrants. Total query time is $O\left(n^{\log_2(1+\sqrt{5})-1}\right) \approx O(n^{0.695})$.

If we want to save query time, then we can use a *Level Arrangement*[14]. This structure has a query time of only $O(\log n)$ but requires $O(n^2)$ storage. The level arrangement structure is constructed in a dual space. Given a set S of n points $p_1, p_2, \ldots, p_n$ where each $p_i = (a_i, b_i)$, we map each point to a corresponding dual line $l_i : y = a_i \cdot x - b_i$.

Considering all of the intersection points formed by the set of dual lines crossing each other, and the edges between them, we can easily see the $O(n^2)$ size. Preprocessing continues by enclosing the environment in a bounding box which contains all of the intersections. By following the monotone chains of intersections from left to right, we can determine the "level" of each edge, that is, the number of edges below (or above) each edge. Total preprocessing time is $O(n^2)$.

In this dual space, the line which defines the boundary to our query half-plane becomes mapped to a point. The query proceeds by selecting the median level of the arrangement, and then performing a binary search over its edges to find the one above or below the query point. This process is repeated in a binary search sort of fashion on the subset of levels containing the query point until we have found the levels directly above and below the query point. Depending on the half-plane, the levels lying to one side of the query point or the other correspond to the input points satisfying the query half-plane. As we have described it here, this query requires $O\big(\log^2 n\big)$ time, but this can be reduced to $O(\log n)$.[21]

A different approach to solving half-plane range searches altogether is to consider a half-plane as a degenerate simplex, and then use any simplex range searching structure to answer a query. Such structures are discussed below and in Section 2.2.

One final method for half-plane searches we would like to mention is by Chazelle *et al.*[8, 5] as it does not obey the general trade-off between space and query time that we mentioned above. This method uses *convex layers* and gives $O(n)$ storage and $O(\log n + k)$ query time, where k is the number of points found.

Convex layers of a pointset are easiest to explain by their construction, which is performed iteratively by calculating the convex hull, deleting it, and then taking the convex hull of what remains, until all points are deleted. Convex layers can be constructed in $O(n \log n)$ time, and require $O(n)$ space. In order to use convex layers for half-plane queries, we will translate the pointset such that the origin point of the plane occurs inside the innermost convex layer.

A query line which passes through the pointset will intersect some or all of the layers, from l_1 at the outside to l_m, the innermost layer which is intersected. The query is performed in two main steps: identify l_m, and then report all of the points.

Locating l_m is done in a dual space transformation. Our pointset becomes an arrangement of lines, and our query line becomes a point. A consequence of the origin point being inside the innermost convex layer in the primal space is that the dual space will be comprised of corresponding levels, although in reverse order. We can then perform any planar point location method we like on the query point to determine the level containing it; Chazelle *et al.* give several $O(\log n)$ choices.

With the convex layer l_m known, we can compute the intersections of the query line with l_m in $O(\log n)$ time, and in so doing, identify at least one point which is inside our query half-plane. During preprocessing, we augment every vertex with pointers to the edges which appear above and below them in neighbouring convex layers, accomplished by walking around two neighbouring layers at a time

simultaneously. Depending on which side of the query line our half-plane extends, we can walk these pointers from l_m across the remaining levels, whether that first takes us deeper into the layers, or directly out. At each level, we then traverse left and right around the level for as long as we remain in the query half-plane reporting our matching points.

Simplex Range Searching

In simplex range searching in the plane, there is no known data structure which can answer a query in polylogarithmic time using linear (or nearly linear) storage.[1]

Considering structures with linear or near linear storage, the first sublinear query time algorithm in this area was by Willard in 1982[22]. His paper introduced a method using *partition trees*. The general idea is to partition space into several regions each containing a roughly equal number of input points. Each region is then recursively built in the same way. The goal is to choose our regions in such a way that the largest number of regions which may be intersected by any line is minimized. This property is known as the *crossing number*. Willard's approach requires a query time of $O(n^{0.774})$. Moreover, the partition tree approach in general expresses a "nice" recursive structure which lends itself to multi-level query structures.

Following this first approach, a series of improvements were made to the partition tree method, mostly concentrating on developing a lower crossing number. For many years, the best known approach was by Matousek[17] which requires $O(n \log n)$ preprocessing time and $O\left(\sqrt{n} \log^{O(1)} n\right)$ query time. Recently, this has been improved to a query time of just $O(\sqrt{n})$ by Chan[4] with the same preprocess-

ing time. We use this latter method throughout Chapters 3 and 4, and discuss it in more detail in Section 2.2.

Intersection Searching

Intersection searching is similar to range searching, except that we consider different types of geometric input objects aside from points. Given a query region, we are looking for any objects which have even a single point of intersection in common with the query itself; that is, the object need not be entirely enclosed with the query region. In this way, we can think of range searching as a specialization of intersection searching. We introduce just a few of the many variations of intersection searching problems.

In *Segment Intersection Searching*, the query is a line segment. The case where both the input objects and the query are orthogonal line segments has many applications in VLSI design. Orthogonal or otherwise, segment intersection searching can be efficiently solved using plane-sweep algorithms.

Rectangle Intersection Searching, also known as *Windowing Queries* identify polygons which intersect a rectangular query region. Many computer graphics problems are related to this problem as, for example, we might want to determine what items need to be rendered on screen from a 3D environment.

Finally, *Point Intersection Searching* is sort of the opposite of a normal range searching query. In this problem, we are interested in reporting all objects which contain a query point.

Some of these methods, notably windowing queries, initially seemed as though they would be helpful in solving PERS problems. For example, looking at our line

segment problems, we could start by considering only those segments that at all intersect a query region, and then perform additional steps to identify only those which are sufficiently enclosed. However, in the worst case, the number of segments intersecting the query region could be much, much higher than the number satisfying the partial enclosure property, such that the added steps of the intersection query do not help at all.

Completing a Search

Throughout this section, we've used the term "identify" rather loosely. For the hierarchical structures above, "identifying" points satisfying a query really comes down to isolating those partitions, or disjoint subsets, which contain them. What we do with the partitions at that point is somewhat flexible. For example, we can simply count the points we have identified with only a constant factor of extra time and space by having the partitions store their own sizes. We could instead report the points by traversing the contents of the partition, although this will cost us an extra linear factor with respect to to the number of reported items. We use the term "identify" in this way throughout the thesis.

We conclude this section by comparing and contrasting our own problem with the ones we have just introduced.

Our problem is notably different from standard range searching problems because we are not just looking for items which are entirely contained inside of a query region. This immediately rules out a simple application of orthogonal range searching like we saw with the rectangles-in-rectangles problems.

Our problem also differs from intersection searching, since we have the extra

Table 1.1: Summary of Contributions

Object	Query	Theorem	Space	Time	Query
AP Segment	AP Rectangle	Th 3.1	$n \log^3 n$	$n \log^3 n$	$\log^3 n$
AO Segment	AP Rectangle	Th 3.8	$n \log^7 n$	$n \log^7 n$	$\sqrt{n} \log^7 n$
AP Segment	AO Slab	Th 4.5	$n \log^2 n$	$n \log^3 n$	$\sqrt{n} \log^3 n$
AP Segment	2 AO Slabs	Th 4.9	$n \log^3 n$	$n \log^3 n$	$\sqrt{n} \log^3 n$
Convex P	Rectangle	Th 5.1	n	n	$\log n$
Convex P	Convex k-gon	Cor 5.3	n	n	$k \log n$
Monotone P	AP Rectangle	Th 6.3	$n \log n$	$n \log n$	$\log n$
Monotone P	AP Rectangle	Th 6.5	n	$n \log n$	$\sqrt{n}$
Simple P	Horiz Slab	Cor 6.6	n	n	$\log n$

constraint of requiring a specific proportion of the input objects to be contained in the query region, rather than just any point.

1.4 Summary of Contributions

In this thesis, we develop contributions to several variations of the PERS problem, along with some noteworthy ancillary methods. Table 1.1 gives a broad overview of these techniques. The table uses 'AP' for "Axis-Parallel", 'AO' for "Arbitrary Orientation", 'P' for "Polygon", and omits "Big Oh" for clarity.

In the line segment cases, we develop methods for restating the partial enclosure property as something which can be evaluated as a one or two variable inequality. These expressions can then be queried by known structures for orthogonal or half-plane range searching. We discuss the transformations to appropriate dual-spaces and the design of data structures which can answer the necessary multi-part queries.

In the polygon cases, we primarily develop methods for calculating area within a simple region, e.g., below a query line. These methods are then repeated and

combined to find the area enclosed by the actual query region. Once the enclosed area is known, determining the partial enclosure property is straight-forward.

1.5 Organization of the Thesis

The remainder of this thesis is organized in the following way. Chapter 2 reviews existing data structures and range searching techniques which we utilize in our own contributions. The next four chapters cover partial enclosure range searching queries on successively more sophisticated geometric objects, with Chapter 3 focusing on axis-parallel rectangles, Chapter 4 on arbitrarily-oriented slabs, Chapter 5 on convex polygons, and Chapter 6 on monotone polygons. The thesis concludes with Chapter 7 which summarizes the contributions and future work presented in earlier chapters.

Chapter 2

Preliminaries

In this chapter, we outline the data structures and other techniques which we rely on for solutions to our own contributions.

2.1 Range Trees

A range tree is a binary search tree that allows us to search for elements satisfying a 1D query interval. Given a set of n elements, $A = \{a_1, a_2, \ldots, a_n\}$, each expressing a scalar "key" value $v_1, v_2, \ldots, v_n$, a range tree T can identify the elements whose keys fall between two values α and β, where $\alpha \leq \beta$. That is, the search identifies the following set:

$$\{a_i \in A \mid \alpha \leq v_i \leq \beta\}$$

This search can be done in $O(\log n)$ time with only $O(n)$ space and $O(n \log n)$ preprocessing time. Reporting the matching elements can be done in $O(\log n + k)$ time, where k is the number of elements reported.

However, all of these properties are also true of a regular binary search on a sorted array, so why are range trees interesting? To answer this, we look at a simple example. Suppose A is a set of 2D points embedded in the plane, and that, for our query, we wish to identify all points whose x-coordinate is between α and β. To answer this query, we can construct a range tree T where each point a_i, $1 \leq i \leq n$ is represented by leaf node whose key value is set to the x-coordinate of a_i; that is, where $v_i = a_i.x$.

When we query T for α and β, we identify two paths through T in $O(\log n)$ time. The leaf with key value α (or with the successor value to α, if the exact value is not present) is a_α and is the leftmost point in A matching our query. Similarly, the leaf with key value β (or with the predecessor value to β, if the exact value is not present) is a_β and is the rightmost point in A matching our query. Since the leaves in a binary search tree are sorted, the sequence of points from a_α to a_β are exactly the points we need to identify. We can report these points with an inorder traversal.

What makes range trees such versatile structures is not that they identify appropriate elements, but rather due to the way that they group the identified elements into at most $O(\log n)$ disjoint subsets. These subsets are precisely the elements found in the "inner" subtrees along the path from a_α to a_β, as illustrated in Figure 2.1.

With each non-leaf node of T, we can store additional information about the subset represented by its subtree. Then, during query time, we can aggregate information relating to the entire matching subset by considering the preprocessed information stored in only $O(\log n)$ nodes. Continuing our example on 2D points, with only $O(1)$ extra space per node, we could store which point has the highest

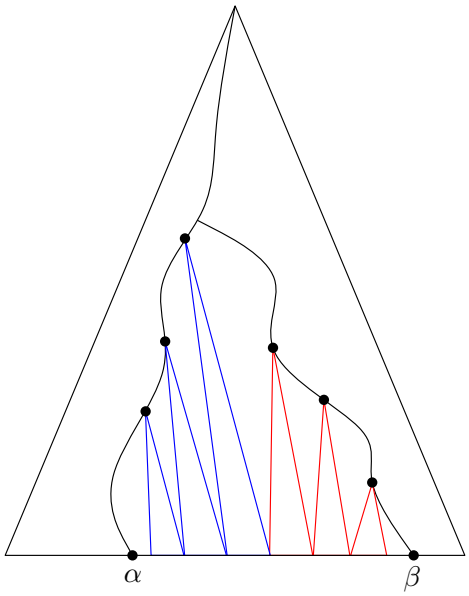

Figure 2.1: A 1D range tree and the subtrees identified by a query.

y-coordinate in each subtree. Then, we can use our range tree to find the highest point between two x-coordinates in $O(\log n)$ time.

We are not limited to just storing simple values with each node, however. We could also associate an entire additional data structure, constructed on the elements represented by that node. In particular, we can store another range tree which is based on some other property of each element. When we query this two-level range tree, our query at the first level identifies $O(\log n)$ matching subsets. Then, for each subset, we query the associated data structure to identify elements satisfying a second query condition. The elements identified here satisfy both conditions. This technique can be repeated to create *multi-level range trees* with several levels.

The following theorems on range trees, restated from [10, Chapter 5], detail the construction and query times of multi-level range trees.

Theorem 2.1. *Let P be a set of n points in d-dimensional space, with $d \geq 2$. A multi-level range tree for P uses $O\big(n \log^{d-1} n\big)$ storage and it can be constructed in $O\big(n \log^{d-1} n\big)$ time. With this range tree one can report the points in P that lie in a rectangular query range in $O\big(\log^{d-1} n + k\big)$ time where k is the number of reported points.*

If each node of the range tree is augmented with information about the size of its subtree, then we can count the number of items within a query rectangle rather than reporting them. The following corollary summarizes this.

Corollary 2.2. *Let P be a set of n points in d-dimensional space, with $d \geq 2$. A multi-level range tree for P uses $O\big(n \log^{d-1} n\big)$ storage and it can be constructed in $O\big(n \log^{d-1} n\big)$ time. With this range tree one can count the points in P that lie in a rectangular query range in $O\big(\log^{d-1} n\big)$ time.*

2.2 Canonical Subsets Data Structure

The range trees introduced in the previous section can be used to query any range which can be expressed by a single query variable expression. In some cases, however, our expressions will take the form of half-planes comprised of two query variable expressions, and for those, we use the following structure described in Chan[4], restated here with $d = 2$.

Theorem 2.3 (Corollary 7.3, part i, in Chan[4])**.** *We can form $O(n)$ canonical subsets of total size $O(n \log n)$ in $O(n \log n)$ time, such that the subset of all points inside any query simplex can be reported as a union of disjoint canonical subsets C_i with $\sum_i \sqrt{|C_i|} \leq O(\sqrt{n} \log n)$ in time $O(\sqrt{n} \log n)$ with high probability with respect to n.*

In this theorem, "with high probability with respect to n", abbreviated "w.h.p. (n)", refers to the query phase of the data structure, and specifically that the total size of the canonical sets containing the points matching our simplex query will satisfy $\sum_i \sqrt{|C_i|} \leq O(\sqrt{n} \log n)$. It is possible that the total size may be larger, but the probability that the size of the results is bounded as stated approaches 1 as n tends to infinity.

This structure is particularly well-suited to multi-part queries where we need to identify objects which satisfy a half-plane query, and which also satisfy some other arbitrary query condition for which we have an existing query method. To support such a query, we construct a canonical subsets structure to perform the half-plane component of the query. Then, with the objects of each canonical subset, we construct secondary query structures. A similar structure with slightly worse query time bounds by Matousek[17] is presented in [10, Chapter 16], along with a method for using it to construct multi-part queries.

A multi-level query is executed by first querying the canonical subsets structure for all points satisfying the half-plane. With each subset identified by the first step, we then evaluate the associated secondary structure. The result is the set of all objects which satisfy both the half-plane query *and* whatever conditions the second-level query may test. If the secondary structure is itself a multi-level structure, then this procedure effectively adds one more layer, and we can repeat this process to any arbitrary number of levels. The following corollary formalizes the use of canonical subset structures to build multi-level structures and details the preprocessing and query requirements.

Corollary 2.4. *Given a set of n objects $a_1, a_2, \ldots, a_n$ which can be queried with a*

data structure A requiring preprocessing space $S(n)$, preprocessing time $T(n)$, and query time $Q(n)$, and where each a_i also has an associated point p_i, we can construct a multi-level data structure which can identify all objects whose associated point p_j satisfies a half-plane and *where a_j satisfies a query on A.*

1. *If $S(n) \in O\big(n \log^f n\big)$, $T(n) \in O(n \log^g n)$, and $Q(n) \in O\big(\sqrt{n} \log^h n\big)$, where $f, g, h \in O(1), 0 \leq f \leq g$, then the resulting multi-level data structure requires $O\big(n \log^{f+1} n\big)$ preprocessing space, $O\big(n \log^{g+1} n\big)$ preprocessing time, and $O\big(\sqrt{n} \log^{h+1} n\big)$ query time w.h.p. (n).*

2. *If $S(n) \in O\big(n \log^f n\big)$, $T(n) \in O(n \log^g n)$, and $Q(n) \in O\big(\log^h n\big)$, where $f, g, h \in O(1), 0 \leq f \leq g$, then the resulting multi-level data structure requires $O\big(n \log^{f+1} n\big)$ preprocessing space, $O\big(n \log^{g+1} n\big)$ preprocessing time, and $O\big(\sqrt{n} \log^{h+1} n\big)$ query time w.h.p. (n).*

Proof. The preprocessing space of the multi-level structure requires $O(n \log n)$ space for the canonical subsets structure itself, plus

$$\begin{aligned}
\sum_{i=1}^{k} S(|C_i|) &\leq \sum_{i=1}^{k} O\big(|C_i| \log^f |C_i|\big) \\
&\leq O\left(\sum_{i=1}^{k} |C_i| \log^f n\right) \\
&\leq O\left(\log^f n \cdot \sum_{i=1}^{k} |C_i|\right) \\
&\leq O\big(\log^f n \cdot n \log n\big) && \text{(by Theorem 2.3)} \\
&\leq O\big(n \log^{f+1} n\big)
\end{aligned}$$

for the associated structures, for a total space complexity of $O\big(n \log^{f+1} n\big)$.

Preprocessing time is calculated in the same manner. We start with $O(n \log n)$ time for building the canonical subsets structure itself, plus

$$\begin{aligned}
\sum_{i=1}^{k} T(|C_i|) &\leq \sum_{i=1}^{k} O(|C_i| \log^g |C_i|) \\
&\leq O\left(\sum_{i=1}^{k} |C_i| \log^g n \right) \\
&\leq O\left(\log^g n \cdot \sum_{i=1}^{k} |C_i| \right) \\
&\leq O(\log^g n \cdot n \log n) && \text{(by Theorem 2.3)} \\
&\leq O\big(n \log^{g+1} n\big)
\end{aligned}$$

for building the associated structures, for a total time complexity of $O\big(n \log^{g+1} n\big)$.

Querying requires $O(\sqrt{n} \log n)$ time w.h.p. (n) to find the k' disjoint canonical subsets representing the elements found by the top-level canonical subsets query, plus the time required to query the associated data structures. If $Q(n) \in O\big(\sqrt{n} \log^h n\big)$ then the total time to query the appropriate associated structures is

$$\begin{aligned}
\sum_{i=1}^{k'} Q(|C_i|) &\leq \sum_{i=1}^{k'} O\Big(\sqrt{|C_i|} \log^h |C_i|\Big) \\
&\leq O\left(\sum_{i=1}^{k'} \sqrt{|C_i|} \log^h n \right) \\
&\leq O\left(\log^h n \cdot \sum_{i=1}^{k'} \sqrt{|C_i|} \right)
\end{aligned}$$

$$\begin{aligned} &\leq O\big(\log^h n \cdot \sqrt{n}\log n\big) && \text{(by Theorem 2.3)} \\ &\leq O\big(\sqrt{n}\log^{h+1} n\big) \end{aligned}$$

Otherwise, if $Q(n) \in O\big(\log^h n\big)$ then the total time to query the appropriate associated structures is

$$\begin{aligned} \sum_{i=1}^{k'} Q(|C_i|) &\leq \sum_{i=1}^{k'} O\big(\log^h |C_i|\big) \\ &\leq O\left(\sum_{i=1}^{k'} \log^h n\right) \\ &\leq O\left(\log^h n \cdot \sum_{i=1}^{k'} 1\right) \\ &\leq O\big(\log^h n \cdot \sqrt{n}\log n\big) && \text{(by Theorem 2.3)} \\ &\leq O\big(\sqrt{n}\log^{h+1} n\big) \end{aligned}$$

Thus, the total query time, including the associated structures, of $O\big(\sqrt{n}\log^{h+1} n\big)$ w.h.p. (n) is identical for both complexity classes of $Q(n)$ that we consider. □

We use this theorem and corollary as "black boxes" throughout Chapters 3 and 4, and omit the "w.h.p. (n)" for simplicity. We now give an example of how we can use these structures.

Suppose we have n line segments in the plane. Each line segment s_i, $1 \leq i \leq n$ is given by its left endpoint $p_i = (a_i, b_i)$ and its right endpoint $q_i = (c_i, d_i)$. We would like to identify all segments intersected by a query line $L : y = \alpha x + \beta$.

We will solve this query in two steps. First, we identify all segments whose left endpoint is left of L. Of those, we will then identify which have their right endpoint right of L. Describing the construction of a multi-level data structure is often clearer when we begin with the innermost level. With that in mind, we consider the right endpoints first. Considering only the points q_i, $1 \leq i \leq n$, and using L to define a half-plane, we apply Theorem 2.3 to create a structure in $O(n \log n)$ time and space which will identify all points inside the half-plane representing the right side of L in $O(\sqrt{n} \log n)$ time.

Considering the left endpoints now, applying Theorem 2.3 again gives us another structure with identical preprocessing and query times. On this structure, we associate an instance of the structure from step 1 with every subset, constructed on only the elements of its respective subset. By Corollary 2.4, this multi-level data structure will require $O\big(n \log^2 n\big)$ preprocessing time and space, and will identify all segments whose left endpoint is left of L *and* whose right endpoint is right of L in $O\big(\sqrt{n} \log^2 n\big)$ time.

Chapter 3

Partial Enclosure Range Searching with Axis-Parallel Rectangles

In this chapter, we begin with a simple environment for performing partial enclosure range searching. Even with this simplified model, we will be able to identify several challenges inherent to this problem domain and develop some techniques which we will use to solve many of the more complex problems.

We will present two variations of the PERS problem in this chapter. Both use axis-parallel rectangles as their query and line segments as their input. In the first variation, the line segments will also be axis-parallel, whereas in Section 3.2, we allow segments of arbitrary orientation.

3.1 Axis-Parallel Segments

In this section, our input is a set of axis-parallel line segments in the plane and our queries are in the form of axis-parallel rectangles. This is the simplest 2D envi-

ronment on which we can perform PERS as the objects and query region are both simple geometric objects, and we can rely on their fixed orientations while expressing equations about them. We will discuss only horizontal segments here. The solution for vertical segments is identical, and we discuss how to combine the two approaches at the end of the section. We define the problem as follows.

Problem 3.1. Given a set of n axis-parallel line segments in the plane, and a fixed parameter ρ such that $0 < \rho \leq 1$, we want to identify those segments which are sufficiently enclosed by an axis-parallel query rectangle Q so as to satisfy the partial enclosure property. A segment s satisfies this property if and only if $|s \cap Q| \geq \rho \cdot |s|$.

3.1.1 Definitions

Let S be a set of n axis-parallel line segments in the plane. Each segment $s_i \in S$, $1 \leq i \leq n$ is defined by its two endpoints $p_i = (a_i, b_i)$ and $q_i = (c_i, d_i)$. Without loss of generality, we will assume that p_i is the leftmost endpoint of s_i (or the lower endpoint if s_i is vertical). We define $l_i = |s_i|$ as the length of s_i; for horizontal segments the length is easily calculated as $l_i = c_i - a_i$. Our query Q is given by its lower-left corner (α, β) and its upper-right corner (γ, δ). We say that $s_i \in_\rho Q$ if and only if s_i satisfies the partial enclosure property w.r.t. Q, otherwise $s \notin_\rho Q$.

For any x-coordinate, $\overline{x}$ is the vertical line through x. Similarly, for any y-coordinate, $\overline{y}$ is the horizontal line through y. For example, $\overline{\beta}$ is the horizontal line through the bottom boundary of Q, while $\overline{a_i}$ is the vertical line through a_i, and thus, through the left endpoint of s_i. Finally, we define $\overline{s_i}$ as the line through the segment s_i; for a horizontal segment s_i, the lines $\overline{s_i}$, $\overline{b_i}$, and $\overline{d_i}$ are all equivalent. When we focus on any single, general segment, we omit the i subscripts for clarity.

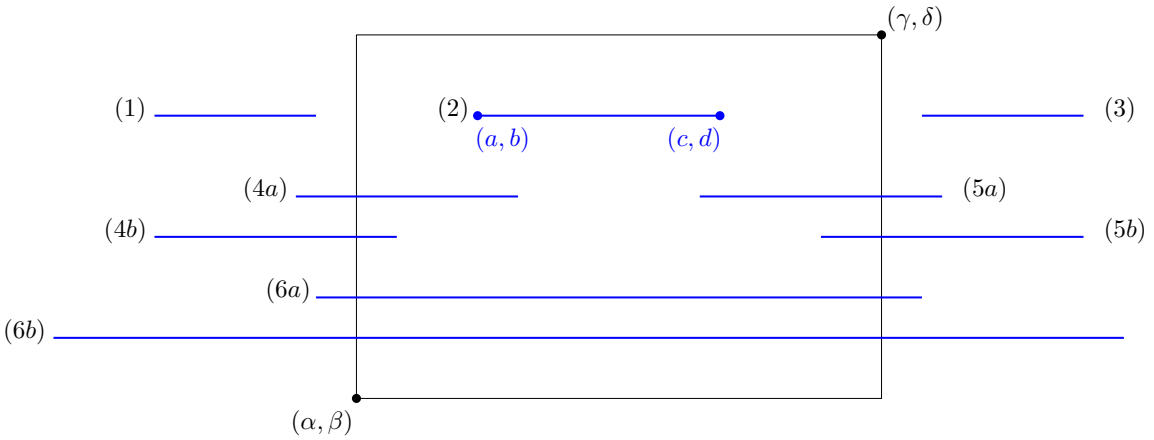

Figure 3.1: An axis-parallel query on axis-parallel segments. Different cases of horizontal segments interacting with the query region are shown.

3.1.2 Decomposing the Problem

As Figure 3.1 illustrates, there are several cases to consider regarding how a horizontal segment $s \in S$ may interact with Q. Cases (1), (2), and (3), where s is entirely left, entirely within, or entirely right of Q, respectively, are exactly an instance of standard range searching problems where we look to find elements which are completely inside or completely outside of a query region.

Case (4) considers segments where s crosses only the left boundary of Q. Looking at it another way, s has its left endpoint left of α, while its right endpoint satisfies $\alpha \leq c \leq \gamma$. We further subdivide case (4) into cases where, for a fixed value of ρ, $s \in_\rho Q$ in case $(4a)$ and $s \notin_\rho Q$ in case $(4b)$. Cases $(5a)$ and $(5b)$ are similar to cases $(4a)$ and $(4b)$, but with respect to γ. Specifically, s falls into case $(5a)$ or $(5b)$ when $\alpha \leq a \leq \gamma$ and $c > \gamma$.

In case (6), s crosses both the left and right boundaries of Q, and, crucially, neither p nor q are inside Q. Just as in cases (4) and (5), s falls into case $(6a)$ if $s \in_\rho Q$ and into case $(6b)$ if $s \notin_\rho Q$.

Our goal is to identify all segments belonging to cases (2), $(4a)$, $(5a)$, and $(6a)$, and none of the segments belonging to any other case.

Our approach to solving this problem involves restricting and classifying segments step-by-step until, for some subset of segments, we are able to identify an expression representing its partial enclosure property. Our solution proceeds as follows.

Step 1. As we are only considering horizontal segments for now, we only need to consider those segments which are neither above nor below Q, since they are the only ones which can intersect Q at all. Let $S_1 = \{s \in S \mid \beta \leq b \leq \delta\}$.

Step 2. Identify those segments which are not "too long". Let $w = \gamma - \alpha$ be the width of Q. Any segment s with length l where $l > \frac{w}{\rho}$ can never have its partial enclosure property satisfied. Let $S_2 = \{s \in S_1 \mid l \leq \frac{w}{\rho}\}$.

Step 3. Partition the remaining segments according the location of their left endpoint with respect to α. Specifically, let $S_L = \{s \in S_2 \mid a < \alpha\}$ and let $S_R = \{s \in S_2 \mid a \geq \alpha\}$.

Step 4. Test an appropriate partial enclosure expression to determine if s should be counted. For segments whose left endpoint is left of α, we want to ensure that "not too much of s is outside of Q"; Let $S'_L = \{s \in S_L \mid \alpha - a < (1 - \rho) \cdot l\}$. For segments whose left endpoint is at or right of α, we want to ensure that "enough of s is inside Q"; Let $S'_R = \{s \in S_R \mid \gamma - a \geq \rho l\}$. Our claim is that the subset of segments satisfying the partial enclosure property is $S_\rho = S'_L \cup S'_R$.

Proof. The correctness of Steps 1 and 2 is straight-forward, and Step 3 merely partitions the remaining segments. Thus, we focus on Step 4 and show that, for each case, a segment belonging to that case is correctly accepted or rejected.

Consider the segments in S_L, which we check against the partial enclosure expression $\alpha - a < (1 - \rho) \cdot l$ measuring how much of s is outside of Q. Segments in case (1) have $\alpha - a > l$, so they are correctly rejected. Segments in cases $(4a)$ and $(4b)$ are accepted or rejected straight-forwardly by this condition as we measure precisely how much of s is outside of Q, and thus, how much is inside.

The most interesting cases are $(6a)$ and $(6b)$, and they are the motivation behind Step 2. After that step, for any segment we are still considering, we know that $l \leq \frac{w}{\rho}$. This allows us to avoid explicitly deciding if s crosses γ, since if not too much of s is left of Q, then s either crosses only α and case (4) holds, or it crosses α and γ, $|s \cap Q| = w$, and w is sufficient.

Next, we consider the segments in S_R, which we check against the partial enclosure expression $\gamma - a \geq \rho l$. For case (2), $\gamma - a \geq l$, and since $l \geq \rho l$ for all valid values of ρ, these segments are accepted. On the other hand, for case (3), $\gamma - a < 0$ and all such segments are rejected. In the final cases, $(5a)$ and $(5b)$, the value of $|s \cap Q|$ is measured directly, and the segments are classified appropriately.

□

3.1.3 Construction and Analysis

Each step of Section 3.1.2 includes or excludes segments by examining inequalities. Each inequality is comprised of segment parameters which we know during preprocessing, and a single query variable expression, i.e., an expression consisting only

of query variables which can be calculated at query time. To perform the query, we map each segment to a point in 4D where each component of the point is based on the segment input parameters corresponding to one of our inequalities, and map the query variables given by Q to a 4D box. The query can then be answered using a multi-level range tree.

Recall that for every $s \in S$, we know a, b, and l, which are the x and y-coordinates of the left endpoint, and the length of the segment, respectively. The query region Q has width $w = \gamma - \alpha$. Let v be the 4D point we create which corresponds to s, constructed as follows.

The first two conditions we need to check are the same for all cases. We first isolate segments based directly on their y-coordinate; thus, the first component of v will be b, and the first component of our 4D query box will be $[\beta, \delta]$. We next want to include only segments where $l \leq \frac{w}{\rho}$. We can rewrite this as $\rho l \leq w$. Thus, the second component of v will be ρl, and the second component of our query box will be $(0, w]$.

Our segments are next classified according to which side of α their left endpoints are found. We set the third component of v to a. The third component of the query box will be either $(-\infty, \alpha)$ or $[\alpha, \infty)$. Our overall query must execute both cases and combine the results later.

Finally, we need to check the partial enclosure expressions. We have two cases to check. The first expression is for segments whose left endpoint is left of α. The partial enclosure expression $\alpha - a < (1 - \rho) \cdot l$ can be rewritten as $\alpha < a + (1 - \rho) \cdot l$. Therefore, we set the fourth component of v to $a + (1 - \rho) \cdot l$, and the fourth component of the query box to (α, ∞). All together, to answer this case, every

segment $s_i \in S$ is mapped to the 4D point

$$v_i = (b_i, \rho l_i, a_i, a_i + (1 - \rho) \cdot l_i)$$

and is queried by the 4D query box

$$[\beta, \delta] \times (0, w] \times (-\infty, \alpha) \times (\alpha, \infty)$$

The next expression is for segments whose left endpoints are at or right of α. The partial enclosure expression $\gamma - a \geq \rho l$ can be rewritten as $\gamma \geq a + \rho l$. Therefore, we set the fourth component of v to $a + \rho l$, and the fourth component of the query box to $(-\infty, \gamma]$. All together, to answer this case, every segment $s_i \in S$ is mapped to the point

$$v_i' = (b_i, \rho l_i, a_i, a_i + \rho l_i)$$

and is queried by the box

$$[\beta, \delta] \times (0, w] \times [\alpha, \infty) \times (-\infty, \gamma]$$

By Corollary 2.2, we can answer either of these queries by constructing a multi-level range tree. As our points are in 4D, such a tree requires $O\big(n \log^3 n\big)$ time and space for preprocessing and can answer queries in $O\big(\log^3 n\big)$ time. To find all horizontal segments satisfying their partial enclosure property, we need to perform both queries and combine the results.

A second data structure and query does not change the asymptotic requirements of the algorithm, however, we note that since both point mappings start with the

same three components, we can save a constant factor of space by sharing the first three levels of the multi-level range tree. At the third level of the tree, we create *two* associated structures, one for each of the partial enclosure expressions, and query each one as needed. In a similar way, we can save a constant factor of time during our queries, since both queries begin with the same two values.

Vertical Segments. The method for querying vertical segments is very similar to that for horizontal segments, except that we care about the height of Q rather than its width, and need to consider symmetric coordinates of each segment when going through each of the 4 steps. Preprocessing and querying this extra orientation does not change our asymptotic analysis, since the number of orientations remains fixed.

Combining the Steps. We maintain the structures necessary for both the horizontal and vertical portions of the query. At query time, each structure is queried independently. The subtrees found in the last level of each structure contain matching segments, and these results can be combined in a straight-forward manner for counting or reporting queries. The following theorem summarizes the solution.

Theorem 3.1. *Given a set of n axis-parallel line segments, we can identify a set of disjoint subsets containing all segments which satisfy the partial enclosure property for an axis-parallel query rectangle in $O\big(\log^3 n\big)$ time, with a data structure requiring $O\big(n \log^3 n\big)$ preprocessing time and space.*

3.2 Arbitrarily-Oriented Segments

In this section, we relax the restriction that all segments should be axis-parallel, allowing each segment to have any arbitrary orientation with respect to one another. It does not matter if segments intersect. The problem statement is as follows.

Problem 3.2. Given a set of n arbitrarily-oriented line segments in the plane, and a fixed parameter ρ such that $1/2 < \rho \leq 1$, we want to identify those segments which are sufficiently enclosed by an axis-parallel query rectangle Q so as to satisfy the partial enclosure property. A segment s satisfies this property if and only if $|s \cap Q| \geq \rho \cdot |s|$.

Throughout this section, the query region Q and the set of segments S follow the same definitions as in Section 3.1.1. Observe that the definition for ρ is different than in Section 3.1; we discuss this difference in the conclusion of this chapter.

3.2.1 Decomposing the Problem

There are three principal cases to consider; those segments which have both endpoints inside Q, those with only one endpoint inside Q, and those with both endpoints outside Q.

The first case, where both endpoints are inside Q, is answered by mapping each segment to a point in 4D space and then using a multi-level range tree, much like we did in the previous section. Specifically, we map each segment s_i to the point $v_i = (a_i, b_i, c_i, d_i)$. The query Q is mapped to the following 4D box

$$[\alpha, \gamma] \times [\beta, \delta] \times [\alpha, \gamma] \times [\beta, \delta]$$

In essence, we are just testing to ensure that both endpoints appear within query rectangle. The second case, where only one endpoint is inside Q is solved using the same method as the first case, but on a "virtual" segment. For each segment s_i, we create virtual segments which share endpoints with s_i. Specifically, we define the segment $s_{i,p} \subseteq s_i$ where s_i and $s_{i,p}$ share the endpoint p_i and $|s_{i,p}| = \rho \cdot |s_i|$. Likewise, we define the segment $s_{i,q} \subseteq s_i$ where s_i and $s_{i,q}$ share the endpoint q and $|s_{i,q}| = \rho \cdot |s_i|$.

Then, as a direct consequence of their construction, if $s_{i,p} \in Q$, then $s_i \in_\rho Q$, and if $s_{i,q} \in Q$, then $s_i \in_\rho Q$. Counting both cases will double-count any segment which has both endpoints in Q, so we subtract accordingly.

We can query these higher dimensional points by applying the multi-level range tree method given by Theorem 2.1.

3.2.2 Handling Endpoints Outside of Q

The case where both endpoints of a segment are outside Q is the most challenging case to handle. We begin by partitioning the space around Q into 8 regions by extending lines through its horizontal and vertical boundaries. These regions are labelled anti-clockwise from left-middle as $I, II, \ldots, VIII$; see Figure 3.2. Any segment which passes through Q but which has neither endpoint in Q will have its endpoints in two distinct regions. Not every pair of regions is legal, however. For example, a segment with one endpoint in region I and another in region II cannot intersect Q.

A segment which passes through Q may involve any of the following pairs of regions, falling into 4 classes:

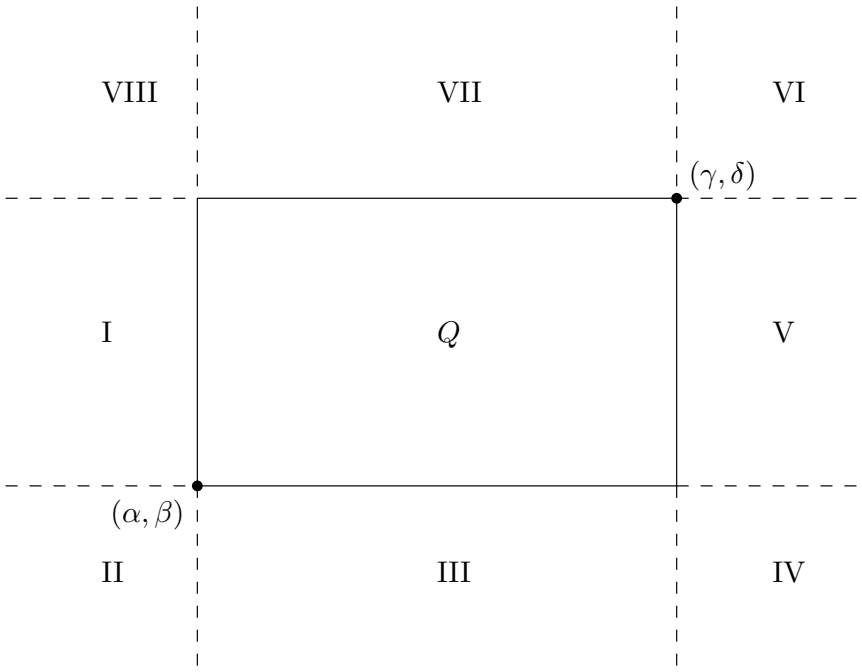

Figure 3.2: The 8 regions surrounding Q.

i. Parallel sides: $(I, V), (III, VII)$.

ii. Perpendicular sides: $(I, III), (III, V), (V, VII), (VII, I)$.

iii. Diagonal to Orthogonal: (II, V), (II, VII), (IV, I), (IV, VII), (VI, I), (VI, III), $(VIII, III)$, $(VIII, V)$.

iv. Diagonal to Diagonal: $(II, VI), (IV, VIII)$.

We will query for each case separately and combine our results at the end. For a segment $s \in S$, we can identify the regions of p and q using a 4-dimensional orthogonal range search. This classification determines which partial enclosure expressions we need to test in the second phase of the query. We develop expressions for each case below.

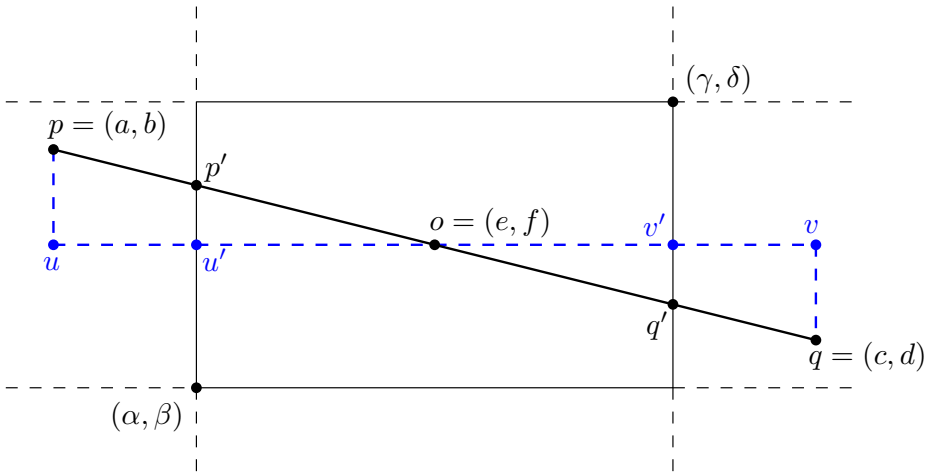

Figure 3.3: An example of a segment in class (i), case (I, V).

In all cases, we require the following additional definitions. Let $o_i = (e_i, f_i)$ be the centrepoint of s_i. We define the function $d(u, v)$ as the Euclidean distance between any two points u and v. We define $L_i = d(p_i, q_i)/2$ to be exactly half the length of s_i (therefore, $d(p_i, o_i) = d(q_i, o_i) = L_i$). As with our previous definitions, when we discuss any single, general segment $s \in S$, we will drop the i subscripts for clarity.

Class (i); e.g., $(p, q) \in (I, V)$

Class (i) is concerned with segments which cross parallel sides of Q. We present the details of the partial enclosure property for the case where $p \in I$ and $q \in V$; the solution for the (III, VII) case is similar.

Let $s \in S$ be a segment belonging to class (i), case (I, V). Assume for now that $o \in Q$ and let p' (q') be the intersection of s with Q closest to p (q). Let $u = (a, f)$ be the point vertically aligned with p and horizontally aligned with o. Then, Δpuo is a

right triangle, and $u' = (\alpha, f)$ is the intersection point of $\overline{uo}$ with the boundary of Q. The triangle $\Delta p'u'o$ is a right triangle and is similar to Δpuo. Likewise, let $v = (c, f)$ be the point vertically aligned with q and horizontally aligned with o. Then, Δqvo is a right triangle, and $v' = (\gamma, f)$ is the intersection point of $\overline{vo}$ with the boundary of Q. The triangle $\Delta q'v'o$ is a right triangle and is similar to Δqvo. See Figure 3.3.

We will test the partial enclosure property by checking that "not too much of s is outside of Q". Specifically, we need to find those segments where $\frac{d(p,p')+d(q,q')}{d(p,q)} \leq 1 - \rho$. By similarity of triangles,

$$\frac{d(p,p')}{d(p,o)} = \frac{d(p,p')}{L} = \frac{d(u,u')}{d(u,o)} = \frac{2d(u,u')}{2d(u,o)} = \frac{2(\alpha - a)}{c - a}$$

and

$$\frac{d(q,q')}{d(q,o)} = \frac{d(q,q')}{L} = \frac{d(v,v')}{d(v,o)} = \frac{2d(v,v')}{2d(v,o)} = \frac{2(c - \gamma)}{c - a}$$

Thus,

$$\begin{aligned}
\frac{d(p,p') + d(q,q')}{d(p,q)} &= \frac{d(p,p') + d(q,q')}{2L} \\
&= \frac{1}{2}\left(\frac{d(p,p')}{L} + \frac{d(q,q')}{L}\right) \\
&= \frac{1}{2}\left(\frac{2(\alpha - a)}{c - a} + \frac{2(c - \gamma)}{c - a}\right) \\
&= \frac{\alpha - a}{c - a} + \frac{c - \gamma}{c - a} \\
&= \frac{\alpha - a + c - \gamma}{c - a} \\
&= 1 + \frac{\alpha - \gamma}{c - a}
\end{aligned}$$

Therefore, the inequality $\frac{d(p,p')+d(q,q')}{d(p,q)} \leq 1 - \rho$ can instead be evaluated as $1 + \frac{\alpha-\gamma}{c-a} \leq$

$1-\rho$, which is based on two query variables α and γ. We can further simplify the expression to

$$\rho(c-a) \leq \gamma - \alpha$$

where the value of $\gamma - \alpha$ is a single query variable expression calculated at query time. This inequality can therefore be checked by an orthogonal range search similar to the techniques used in Section 3.1.2.

When $o \notin Q$, we cannot construct similar triangles in the same way, since we require the points of intersection between the triangles and Q. However, since we know that any segment which *is* sufficiently enclosed by Q will have its centrepoint in Q, it is enough for our purposes if our test condition always reports a failure when o is outside of Q.

We know that both p and q are outside of Q, so it must be the case that $\frac{\alpha-a}{c-a} > 0$ and $\frac{c-\gamma}{c-a} > 0$. Observe that if o is left of α, then $\frac{\alpha-a}{c-a} > 1/2$. Likewise, if o is right of γ, then $\frac{c-\gamma}{c-a} > 1/2$. Therefore, the sum $\frac{\alpha-a}{c-a} + \frac{c-\gamma}{c-a} > 1/2$ and is strictly larger than any allowable value for $(1-\rho)$, resulting in the segment being rejected as desired.

Class (ii); e.g., $(p, q) \in (I, III)$

Class (ii) is concerned with segments which cross perpendicular sides of Q. We present the details of the partial enclosure property for the case where $p \in I$ and $q \in III$; the solutions for other cases of class (ii) are similar. These expressions are not so different from the ones given in class (i), however we will see that we get two query variable expressions instead of one, and this has ramifications for how we will construct our data structure.

Assume for now that $o \in Q$ and let p' (q') be the intersection of s with Q closest

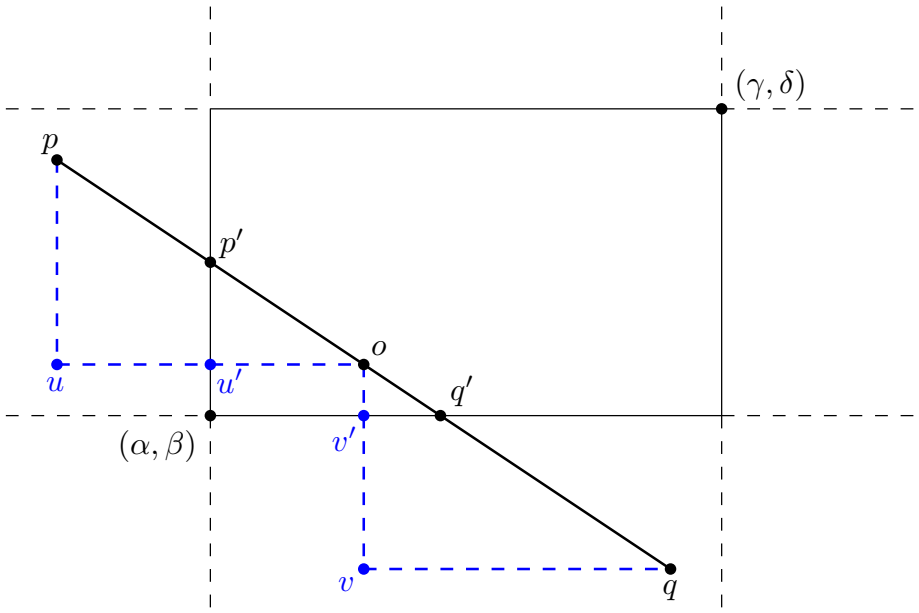

Figure 3.4: An example of a segment in class (ii), case (I, III). This particular segment is not sufficiently enclosed by Q for any allowable value of ρ.

to p (q). Let $u = (a, f)$ be the point under p directly left of o. Then, Δpuo is a right triangle, and $u' = (\alpha, f)$ is the intersection point of $\overline{uo}$ with the boundary of Q. The triangle $\Delta p'u'o$ is a right triangle and is similar to Δpuo. Likewise, let $v = (e, d)$ be the point left of q directly under o. Then, Δqvo is a right triangle, and $v' = (e, \beta)$ is the intersection point of $\overline{vo}$ with the boundary of Q. The triangle $\Delta q'v'o$ is a right triangle and is similar to Δqvo. See Figure 3.4.

We need to find those segments where $\frac{d(p,p')+d(q,q')}{d(p,q)} \leq 1 - \rho$. By similarity of triangles,

$$\frac{d(p,p')}{d(p,o)} = \frac{d(p,p')}{L} = \frac{d(u,u')}{d(u,o)} = \frac{2d(u,u')}{2d(u,o)} = \frac{2(\alpha - a)}{c - a}$$

and

$$\frac{d(q,q')}{d(q,o)} = \frac{d(q,q')}{L} = \frac{d(v,v')}{d(v,o)} = \frac{2d(v,v')}{2d(v,o)} = \frac{2(\beta - d)}{b - d}$$

Thus,

$$\begin{aligned}\frac{d(p,p')+d(q,q')}{d(p,q)} &= \frac{d(p,p')+d(q,q')}{2L} \\ &= \frac{1}{2}\left(\frac{d(p,p')}{L}+\frac{d(q,q')}{L}\right) \\ &= \frac{1}{2}\left(\frac{2(\alpha-a)}{c-a}+\frac{2(\beta-d)}{b-d}\right) \\ &= \frac{\alpha-a}{c-a}+\frac{\beta-d}{b-d}\end{aligned}$$

Therefore, the inequality $\frac{d(p,p')+d(q,q')}{d(p,q)} \leq 1-\rho$ can be tested by checking the equivalent inequality $\frac{\alpha-a}{c-a}+\frac{\beta-d}{b-d} \leq 1-\rho$, defined on the two query variables α and β. We will perform this test using a half-plane range query. To construct an appropriate dual-space for the half-plane query, we reorder the expression as follows:

$$\begin{aligned}\frac{\alpha-a}{c-a}+\frac{\beta-d}{b-d} &\leq 1-\rho \\ \frac{\alpha}{c-a}-\frac{a}{c-a}+\frac{\beta}{b-d}-\frac{d}{b-d} &\leq 1-\rho \\ \alpha\cdot\frac{1}{c-a}+\beta\cdot\frac{1}{b-d} &\leq (1-\rho)+\frac{a}{c-a}+\frac{d}{b-d} \\ \alpha+\beta\cdot\frac{c-a}{b-d} &\leq \left((1-\rho)+\frac{d}{b-d}\right)\cdot(c-a)+a\end{aligned}$$

We then map each segment s to a point with coordinates

$$\left(\frac{c-a}{b-d}, \left((1-\rho)+\frac{d}{b-d}\right)\cdot(c-a)+a\right)$$

In this space, segments matching the partial enclosure expression correspond to dual-points satisfying the half-plane $y \geq \beta x+\alpha$

Just as in class (i), when $o \notin Q$, we cannot construct appropriate similar trian-

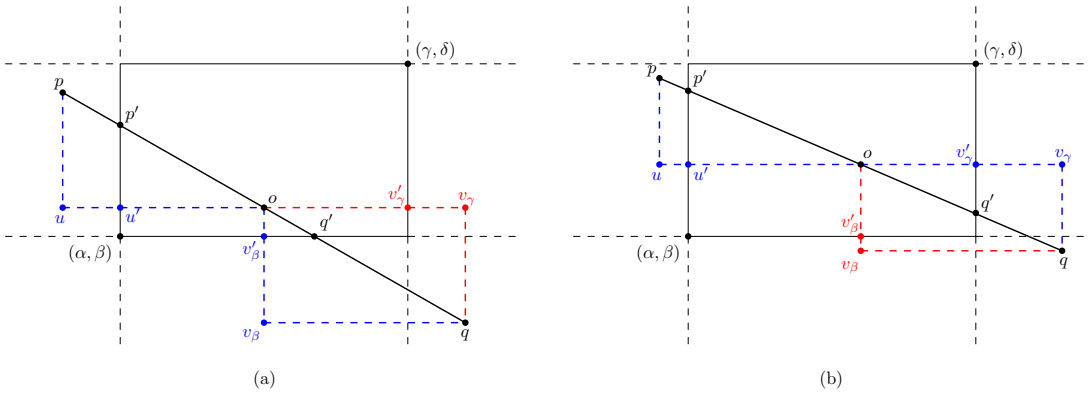

Figure 3.5: An example of segments from each subcase of class (iii), case (I, IV). The blue follows the proper handling of the subcase, while the red shows the incorrect handling.

gles for our method, but it remains sufficient for the expressions to reject any such segment. Observe that if o is left of α, then $\frac{\alpha - a}{c - a} > 1/2$. Likewise, if o is below β, then $\frac{\beta - d}{b - d} > 1/2$. Their sum is therefore strictly greater than $1/2$, and larger than any allowable value for $(1 - \rho)$, rejecting the segment as desired.

Class (iii); e.g., $(p, q) \in (I, IV)$

Class (iii) is concerned with segments which have one endpoint in a corner region of Q, and the other endpoint in an orthogonal region of Q. We present the details of the partial enclosure property for the case where $p \in I$ and $q \in IV$; the solutions for other cases of Class (iii) are similar.

Assume for now that $o \in Q$. We need to consider two subcases: (a) s crosses α and β or (b) s crosses α and γ. See Figure 3.5 for examples. Subcase (a) is very nearly exactly the example presented for class (ii). In that case, the only use of the fact that the endpoints of s were located in regions I and III was to imply that s crossed α and β. As such, this subcase can use the same expression for testing s.

Likewise, subcase (b) is similar to the example presented for class (i) and can use exactly that expression for testing s.

Our initial query for identifying the regions of p and q does not allow us to differentiate between subcases. Instead, we will check both subcases simultaneously. From class (i) and class (ii), we have the following expressions:

$$\frac{\alpha - a}{c - a} + \frac{c - \gamma}{c - a} \leq 1 - \rho$$

and

$$\frac{\alpha - a}{c - a} + \frac{\beta - d}{b - d} \leq 1 - \rho$$

If s belongs to subcase (a), then $\frac{c-\gamma}{c-a} \leq \frac{\beta-d}{b-d}$ since γ is farther right than the point where s exits Q. Likewise, in subcase (b), $\frac{\beta-d}{b-d} \leq \frac{c-\gamma}{c-a}$. Therefore, in either subcase, the result of the correct expression is larger than the incorrect one, allowing us to correctly reject segments by blindly checking both conditions. This also holds when $o \notin Q$, since the expression for at least one subcase would exclude the segment.

Class (iv); e.g., $(p, q) \in (II, VI)$

Class (iv) is concerned with segments whose endpoints appear in diagonally opposite corner regions of Q. We present the details of the partial enclosure property for the case where $p \in II$ and $q \in VI$; the solution for the $(VI, VIII)$ case is similar.

Assume for now that $o \in Q$. As with class (iii), we need to consider several subcases depending on which sides of Q are intersected by s:

(a) s crosses α and δ,

(b) s crosses α and γ,

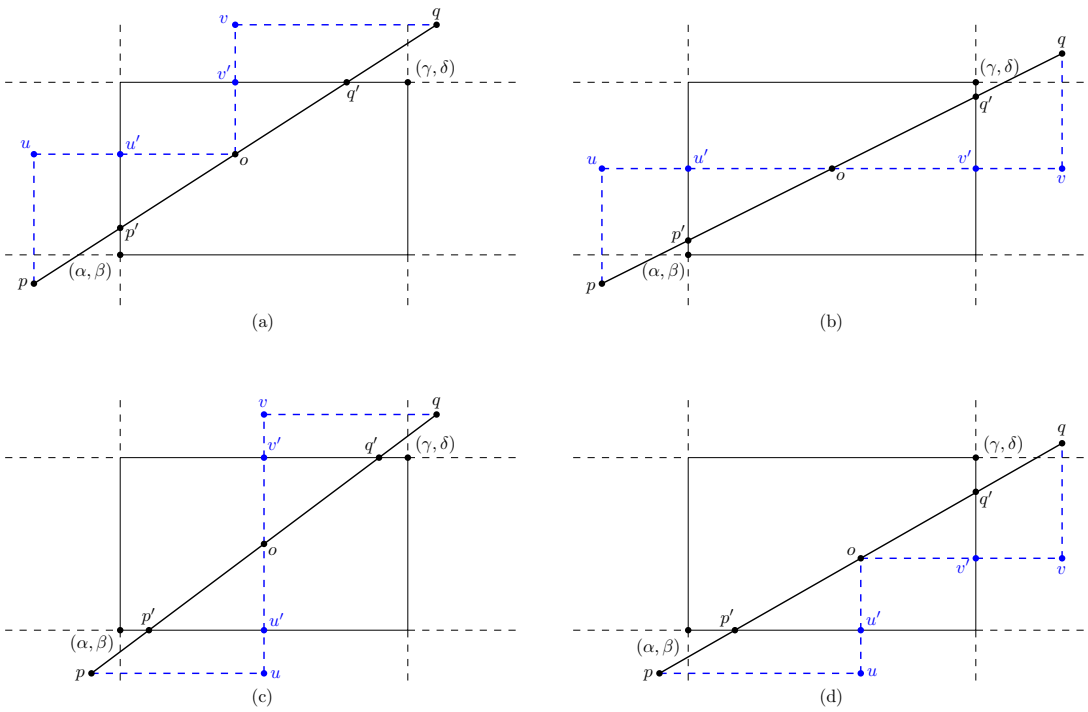

Figure 3.6: An example of segments from each subcase of class (iv), case (II, VI). The blue lines show the proper handling of each.

(c) s crosses β and δ,

(d) s crosses β and γ,

See Figure 3.6 for examples of each subcase. Our initial query for identifying the regions of p and q does not allow us to differentiate between subcases. However, it is sufficient to check all subcases simultaneously. The expression for subcase (b) comes directly from class (i). Using similar techniques as we did for classes (i) and (ii), we develop the following set of partial enclosure expressions:

$$(a) \quad \frac{\alpha - a}{c - a} + \frac{d - \delta}{d - b} \leq 1 - \rho \qquad (b) \quad \frac{\alpha - a}{c - a} + \frac{c - \gamma}{c - a} \leq 1 - \rho$$

$$(c) \quad \frac{\beta - b}{d - b} + \frac{d - \delta}{d - b} \leq 1 - \rho \qquad (d) \quad \frac{\beta - b}{d - b} + \frac{c - \gamma}{c - a} \leq 1 - \rho$$

Subcases (b) and (c) can be simplified to orthogonal range queries, while subcases (a) and (d) will need to be mapped to appropriate dual-spaces and queried with half-planes as in class (ii).

To illustrate that checking all four conditions simultaneously yields correct results, consider what happens if s belongs to subcase (a), where s crosses α and δ. In these circumstances, u' is above β, and we have that $\frac{\alpha - a}{c - a} \geq \frac{\beta - b}{d - b}$. Likewise, v' is left of γ, so $\frac{d - \delta}{d - b} \geq \frac{c - \gamma}{c - a}$. Therefore, the expression for (a) dominates the other expressions (i.e., (a) $\geq$ (b), (a) $\geq$ (c), and (a) $\geq$ (d)), allowing us to identify qualifying segments for subcase (a) by blindly checking all conditions. This property is true for segments belonging to subcases (b), (c), and (d) as well. Furthermore, this test also holds when $o \notin Q$, since the expression for at least one subcase would exclude the segment.

3.2.3 Construction and Analysis

Our method takes a very case-by-case approach to solving the problem, and executes in two phases. The first phase must classify segments belonging to each of the 16 pairs of regions around Q which are of interest. For each case, the second phase must then identify those segments satisfying their respective partial enclosure property.

Broadly, our solution to this problem uses a multi-level range tree (Section 2.1) for the classification phase, and a combination of range trees and canonical subsets

structures (Section 2.2) to check the partial enclosure expressions.

Identifying the segments which satisfy each case of each class is a matter of testing a set of several conditions, all of which must be true. The order that we test the conditions in makes no difference to the correctness of the algorithm, but can have an impact on its space requirements.

Class (i), $(p, q) \in (I, V)$ **case.** To identify these segments, we need to find those segments whose endpoints are in the appropriate regions, and which satisfy the single, orthogonal partial enclosure expression for the case. Using an approach similar to Section 3.1.3, for each $s_i \in S$, we construct a corresponding 5D point v_i as follows.

$$v_i = (a_i, b_i, c_i, d_i, \rho(c_i - a_i))$$

We then query these points with the 5D box:

$$(-\infty, \alpha) \times [\beta, \delta] \times (\gamma, \infty) \times [\beta, \delta] \times (-\infty, \gamma - \alpha]$$

The following lemma summarizes how we can query this class of segments.

Lemma 3.2. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments belonging to class (i)* and *which satisfy their partial enclosure property in $O(\log^4 n)$ time using a data structure requiring $O(n \log^4 n)$ preprocessing time and space.*

Class (ii), $(p, q) \in (I, III)$ **case.** As with all cases, part of the problem involves first identifying those segments with endpoints in the appropriate regions. In this case, however, the partial enclosure expression is evaluated using a half-plane query.

The classification portion is performed just as we have seen above. We map each segment to the 4D point $v_i = (a_i, b_i, c_i, d_i)$. The partial enclosure expression will be queried by the dual-space we developed in Section 3.2.2. For each $s_i \in S$, we define $h_i = \left(\frac{c-a}{b-d}, \left((1-\rho) + \frac{d}{b-d}\right) \cdot (c-a) + a\right)$. We need to construct a data structure which can answer queries on pairs (v_i, h_i).

By Theorem 2.1, we can query the classification component using a range tree according to the following lemma.

Lemma 3.3. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments belonging to class (ii) in $O\left(\log^3 n\right)$ time using a data structure requiring $O\left(n \log^3 n\right)$ preprocessing time and space.*

By Theorem 2.3, we can query the half-plane component of our query objects using a canonical subsets data structure, giving the following lemma.

Lemma 3.4. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments satisfying a half-plane representation of a partial enclosure expression in $O(\sqrt{n} \log n)$ time, using a data structure requiring $O(n \log n)$ preprocessing time and space.*

Since the order that we check our conditions in does not affect correctness, we will check the half-plane condition first, then the endpoint classification. By Corollary 2.4, we can accomplish this by associating a classification structure with each subset of the half-plane structure. The resulting structure is summarized by the following lemma.

Lemma 3.5. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments belonging to class (ii)* and *which satisfy their partial enclosure*

property in $O\big(\sqrt{n}\log^4 n\big)$ time using a data structure requiring $O\big(n\log^4 n\big)$ preprocessing time and space.

Class (iii); $(p,q) \in (I, IV)$ case. This case can use the same orthogonal structure that we developed in Lemma 3.2 and the same half-plane expression as in Lemma 3.4. One component of the query box needs to be updated to account for region IV, giving the following.

$$(-\infty, \alpha) \times [\beta, \delta] \times (\gamma, \infty) \times (-\infty, \beta) \times (-\infty, \gamma - \alpha]$$

By Corollary 2.4, we can combine these two data structures to create a new structure which can answer this case as summarized by the following lemma.

Lemma 3.6. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments belonging to class (iii)* and *which satisfy their partial enclosure property in $O\big(\sqrt{n}\log^5 n\big)$ time using a data structure requiring $O\big(n\log^5 n\big)$ preprocessing time and space.*

Class (iv); $(p,q) \in (II, VI)$ case. This case has the largest number of partial enclosure expressions which need checking, in addition to the usual endpoint classification step. As a result, it will be the largest data structure to build.

The basic steps are just as in the last three cases. To cover endpoint classification and the two orthogonal partial enclosure expressions, we will use a data structure and query box similar to Lemma 3.2, but extended to 6D to cover the extra partial enclosure expression. We then apply Lemma 3.4 and Corollary 2.4 twice to handle the two half-plane partial enclosure expressions and associate the orthogonal range

tree. The entire structure for this case is summarized by the following lemma.

Lemma 3.7. *Given a set of n line segments, we can identify a set of disjoint subsets containing all segments belonging to class (iv)* and *which satisfy their partial enclosure property in $O\big(\sqrt{n}\log^7 n\big)$ time using a data structure requiring $O\big(n\log^7 n\big)$ preprocessing time and space.*

Combining the Steps. Querying the entire environment requires us to create the structures for handling the cases where one or both endpoints of a line segment lie entirely inside a query Q, as well as the structures for each of the cases when both endpoints lie outside of Q. Overall, this process is dominated by the structure required for the class (iv) segments. The following theorem summarizes the overall solution.

Theorem 3.8. *Given a set of n arbitrarily-oriented line segments, we can identify a set of disjoint subsets containing all segments which satisfy the partial enclosure property for an axis-parallel query rectangle in $O\big(\sqrt{n}\log^7 n\big)$ time, using a data structure requiring $O\big(n\log^7 n\big)$ preprocessing time and space.*

3.3 Conclusion

In this chapter, we have developed methods for answering the partial enclosure range searching problem on both axis-parallel and arbitrarily-oriented line segments with axis-parallel query rectangles. We have seen that the most challenging cases occur when both endpoints of a segment exist outside of the query region. We have also seen how we can transform partial enclosure expressions into dual-spaces

which can be easily queried with a half-plane.

There are several possibilities for future work. Most notably is the condition that $\rho > 1/2$ for the arbitrary-orientation problem, which is the only problem with such a limitation presented in this thesis. The limitation stems from our use of similar triangles anchored on the centrepoint of the segment. When $\rho > 1/2$, the centrepoint of any segment satisfying the partial enclosure property will be found inside the query region, guaranteeing intersections between Q and our similar triangles.

We would also like to reduce how many simultaneous partial enclosure expressions need to be checked. Having to check multiple expressions is a direct result of the uncertainty of which boundary of Q a segment intersects. If we could directly identify the intersected boundaries with a better classification strategy, the number of nested levels required by our data structures could be reduced.

Finally, with the axis-parallel case, we are able to save a constant factor of space by sharing the classification steps between queries. In the arbitrary-orientation case, we perform our half-plane queries first as this saves us a $\log n$ factor in our asymptotic preprocessing needs. A side effect of this choice is that we cannot share the endpoint classification structures which are common to all cases. Some experimentation is needed to identify for which inputs this trade-off makes practical sense.

Chapter 4

Partial Enclosure Range Searching with Arbitrarily-Oriented Slabs

In this chapter, we show how we can answer partial enclosure range searches over horizontal line segments where queries are in the form of an arbitrarily-oriented slab. In Section 4.2, we extend our solution to query regions which are the intersection of two slabs; this type of query is a generalization of querying with an arbitrarily-oriented rectangle.

4.1 Querying with One Slab

In this section, we develop a method for identifying horizontal line segments which are sufficiently enclosed within a single, arbitrarily-oriented query slab.

4.1.1 Problem Definition

The formal statement of our problem is as follows.

Problem 4.1. Given a set of n horizontal line segments in the plane, and a fixed parameter ρ such that $0 < \rho \leq 1$, we want to identify those segments which are sufficiently enclosed by an arbitrarily-oriented (i.e., slanted) slab Q to satisfy the partial enclosure property. A segment s satisfies this property if and only if $|s \cap Q| \geq \rho \cdot |s|$.

Throughout this section, we use the following definitions. Let S be a set of n horizontal line segments. Each segment $s_i \in S, 1 \leq i \leq n$ is defined by three values a_i, b_i, l_i which in turn define the endpoints of the segment as $p_i = (a_i, b_i)$ and $q_i = (a_i + l_i, b_i)$. When discussing a single, general segment, we omit the i subscripts for clarity.

A query Q is given as three inputs: α, β, w which we use to define an arbitrarily-oriented slab. The left edge of the slab is defined by $L_1 : y = \alpha x + \beta$, while the right edge defined by $L_2 : y = \alpha x + \beta - \alpha w$. Thus, w is the horizontal width of Q.

We say that $s_i \in_\rho Q$ if and only if s_i satisfies the partial enclosure property with respect to Q, otherwise $s \notin_\rho Q$.

4.1.2 Identifying the Segments

Identifying segments which satisfy the partial enclosure property requires three broad steps:

1. Restrict segments to those which are “not too long” to fit sufficiently inside Q.

2. Classify all segments by whether their left endpoints are left or right of L_1.

3. For each class of segments, test an appropriate partial enclosure expression.

To answer these queries, we will use a multi-level canonical sets data structure, as described in Section 2.2. We describe the different steps of the query in more detail next, while Section 4.1.3 describes the construction and analysis of the overall data structure.

Restrict Length

The first step of the query is to perform a length test, as it simplifies future steps. We want to identify all segments which are not so long that they could never be sufficiently enclosed by Q. With the query parameter w given, only segments with length $l \leq \frac{w}{\rho}$ can satisfy the partial enclosure property. Reworking the equation to $w \geq \rho l$ provides a 1-dimensional orthogonal range to query. Let $S_1 = \{s \in S \mid \rho l \leq w\}$.

Classify Endpoints

There are three different regions where we may find the left endpoints of our line segments: (1) left of L_1, (2) between L_1 and L_2, and (3) right of L_2. However, only those segments belonging to cases (1) and (2) are interesting to us, since any segment in case (3) cannot intersect Q at all. We will see that partitioning segments as left or right of L_1 is sufficient, as we can discriminate between cases (2) and (3) when testing partial enclosure expressions in the next step.

Identifying segments whose left endpoints appear to one side of L_1 can be

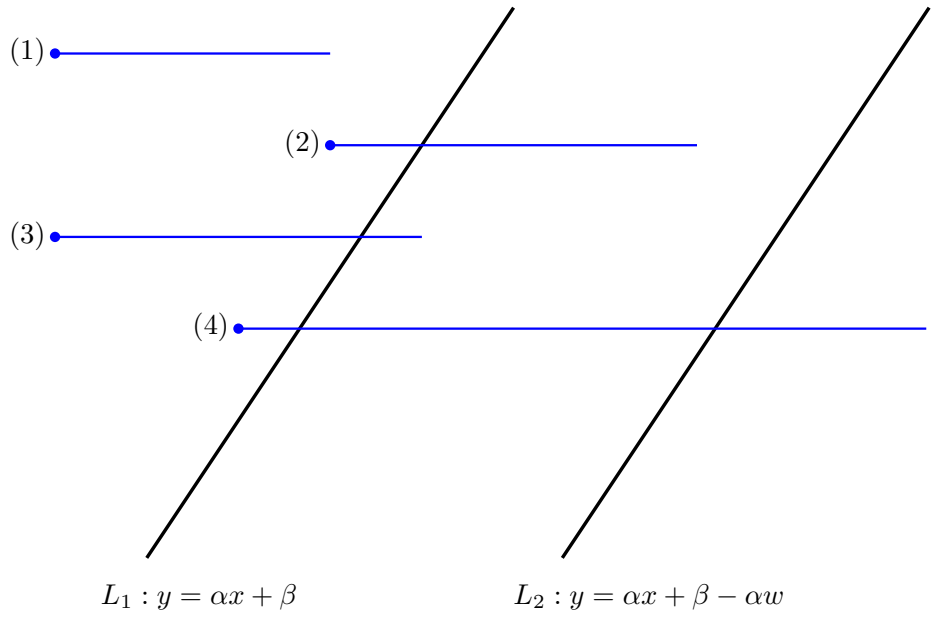

Figure 4.1: Segments whose left endpoints are left of L_1.

accomplished using a half-plane query directly on the positions of the left endpoints and the definition of L_1 itself. Let $S_L = \{s \in S_1 \mid p \text{ is left of } L_1\}$, and let $S_R = \{s \in S_1 \mid p \text{ is right of } L_1\}$.

Check the Partial Enclosure Property

For each of S_L and S_R, the final step is to identify those segments which satisfy the partial enclosure property.

Set S_L, left endpoint left of L_1. There are 4 cases of segments whose left endpoints are left of L_1, as shown in Figure 4.1.

1. The entire segment is left of L_1 (and should not be counted),

2. The segment intersects L_1, and the segment is sufficiently enclosed by Q,

3. The segment intersects L_1, but the segment is insufficiently enclosed by Q (and should not be counted),

4. The segment intersects both L_1 and L_2.

Given a segment $s \in S_L$ with left endpoint $p = (a, b)$, let $\overline{s}$ be the line through s, given by the equation $y = b$ (the slope is 0). Let (a', b) be the intersection point of $\overline{s}$ and L_1. Solving the equations of these two lines gives $a' = \frac{b-\beta}{\alpha}$. It is sufficient to show that $s \in_\rho Q$ when "not too much of s is outside of Q", which we can test with the expression $a' - a < (1 - \rho)l$.

Proof. We look at each case to show that this single expression is enough to identify all segments correctly. First, the test directly identifies segments belonging to cases (2) or (3) since $a' - a$ is precisely the amount of s outside of Q. This test will also reject segments, as we desire, from case (1) since $a' - a > l$ and cannot satisfy the partial enclosure property for any allowed value of ρ.

Finally, this test is also sufficient for case (4) owing to the length restriction step we perform in Section 4.1.2. If a segment is not too far left of L_1, then either it crosses only L_1, and case (2) holds, or it crosses L_1 and L_2. In the latter case we know that $|s| < \frac{w}{\rho}$, where w is the width of the query slab. Since any segment in case (4) has the property $|s \cap Q| = w$, this implies that $s \in_\rho Q$. □

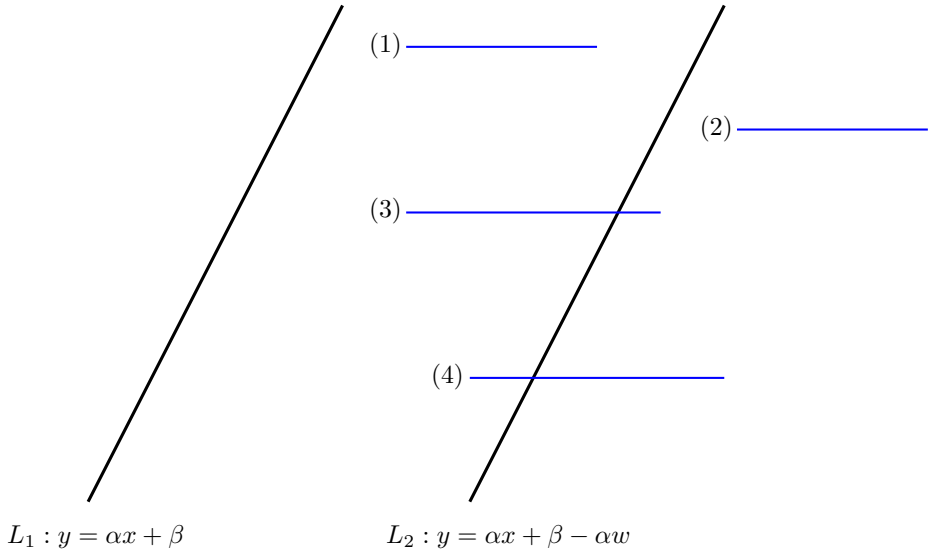

Figure 4.2: Segments whose left endpoints are between L_1 and L_2.

We test for segments satisfying $a' - a < (1-\rho)l$ by transforming it to a half-plane query. Expanding this expression gives:

$$\begin{aligned} a' - a &< (1-\rho)l \\ \frac{b-\beta}{\alpha} - a &< (1-\rho)l \\ a + (1-\rho)l &> \frac{1}{\alpha}b - \frac{\beta}{\alpha} \end{aligned}$$

From this inequality, we can construct an appropriate dual-space to perform the half-plane query. We map each segment s to a dual-point with coordinates $(b, a+(1-\rho)l)$. Segments matching the partial enclosure expression then correspond to the dual-points satisfying the half-plane $y > \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

Set S_R, left endpoint right of L_1. There are 4 cases of segments whose left-endpoints are right of L_1, as shown in Figure 4.2.

1. The entire segment is between L_1 and L_2. Since $s \in Q$ implies $s \in_\rho Q$, this segment should be counted.

2. The entire segment is right of L_2. Since $s \cap Q = \emptyset$ implies $s \notin_\rho Q$, this segment should not be counted.

3. The segment intersects L_2, and $s \in_\rho Q$.

4. The segment intersects L_2, but $s \notin_\rho Q$.

For each $s \in S_R$, we test the partial enclosure property by checking that "enough of the segment is inside Q". Just as we did with the segments of S_L, let $\overline{s}$ be the horizontal line through s. Let (a'', b) be the intersection point of $\overline{s}$ with L_2. Solving their respective equations gives $a'' = \frac{b-\beta}{\alpha} + w$. We will show that a segment satisfies the partial enclosure property when $a'' - a \geq \rho l$.

Proof. If $s \in S_R$ belongs to case (1), we have that $a'' - a > l$, which is certainly greater than ρl. If s belongs to case (2), then $a'' - a < 0$ since a is right of a'', and s will not be counted. Finally, the correctness of this test is straight-forward when s is in cases (3) or (4), since the expression directly measures $|s \cap Q|$. □

We test for segments satisfying $a'' - a \geq \rho l$ using a half-plane query. Expanding this expression gives:

$$\begin{aligned} a'' - a &\geq \rho l \\ \frac{b-\beta}{\alpha} + w - a &\geq \rho l \\ \rho l + a &\leq \frac{1}{\alpha} b - \frac{\beta}{\alpha} + w \end{aligned}$$

From this inequality, we can construct an appropriate dual-space to perform the half-plane query. We map each segment s to a dual-point with coordinates $(b, \rho l + a)$. Segments matching the partial enclosure expression then correspond to the dual-points satisfying the half-plane $y \leq \frac{1}{\alpha}x - \frac{\beta}{\alpha} + w$.

4.1.3 Construction and Analysis

We will use the multi-level canonical sets data structure described in Section 2.2 to perform parts of this query. Each of the three steps given in Section 4.1.2 will correspond to one nested level of the final data structure. It is easiest to describe the structure inside-out, so we begin with the innermost structure.

Length Restriction

The innermost structure answers the length restriction step of the overall query. This is easily answered using a 1-dimensional range tree, as described in Section 2.1, keyed on the segment lengths. The following lemma summarizes the time and space requirements of this structure.

Lemma 4.1. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments with length at most $\frac{w}{\rho}$ in $O(\log n)$ time, using a data structure of size $O(n)$, which can be built in $O(n \log n)$ preprocessing time.*

Partial Enclosure Property

To identify segments satisfying the partial enclosure property, we use the half-plane expressions we developed in Section 4.1.2. There are two expressions we need to

test, depending on whether the left endpoint of a segment is left or right of L_1. By Theorem 2.2, we can answer this type of half-plane query directly using a canonical subsets data structure, yielding the following lemma applicable to both cases.

Lemma 4.2. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which are not "too much" to one side of a query line in $O(\sqrt{n}\log n)$ time, using a data structure of size $O(n\log n)$, which can be built in $O(n\log n)$ preprocessing time.*

With each subset created for this structure, we will associate the structure required for Lemma 4.1. Applying Corollary 2.4 gives us the following result.

Lemma 4.3. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which are not "too much" to one side of a query line* and *which do not exceed a maximum length in $O\big(\sqrt{n}\log^2 n\big)$ time, using a data structure of size $O(n\log n)$, which can be built in $O\big(n\log^2 n\big)$ preprocessing time.*

Left-endpoint classification

We need to classify the left-endpoints of the segments as left or right of L_1. By Theorem 2.3, each of these queries can be answered by a canonical subsets data structure. With each subset created by this structure, we will associate the structure from Lemma 4.3. Applying Corollary 2.4 gives us the following result.

Lemma 4.4. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments whose left endpoints are to one side of a line, which are not "too much" to one side of a query line,* and *which do not exceed a maximum length in $O\big(\sqrt{n}\log^3 n\big)$ time, using a data structure of size $O\big(n\log^2 n\big)$, which can be built in $O\big(n\log^3 n\big)$ preprocessing time.*

Combining the Steps

To fully answer any query Q, we need to preprocess two such data structures, choosing our expressions appropriately for the left and right cases. During query time, we will query both structures and combine their results. The following theorem summarizes the overall solution.

Theorem 4.5. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which satisfy the partial enclosure property for an arbitrarily-oriented query slab in $O\big(\sqrt{n}\log^3 n\big)$ time, using a data structure of size $O\big(n\log^2 n\big)$, requiring $O\big(n\log^3 n\big)$ preprocessing time.*

4.2 Querying with Two Slabs

In this section, we consider a variation of the slab query. Our query is input as two slabs, and we are interested in those segments which satisfy the partial enclosure property with respect to their intersection. When the two slabs are orthogonal to each other, this method will solve the problem of finding segments sufficiently enclosed by an arbitrarily-oriented rectangle. The overall approach is very similar to Section 4.1, and the data structure we will build to answer these queries is again based on the canonical sets structure outlined in Section 2.2.

4.2.1 Problem Definition

The formal statement of this problem is as follows.

Problem 4.2. Given a set of n horizontal line segments in the plane, and a fixed

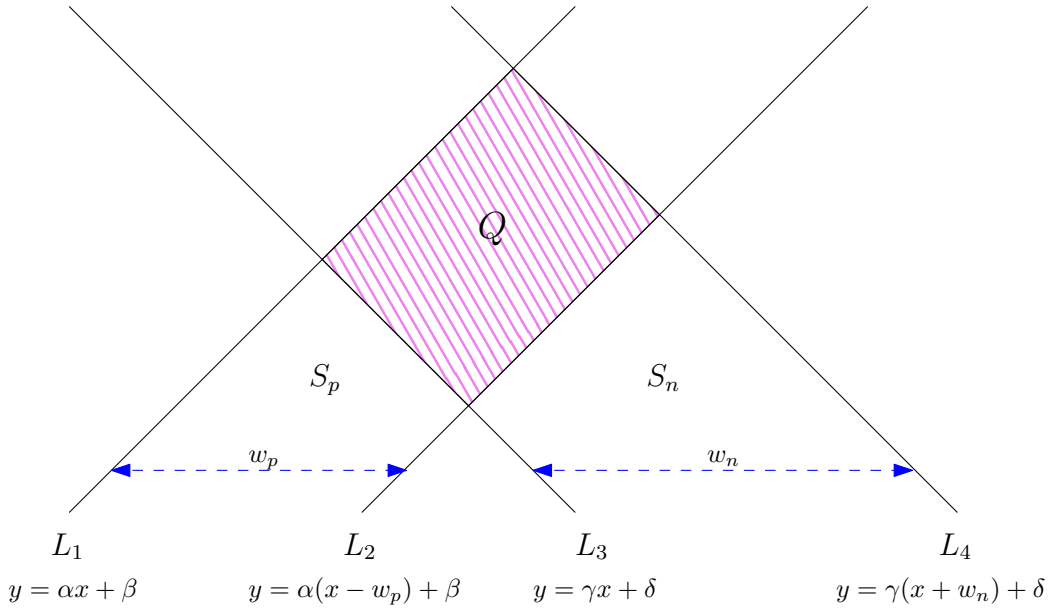

Figure 4.3: A query parallelogram Q formed by the inputs α, β, w_p, γ, δ, and w_n.

parameter ρ such that $0 < \rho \leq 1$, we want to identify those segments which are sufficiently enclosed by an arbitrarily-oriented parallelogram Q to satisfy the partial enclosure property. A segment s satisfies this property if and only if $|s \cap Q| \geq \rho \cdot |s|$.

Throughout this section, we use the following definitions. Let S be a set of n horizontal line segments. Each segment $s_i \in S, 1 \leq i \leq n$ is defined by three values a_i, b_i, l_i which in turn define the endpoints of the segment as $p_i = (a_i, b_i)$ and $q_i = (a_i + l_i, b_i)$.

A query region is an arbitrarily-oriented parallelogram defined by the intersection of two slabs, S_p and S_n, which have positive and negative slopes, respectively. See Figure 4.3.

Specifically, a query is given as six inputs $\alpha, \beta, w_p, \gamma, \delta, w_n$, where $\alpha > 0$, $w_p > 0$, $\gamma < 0$, and $w_n > 0$. With these inputs, we define:

- The line $L_1 : y = \alpha x + \beta$, the left edge of S_p,
- The line $L_2 : y = \alpha(x - w_p) + \beta$, the right edge of S_p,
- The line $L_3 : y = \gamma x + \delta$, the left edge of S_n,
- The line $L_4 : y = \gamma(x + w_n) + \delta$, the right edge of S_n,
- The slab $S_p = \{u \in \mathbb{R}^2 | (u \text{ is on or right of } L_1) \wedge (u \text{ is on or left of } L_2)\}$, a slab with positive slope,
- The slab $S_n = \{u \in \mathbb{R}^2 | (u \text{ is on or right of } L_3) \wedge (u \text{ is on or left of } L_4)\}$, a slab with negative slope,
- The parallelogram $Q = S_p \cap S_n$.

We say that $s_i \in_\rho Q$ if and only if s_i satisfies the partial enclosure property with respect to Q, otherwise $s_i \notin_\rho Q$.

4.2.2 Identifying the Segments

Just as in the single slab problem, identifying segments satisfying the partial enclosure property is accomplished by classifying the endpoints of our segments by how they interact with Q and then testing an appropriate partial enclosure expression. We first describe the different steps of the query in more detail, and then describe the construction and analysis of an appropriate data structure.

Decomposition

The query Q is decomposed into three regions, Q_a, Q_b, and Q_c, by extending horizontal lines through each vertex of Q. Q_a and Q_c are triangular regions, while

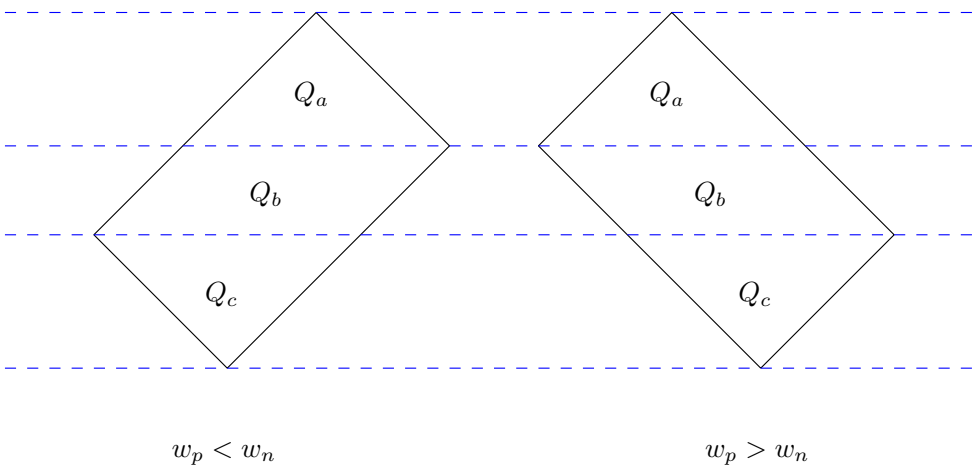

Figure 4.4: Decomposition of the query region Q. The orientation of Q depends on the relative widths of the slabs which define it.

Q_b is a parallelogram. Q_a is always defined by L_1 on the left and L_4 on the right. Likewise, Q_c, is always defined by L_3 on the left and L_2 on the right. The definition of Q_b depends on the overall orientation of Q, which depends on the relationship between w_p and w_n. Specifically, Q_b is defined by L_1 and L_2 when $w_p < w_n$ and defined by L_3 and L_4 when $w_p > w_n$. When $w_p = w_n$, Q_b disappears, and we don't need to continue further query steps on it.

Endpoint Classification

Once we know the definitions of each region, we can proceed with endpoint classification. The goal of this step is to identify which partial enclosure expression is appropriate to test with each line segment by considering which borders of Q it interacts with. Each region has a left, center, and right classification zone, Z_L, Z_C, and Z_R, where the center zone is the query region itself. Figure 4.5 gives an illustration; Z_L, Z_C, and Z_R are coloured blue, green, and red, respectively.

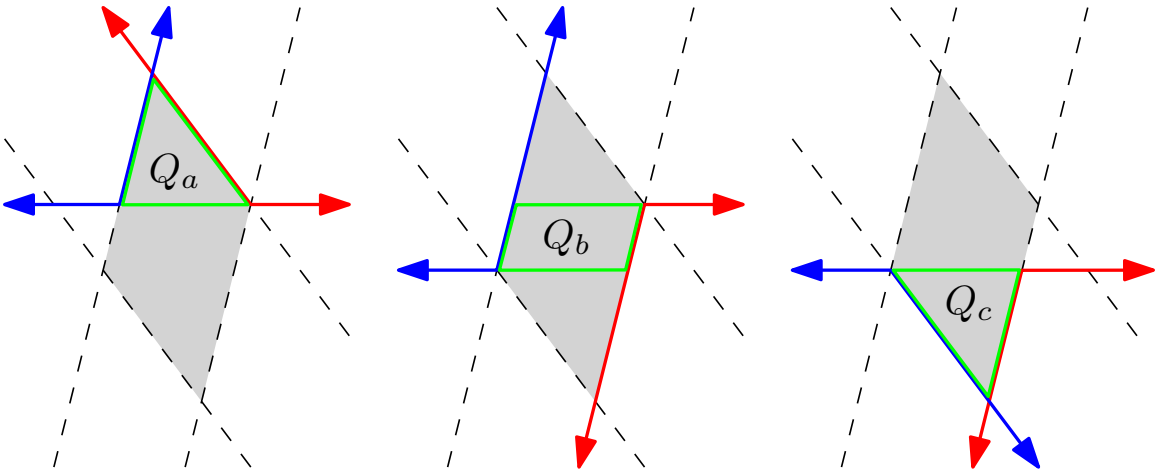

Figure 4.5: Classification zones for each region of Q. Each region has a left, center, and right zone, coloured blue, green, and red, respectively, where we may find segments which need further testing.

For Q_a, Z_L is aligned with the base of Q_a, and extends up L_1. Z_R is also aligned with the base of Q_a, and extends up L_4. Both of these zones are open away from Q. The zones for Q_c are symmetric, being aligned with the top of the region, and with Z_L and Z_R extending down L_3 and L_2, respectively.

For Q_b, the classification zones we use depend on the orientation of the region. When $w_p < w_n$, Z_L is aligned with the base of Q_b and extends up L_1, while Z_R is aligned with the top of Q_b and extends down L_2. Conversely, when $w_p > w_n$, Z_L is aligned with the top of Q_b and extends down L_3, while Z_R is aligned with the bottom of Q_b and extends up L_4. The former case is what is illustrated in Figure 4.5. Z_C is Q_b itself, which we decompose into two triangular subzones so that all of our zones are triangular in nature. During the query, any time that we consider Z_C for Q_b, we will query both subzones and consider their union.

Any segment interacting with a particular query region must have its endpoints

in one of 6 combinations of classification zones: (Z_L, Z_L), (Z_L, Z_C), (Z_L, Z_R), (Z_C, Z_C), (Z_C, Z_R), or (Z_R, Z_R). For a segment $s \in S$ with endpoints p and q:

- If $p, q \in (Z_C, Z_C)$, then s is entirely inside Q and $s \in_\rho Q$.
- If $p, q \in (Z_L, Z_L)$, or $p, q \in (Z_R, Z_R)$ then s is entirely outside Q and $s \notin_\rho Q$.
- Otherwise, s crosses one or both boundaries of the query region and we need to test the appropriate partial enclosure expression.

Partial Enclosure Property

We begin by examining Q_a in detail. Let $s \in S$ be a horizontal line segment and let $\overline{s}$ be the line through s, defined by the equation $y = b$. Let $(a', b) = \overline{s} \cap L_1$ and $(a'', b) = \overline{s} \cap L_4$, be the intersection points of $\overline{s}$ with L_1 and L_4, respectively, then $a' = \frac{b-\beta}{\alpha}$ and $a'' = \frac{b-\delta}{\gamma} - w_n$. We have three combinations of classification zones that need further testing.

For the (Z_L, Z_C) case, we know that s only crosses L_1. We check that not too much of s is outside of Q_a, giving the following partial enclosure expression.

$$\begin{aligned} a' - a &< (1-\rho)l \\ \frac{b-\beta}{\alpha} - a &< (1-\rho)l \\ b\frac{1}{\alpha} - \frac{\beta}{\alpha} &< a + (1-\rho)l \end{aligned}$$

We can query for segments matching this expression by performing a half-plane query on an appropriate dual-space, just as we did in Section 4.1.2. For this expression in particular, we map each segment s to a dual-point with coordinates $(b, a + (1-\rho)l)$, and query with the half-plane $y > \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

For the (Z_C, Z_R) case, we know that s only crosses L_4. We check that enough of s is inside Q_a, which gives the following partial enclosure expression.

$$\begin{aligned} a'' - a &\geq \rho l \\ \frac{b-\delta}{\gamma} - w_n - a &\geq \rho l \\ b \cdot \frac{1}{\gamma} - \frac{\delta}{\gamma} - w_n &\geq a + \rho l \end{aligned}$$

We can test this expression by mapping each segment s to a dual-point with coordinates $(b, a + \rho l)$ and then querying with the half-plane $y \leq \frac{1}{\gamma}x - \frac{\delta}{\gamma} - w_n$.

Finally, for the (Z_L, Z_R) case, we know that both endpoints of s are outside of Q_a, so we only need to measure the width of $s \cap Q_a$. Specifically, we require that:

$$\begin{aligned} a'' - a' &\geq \rho l \\ \frac{b-\delta}{\gamma} - w_n - \frac{b-\beta}{\alpha} &\geq \rho l \\ \frac{b}{\gamma} - \frac{\delta}{\gamma} - w_n - \frac{b}{\alpha} + \frac{\beta}{\alpha} &\geq \rho l \\ b \cdot \left(\frac{1}{\gamma} - \frac{1}{\alpha}\right) + \left(\frac{\beta}{\alpha} - \frac{\delta}{\gamma} - w_n\right) &\geq \rho l \end{aligned}$$

We can test this expression by mapping each segment s to a dual-point with coordinates $(b, \rho l)$ and then querying with the following half-plane.

$$y \leq \left(\frac{1}{\gamma} - \frac{1}{\alpha}\right) \cdot x + \left(\frac{\beta}{\alpha} - \frac{\delta}{\gamma} - w_n\right)$$

Classification into the left and right zones is somewhat "rough", as the zone continues above Q_a itself. This is not a problem in the (Z_L, Z_C) and (Z_C, Z_R) cases since one endpoint of s is classified directly in the closed zone Z_C and the segments

are horizontal. However, it can happen that a segment classified into (Z_L, Z_R) is entirely above Q_a. In this case, since we are measuring $a'' - a'$, and $a'' < a'$ above the apex of Q_a, the expression will be negative and the segment will be rejected.

The partial enclosure expressions for Q_b and Q_c are developed using exactly the same reasoning as for Q_a, differing only by which of the lines L_1, L_2, L_3, and L_4 we use to define a' and a''.

4.2.3 Construction and Analysis

We will use a multi-level query structure for this problem, just as we did for the single slab query. Each of the steps given in Section 4.2.2 will correspond to one nested level of the data structure. It is easiest to describe the structure inside-out, so we begin with the innermost structure.

Check Partial Enclosure Expression

To identify segments satisfying a partial enclosure expression, we use the half-plane dual-spaces we developed in Section 4.2.2. By Theorem 2.2, we can answer such a half-plane query directly using a canonical subsets data structure, yielding the following lemma.

Lemma 4.6. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which are not "too much" to one side of a query line in $O(\sqrt{n}\log n)$ time, using a data structure of size $O(n \log n)$, which can be built in $O(n \log n)$ time.*

Classify Right Endpoint

We need to be able to classify the right endpoints of our segments into any of the classification zones of each query region. Each of these regions are triangular in nature and can be queried with a canonical subsets data structure. With each subset created by this structure, we will associate the structure from Lemma 4.6. Applying Corollary 2.4 gives us the following result.

Lemma 4.7. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which have their right endpoint in a query triangle, and which are not "too much" to one side of a query line, in $O\big(\sqrt{n}\log^2 n\big)$ time, using a data structure of size $O\big(n\log^2 n\big)$, which can be built in $O\big(n\log^2 n\big)$ time.*

Classify Left Endpoint

We need to be able to classify the left endpoints of our segments into any of the classification zones as well, which will use another canonical subsets data structure. With each subset of this structure, we continue to build up our multi-level structure by associating the structure from Lemma 4.7. By applying Corollary 2.4 again, we have the following.

Lemma 4.8. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which have their left and right endpoints in given query triangles, and which are not "too much" to one side of a query line, in $O\big(\sqrt{n}\log^3 n\big)$ time, using a data structure of size $O\big(n\log^3 n\big)$, which can be built in $O\big(n\log^3 n\big)$ time.*

Combining the Steps

To fully answer a query Q, we only need one instance of the structures for classifying the left and right endpoints, since these structures can support all of the different triangular classification queries we need to perform. We will need to preprocess several versions of the innermost level which determines the partial enclosure expression, however.

During query time, we check (Z_C, Z_C), (Z_L, Z_C), (Z_C, Z_R) and (Z_L, Z_R) for every query region. For any of the latter three zone combinations which are non-empty, we check the innermost structure corresponding to the appropriate partial enclosure expression. The following theorem summarizes this process.

Theorem 4.9. *Given a set of n horizontal line segments, we can identify a set of disjoint subsets containing all segments which satisfy the partial enclosure property with respect to the intersection of two query slabs in $O\big(\sqrt{n}\log^3 n\big)$ time, using a data structure of size $O\big(n\log^3 n\big)$, requiring $O\big(n\log^3 n\big)$ preprocessing time.*

4.3 Remarks on Arbitrarily-Oriented Segments

Our general approach can be extended to identify arbitrarily-oriented line segments which satisfy the partial enclosure property for one or two slabs, however, the resulting structure is very "case heavy" and involves a large number of query variables.

Arbitrarily-oriented segments can cross through the borders of Q in many more ways than horizontal ones. This prevents us from partitioning the environment into a simple set of classification zones, and increases the number of cases to consider.

The partial enclosure expressions for each case will also involve higher degree

polynomials. For example, a segment s which crosses L_1 and L_2 has intersection points $p' = (a', b')$ with L_1 and $q' = (a'', b'')$ with L_2. To consider how much of the segment is inside the slab, we need to calculate the length of the segment from p' to q', as follows.

$$\begin{aligned}\|p'q'\|^2 &= \left(\sqrt{(a''-a')^2+(b''-b')^2}\right)^2 \\ &= (a''-a')^2+(b''-b')^2 \\ &= (a'')^2 - 2a'a'' + (a')^2 + (b'')^2 - 2b'b'' + (b')^2\end{aligned}$$

Each term in the above expression represents several query variables when we consider the actual values for a', b', a'', and b''. Let $\overline{s}$ be the line through s, and assume that it is defined as $y = mx + t$, then

$$a' = \frac{\beta - t}{m - \alpha} \qquad b' = m\frac{\beta - t}{m - \alpha} + t$$
$$a'' = \frac{\beta - t - \alpha w}{m - \alpha} \qquad b'' = m\frac{\beta - t - \alpha w}{m - \alpha} + t$$

Multiplying and squaring these expressions to calculate $\|p'q'\|^2$ results in a high degree polynomial with respect to the number of independent query variable expressions. The situation worsens when we consider intersections between L_1 or L_2 with L_3 or L_4, as we have to consider γ and δ as well. Such an expression *can* be mapped to a high-degree half-space and then be queried with a canonical subset structure, but the space and query times suffer.

4.4 Conclusion

In this chapter, we have developed methods for identifying horizontal segments which satisfy the partial enclosure property with respect to an arbitrary slab or parallelogram. The latter case is a generalization of arbitrarily-oriented rectangles when the two slabs defining the parallelogram are orthogonal to each other.

In future work, we would like to find a more straight-forward method for relaxing the horizontal restriction on the input segments. A less case-heavy method for querying arbitrarily-oriented segments against slabs would automatically improve our results from Section 3.2, as well.

Chapter 5

Partial Enclosure Range Searching on Convex Polygons

In this chapter, we show how we can answer the partial enclosure range searching problem on convex polygons. The main contribution of this section is a method to calculate the area of a convex polygon appearing within a query rectangle. With the enclosed area in hand, deciding on the partial enclosure property is straightforward.

5.1 Problem Definition

Problem 5.1. We are given a convex polygon P consisting of n edges and a fixed parameter ρ such that $0 < \rho \leq 1$. We want to determine whether at least $\rho \cdot \mathrm{area}(P)$ is enclosed by a query rectangle Q.

Throughout this chapter we use the following definitions. The polygon P is

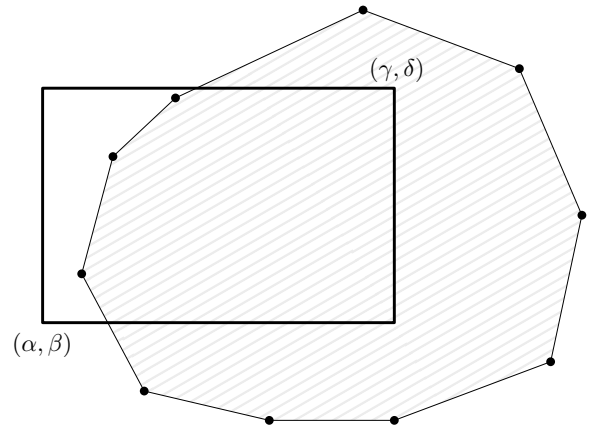

Figure 5.1: The query region Q formed by the inputs α, β, γ, δ, and a convex polygon P.

defined by its edges $E = e_1, e_2, \ldots, e_{n-1}, e_n$, given in clockwise order, where e_n and e_1 share a vertex to close the polygon. A *chord* of P is a straight line segment incident to two edges which partitions P into two regions.

A query Q is given by its lower-left and upper-right corners (α, β) and (γ, δ), respectively. See Figure 5.1 for an example.

We say that P is sufficiently enclosed by Q to satisfy the partial enclosure property if and only if $\text{area}(Q \cap P) \geq \rho \cdot \text{area}(P)$. Throughout this chapter we will assume that Q is an axis-parallel rectangle for ease of discussion. In Section 5.7 we will show how to use our method even when relaxing some restrictions on Q.

5.2 Overview of the Algorithm

To achieve a fast query time, our algorithm requires some preprocessing on P to speed up certain area calculations. Before describing the preprocessing, we outline the query algorithm here.

1. Identify which edges of P are intersected by Q.

2. Enumerate the intersecting edges in clockwise order to determine which areas of Q are inside of P, and which are outside. There are at most 4 pairs of intersecting edges where P crosses into Q, and subsequently back out.

3. For each entrance/exit pair:

   (a) Consider the point s on the entrance edge and the point t on the exit edge where P intersects the boundary of Q. The segment st through the interior of Q forms a *chord* of P.

   (b) On one side of st, we have the subpolygon P_{st} defined by a subchain of P (including some partial component of the entrance and exit edges) and the segment st itself. $P_{st} \subseteq Q$, so we must calculate its area as part of our overall query.

4. Assuming that P and Q *do* intersect, the assembled collection of all such entrance and exit points from the previous step, as well as any corners of Q which are inside P define P_I. We can also think of this as the space on the "insides" of all the chords. P_I is a polygon of not more than 8 edges. $P_I \subseteq Q$, so we must calculate its area as part of our overall query.

5.3 Preprocessing

In order to help with calculating the area of the subpolygons identified by step 3(b), above, we will preprocess the area of several chords of P, storing the results in a binary search tree T_P.

We will look at chords between pairs of edges. A chord from edge e_i to e_j, $i < j$, is defined as the segment through the interior of P from the vertex of e_i which is not shared with e_{i+1} to the vertex of e_j which is not shared with e_{j-1}. For certain values of i and j (detailed below), we will calculate the area of the subpolygon defined by the chord segment itself and the edges $e_i, e_{i+1}, \ldots, e_j$.

During our query, it may be helpful to query chords between two edges e and e' such that the sequence of edges includes e_n. That is, the edge sequence is of the form $e, \ldots, e_{n-1}, e_n, e_1, e_2, \ldots e'$. To perform these queries easily, we will double the edge list for the sake of preprocessing to $e_1, e_2, \ldots, e_{n-1}, e_n, e'_1, e'_2, \ldots, e'_{n-1}, e'_n$ where $e_i = e'_i$ for $1 \leq i \leq n$. Any chord query between two edges e_i and e_j where $j < i$ is instead queried as e_i and e'_j.

Algorithm 5.1 gives the details of the construction of T_P, while Figure 5.2 illustrates what it is calculating.

Essentially, T_P is defined recursively by splitting E into halves. The resulting tree has a depth of $O(\log n)$, and requires only $O(1)$ processing and storage per node for a total preprocessing time of $O(n)$. Each node t of T_P records the labels of the left and right edges of P that it spans, and the total area of the subpolygon between those edges. Given two edge labels i and j, we can query T_P and identify $O(\log n)$ subtrees between those labels which correspond to precalculated subpolygons. This will not give us the total area of the subpolygon between any arbitrary e_i and e_j, but it does reduce the problem significantly, as the remaining area to be calculated is bounded by a path of only $O(\log n)$ chords rather than $O(n)$ edges of P.

Algorithm 5.1: BuildChordTree
Input: List of edges $E = e_1, e_2, \ldots, e_n$

if $|E| = 1$ **then**
  $T \leftarrow$ create new leaf node
  $l(T) \leftarrow$ label of the single edge in E
  $r(T) \leftarrow$ label of the single edge in E
  $a(T) \leftarrow 0$
  return T
end
$m \leftarrow \left\lfloor \frac{|E|}{2} \right\rfloor$
$E_l = \{e_1, e_2, \ldots, e_m\}$
$E_r = \{e_{m+1}, e_{m+2}, \ldots, e_n\}$
$T_l \leftarrow$ `BuildChordTree(`E_l`)`
$T_r \leftarrow$ `BuildChordTree(`E_r`)`
$T \leftarrow$ Create new node with left child T_l and right child T_r
$l(T) \leftarrow l(T_l)$
$r(T) \leftarrow r(T_r)$
$v_m \leftarrow$ vertex between e_m and e_{m+1}
$v_l \leftarrow$ vertex of e_1 not shared with e_2
$v_r \leftarrow$ vertex of e_n not shared with e_{n-1}
$a \leftarrow$ area of triangle $\Delta v_l v_m v_r$
$a(T) \leftarrow a + a(T_l) + a(T_r)$
return T

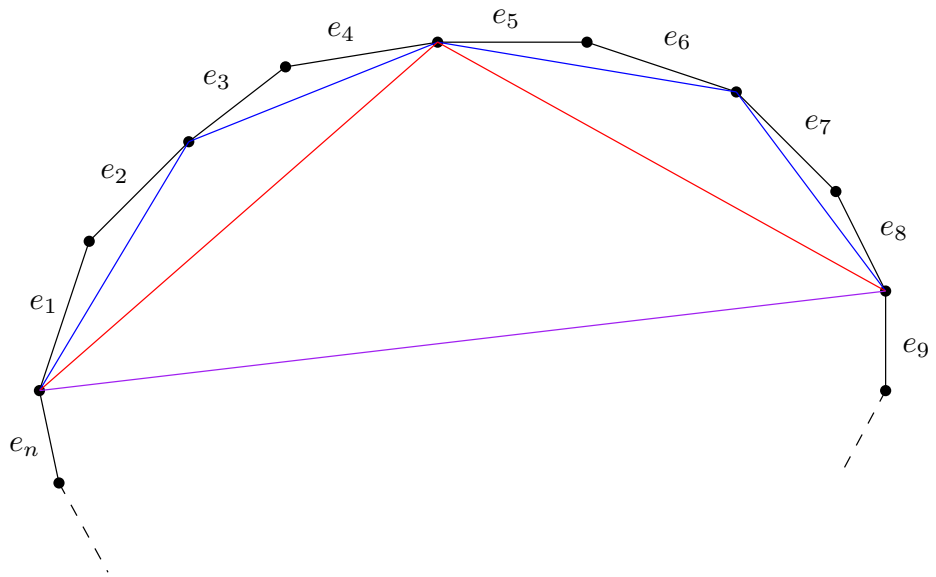

Figure 5.2: Preprocessing the area of chords of P. Each level forms a triangle with the chords of the previous level.

5.4 Locating Intersections

Following from the convexity of P, there can be at most 8 intersections between P and Q. Any straight-line through P will intersect at most twice, so extending each of the 4 boundaries of Q into lines yields at most 8 intersections.

Each intersection can be found in the following way. Let b_α be the vertical boundary of Q with x-coordinate at α. Let $\overline{\alpha}$ be the vertical line through b_α. Using a binary search on the edges of P, we can identify the edges e_α and e'_α which intersect $\overline{\alpha}$, if they exist, such that e_α is clockwise from e'_α along P. Specifically, we test only the top chain of P to find e_α and only the bottom chain of P to find e'_α. We then determine if e_α and e'_α are intersected by the segment b_α itself by checking the y-coordinate of the intersection point. We will store a 'nil' value with e_α and/or e'_α if the intersections do not exist or do not intersect b_α.

Similarly, let b_γ be the vertical boundary of Q with x-coordinate at γ, and let b_β and b_δ be the horizontal boundaries of Q with y-coordinates at β and δ, respectively. Using the same binary search technique on appropriate half-chains of P, we can collect e_β, e'_β, e_γ, e'_γ, e_δ, and e'_δ.

If all of these values are nil, then P does not intersect Q at all, and one of the following is true:

- $P \subset Q$, determined by selecting any edge of P and checking whether it is contained in Q. This check takes $O(1)$ time.
- $Q \subset P$, determined by selecting any point $q \in Q$, projecting horizontal and vertical lines through q, and checking that the intersections of these lines with P occur on opposite sides of q. This check takes $O(\log n)$ time.
- $P \cap Q = \emptyset$, determined to be true if the previous two tests are false.

5.5 Chain Decomposition

For the remainder of the query, we assume that P and Q *do* intersect. Any intersections between them must come in pairs; i.e., if P enters Q, it must leave elsewhere. Our goal is to determine how P and Q interact, and then calculate the area of $P \cap Q$. Figure 5.3 shows some examples of how the two may interact.

Continuing to examine the problem in a clockwise direction, P can enter Q at e_α, e_β, e_γ, and/or e_δ. Once P enters Q, it must exit at one of e'_α, e'_β, e'_γ, or e'_δ before any other entrance can occur.

We can find all entrance/exit pairs with the following steps.

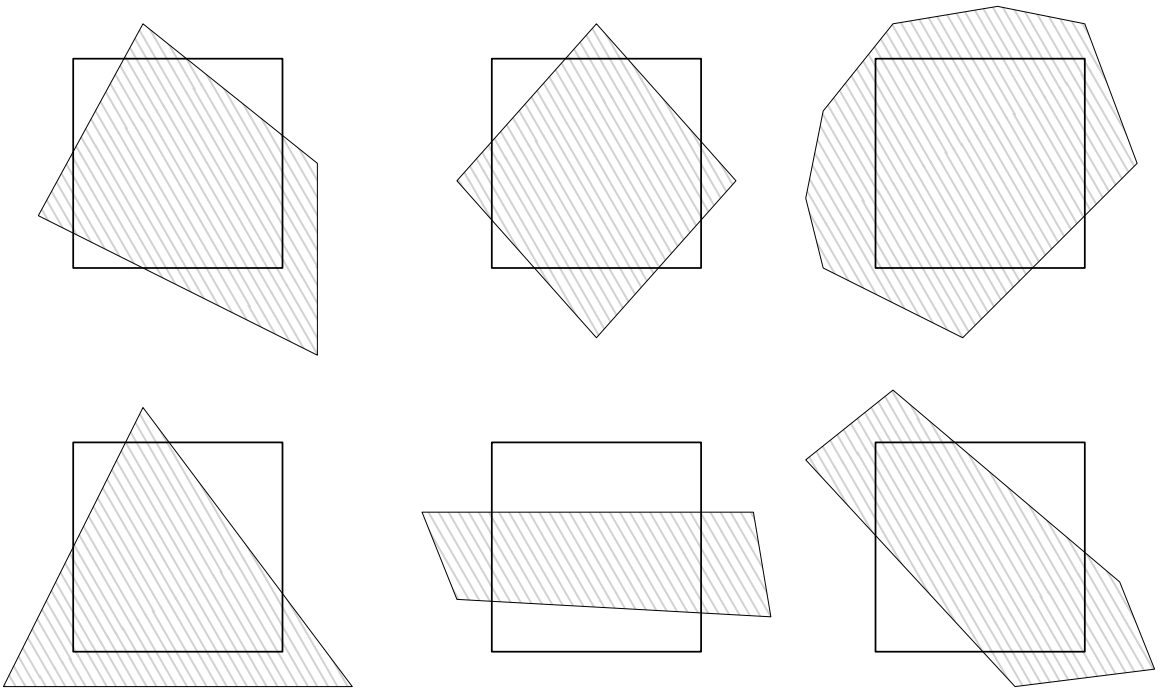

Figure 5.3: Examples of how P and Q may interact.

1. Consider the following clockwise ordering of the entrance and exit labels as a circular list: e_α, e'_δ, e_δ, e'_γ, e_γ, e'_β, e_β, e'_α.

2. Starting anywhere in the list, scan for a non-nil entrance label (a non-prime edge).

3. Find the subsequent non-nil exit label, by continuing to walk the list in circular order.

4. Mark any corners of Q which are bypassed by step 2 as outside of P. The corners which remain inside will be important later.

5. We are finished when we visit a non-nil entry label for the second time.

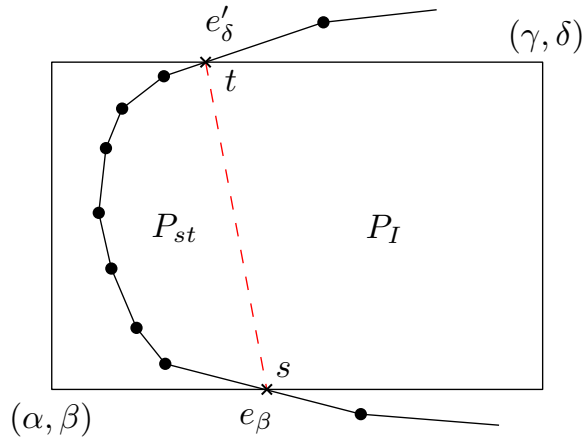

Figure 5.4: A query Q intersecting P at points s and t. The segment st is a chord of P through the interior of Q.

Figure 5.4 shows an example. In the figure, e_β is an edge that enters into Q, while e'_δ is the corresponding exit edge. Both e'_α and e_α are nil. The corners of Q at (α, β) and (α, δ) are outside of P since we know P entered Q before (α, β), and therefore all corners of Q must be outside until the next non-nil entrance label.

For an entrance edge e_i and exit edge e_j, let s and t be the intersection points of e_i and e_j with the boundary of Q, respectively. The segment st partitions Q into two parts and forms a chord of P.

Let P_{st} be the subpolygon "cap" of P defined by the segment st, the portion of e_i from s towards e_{i+1}, the edges $e_{i+1}, e_{i+2}, ..., e_{j-1}$, and the portion of e_j from e_{j-1} to t. Since $P_{st} \subset Q$, we need to calculate its area.

We first check if $e_i = e_j$. In that case, a single edge of P has formed both the entrance and exit of Q, the chord st is a subsegment of that edge, and the area of P_{st} is 0. Otherwise, P_{st} is a convex polygon. Using the preprocessed subpolygon areas stored in T_P, the chain of edges e_{i+1} to e_{j-1} can be decomposed into $O(\log n)$

preprocessed subpolygons whose total area is found in the tree. The remaining region of P_{st} for which we do not yet know the area is comprised of a path of $O(\log n)$ chords from T_P, the partial edges of P_{st} up to s and t, and the segment st. This region is a convex polygon with $O(\log n)$ vertices and its area can be calculated in $O(\log n)$ time.

The above process is repeated for each entry and exit pair of intersections, giving at most four caps of P. We still need to calculate the area to the inside of each chord (the side away from the chord's cap). This interior area may also be defined in part by the edges and corners of Q.

More precisely, let P_I be the polygon defined by all of the intersection points of P with the boundary of Q (e.g., the points s and t for each cap), as well as any corners of Q which are inside P. $P_I \subset Q$ and is comprised of $O(1)$ vertices, all of which have been previously identified in earlier steps of the query. Its area can be calculated in $O(1)$ additional time.

5.6 Analysis

Construction of T_P, detailed in Section 5.3 is done using a recursive approach in only linear time as only $O(1)$ time is needed to merge the results of the recursion steps and calculate the area of one new triangle. Thus, total preprocessing time and space is only $O(n)$. Querying involves the following steps, each requiring only $O(\log n)$ time.

1. Binary search on P to locate intersections with lines through the boundaries of Q in $O(\log n)$ time.

2. Identifying the caps of P inside Q; $O(1)$ time.

3. Finding the precalculated area within each cap in T_P; $O(\log n)$ time.

4. Calculating the rest of the area inside each cap; $O(\log n)$ time.

5. Testing the corners of Q for inclusion in P; $O(1)$ if found implicitly while scanning the circular list of entrance and exit points, or $O(\log n)$ time if done explicitly by binary search on the edges of P.

6. Sorting the intersection points and corners together to form P_I, and then calculating its area; $O(1)$ time.

7. Combining all results together and making a determination about Q enclosing a sufficient portion of P; $O(1)$ time.

The following theorem and corollary summarize our overall approach.

Theorem 5.1. *Let P be a convex polygon consisting of n edges. In $O(n)$ time and space, we can create a data structure which allows us to determine $\mathrm{area}(Q \cap P)$ in $O(\log n)$ time, for any axis-parallel rectangular query region Q.*

Corollary 5.2. *Let P be a convex polygon consisting of n edges, and let $0 < \rho \leq 1$ be a fixed parameter. With $O(n)$ time and space, we can determine if $\mathrm{area}(Q \cap P) \geq \rho \cdot \mathrm{area}(P)$ in $O(\log n)$ time for any axis-parallel rectangular query Q.*

5.7 Remarks

Our method works for a query rectangle which has any orientation; that is, Q need not be axis-parallel. Nearly all of the steps involved in calculating the area are only

concerned with edges of P and intersection points of lines through the boundaries of Q. The only place we use the axis-parallel property of Q is in Section 5.4 where we calculate those intersection points.

We can allow for Q to be arbitrarily-oriented by modifying our intersection testing in the following way. During preprocessing, we calculate c, the centrepoint of P, which can be done in $O(n)$ time. Let $\bar{l}$ be a line through any boundary of Q and let $\bar{c}$ be the line perpendicular to $\bar{l}$ through the centrepoint of P. The line $\bar{c}$ crosses the boundary of P in two places, and, using a binary search on the edges of P, we can identify the edges e_a and e_b that $\bar{c}$ crosses. These two edges divide P into two chains, E_{ab} and E_{ba}. We can now perform a binary search on each of these chains looking for intersections with $\bar{l}$ (which may or may not cross P at all). Figure 5.5 illustrates this process.

Our method can also be extended to allow for querying by an arbitrary convex k-gon. The number of intersections we have to locate increases to $O(k)$, as does the number of chords across P. Other than that, we use exactly the same preprocessing and general query method as for rectangles. We require $O(k \log n)$ time to find all intersection points between P and Q and to calculate the areas of each cap of P. A further $O(k)$ time is needed to calculate the area of P_I. The total query time for a convex k-gon is thus $O(k \log n)$. The overall approach is summarized by the following corollary.

Corollary 5.3. *Let P be a convex polygon consisting of n edges. In $O(n)$ time and space, we can create a data structure which allows us to determine $\mathrm{area}(Q \cap P)$ in $O(k \log n)$ time, for any convex k-gon query region Q.*

Work by Czyzowicz *et al.*[9] and then Iacono and Langerman[15] gives an al-

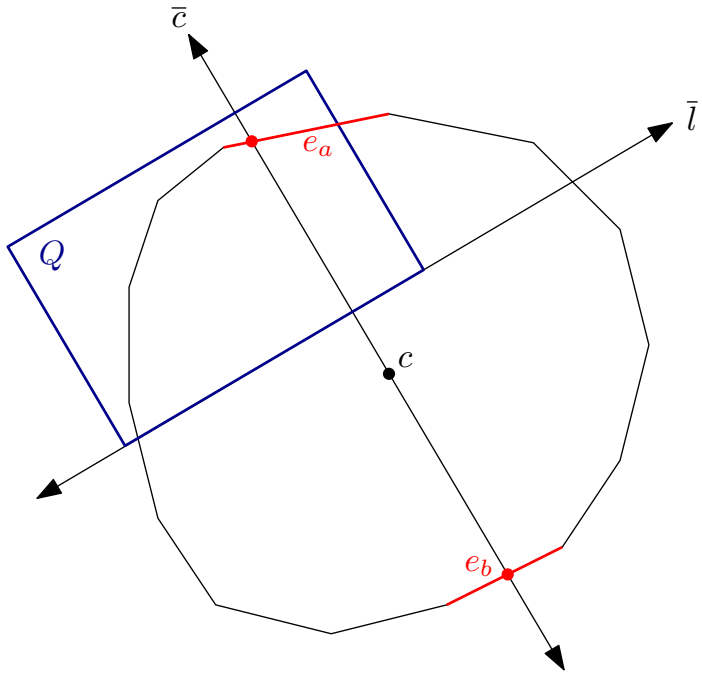

Figure 5.5: Locating the intersections of P with an arbitrarily-oriented Q.

ternate method for calculating the area of a convex polygon cut by a chord. This method requires only $O(n)$ preprocessing time and space, and can answer area queries in $O(1)$ time. We could use this method to reduce the time needed to find the area within each cap of P to $O(1)$ with no increase in the cost of our preprocessing. However, calculating the area of a rectangle still requires us to expend $O(\log n)$ time locating the points of intersection between P and Q, so our overall query time remains unchanged.

5.8 Conclusion

In this chapter we have developed a method for calculating the area of a convex polygon enclosed by a query rectangle or convex k-gon. Our solution does not

require complex data structures, and uses only a binary tree and some line intersection tests. Specifically, we have described:

1. A method for calculating the area of a convex polygon enclosed by a rectangle, Theorem 5.1.

2. A method for calculating the area of a convex polygon enclosed by a convex k-gon, Corollary 5.3

In both cases, once the area is known, calculating the partial enclosure property is straight-forward.

Chapter 6

Partial Enclosure Range Searching on Monotone Polygons

In this chapter, we show how we can answer the partial enclosure range searching problem on monotone polygons. The main contribution of this chapter is a method to calculate the area of a monotone polygon appearing within a query rectangle. Once the enclosed area is calculated, determining if the partial enclosure property is satisfied is straight-forward.

6.1 Problem Definition

The following problem statement describes our goal for this chapter.

Problem 6.1. We are given a monotone polygon P consisting of n vertices and a fixed parameter ρ such that $0 < \rho \leq 1$. We want to determine whether at least $\rho \cdot \text{area}(P)$ is enclosed by an axis-parallel query rectangle Q.

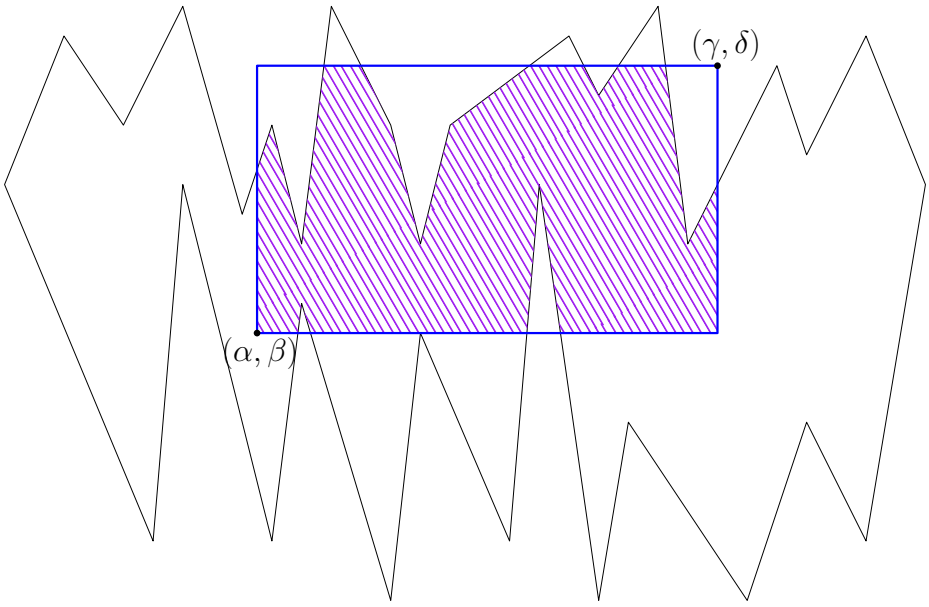

Figure 6.1: A monotone polygon P with query Q. The area of P enclosed by Q, $Q \cap P$, is highlighted.

Throughout this chapter, we will use the following definitions. The input is a polygon P embedded in the plane which, without loss of generalization, is monotone with respect to the x-axis (i.e., from left to right). P is defined by its vertices $v_1, v_2, \ldots, v_n$, where v_n and v_1 share an edge to close the polygon.

A query Q is given by its lower-left and upper-right corners (α, β) and (γ, δ), respectively. See Figure 6.1 for an example.

We say that P is sufficiently enclosed by Q to satisfy the partial enclosure property if and only if $\text{area}(Q \cap P) \geq \rho \cdot \text{area}(P)$.

6.2 A First Problem

We will first describe an algorithm which solves a simpler problem, and then extend that solution into one for our main problem.

Problem 6.2. Given a monotone polygon P, and a query in the form of a horizontal slab S, what is the total area of P enclosed by S? i.e., what is $\mathrm{area}(S \cap P)$?

Our overall approach for this problem is to define a function (actually, a collection of functions) which returns the area below a given horizontal query line.

6.2.1 Decomposing P

In order to create an area formula for the entire polygon, we decompose P into a linear number of regions and calculate area formulas for each. These regional formulas are then combined into an overarching *multi-region formula*.

We begin by sorting all of the vertices by their x-coordinates. If two vertices share the same x-coordinate (no more than two can do so, since P is monotone), we can skip the duplicate without any other change to the algorithm. However, for ease of discussion, we will assume that every vertex has a distinct x-coordinate. For the remainder of this chapter, we relabel the x-coordinates by their sorted order so that x_1 is the x-coordinate of the leftmost vertex of P, x_2 the next leftmost, etc.

Let $\overline{x}$ be the vertical line through any x-coordinate, and let P_U and P_L be the upper and lower chains of P, respectively. Consider the sequence of lines $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$ through the vertices of P. For every line $\overline{x_i}$, let $a_i = \overline{x_i} \cap P_U$ and $a'_i = \overline{x_i} \cap P_L$ be the intersections of $\overline{x_i}$ with the upper and lower chains of P, respectively.

We will calculate all of these intersections as we walk from left to right over both P_U and P_L simultaneously. We also keep track of e_i and e'_i, the edges of P_U and P_L, respectively, which are intersected by $\overline{x_i}$, and upon which reside the points a_i and a'_i. Where $\overline{x_i}$ intersects a vertex of P_U or P_L, we store the edge to the right of that vertex as the corresponding value of e_i or e'_i. This walk allows us to generate a sequence of regions $R_1, R_2, \ldots, R_{n-1}$ which have the following, equivalent definitions.

- For each $1 \leq i \leq n-1$, let X_i be the vertical slab between $\overline{x_i}$ and $\overline{x_{i+1}}$, then $R_i = X_i \cap P$.
- R_i is the area of P bounded by the vertical segments $\overline{x_i} \cap P$ and $\overline{x_{i+1}} \cap P$ to the left and right, and some portion of the edges e_i and e'_i to the top and bottom (while, in general, e_i and e'_i extend beyond $\overline{x_i}$ and $\overline{x_{i+1}}$, this excess is irrelevant to the definition of R_i).
- R_i is the polygon formed by the cycle on $a_i, a_{i+1}, a'_{i+1}, a'_i$.

Thus, each region is a (possibly degenerate) trapezoid.

6.2.2 Area of a Region

For each region $R_i \in \{R_1, R_2, \ldots R_{n-1}\}$, we create a function $F(R_i, h)$ which will return the area of R_i below a horizontal line query line having a y-component of h. Figure 6.2 shows an example region. From bottom to top, we can identify at most 4 "critical heights", where the nature of how the area of a region grows with respect to h changes. For a general region R, these 4 critical heights are:

- h_0, where R begins.

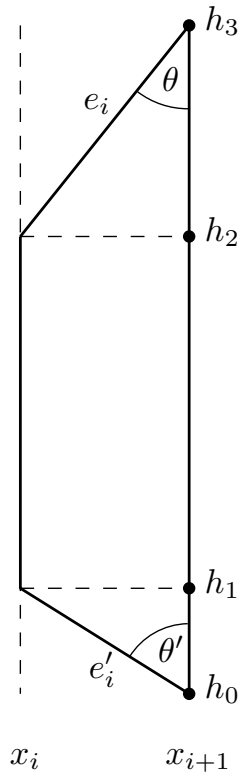

Figure 6.2: A trapezoidal region of P for which we can derive an area function dependant on h.

- h_1, where R stops growing as a triangle and begins growing as a rectangle.
- h_2, where R stops growing as a rectangle and begins growing as the base trapezoid of a triangle.
- h_3, where R ends.

These 4 critical heights give rise to a piecewise function representing the area, defined as follows.

$$F(R,h) = \begin{cases} 0 & \text{if } h < h_0 \\ A(h')^2 & \text{if } h_0 \leq h < h_1 \text{ for } h' = h - h_0 \\ Bh' + C & \text{if } h_1 \leq h < h_2 \text{ for } h' = h - h_1 \\ A'(h')^2 + C' & \text{if } h_2 \leq h < h_3 \text{ for } h' = h_3 - h \\ C'' & \text{if } h_3 \leq h \end{cases}$$

In essence, the area function is a quadratic one, although, for different values of h, some or all of the coefficients are 0. The details of each part of this function, and the definitions of the constants A, A', B, C, C' and C'' are as follows.

1. When $h < h_0$, the area of R below h is 0, since no part of R exists below that measure.

2. When $h_0 \leq h < h_1$, the area of R below h grows as a triangle. Let $h' = h - h_0$. The angle θ' between e_i' and the vertical can be calculated as

$$\tan(\theta') = \frac{x_{i+1} - x_i}{h_1 - h_0}$$

   Let t' be this ratio, then at h', the base of the triangle has width $h' \cdot t'$. The area of the triangle is therefore $\frac{1}{2}(h')^2 t'$ and we set the constant $A = \frac{1}{2}t'$.

3. When $h_1 \leq h < h_2$, the area of R includes all of the area between h_0 and h_1, which we store as the constant C. The remaining area of R up to h grows as

a rectangle. Let $h' = h - h_1$, then the rectangular area is $(x_{i+1} - x_i) \cdot h'$, and we define the constant $B = (x_{i+1} - x_i)$. Thus, the total area of R_i up to h' is $Bh' + C$.

4. When $h_2 \leq h < h_3$, the area of R includes all of the area between h_0 and h_2, which we store as the constant C'_a. The remaining area of R up to h grows as the base trapezoid of a triangle. Let $h' = h_3 - h$ (note that this definition is somewhat reversed from the other cases). We can calculate this area by taking the total area of the triangle, and subtracting the area of the triangle from h_3 down to h.

   The overall triangle between h_2 and h_3 has area

   $$a = \frac{1}{2} \cdot (h_3 - h_2) \cdot (x_{i+1} - x_i)$$

   The angle θ between e_i and the vertical can be calculated as

   $$\tan(\theta) = \frac{x_{i+1} - x_i}{h_3 - h_2}$$

   Let t be this ratio, then the smaller triangle from h_3 down to h can be calculated as in case 2, giving $a' = \frac{1}{2}(h')^2 t$. The overall area from h_2 up to h is then

   $$a - a' = \frac{1}{2} \cdot (h_2 - h_3) \cdot (x_{i+1} - x_i) - \frac{1}{2}(h')^2 t$$

   This expression is of the form $C'_b + A'(h')^2$. Thus we set the constants $A' = -\frac{1}{2}t$,

$C'_b = \frac{1}{2} \cdot (h_2 - h_3) \cdot (x_{i+1} - x_i)$, and $C' = C'_a + C'_b$ to complete the formula for this part of $F(R, h)$.

5. When $h_3 \leq h$, the total area of R should be reported, which we store in C''.

6.2.3 Creating Multi-Region Formulas

After completing the previous step, we have a set of regions, and their area formulas, R_i and $F(R_i, h)$ respectively for $1 \leq i \leq n - 1$.

To complete the preprocessing for our horizontal slab query, we will process our regional area formulas into a list of multi-region formulas which can report the area of the entire polygon under a query line.

We begin by collecting tuples of all of the critical heights from our regional formulas. For every region R_i, we collect (y, R) for $y \in \{h_0, h_1, h_2, h_3\}$ and $R = R_i$. Let $\mathcal{Y}$ be the list of all such tuples across all regions, sorted by y-values from lowest to highest.

If we suppose for a moment that each y-value is distinct, then for each one there is exactly one region R which is transitioning from one phase of growth to another. That is, looking at the piecewise function $F(R, h)$ which determines the area of R below a query line, a new piece of that function is taking over. To create the multi-region formula for a new critical height, we copy the formula for the predecessor height (the lowest y-value starts with a formula set to 0). To update the multi-region formula, we subtract the coefficients of R for its previous phase of growth so that they no longer influence the formula, and then add the coefficients for the new phase.

Since neighbouring regions share vertices, the y-values will not be distinct. In-

stead, several y-values will be collapsed into a single critical height in the list of multi-region formulas, with all appropriate constants from contributing regions added or subtracted as necessary. Algorithm 6.1 gives the details of this process more formally.

Algorithm 6.1: BuildMultiRegionFormula
Input: List of regions $\mathcal{R} = R_1, R_2, \ldots, R_{n-1}$

Initialize a list $\mathcal{Y}$
foreach R *in* $\mathcal{R}$ **do**
  $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{(y, R) \mid y \in \{R.h_0, R.h_1, R.h_2, R.h_3\}\}$
end
`sort(`$\mathcal{Y}$ *on* y`)`
$y' \leftarrow$ minimum y-value in $\mathcal{Y}$
$F(P, y') \leftarrow$ coefficients A, B, C set to 0
foreach (y, R) *in* $\mathcal{Y}$ **do**
  $F(P, y) = F(P, y')$
  $i \leftarrow \{0, 1, 2, 3\}$ such that $y = h_i \in \{R.h_0, R.h_1, R.h_2, R.h_3\}$
  if $i > 0$ **then**
    $F(P, y).A \leftarrow F(P, y).A - R.h_{i-1}.A$
    $F(P, y).B \leftarrow F(P, y).B - R.h_{i-1}.B$
    $F(P, y).C \leftarrow F(P, y).C - R.h_{i-1}.C$
  end
  $F(P, y).A \leftarrow F(P, y).A + R.h_i.A$
  $F(P, y).B \leftarrow F(P, y).B + R.h_i.B$
  $F(P, y).C \leftarrow F(P, y).C + R.h_i.C$
  $y' \leftarrow y$
end
$H \leftarrow$ the list of all $F(P, h)$ created above
return H

The output of this algorithm is the list H of multi-region formulas. Since $\mathcal{Y}$ was sorted by y-value, H will be as well.

6.2.4 Querying Multi-Region Formulas

We are now ready to query horizontal slabs against P. Let $F(P, h)$ be the multi-region area function for P, which is essentially a very big piecewise function given by Algorithm 6.1 in the form of a sorted list H. Given two horizontal lines $\overline{\beta}$ and $\overline{\delta}$ defining our query slab, which have y-components β and δ, respectively with $\beta \leq \delta$, our query requires only a few steps.

1. Our query parameter β will not generally correspond to a critical height in H. Perform a binary search on H to find the formula stored with the *predecessor* of β. Denote this formula as F_β. Evaluating $F_\beta(P, \beta)$ tells us the area of P below β.

2. For δ, we perform another binary search on H to find F_δ such that $F_\delta(P, \delta)$ tells us the area of P below δ.

3. We can now calculate the area inside the query slab as

$$\text{area}(S \cap P) = F_\delta(P, \delta) - F_\beta(P, \beta)$$

We summarise our results with the following theorem and corollary.

Theorem 6.1. *Let P be a monotone polygon consisting of n vertices. In $O(n \log n)$ time and $O(n)$ space, we can create a data structure that allows us to determine $\text{area}(S \cap P)$ in $O(\log n)$ time for any horizontal slab query region S.*

Corollary 6.2. *Let P be a monotone polygon consisting of n vertices, and let $0 < \rho \leq 1$ be a fixed parameter. With $O(n \log n)$ preprocessing time and $O(n)$ space, we can determine if $\text{area}(S \cap P) \geq \rho \cdot \text{area}(P)$ in $O(\log n)$ time for any horizontal slab S.*

6.3 Extending to Rectangular Queries

Our actual query, as introduced in Figure 6.1, is a rectangular area. Our solution to this type of query is extended from the methods we used to solve a horizontal slab query.

6.3.1 Preprocessing

We develop a list of regions $\mathcal{R} = R_1, R_2, \ldots, R_{n-1}$ just as in Section 6.2.1. Now, instead of constructing a single list of multi-region formulas to cover all of P, we will construct a tree of such lists so that for any $1 \leq i \leq j \leq n-1$, we can query the subpolygon $\bigcup_{k=i}^{j} R_k$ for its area below a query line h in $O(\log n)$ time.

Let this multi-region formula tree be T; we construct T in the following way. First, construct the multi-region formula tree for each half of the regions recursively, giving the subtrees T_l and T_r. If $|\mathcal{R}| = 1$ for any recursive step, we create a leaf and return the area formula for that single region.

Let $H(T_l)$ and $H(T_r)$ be the multi-region formula lists for T_l and T_r, respectively. We need to create a merged list, $H(T)$ representing the regions of both subtrees together. Unfortunately, we cannot just naively merge the formulas into a combined sorted list as each formula is only concerned with the regions upon which it was originally created.

Instead, we need to generate new coefficients for each critical height of h which will be valid for all regions of the combined area. This process is precisely Algorithm 6.1 *without* the initial sorting step, which we can avoid since $H(T_l)$ and $H(T_r)$ are already sorted. Thus, we can build $H(T)$ in time $O(|H(T_l)| + |H(T_r)|)$. Algorithm

6.2 gives more formal details.

Algorithm 6.2: BuildMultiRegionFormulaTree
Input: List of regions $\mathcal{R} = R_1, R_2, \ldots, R_{n-1}$

if $|\mathcal{R}| = 1$ **then**
  $T \leftarrow$ create new leaf node
  $H(T) \leftarrow F(R, h)$
  return $T, H(T)$
end
$m \leftarrow \left\lfloor \frac{|\mathcal{R}|}{2} \right\rfloor$
$\mathcal{R}_l = \{R_1, R_2, \ldots, R_m\}$
$\mathcal{R}_r = \{R_{m+1}, R_{m+2}, \ldots, R_{n-1}\}$
$T_l, H(T_l) \leftarrow$ `BuildMultiRegionFormulaTree(`R_l`)`
$T_r, H(T_r) \leftarrow$ `BuildMultiRegionFormulaTree(`R_r`)`
$T \leftarrow$ Create new node with left child T_l and right child T_r
$H(T) \leftarrow$ Merge $H(T_l)$ and $H(T_r)$ using Algorithm 6.1
return $T, H(T)$

In the last step of the algorithm we create a list of formulas over all regions, which implies that we can answer horizontal slab queries from the root of T. Recursing on half of the regions at each step gives us a tree which will be balanced with depth $O(\log n)$. At each level of T, every critical height for every region is considered, resulting in $O(n)$ area formulas. As mentioned, the merging step at each node is completed in linear time with respect to the number of regions processed. Therefore, the total time and storage required to build T is $O(n \log n)$.

6.3.2 Querying

Our query rectangle Q is given by the lower-left and upper-right coordinates (α, β) and (γ, δ), respectively. We define $\overline{\beta}$ and $\overline{\delta}$ as the horizontal lines through β and δ, and $\overline{\alpha}$ and $\overline{\gamma}$ as vertical lines through α and γ, respectively.

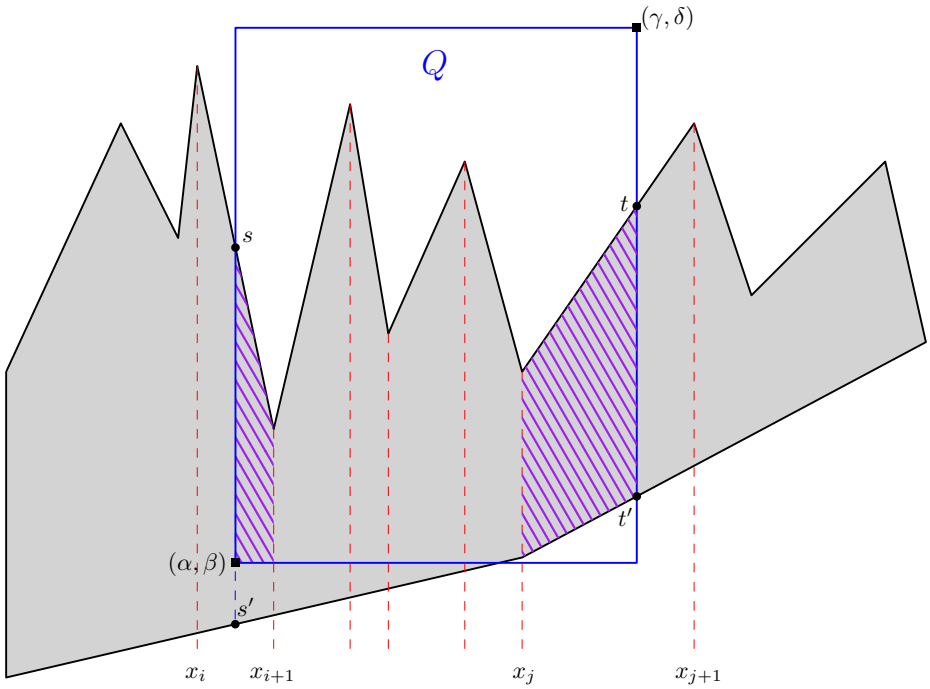

Figure 6.3: A monotone polygon P with query Q. The tiled areas cannot be queried using preprocessed area formulas and will require special handling.

Using a binary search on the x-values of P, we can identify the regions which $\overline{\alpha}$ and $\overline{\gamma}$ pass through; let these regions be R_i and R_j, respectively. Let $V(Q)$ be the vertical slab defined by Q; that is, the vertical area between $\overline{\alpha}$ and $\overline{\gamma}$. We can calculate the area of $Q \cap P$ by considering how Q interacts with the following areas:

1. The leftmost region R_i, where $R_i \not\subset V(Q)$. Let $A_i = \text{area}(Q \cap R_i)$.

2. The center regions $R_{i+1}, R_{i+2}, \ldots, R_{j-1}$, where $R_k \subset V(Q)$ for $i+1 \leq k \leq j-1$. Let $A_c = \text{area}\left(Q \cap \left(\bigcup_{k=i+1}^{j-1} R_k\right)\right)$.

3. The rightmost region R_j, where $R_j \not\subset V(Q)$. Let $A_j = \text{area}(Q \cap R_j)$.

See Figure 6.3 for an example. To calculate A_i, A_c, and A_j, we begin with the

center regions. All of these regions are entirely within $V(Q)$, and so we can use their precalculated area formulas directly. That is:

$$A_c = \text{area}\left(Q \cap \left(\bigcup_{k=i+1}^{j-1} R_k\right)\right) = \sum_{k=i+1}^{j-1} F(R_k, \delta) - F(R_k, \beta)$$

Performing this sum naively takes $O(n)$ time as there may be a linear number of regions spanned by Q. However, using T, we can answer this query for any values of i and j by checking at most $O(\log n)$ subtrees. At each subtree, we require $O(\log n)$ time to find the correct formula, for a total query time of $O\big(\log^2 n\big)$. However, since each subtree queries for the same value of h, we can reduce this query time to only $O(\log n)$ by using fractional cascading.[6, 7]

Considering A_i now, if $\alpha = x_i$, then $A_i = F(R_i, \delta) - F(R_i, \beta)$. In general, however, $x_i < \alpha < x_{i+1}$, and we cannot use our precalculated area formulas. Fortunately, $Q \cap R_i$ is a polygon of $O(1)$ complexity, specifically a trapezoid, so its area can be calculated directly in constant time. Recall from the construction of the list of regions, $\mathcal{R}$, that if $\overline{\alpha}$ passes between x_i and x_{i+1}, then it passes through region R_i, which stores the edges e_i and e_i' defining its top and bottom. We can calculate the intersection points with $\overline{\alpha}$ as $s = e_i \cap \overline{\alpha}$ and $s' = e_i' \cap \overline{\alpha}$. Recall that $a_{i+1} = \overline{x_{i+1}} \cap e_{i+1}$, and $a_{i+1}' = \overline{x_{i+1}} \cap e_{i+1}'$. Thus, the vertices of $Q \cap R_i$ are:

- Its top-left: the lower of (α, s) and (α, δ),
- Its bottom-left: the higher of (α, s') and (α, β),
- Its top-right: the lower of a_{i+1} and $\overline{\delta} \cap \overline{x_{i+1}}$, and
- Its bottom-right: the higher of a_{i+1}' and $\overline{\beta} \cap \overline{x_{i+1}}$.

Likewise, if $\gamma = x_j$, then $A_j = 0$. But, in general, $x_j < \gamma < x_{j+1}$, and, again, we cannot use our precalculated area formulas. $Q \cap R_j$ is also a trapezoid, however, so its area can be calculated directly in a similar way to A_i.

With all three area calculations completed, a simple sum completes our query. We summarize our results with the following theorem and corollary.

Theorem 6.3. *Let P be a monotone polygon consisting of n vertices. In $O(n \log n)$ time and $O(n \log n)$ space, we can create a data structure which allows us to determine $\mathrm{area}(Q \cap P)$ in $O(\log n)$ time, for any axis-parallel rectangular query region Q.*

Corollary 6.4. *Let P be a monotone polygon consisting of n vertices, and let $0 < \rho \leq 1$ be a fixed parameter. With $O(n \log n)$ preprocessing time and $O(n \log n)$ space, we can determine if $\mathrm{area}(Q \cap P) \geq \rho \cdot \mathrm{area}(P)$ in $O(\log n)$ time for any axis-parallel rectangular query Q.*

6.4 An Alternate Approach

In this section, we detail an alternate approach to solving this problem which sacrifices some query time in favour of lowering our space requirements to only $O(n)$.

We begin by decomposing P into $O(n)$ trapezoidal regions and creating area formulas for each just as in Section 6.2. Next, we divide P into $O(\sqrt{n})$ multi-regions, from left to right, each comprised of $O(\sqrt{n})$ trapezoidal regions. For each multi-region, we calculate its multi-region area formula using Algorithm 6.1. Altogether, this process requires $O(n \log n)$ time, and $O(n)$ storage.

Our query Q is a box which is open to the left and bottom. We can specify Q by a single point with coordinates (γ, δ). Figure 6.4 shows an example query and a

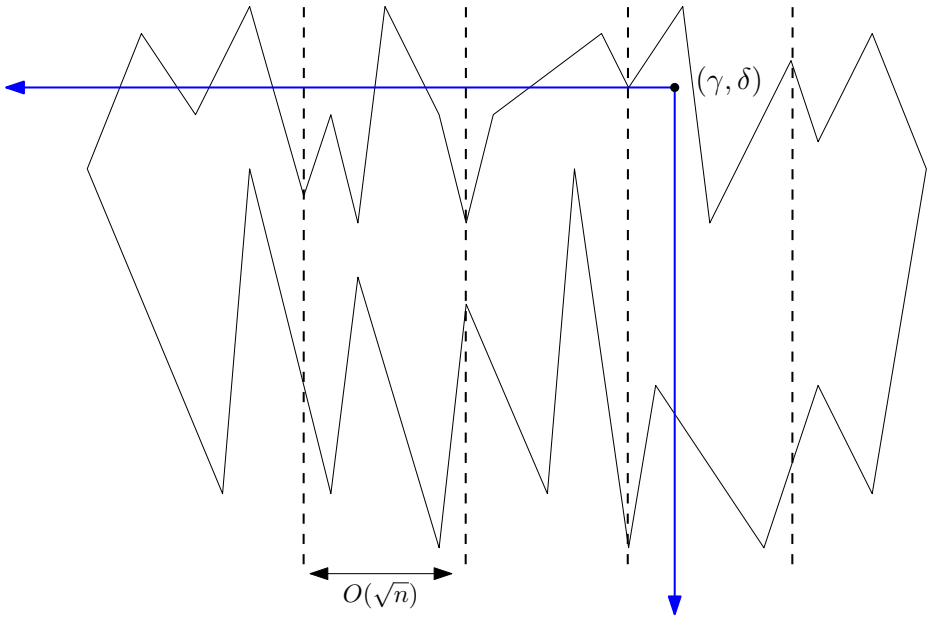

Figure 6.4: A monotone polygon P with query Q. The query region is box which is open to the left and bottom.

decomposition of P into $O(\sqrt{n})$ multi-regions.

Performing our query requires only a few steps which consider successively finer regions of our decomposition of P. With a binary search, we can identify the multi-regions $S_1, S_2, \ldots, S_{k-1}$, where $1 \leq k < \sqrt{n}$, which are *entirely* to the left of γ, and the at most one multi-region S_k which is intersected by γ. For each S_i, $1 \leq i \leq k-1$, we evaluate the associated multi-region area formula with respect to δ. This requires $O(\log |S_i|) = O(\log n)$ time per multi-region for a total of $O(\sqrt{n} \log n)$ time.

We then consider S_k, which is intersected by γ. S_k is comprised of $O(\sqrt{n})$ trapezoidal regions, some of which are entirely left of γ, and at most one of which is intersected by γ. For all of those regions entirely left of γ, we evaluate the regional area formulas associated with them with respect to δ. The intersection of the final, partial trapezoidal region which is intersected by γ has only $O(1)$ complexity, so we

can evaluate its area directly.

This completes the query. In total, we require the following time to calculate the area inside of Q:

$$O(\log n) + O\big(\sqrt{n}\log n\big) + O\big(\sqrt{n}\big) + O(1) = O\big(\sqrt{n}\log n\big)$$

We can extend this method to query on a closed rectangle by combining the results of four separate open box queries. Let Q' be a closed rectangle defined by its lower-left point (α, β) and its upper-right point (γ, δ). We can find $\mathrm{area}(Q')$ in the following way:

- Let Q_1 be the open query box on (γ, δ).
- Let Q_2 be the open query box on (γ, β), which captures excess area under Q'.
- Let Q_3 be the open query box on (α, δ), which captures excess area left of Q'.
- Let Q_4 be the open query box on (α, β), which captures the area which is double counted by Q_2 and Q_3.

Then,

$$\mathrm{area}(Q') = \mathrm{area}(Q_1) - \mathrm{area}(Q_2) - \mathrm{area}(Q_3) + \mathrm{area}(Q_4)$$

6.4.1 Improving Query Time

We can reduce our query time to only $O(\sqrt{n})$ by grouping the regions into three different sized groups instead of just two.

As before, we decompose P into $O(n)$ trapezoidal regions and create their area formulas. Next, from left to right, we group the regions into $\frac{\sqrt{n}}{\log n}$ multi-regions

each containing $O(\sqrt{n}\log n)$ trapezoidal regions. We refer to these as the A-level regions, and calculate a multi-region formula for each using Algorithm 6.1. Total preprocessing time and space for the A-level regions is $O(n\log n)$ and $O(n)$, respectively. Within each A-level region, we create B-level multi-regions consisting of $O(\sqrt{n})$ trapezoidal regions each, again from left to right, resulting in $O(\log n)$ B-level regions for each A-level region. Total preprocessing time and space for the B-level regions is also $O(n\log n)$ and $O(n)$, respectively, across all A-levels regions.

Our queries use the same definitions as from Section 6.4. Given a query Q defined by (γ, δ), we first consider what areas are left of γ, and then use δ as the input to our area formulas.

We start by considering the A-level regions which are entirely left of γ. Each such region contains $O(\sqrt{n}\log n)$ trapezoidal regions, so querying the multi-region area formula stored there will require the following time.

$$\log O\big(\sqrt{n}\log n\big) = O(\log n)$$

Therefore, the worst case time required to query all necessary A-level regions is:

$$\frac{\sqrt{n}}{\log n} \cdot O(\log n) = O\big(\sqrt{n}\big)$$

At most one A-level region is intersected by γ, and if that is the case, we proceed to its respective B-level regions. Of these B-level regions, at most $O(\log n)$ are entirely to the left of γ. Each has size $O(\sqrt{n})$, so querying the multi-region area formulas stored with them requires $O(\log n)$ time each. Thus, the total query time for B-level regions is $O\big(\log^2 n\big)$.

At the last level, at most one B-level region is intersected by γ. This region contains $O(\sqrt{n})$ trapezoidal regions which are entirely left of γ. We can evaluate the area formula stored with each region in $O(\sqrt{n})$ total time. Finally, there may be one trapezoidal region which is intersected by γ. This intersection produces an area of $O(1)$ complexity which we can calculate directly.

Total time required for this query is as follows.

$$O(\log n) + O\big(\sqrt{n}\big) + O\big(\log^2 n\big) + O\big(\sqrt{n}\big) + O(1) = O\big(\sqrt{n}\big)$$

Total preprocessing time is $O(n \log n)$ time for sorting and building the region and multi-region area formulas. Total space required is $O(n)$ as each trapezoidal region is considered in only one A-level region and in only one B-level region. We summarize these results with the following theorem.

Theorem 6.5. *Let P be a monotone polygon consisting of n vertices. In $O(n \log n)$ time and $O(n)$ space, we can create a data structure which allows us to determine $\mathrm{area}(Q \cap P)$ in $O(\sqrt{n})$ time, for any axis-parallel rectangular query region Q.*

6.5 Remarks on Simple Polygons

We can apply our method for horizontal slab queries from Section 6.2 “as is” to simple polygons since nothing about the way that we decompose P, build the multi-region area formulas, or query them, depends on the monotone property. Put another way, we do not require any special handling for regions which appear above or below other regions.

To answer slab queries on simple polygons, we decompose P by extending rays

from each vertex into the interior of P until they strike the next boundary. The regions will have the same 4-sided shape as in the monotone polygon case, and we can create an area formula for each one over the, at most, four critical heights of h.

Each region still maintains its own critical heights for h, and we use Algorithm 6.1 without modification, first sorting all critical heights together, then maintaining a set of coefficients which we adjust by region as we sweep from bottom to top.

Finally, querying the resulting list of multi-region formulas, H, is done using the same binary search method as in the monotone case. This result is summarized in the following corollary.

Corollary 6.6. *Let P be a simple polygon consisting of n vertices. In $O(n \log n)$ time and $O(n)$ space, we can create a data structure which allows us to determine $\text{area}(S \cap P)$ in $O(\log n)$ time for any horizontal slab query region S.*

Unfortunately, we cannot extend our method for rectangular queries to work with simple polygons so easily. While the multi-region formulas themselves do not use the monotone property, our tree of multi-region formulas does. The tree functions by partitioning the trapezoidal regions with respect to vertical lines, however, in a simple polygon, a vertical line passing through the boundary of one region may pass through the interior of another. This lack of clean partitioning prevents the multi-region formula from working correctly for all possible horizontal query lines which may be given as input to the formula.

6.6 Conclusion

In this chapter we have developed methods for calculating the area of a monotone polygon enclosed by a query slab or rectangle. Specifically, we have described:

1. A method for calculating the area of a monotone polygon enclosed in a horizontal slab, Theorem 6.1.

2. A method for calculating the area of a monotone polygon enclosed in an axis-parallel rectangle, Theorem 6.3.

3. An alternate method for calculating the area of a monotone polygon enclosed in an axis-parallel rectangle which is more space efficient, but requires more query time; Theorem 6.5.

4. A method for calculating the area of a simple polygon enclosed in a horizontal slab, Corollary 6.6.

In every case, once the enclosed area is known, deciding whether the partial enclosure property is satisfied is just a matter of testing a simple inequality.

One possible direction for future research is to consider our options for extending Theorem 6.3 to simple polygons. We've already seen that it does not work directly, but there may be alternate methods for partitioning the multi-region formulas or accounting for partially enclosed regions.

Chapter 7

Conclusion

In this final chapter, we summarize the contributions from the previous chapters as well as possible directions for future research.

7.1 Summary of Contributions

This thesis presents several contributions in the area of partial enclosure range searching.

In Chapter 3, we developed methods for querying line segments with axis-parallel rectangles. We first considered axis-parallel segments and gave a method for expressing the partial enclosure property as an orthogonal range query. Next we considered arbitrarily-oriented line segments and we saw that the uncertainty of where a segment may intersect the query region significantly increases the complexity of testing partial enclosure expressions.

In Chapter 4, we consider axis-parallel line segments against a query region which is either an arbitrarily-oriented slab, or the intersection of two such slabs;

the latter is a generalization of querying with an arbitrarily-oriented rectangle. We gave a method for expressing the partial enclosure property as a half-plane query in a dual-space.

In Chapter 5, we moved away from line segments to consider convex polygons and partial enclosure range searches involving area. We developed a method for preprocessing a convex polygon such that the area to one side of any chord can be found in logarithmic time with respect to the number of edges in the polygon. We gave an algorithm for decomposing the intersection of a rectangle with a convex polygon in such a way that the area of their intersection could be found in logarithmic time with respect to the number of edges in the polygon.

In Chapter 6, we considered monotone polygons. We gave an algorithm which produces a list of area formulas which can be queried for the area of the polygon underneath a horizontal query line. We combined this approach with a recursive decomposition of the polygon to create a method for calculating the area of a monotone polygon within a query rectangle in logarithmic time with respect to the number of vertices in the polygon.

7.2 Future Work

In this section, we summarize the future work that has appeared in previous chapters.

1. From Chapter 3, can we remove the limitation that $1/2 < \rho$ when querying on arbitrarily-oriented line segments? This limitation stems from our current solution relying on the centrepoint of any matching segment residing inside

the query region.

2. From Chapter 3, can we classify the endpoints of the segments in a way that reduces the number of partial enclosure expressions that we need to test? The correct partial enclosure expression depends on which boundaries of the query rectangle are crossed by a segment, but our current classification technique is limited in how well it can identify these.

3. From Chapter 3, in a practical setting, when we are better off saving a constant factor of space by sharing the endpoint classification structures versus saving a $\log n$ factor by performing the half-plane queries first?

4. From Chapter 4, extending our method to arbitrarily-oriented segments results in a very case-heavy data structure. Moreover, the partial enclosure expressions involve several query variable expressions resulting in high-degree half-space queries. Can a better method of classification reduce this?

5. From Chapter 6, can we modify our approach to building a multi-region formula tree to support querying on simple polygons? Our tree relies on clean boundaries between regions so that we can query a single formula for any horizontal line query. In a simple polygon setting, two regions may be located one over top of the other without sharing common boundaries. Attempting to partition the polygon by the boundary of one region can therefore result in subregions which break our area formulas.

Bibliography

[1] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1999.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] Jon Louis Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5): 244–251, 1979.

[4] Timothy M. Chan. Optimal partition trees. *Discrete & Computational Geometry*, 47(4):661–690, 2012.

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

[6] Bernard Chazelle and Leonidas Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[7] Bernard Chazelle and Leonidas Guibas. Fractional cascading: II. applications. *Algorithmica*, 1:163–191, 1986.

[8] Bernard Chazelle, Leonidas J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.

[9] Jurek Czyzowicz, F. Contreras-Alcalá, and Jorge Urrutia. On measuring areas of polygons. In *Proceedings of the 10th Canadian Conference on Computational Geometry, McGill University, Montréal, Québec, Canada*, 1998.

[10] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry*. Springer-Verlag, Berlin, third edition, 2008. ISBN 978-3-540-77973-5. Algorithms and applications.

[11] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1987. ISBN 978-3-642-64873-1.

[12] Herbert Edelsbrunner and Roman Waupotitsch. Computing a ham-sandwich cut in two dimensions. *J. Symb. Comput.*, 2(2):171–178, 1986.

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

[14] Partha P. Goswami, Sandip Das, and Subhas C. Nandy. Triangular range counting query in 2d and its application in finding k nearest neighbors of a line segment. *Comput. Geom.*, 29(3):163–175, 2004.

[15] John Iacono and Stefan Langerman. Volume queries in polyhedra. In *Discrete and Computational Geometry, Japanese Conference 2000, Tokyo, Japan, November, 22-25, 2000, Revised Papers*, pages 156–159, 2000. doi: 10.1007/3-540-47738-1_13. URL `http://dx.doi.org/10.1007/3-540-47738-1_13`.

[16] R.J. Jarrett, G.A. Schobbe, M. Iwema, C.E. Lui, F.D. Jones, E.K. Rimas, B. Dresevic, and S. Bhattacharyay. Lasso select, October 11 2011. US Patent 8,037,417.

[17] Jirí Matousek. Efficient partition trees. *Discrete & Computational Geometry*, 8: 315–334, 1992.

[18] Jirí Matousek. Range searching with efficient hiearchical cutting. *Discrete & Computational Geometry*, 10:157–182, 1993.

[19] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[20] Nimrod Megiddo. Partitioning with two lines in the plane. *J. Algorithms*, 6 (3):430–433, 1985.

[21] Subhas C. Nandy, Sandip Das, and Partha P. Goswami. An efficient k nearest neighbors searching algorithm for a query line. *Theor. Comput. Sci.*, 1-3(299): 273–288, 2003.

[22] Dan E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11(1):149–165, 1982.

[23] Dan E. Willard. Applications of range query theory to relational data base join and selection operations. *J. Comput. Syst. Sci.*, 52(1):157–169, 1996.