Exploring the Problem Space of Orthogonal Range Searching

by

Bryan T. Wilkinson

A dissertation presented to the
Faculty of Science and Technology at Aarhus University
in fulfillment of a requirement for the degree of
Doctor of Philosophy in Computer Science

Aarhus, Denmark, 2015

Abstract

Orthogonal range searching is a fundamental problem in computational geometry: preprocess a set of points into a data structure such that we can efficiently answer questions about the points that lie in axis-aligned rectangles called query ranges. There are many interesting variants of orthogonal range searching with differing complexities. We consider classic fundamental variants, new variants inspired by practical applications, and special cases that admit efficient solutions. In general, we explore the problem space of orthogonal range searching to refine our understanding of which problems can be solved efficiently. We thus become better informed of the feasibility of applying orthogonal range searching techniques to solve both practical and theoretical problems.

We give improved bounds for dynamic range reporting, a fundamental variant in which points can be inserted and deleted and a query asks for a list of the points in the query range. We consider the problem in the word RAM model. The best previous data structure uses optimal space and query time, and $O(\log^c n)$ update time for a positive constant $c < 1$. We reduce the exponent in the update time for the two- and three-sided special cases. We also obtain significantly more efficient data structures for the very special case of insertion-only range emptiness. Here, we need only decide whether a query range is empty. This result exemplifies that there are many simple variants of range searching that can still be improved. In arriving at our results, we first design an external memory data structure. Our word RAM solutions pack points into words and simulate the external memory solution. This approach allows for clean specifications of packed word algorithms, which otherwise tend to require more details.

In colored range counting, points are colored and a query asks for the number of distinct colors in the query range. There are no known efficient data structures for colored range counting and a conditional lower bound for the offline case gives evidence that it is hard. We consider the special case in which all query ranges are squares, solving it as efficiently as uncolored range counting with arbitrary query ranges. We give a conditional lower bound for the offline cube case in three dimensions, giving evidence that our upper bound may not extend to higher dimensions. We also consider a related algorithmic problem of interest in GIS and ecology: given a raster of colors, output the number of distinct colors in the square or circular window around each cell. We give theoretically and practically efficient algorithms for these problems.

In concurrent range reporting, we again have colored points. A concurrent query includes a subset of colors and the answer to the query includes, for each query color c, the range reporting result for the points of color c. This type of query is prevalent in practice (e.g., consumer websites that allow filtering search results by preferred brands) but there is little relevant theoretical work. We give near-optimal pointer machine data structures. We also show that two different and reasonable representations of a query are in fact equivalent, clearing up the problem's theoretical foundations.

Resumé

Ortogonal område søgning er et grundlæggende problem i geometrisk databehandling: præprocesser et sæt af punkter til en datastruktur, så vi effektivt kan besvare spørgsmål om de punkter, der ligger i akse parallelle rektangler kaldet forespørgselsområde. Der er mange interessante varianter af ortogonal område søgning med forskellige kompleksiteter. Vi betragter klassiske grundlæggende varianter, nye varianter inspireret af praktiske anvendelser, og særlige tilfælde, som har effektive løsninger. Generelt undersøger vi problemrummet for ortogonal område søgning for at forfine vores forståelse af, hvilke problemer der kan løses effektivt. Vi bliver således bedre informeret om muligheden for at anvende ortogonal område søgningsteknikker til at løse både praktiske og teoretiske problemer.

Vi giver forbedrede grænser for dynamisk område rapportering, en grundlæggende variant, hvor punkter kan indsættes og slettes, og en forespørgsel beder om en liste over de punkter i forespørgslens område. Vi betragter problemet i word RAM model. Den bedste forrige datastruktur bruger optimal plads og forespørgsel tid, og $O(\log^c n)$ opdaterings tid for en positiv konstant $c < 1$. Vi reducerer eksponenten i opdateringenstiden for to- og tre-sidede forespørgselsområder. Vi opnår også betydeligt mere effektive datastrukturer for det meget specielle tilfælde af område-tomhed hvor kun indsættelser er tilladt. Her behøver vi kun at beslutte, om et forespørgselsområde er tomt. Dette resultat eksemplificerer, at der er mange simple varianter af område søgning, der stadig kan forbedres. Undervejs i vores resultater designer vi en ekstern hukommelse data struktur. Vores word RAM løsninger pakker punkter ind i ord og simulerer så den eksterne hukommelses data struktur. Denne fremgangsmåde giver mulighed for rene specifikationer af pakkede ord algoritmer, som ellers har tendens til at kræve flere detaljer.

I farvet område optælling er punkterne farvede og en forespørgsel beder om antallet af forskellige farver i forespørgslens område. Der er ingen kendte effektive datastrukturer for farvet område optælling og en betinget nedre grænse for offline varianten giver belæg for, at det er svært. Vi betragter det særlige tilfælde, hvor alle query intervaller er kvadrater, og løser det så effektivt som ufarvet område optælling med vilkårlige forespørgselsområde. Vi giver en betinget nedre grænse for offline kube varianten i tre dimensioner, der giver dokumentation for, at vores øvre grænse ikke kan strække sig til højere dimensioner. Vi betragter også et relateret algoritmisk problem som er interessant for GIS og økologi: givet et raster af farver, beregn antallet af forskellige farver i et kvadrat eller et cirkulært vindue omkring hver celle. Vi giver teoretisk og praktisk effektive algoritmer for disse problemer.

I samtidig område rapportering har vi igen farvede punkter. En samtidig forespørgsel indeholder en delmængde af farver og svaret på forespørgslen omfatter for hver forespørgselsfarve c, område rapporteringsresultatet for de punkter med farven c. Denne type af forespørgsel er udbredt i praksis (f.eks, forbruger hjemmesider, der tillader filtrering af søgeresultater ved foretrukne mærker), men der er kun lidt relevant teoretisk arbejde. Vi giver nær-optimale datastrukturer i pointer maskine modellen. Vi viser også, at to forskellige og rimelige repræsentationer af en forespørgsel i virkeligheden er ækvivalente hvilket rydder op i problemets teoretiske fundament.

Acknowledgements

The last few years have been an incredible journey and it would not have been possible without the support of numerous individuals. I would like to extend my thanks to these wonderful mentors, colleagues, and friends.

I would first like to thank my supervisors Lars Arge and Peyman Afshani. Thanks Lars for creating a research environment that provides freedom, opportunity to collaborate, and a lot of fun. Thanks Peyman for our many brainstorming sessions, for being a steady source of an overabundance of interesting problems, and for guiding me to better understand and strive for qualities that make a strong researcher.

Thanks to all of my coauthors for our fruitful and enjoyable collaborations. I have learned a lot from working with each of you. Thanks Peyman Afshani, Sang Won Bae, Siu-Wing Cheng, Otfried Cheong, Mark de Berg, Kevin Matulef, Cheng Sheng, Yufei Tao, Constantinos Tsirogiannis, Emo Welzl, and Juyoung Yon.

Thanks to Rolf Fagerberg who came from Odense to be on the committee for my qualifying exam. Thanks to John Iacono, Stephen Alstrup, and Susanne Bødker for agreeing to be on my PhD assessment committee.

From September 2014 to January 2015, I had the opportunity to visit the Discrete and Computational Geometry Laboratory at KAIST in Daejeon, South Korea. Thanks to my gracious host Otfried Cheong and to all of the members of his lab—Yujeong Cho, Michael Dobbins, Ji-won Park, Yujin Shin, and Juyoung Yon—who made me feel at home in their lab. Special thanks go to my amazingly generous friends Ben McCoy and Moohwa Lee for helping me get by in South Korea.

Working at MADALGO has been an unusually smooth experience due in large part to its incredible administrative staff, especially Else Magård and Trine Ji Holmgaard Jensen. Thanks for all of the support. I enjoyed the time I spent sharing offices with both Kasper Green Larsen and Sarfraz Raza; thank you both for our interesting discussions. There are and were many others at MADALGO who contributed to my experience at the lab in meaningful ways. Thanks especially to Edvin Berglin, Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, Jesper Asbjørn Sindahl Nielsen, Darius Sidlauskas, Jakob Truelsen, Ingo van Duijn, Freek van Walderveen, and Jungwoo Yang.

I am grateful for advances in technology that have allowed me to communicate with my parents from a distance of over 5000 km. Thanks for your continued support.

Bryan T. Wilkinson,
Aarhus, July 31, 2015.

Preface

This dissertation is the culmination of my three years of study at the Center for Massive Data Algorithmics (MADALGO) under the supervision of Lars Arge and Peyman Afshani. MADLAGO is a research center of the Faculty of Science and Technology at Aarhus University in Aarhus, Denmark.

This dissertation includes only my research related to orthogonal range searching, which constitutes a significant proper subset of the research completed during my studies. In particular, it is based on the following three papers:

[1] Peyman Afshani, Cheng Sheng, Yufei Tao, and Bryan T. Wilkinson. Concurrent range reporting in two-dimensional space. In *Symposium on Discrete Algorithms (SODA)*, pages 983–994, 2014.

[2] Bryan T. Wilkinson. Amortized bounds for dynamic orthogonal range reporting. In *European Symposium on Algorithms (ESA)*, pages 842–856, 2014.

[3] Mark de Berg, Constantinos Tsirogiannis, and Bryan T. Wilkinson. Fast computation of categorical richness on raster data sets and related problems. Manuscript, 2015.

Parts of this dissertation are thus collaborations with Peyman Afshani, Mark de Berg, Cheng Sheng, Yufei Tao, and Constantinos Tsirogiannis. At the time of writing, the manuscript [3] is in submission to the 23rd ACM International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2015).

I also worked on the following results, which are not currently published:

- A convex program that finds the homothete of a convex body that minimizes the volume of the symmetric difference between it and another fixed convex body. This is joint work with Sang Won Bae, Siu-Wing Cheng, Otfried Cheong, Emo Welzl, and Juyoung Yon.
- External memory algorithms for problems on data stored in linked lists that are laid out compactly on disk such that consecutive nodes are not necessarily consecutive on disk. We can answer a predecessor search query in $O(\sqrt{N/B})$ expected I/Os or test whether two convex polygons intersect in $O(\sqrt{\text{sort}(N)})$ expected I/Os. This is joint work with Peyman Afshani and Kevin Matulef.
- A linear bound on the length of generalized Davenport-Schinzel sequences avoiding the family of patterns $\{axaxa, axababxb, axababcbcxc, \dots\}$.
- An $O(n\alpha(n))$-time pointer machine algorithm to compute the lower envelope of disjoint line segments given their endpoints in sorted order by x-coordinate.

Table of Contents

Chapter 1

Introduction

1.1 Field of Study

Theoretical computer science. In *theoretical computer science*, we design and study mathematical models of the capabilities of real world computers. Such a mathematical model is called a *model of computation*. In the real world, computers are physical objects that operate by the rules of physics. The hope is that our models of computation behave similarly enough to real world computers such that knowledge that we obtain about our models of computation is easily transformed into knowledge of real world computers. In the early days of theoretical computer science, the most important question about a model of computation was: what can be computed? In other words, which problems can be solved?

This type of question is at the heart of computability theory. The study of computability theory has been highly successful and for most practical purposes it can be considered more or less complete. The focus of theoretical computer science has since shifted to asking: how efficiently can problems be solved? Models of computation specify various resources which are deemed to be scarce: e.g., processing time, data storage space, random bits, etc... We say that a solution to a problem is *efficient* if it uses resources minimally. This type of question has turned out to be much harder to answer than the questions of computability theory. For example, the most important open problem in theoretical computer science, the P versus NP problem, asks whether or not problems whose solutions can be verified efficiently can also be solved efficiently.

In practice, one way to solve a problem more efficiently is to use a faster computer or wait for a faster computer to exist. If these are adequate solutions, then theoretical computer scientists are not interested. For this reason, we measure resource usage using Big-Oh notation, which hides constant factors and thus abstracts away the specific

computer being used. Big-Oh notation also hides low order terms which become negligible as the input size tends to infinity. The case of large input size is the important case for theoretical computer scientists; if input sizes are small, then it is less important to solve problems efficiently.

Algorithms and data structures. An *algorithm* is a method to obtain the solution to a problem using only the capabilities of a model of computation. The input to an algorithm is data about which we are asking a question and the output of an algorithm is the answer to the question. In the study of algorithms, we seek the most efficient algorithms for specific problems. The problems that we consider are motivated by some practical or theoretical interest. Binary search is an example of a commonly used algorithm. The input to the algorithm is a sorted array of numbers and a query number q. The output is the number in the array that is closest to q but not greater than q. This number is the *predecessor* of q. If the input array contains n numbers, then binary search requires only $O(\log n)$[1] time.

Is binary search the best possible algorithm for the problem that it solves? It depends on the model of computation. Determining the best algorithm for a problem in a model of computation requires a two-pronged approach. First, it requires designing efficient algorithms. Second, it requires studying the limitations of the model of computation. We call the efficiency of an algorithm that solves a problem an *upper bound*. We call a barrier in efficiency that a model of computation cannot break for a problem a *lower bound*. When we have an upper bound for a problem that matches a lower bound for the same problem, then we have found an *optimal* algorithm for the problem. The primary goal of the study of algorithms is to find optimal algorithms for problems.

Sometimes we want to answer many different questions about the same data. Also, sometimes we have data and we know that we will eventually want to answer questions about the data, but we don't yet know the precise questions that will be asked. In practice, this may be because it is not us asking the questions, but rather our clients. In these cases, we want to build a *data structure*. A data structure is a way of organizing data within a model of computation such that, in the future, we can efficiently answer questions about the data. For example, if we are given an unsorted array of numbers, we may choose to structure the data by sorting the array. Then, if we are presented with any query number q, we can find its predecessor efficiently using binary search. We only need to sort the array once in the beginning, and then every query in the future can be handled efficiently.

We call the algorithm that structures the input data in the beginning a *preprocessing* algorithm. Algorithms that answer questions about the structured data are *query*

[1] All logarithms without an explicit base in this dissertation have base two. Furthermore, we follow the convention that every logarithm returns a value of at least one, by defining $\log x = \max\{1, \log_2 x\}$.

algorithms. Data structures may also come equipped with *update* algorithms, which modify the original input data and update the structured data to reflect the changes. If a data structure includes update algorithms then it is called *dynamic*. Otherwise, it is *static*.

The efficiency of a data structure is typically determined by the running times of its associated algorithms and by the amount of structured data it stores between invocations of the algorithms. This latter metric is called the *space* consumed by the data structure.

Computational geometry. *Computational geometry* is the study of algorithmic problems that are specified with geometric objects. It is easy to see how the physical world around us can be modeled as abstract geometric objects. For example, a terrain can be modeled with a three-dimensional triangle mesh or with a raster (i.e., two-dimensional array) of elevation values. The layout of a floor of a building can be modeled with line segments that represent walls. Trajectories of vehicles through space-time can be modeled with four-dimensional polygonal chains. Once we have a geometric model of the real world, asking questions about the model can give useful information about the real world. How much rain must fall before some location on a terrain becomes flooded? How can we place the minimal number of guards in a building so that every nook and cranny can be seen by some guard? When are two moving vehicles closest to each other? All of these questions can be answered by solving algorithmic problems on geometric models of the physical world.

These applications relating to the physical world are sufficient to make computational geometry an important field of study. However, these applications are just the tip of the iceberg of applications of computational geometry. In fact, almost all data can be interpreted geometrically! Consider, for example, employees of a company. An employee has an age, a date of birth, and a salary. Knowing the values of these attributes, we can model all employees as points in a three-dimensional space in which one axis represents age, another represents date of birth, and the last represents income. Note that age and date of birth are obviously correlated. This correlation manifests itself geometrically in our model by the fact that all points lie on a specific plane that cuts through our three-dimensional space. Specifically, all points lie on the plane satisfying the equation $\texttt{age} = \texttt{current date} - \texttt{date of birth}$.

More generally, objects with d numerical attributes can be modeled as points in d-dimensional space. It may be interesting to ask: what are the objects with the most extreme trade-offs between attributes? Conversely, what object is most central with respect to the attributes (e.g., in the same way that the mean or median of a set of numbers is central)? These are again questions that can be answered by solving algorithmic problems on our geometric model.

Range searching. *Range searching* is a fundamental problem studied in computational geometry. The problems considered in this dissertation are all range searching problems. Range searching is a data structure problem in which the data that we structure is a set of d-dimensional points. As described above, these points can, for example, represent objects with d numerical attributes. A query to a range searching data structure asks a question about the subset of points that lie within a specified range. For example, if the points are two-dimensional, a query might ask for the number of points that lie within a specific circle. This particular range searching problem is called circular range counting. For each combination of dimension, question, and range type there is a corresponding range searching problem. Other questions might be: which points lie in the range? Does a point lie in the range? Points may be augmented with weights or colors. Then, we can ask: what is the total weight of the points in the range? How many distinct colors appear in the range? Types of ranges include axis-aligned rectangles, circles, halfspaces, triangles, and their higher-dimensional analogues.

In addition to parameterizing range searching problems by dimension, question, and range type, we can also consider point sets that can be modified over time. For example, if our points represent employees, then we may want our data structure to support insertions of new points in case of new hires and deletions of existing points in case of termination of employment.

One can strive for a few different goals in the study of range searching. One can:

- specify and solve a new range searching problem motivated by some practical or theoretical interest. A problem that is solved to some extent in practice may not yet be considered in the theoretical literature. Also, a solution to a new theoretical problem can create new directions for research.
- design a data structure that solves a previously studied range searching problem more efficiently. More efficient data structures are of course desirable. If the solution involves a more generally applicable new technique, it can be applied to give improved data structures for other problems as well.
- show that there is a barrier in efficiency that no data structure for a specific range searching problem can break. Efforts to design more efficient data structures can be in vain if there are lower bounds that prevent further progress. Proving lower bounds allows upper bound researchers to focus on solvable problems.

In general, these goals describe different ways of exploring the problem space of range searching. This dissertation includes results in all of these directions. We are interested in characterizing more precisely which range searching problems can be solved efficiently and which cannot. Some range searching problems that are computationally hard may have special cases that are computationally easy. It is important to refine our understanding of which problems are easy and which are hard so that when a

potential application of range searching arises, either in practice or in theory, we are better informed of the feasibility of an efficient solution.

Orthogonal range searching. In particular, we focus on *orthogonal* range searching: the case in which ranges are axis-aligned rectangles (or axis-aligned hyperrectangles in higher dimensions). Orthogonal range searching is highly applicable. Consider the SQL query described in Figure 1.1.

```
SELECT *
  FROM employee
 WHERE age >= 20
   AND age <= 30
   AND salary >= 100000
```

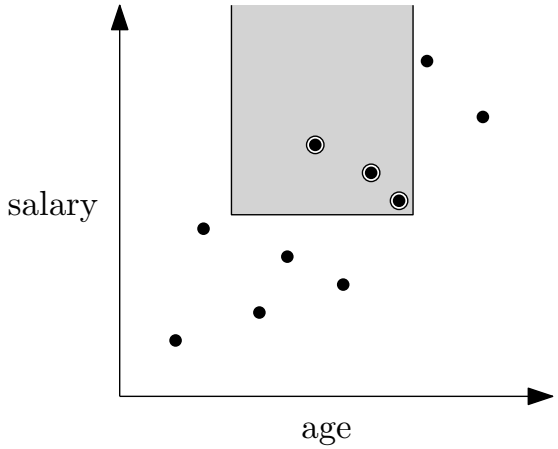


Figure 1.1: An SQL query and a geometric model of the query.

The query maps exactly to a two-dimensional orthogonal range searching query. We can model each employee as a point in a two-dimensional age-salary space. Then, the inequality filters in the SQL query specify an axis-aligned rectangular query in the age-salary space. In this case, since no upper bound on salary is given, the rectangle is unbounded in one direction. In general, any database query with inequality constraints on different columns maps to an orthogonal range searching query.

Orthogonal range searching stands out from other variants of range searching because it is possible in some ways to reason about each dimension independently. As a result, orthogonal range searching is often computationally easier than other variants of range searching. In particular, it is computationally easy enough that the model of computation in which we consider orthogonal range searching problems can have a significant impact. Studying orthogonal range searching requires both understanding of geometry as well as understanding of fundamental algorithms, data structures, and techniques applicable in various models of computation.

1.2 Variants of Orthogonal Range Searching

We consider three variants of orthogonal range searching. In this section, we briefly introduce and motivate these variants, leaving formal specifications of the problems to the next section.

Dynamic range reporting. In range *reporting*, the answer to a query is simply the list of points that lie in the query range. In dynamic range reporting, both insertions and deletions of points are allowed. Dynamic range reporting is one of the most fundamental variants of range searching with countless applications. We investigate data structures for two-dimensional dynamic orthogonal range reporting. We consider special cases in which some of the sides of the query ranges are fixed to infinity. It is often helpful to first consider these cases before moving on to the general case. These special cases are also of independent interest and often they can be solved with more efficient data structures. General two-dimensional queries are called *four-sided* queries. If one side is fixed to infinity, the query is *three-sided*. If two sides (one for each dimension) are fixed to infinity, the query is *two-sided*.

Colored range counting. It is often the case that objects not only have numerical attributes, but also categorical attributes. The values of a categorical attribute do not have any inherent ordering. For example, the brand of a product is a categorical attribute. The department in which an employee works is a categorical attribute. We represent a categorical attribute in our geometric model of a range searching problem by augmenting points with colors. Each color represents one of the possible values of a categorical attribute. In range *counting*, the answer to a query is the number of points that appear in the query range. In *colored* range counting, the answer to a query is the number of distinct colors that appear in the query range. Thus, colored range counting can be applied to compute a measure of the heterogeneity of a range with respect to some categorical attribute.

We investigate data structures for colored range counting in which the query ranges are restricted to be squares, or hypercubes in higher dimensions. This special case is motivated by practical applications in which a square is used to approximate the neighborhood of some position.

Concurrent range reporting. *Concurrent* range searching is another variant of range searching in which points are augmented with colors. Again, these colors can represent values of some categorical attribute. A concurrent range searching query consists not only of a query range, but also of a set of query colors. The answer to a

concurrent range searching query is a tuple of values. For each query color c, we include in the tuple the answer to the standard range searching query over the points of color c.

Thus, a naive solution to concurrent range searching is to simply build a data structure for standard range searching separately for the points of each color. A concurrent range searching query is then answered by querying the data structure for each query color separately. The goal of studying concurrent range searching is to find ways to reuse work between query colors to answer queries faster than this naive solution.

Concurrent range searching is prevalent in practice. Many applications have search interfaces in which users can specify inequality constraints on some numerical attributes while simultaneously filtering results to a certain subset of values of some categorical attribute. For example, many shopping websites allow customers to search for products in a certain price range and to filter their search results based on brands of interest.

1.3 Formal Problem Specifications

Range searching. Let V be a totally ordered set of *coordinate* values. These coordinate values are the primitives of all of our problems and vary between different models of computation. For example, V may be $\mathbb{R}$, or a finite set of integers, or a set of abstract values that can only be inspected via comparisons. For all of our problems, the input includes points from V^d for some constant positive integer d, which we call the *dimension*. Let $\mathcal{P} = V^d$ be the set of all possible input points. The input to all of our problems includes a set $P \subseteq \mathcal{P}$ of size n.

A *range* is a subset of V^d. A range searching problem is parameterized by a class $\mathfrak{R}$ of ranges and an aggregation function f with domain $2^{\mathcal{P}}$. We must preprocess P into a data structure such that we can answer a sequence of queries efficiently. A query consists of a range $R \in \mathfrak{R}$ and the answer to a query is $f(P \cap R)$.

Range types. An *interval* is a one-dimensional range of values of V. An interval $[a, b]$ for $a, b \in V$ such that $a \leq b$ is the set of values between a and b, inclusive. That is, $[a, b] = \{v \in V \mid a \leq v \leq b\}$. Intervals can be unbounded on one or both sides. We denote such intervals as follows: $[a, \infty) = \{v \in V \mid v \geq a\}$, $(-\infty, b] = \{v \in V \mid v \leq b\}$, and $(-\infty, \infty) = V$. *Orthogonal* ranges have the form $I_1 \times I_2 \times \cdots \times I_d$, where each I_i for $i \in [d]$[2] is an interval. Orthogonal range searching considers the special case of range searching in which $\mathfrak{R}$ is the set of all orthogonal ranges. A range is *s-sided* for $d \leq s \leq 2d$ if $s-d$ of its intervals are bounded on both sides and $2d-s$ of its intervals are unbounded on one side. For example, $[a_1, b_1] \times (-\infty, b_2]$ is a three-sided two-dimensional

[2]We denote by $[x]$ the set of integers from 1 to x, inclusive.

range. Let s-sided orthogonal range searching be the range searching problem in which $\mathfrak{R}$ is restricted to only s-sided orthogonal ranges that are all unbounded in the same dimensions and sides. Without loss of generality, we assume that the ranges are bounded in the first $s - d$ dimensions and unbounded in the last $2d - s$ dimensions. We further assume without loss of generality that the unbounded intervals are all of the form $(-\infty, b]$. *Rectangle* range searching is the case $d = 2$ and $s = 4$. *Dominance* range searching is the case $s = d$. In the case that V is a metric space, *hypercube* range searching is orthogonal range searching in which $s = 2d$ and, for each query, all intervals have the same length. *Square* range searching is hypercube range searching for $d = 2$ and *cube* range searching is hypercube range searching for $d = 3$. In orthogonal range searching, we typically assume for simplicity that, along every axis, all points of P have distinct coordinate values.

In the case that V^d is a metric space, *ball* range searching is the special case in which $\mathfrak{R}$ is the set of all balls of dimension d. In two-dimensions, it is called *disk* range searching. In the case that V^d is an affine space, *halfspace* range searching is a special case such that $\mathfrak{R}$ includes a range if it can be specified as a linear inequality in point coordinates.

Query types. Let X be a variable set. In range *reporting*, a range searching data structure computes its result with the identity function $f(X) = X$. In range *counting*, $f(X) = |X|$. In range *emptiness*, $f(X)$ is true if and only if $X = \emptyset$.

In some problems, our points P may be augmented with *colors*. Let a color be a value in $[t]$ for some maximum integer number of colors t. We model a coloring of the points of P with a color function $\chi : P \to [t]$. Thus, the color of a point $p \in P$ is $\chi(p)$. For a color $c \in [t]$, let X_c be all of the points of X of color c. That is, $X_c = \{p \in X \mid \chi(p) = c\}$.

We now define two augmented query types: *colored* range counting and *concurrent* range counting. Given a range searching problem for ranges $\mathfrak{R}'$ and aggregation function f', we define our augmented range searching problems. In an augmented range searching problem, the class of ranges is unchanged, so $\mathfrak{R} = \mathfrak{R}'$. In colored range searching, $f(X) = f'(\{\chi(x) \mid x \in X\})$. Thus, a query asks about the set of distinct colors occurring in a query range rather than about the points that lie in the query range. In concurrent range searching, every query specifies a subset of colors $Q \subseteq [k]$ in addition to a range R. The result of a concurrent query is computed with $f(X) = \{(c, f'(X_c)) \mid c \in Q\}$. Thus, a query asks for the query result with aggregation function f' for the separate points of each query color.

Dynamic point sets. In *dynamic* range searching, update operations can be interleaved in the sequence of queries. Updates are either insertions of points into P or

deletions of points from P. In *incremental* range searching, all update operations are insertions. In *decremental* range searching, all update operations are deletions. If no update operations are allowed, we call the problem *static*.

Online/offline operations. *Online range searching* is a data structure problem in which the sequence of operations is revealed in order, one operation at a time, after a preprocessing phase. *Offline* range searching is an algorithmic problem in which the entire sequence of operations is given as input and the output consists of the answers to all query operations in the sequence. One solution to offline range searching is to use an online range searching data structure, but it is possible to achieve speedups by exploiting that it is possible to inspect all operations from the beginning.

Nomenclature. Since a range searching problem can be parameterized in many ways, specifying a specific problem can be quite verbose. In Table 1.1, we define default values and an ordering for the parameters to simplify the specification of problems. Thus, we may write "three-sided colored range counting" rather than "online three-sided two-dimensional static orthogonal colored range counting." We note that context can override these default values.

Parameter	Values	Default value
Sequence	online, offline	online
Sidedness	s-sided ($s \in \mathbb{Z}$, $d \leq s \leq 2d$)	$2d$-sided
Dimension	d-dimensional ($d \in \mathbb{Z}$, $d \geq 1$)	two-dimensional
Dynamism	static, dynamic, incremental, decremental	static
Range type	orthogonal, rectangle, dominance, hypercube, square, cube, ball, disk, halfspace	orthogonal
Augmentation	colored, concurrent	—
Query type	reporting, counting, emptiness	—

Table 1.1: Range searching parameters

1.4 Models of Computation

Range searching is studied in various models of computation. In this dissertation, we work primarily with three models: the word RAM model, the external memory model, and the pointer machine model. In the following sections, we define the three models. For each model, we also briefly discuss the best known bounds for two fundamental one-dimension problems: *sorting* and *predecessor search*. Sorting is an algorithmic problem:

given a list of n totally ordered elements, permute the elements so that they are ordered from least to greatest. Predecessor search is a data structure problem: maintain a set of n totally ordered elements such that given a query element q, we can efficiently find the greatest element that is no greater than q.

1.4.1 Word RAM Model

The word RAM model was introduced by Fredman and Willard [FW90]. The word RAM model very closely models the capabilities of programs written in programming languages like C that execute on data stored in internal memory. In this model, all data is stored in internal memory, which is modeled as an array A of w-bit words. The base units of computation are these words which can store integers in $[2^w]$. Input elements are initially stored in words of A. We assume that it is possible to store in a word the index into A of each input element. Since there are n input elements, this means that we are implicitly assuming that $w \geq \log n$. When studying range searching, we assume $V = [U]$ where $U \leq 2^w$ so that each coordinate value fits in a word.

Let $A[i]$ denote the word at index i or, depending on context, the value of the word at index i. Given an arbitrary index i, we can access $A[i]$ in constant time. This is a "random access" of the random access memory (RAM) modeled by A. It is possible to perform various operations on words in constant time. For example, given indices i, j, and k, we can store in $A[k]$ the sum of $A[i]$ and $A[j]$ in constant time. Different variants of the word RAM model exist, depending on which operations are allowed. We allow a standard set of operations including all arithmetic operations and bitwise operations that are available in programming languages such as C. All of these operations belong to AC^0 except for multiplication.

In the word RAM model, sorting of values in $[U]$ can be solved in $O(n\sqrt{\log\log n})$ expected time [HT02] or $O(n \log\log n)$ deterministic time [Han04]. There are no lower bounds for sorting except for the trivial $\Omega(n)$ lower bound. Dynamic predecessor search can be solved with linear space and query time $O(\log\log U)$ via y-fast tries [Wil83], $\log_w(n)$ via fusion trees [FW90], or $\sqrt{\log n/\log\log n}$ via exponential search trees [AT07]. The only case in which we can do better in the static case is if we allow the data structure to use polynomial space. Then, static predecessor search can be solved with $O(\log\log U/\log\log\log U)$ query time [BF02]. All of these data structures are optimal for their respective parameters [PT07]. The y-fast trie uses a hashing technique that requires randomization. The best known alternative that does not use randomization requires $O(\log\log n \cdot \log\log U/\log\log\log U)$ time, again via exponential search tress [AT07]. It is open whether or not this bound can be improved.

1.4.2 External Memory Model

The external memory model was introduced by Aggarwal and Vitter [AV88]. In this model, there are two levels of memory, internal memory and external memory, which are both modeled as arrays of elements. Elements can be atomic input elements or integers. Input elements are initially stored in external memory. Both internal memory and external memory are divided into blocks of B consecutive elements. External memory is unbounded but internal memory can contain at most M elements. The input size is denoted N instead of n, by convention. We can transfer a block between internal memory and external memory at unit cost. Such a transfer is called an I/O. Since an I/O is, in the real world, many orders of magnitude slower than performing an operation in internal memory, the model allows arbitrary computation for free as long as it only operates on data currently stored in internal memory. The efficiency of a data structure includes the number of blocks occupied by the data structure and the number of I/Os performed by its operations. When studying range searching, we use the comparison external memory model in which the coordinate space V is simply a totally ordered set that we can inspect only via comparisons. Again, these comparisons are free as long as they are performed on coordinates stored in internal memory.

In the comparison external memory model, sorting can be solved in $O((N/B) \cdot \log_{M/B}(N/B))$ time, which is optimal [AV88]. Predecessor search can be solved in $O(N/B)$ space, $O(\log_B N)$ query time, and $O(\log_B N)$ update time via B-trees [BM72], which is optimal. If a dynamic predecessor search data structure need not immediately supply answers to queries but instead need only supply an answer eventually, the problem can be solved with $O(N/B)$ space and operations that run in $O((1/B)\log_{M/B}(N/B))$ amortized time via buffer trees [Arg03].

1.4.3 Pointer Machine Model

The pointer machine model was introduced by Tarjan [Tar79]. In this model, a data structure is a directed graph in which one vertex is designated as an entry point. Vertices can contain a constant amount of data, which can consist of atomic input elements and integers. Vertices have constant out-degree. To answer a query in which the query result is one or more of the input elements, the query algorithm must navigate the directed graph starting from the entry point by following directed edges to new vertices. The query algorithm must reach nodes containing the necessary output elements. The query algorithm can maintain a working set of a constant number of vertices of the graph. The algorithm's decisions must be made based on the data stored in the working set vertices. The cost of an algorithm is the number of directed edges that it traverses. The space consumed by a data structure is the number of vertices in the graph. When studying range searching, we use the comparison pointer machine model in which the coordinate

space V is simply a totally ordered set that we can inspect only via comparisons. Proving a data structure lower bound in the pointer machine model involves specifying an input such that, no matter the graph, there must exist a query whose output is far from the entry point or scattered around the graph.

In the comparison pointer machine model, sorting can be solved in $O(n \log n)$ time via, for example, merge sort, which is optimal. Predecessor search can be solved in linear space and $O(\log n)$ time per operation in both the static and dynamic settings via balanced binary trees. If the space of values are nodes of a linked list such that nodes earlier in the list are smaller and a query value is specified as a pointer to a linked list node, then predecessor search can be solved in $O(\log \log n)$ time [vEB77], which is optimal [MNA87].

1.5 Our Results in Perspective

In this section, we discuss background and the state-of-the-art for the variants of orthogonal range searching that we consider. After having set the stage, we introduce our new results. In Chapters 2-4, we describe our results in detail for dynamic range reporting, colored range counting, and concurrent range reporting, respectively. Discussion of problems that we leave open for future research is deferred to the concluding remarks of each respective chapter. We begin by summarizing the progress that has been made in the most basic fundamental variants: static orthogonal range reporting, counting, and emptiness. Many techniques that are applied in solving more exotic variants of orthogonal range searching were initially developed for these fundamental variants.

1.5.1 Background

Classical results. Orthogonal range searching has a long history with initial techniques being developed as early as the 1970s. One of the earliest data structures with interesting theoretical guarantees is the k-d tree of Bentley [Ben75]. It requires linear space and $O(n^{1-1/d})$ query time [LW77]. If we specify a query time of T_q but do not specify whether the bound applies for reporting, counting, or emptiness queries, then the bound for counting and emptiness is T_q while the bound for reporting is $T_q + O(k)$, where k is the number of points reported by the query.

Bentley [Ben80] also introduces the range tree, which is arguably the most important data structure for orthogonal range searching. It requires more space than the k-d tree, but queries run exponentially faster. In particular, the range tree requires $O(n \log^{d-1} n)$ space and $O(\log^d n)$ query time. The range tree can also be used as a tool to answer decomposable queries. A query is decomposable if, after partitioning the query range R

into subranges $R_1, \ldots, R_m$, we can compute the query result for R in $O(m)$ time given the query results for subranges $R_1, \ldots, R_m$. (Most of the types of queries that we consider are decomposable; colored range counting queries are a notable exception.) Using a range tree, we can decompose a d-dimensional query into $O(\log n)$ $(d-1)$-dimensional queries. Given a solution to the $(d-1)$-dimensional problem, range trees give a solution for the d-dimensional problem with an additional logarithmic factor in both space and query time. Using a range tree, we can also decompose an s-sided range for $s > d$ into two $(s-1)$-sided queries. Given a solution to the $(s-1)$-sided problem, range trees give a solution for the s-sided problem with an additional logarithmic factor in space and an additional constant factor in query time.

Willard [Wil85] improves the query time of the range tree by a logarithmic factor via downpointers. Downpointers are a special case of fractional cascading [CG86a]. The idea of both is to reduce the cost of repeated predecessor searches. In particular, in the original range tree solution, we preform $O(\log^{d-1} n)$ binary searches in different sets for a total cost of $O(\log^d n)$. With downpointers or fractional cascading, only the first predecessor search requires a full binary search. The rest of the predecessor searches can be solved in $O(1)$ time each by following shortcut pointers that have been added between the elements of different sets.

We now fix our attention to the two-dimensional case. McCreight [McC85] introduces the priority search tree, which solves three-sided range reporting and emptiness in linear space and $O(\log n)$ query time, thus saving a logarithmic factor in space in comparison to the range tree. Range counting is harder than range reporting and emptiness; the priority search tree can only count by reporting all of the points in the query range, which is inefficient in the worst case. In particular, two-, three-, and four-sided range counting are all asymptotically equivalent. This follows from the fact that we can easily compute the number of points in a four-sided query range using the counts for four two-sided ranges: the dominance ranges from each of the corners of the four-sided query range.

Word RAM improvements. The data structures discussed so far can all be implemented in the pointer machine model. Faster data structures are possible in the word RAM model due to the bounded precision of the input, the ability to succinctly represent data in few bits which can then be packed into few words, and the fact that it is possible to operate on $w \geq \log n$ bits in parallel in constant time. In the word RAM model it is beneficial to perform rank space reduction: we replace the x- and y-coordinates of points with their x- and y-ranks. At query time, we must then convert the coordinates of the corners of the query range into rank space, which can be achieved via predecessor search. Then, our space of coordinate values $[U]$ is shrunk to $[n]$, since the maximum rank of a point is n. Recall that U is the maximum coordinate value for

points in the word RAM model. The following bounds all consider range searching in rank space with $U = n$.

Chazelle [Cha88] introduces compressed range trees, which are more space efficient than classical range trees but, depending on the amount of compression, may require some super-constant amount of time to decompress each output point in the case of reporting queries. The bounds obtained are $O(n)$ space and $O(\log n)$ query time for range counting and any of the following bounds for range reporting (for range emptiness set $k = 0$, $\epsilon > 0$ is an arbitrarily small constant):

- $O(n)$ space and $O(\log n + k \log^{\epsilon} n)$ query time,
- $O(n \log\log n)$ space and $O(\log n + k \log\log n)$ query time, or
- $O(n \log^{\epsilon} n)$ space and $O(\log n + k)$ query time.

JaJa et al. [JMS05] reduce query time for range counting to $O(\log n / \log\log n)$, which is optimal [Păt07]. Further improvements for range reporting and emptiness are given by Nekrich [Nek09] and Alstrup et al. [ABR00]. These results are superseded by the work of Chan et al. [CLP11]. This latest work refines the compressed range tree to obtain the following bounds:

- $O(n)$ space and $O(\log^{\epsilon} n + k \log^{\epsilon} n)$ query time,
- $O(n \log\log n)$ space and $O(\log\log n + k \log\log n)$ query time, or
- $O(n \log^{\epsilon} n)$ space and $O(\log\log n + k)$ query time.

It is open whether an emptiness data structure with $O(n)$ space and $O(\log\log n)$ query time is possible.

1.5.2 Dynamic Range Reporting

Previous work. Many of the data structures mention in Section 1.5.1 can also support efficient updates via standard tree balancing and rebuilding techniques. This is true for all of the classical results that we mention, except for the improved range tree with downpointers. Mehlhorn and Näher [MN90] dynamize fractional cascading, which is stronger than downpointers, to obtain a range searching data structure with $O(n \log n)$ space and $O(\log n \log\log n + k)$ time for queries and updates. The $O(\log\log n)$ factors can be eliminated in the incremental or decremental case, but it is open whether or not they can be eliminated in the fully dynamic case.

Dynamizing many of the word RAM data structures is not as simple due to the use of rank space reduction; a single insertion or deletion can cause the ranks of $\Theta(n)$ points to change. Willard [Wil00] reduces the query and update times for the priority search

tree to $O(\log n/\log\log n + k)$ by using techniques from fusion trees [FW90]. Alstrup et al. [AHR98] give a lower bound in the cell probe model, which is stronger than the word RAM model, showing that the maximum of query time and update time must be $\Omega(\log n/\log\log n)$. The lower bound holds even for amortized running times. However, if amortization is allowed, the lower bound does not hold for the incremental and decremental cases. Mortensen [Mor06] shows that update time can in fact be decreased while queries still run in optimal $O(\log n/\log\log n + k)$ time. In particular, he reduces update time to $O(\log^{7/8+\epsilon} n)$. The word RAM model thus allows $\Omega(\text{polylog}\, n)$-factor improvements in update time.

Our results. We give efficient word RAM data structures for two- and three-sided dynamic range reporting, with smaller values for the exponent in the update time than the solution of Mortensen [Mor06]. These are the fastest known data structures for these problems. We also investigate incremental range emptiness, the special case in which only insertions are allowed (i.e., no deletions) and instead of reporting all points in a query range, we must only decide whether or not the query range contains any points. We show that, with amortization, this special case can be solved much more efficiently than the general case. In particular, we obtain data structures in which the maximum of query time and update time is $o(\log n/\log\log n)$.

Many word RAM algorithms and data structures operate by packing information about multiple input points into a single word to achieve speed ups via the word-level parallelism of the word RAM model. Such packed data structures bear many similarities to data structures designed for the external memory model. We explicitly design external memory data structures with the intention of simulating them to create packed word data structures. The descriptions of previous packed word data structures are much more complex.

1.5.3 Colored Range Counting

Previous work. Janardan and Lopez [JL93] introduce colored range searching in the context of one-dimensional range reporting. Gupta et al. [GJS95] explore many other variants of colored range searching. In particular they give a data structure for two-dimensional orthogonal colored range counting that requires $O(n^2 \log^2 n)$ space and $O(\log^2 n)$ time. Despite these bounds being extremely high compared to all other data structures that we have discussed so far, no improvements have been made. In contrast, there are data structures with $O(n\,\text{polylog}\, n)$ space and $O(\text{polylog}\, n)$ query time for both uncolored range counting (e.g. range trees) and colored range reporting (the state-of-the-art in this direction is described in [LvW13]). Kaplan et al. [KRSV08] generalize the result of Gupta et al. [GJS95] to higher dimensions. They solve the d-dimensional

problem with a data structure that requires $O(n^d \log^{2(d-1)} n)$ space and $O(\log^{2(d-1)} n)$ query time. For dominance ranges, they give a more efficient data structure that requires $O(n^{\lfloor d/2 \rfloor} \log^{d-1} n)$ space and $O(\log^{d-1} n)$ query time. For $d \geq 4$ and even, the bounds can be improved to $O(n^{\lfloor d/2 \rfloor} \log^{d-2} n)$ space and $O(\log^{d-2} n)$ query time.

It seems that rectangle colored range counting is computationally harder than many other rectangle range searching problems. There is some evidence for this possibility. First, no data structure with $O(n \operatorname{polylog} n)$ space and $O(\operatorname{polylog} n)$ query time is known. Second, Kaplan et al. [KRSV08] give a linear-time reduction from Boolean matrix multiplication to the offline variant of rectangle colored range counting. Whether Boolean matrix multiplication of $n \times n$ matrices can be solved in $O(n^2)$ time, or even $O(n^2 \operatorname{polylog} n)$ time, is a long-standing open problem. If we were to solve offline rectangle colored range counting in only $O(n \operatorname{polylog} n)$ time, then we would also solve Boolean matrix multiplication in $O(n^2 \operatorname{polylog} n)$ time, settling a long-standing open problem. The reduction can also be interpreted as a conditional lower bound: if Boolean matrix multiplication requires $\Omega(n^{2+\epsilon})$ time for some $\epsilon > 0$, then offline rectangle colored range counting requires $\Omega(n^{1+\epsilon})$ time for some $\epsilon > 0$.

Hypercube colored range counting. Motivated by a practical application in which only square ranges are of interest and by the fact that the conditional lower bound of Kaplan et al. [KRSV08] exploits the ability to construct ranges of various aspect ratios, we consider square colored range counting. In square colored range counting, all queries are squares rather than arbitrary rectangles. We show that square colored range counting reduces to uncolored range counting in a small constant number of dimensions. Thus there are data structures for the square case that require only $O(n \operatorname{polylog} n)$ space and $O(\operatorname{polylog} n)$ query time. Our technique involves a series of transformations of the problem: first to a dual problem, then to an uncolored dual problem, and finally to an uncolored primal problem.

Our data structure does not generalize well to three-dimensions. Motivated by this fact, we consider obtaining a conditional lower bound for offline cube colored range counting. We generalize the technique of Kaplan et al. [KRSV08] and reduce Boolean matrix multiplication to offline cube colored range counting. If all cube ranges are restricted to be of the same fixed size, we show that there is again an efficient data structure for the online case. There is a straightforward reduction from offline rectangle colored range counting to offline four-dimensional hypercube colored range counting. So, for every combination of dimension and whether or not our hypercube query ranges have a fixed size, we either have an efficient data structure for the online case or a conditional lower bound for the offline case.

Categorical richness. The practical application that motivated square colored range counting is called the *categorical richness* problem. It is an algorithmic problem in which the input is a raster of categorical data. We model this input as an axis-aligned uniform grid of colored points. Each square range of a certain fixed size centered on an input point is called a *window*. The output for the categorical richness problem is the number of distinct colors in each window. A scientist with a raster of categorical data can use a solution to categorical richness to, for example, visualize the variance of heterogeneity throughout the raster. Examples of rasters of categorical data for which such a visualization may be of interest include data on soil type or land usage of a terrain. In some cases, scientists may convert numerical data to categorical data by partitioning raster cells into categories based on which of a set of intervals their numerical values lie. This has been done in the case of elevation data for terrains.

Scientists currently use a naive algorithm to solve categorical richness: count the number of colors in each window independently. This naive algorithm requires $O(nr^2)$ time, where r is the fixed radius of a window. We give an algorithm that requires only $O(n)$ time, independent of r, and give an implementation that can process rasters of hundreds of millions of cells efficiently. We also give an algorithm and implementation for circular windows which, in our experiments, is slower by only small factors. It runs in $O((1 + t/r)n)$ time, where t is the number of distinct colors present in the input.

1.5.4 Concurrent Range Reporting

Previous work. Despite the prevalence of concurrent queries in practice in the form of categorical filters, there has been little theoretical research on concurrent range searching problems.

Chazelle and Guibas [CG86b] give a pointer machine data structure for concurrent predecessor search. In concurrent predecessor search, we must find the predecessor of the query value amongst the values of each query color. Chazelle and Guibas [CG86b] give a data structure that requires linear space and $O(\log n + r \log(t/r))$ query time, where t is the number of colors in the input and $r \leq t$ is the number of query colors. They also give a matching lower bound. However, the query representations for the upper and lower bounds differ in a subtle but meaningful way, so their work does not prove the optimality of their data structure. Note that one-dimensional concurrent range reporting easily reduces to concurrent predecessor search. Afshani et al. [AAL09, AAL10] consider concurrent range reporting in two and higher dimensions, again in the pointer machine model. In particular, for three-sided two-dimensional concurrent range reporting, they design a relatively inefficient data structure that happens to be sufficient for their purposes. Their data structure requires linear space and $O(\log n + t \cdot 2^t + k)$ query time, where k is the number of output points across all query colors.

Gupta et al. [GJS95] consider what they call *type-2* colored range counting which bears some resemblance to concurrent range searching. In type-2 colored range counting, for each color c that appears in the query range, the data structure must output the number of points of color c. Thus a type-2 colored range counting query is a concurrent range counting query in which the query colors are not specified explicitly by the query but are instead determined by the colors of the points that lie in the query range.

Our results. First, we consider the issue of query representation that was left open by the work of Chazelle and Guibas [CG86b] and give reductions and a lower bound showing that the two query representations they use are in fact equivalent. Thus, their data structure is indeed optimal.

Second, we give efficient data structures for three-sided two-dimensional concurrent range reporting. One of our data structures requires linear space and only $O(\log n + r \log(t/r)\alpha(n/t) + k)$ query time, where $\alpha(\cdot)$ is proportional to the inverse Ackermann function. The query time is only an $\alpha(n/t)$ factor away from our lower bound. Our data structure is based on a simple space-saving technique combined with an interesting recursive structure.

Chapter 2

Dynamic Range Reporting

We consider various special cases of two-dimensional dynamic orthogonal range searching, a fundamental problem in computational geometry. Range searching problems have many applications in, for example, databases and information retrieval, since it is often useful to model objects with d attributes as points in $\mathbb{R}^d$, where each dimension represents an attribute. In particular, performing an orthogonal range searching query then corresponds to filtering objects with inequality filters on attributes. The two-dimensional orthogonal range searching problem has been studied for over 30 years, but optimal bounds are still not known. We give the first improved bounds in the word RAM model since the work of Mortensen [Mor06] in 2006. In particular, we give data structures with faster updates, maintaining optimal time queries. In practice, fast updates are often as important as fast queries, since they allow the efficient maintenance of a larger number of specialized indices, which can then support a more diverse set of queries efficiently. We also give partially dynamic data structures with faster query times than can be achieved by fully dynamic data structures.

Problems. The *range reporting* problem involves maintaining a set P of n points from $\mathbb{R}^d$ so that given an online sequence of queries of the form $Q \subseteq \mathbb{R}^d$, we can efficiently compute $P \cap Q$. The output size, k, of a range reporting query Q is $|P \cap Q|$. The *range emptiness* problem requires only that the data structure decide whether or not $P \cap Q$ is empty. In *dynamic* range searching problems, insertions and deletions of points to and from P may be interspersed with the online sequence of queries. An *incremental* range searching problem is a dynamic range searching problem in which the set P is initially empty and there are no deletions (i.e., there are only insertions and queries). In *orthogonal* range searching problems, queries are restricted to the set of axis-aligned hypercubes. That is, a query must be of the form $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$. An *s-sided* query range for $d \leq s \leq 2d$, has finite boundaries in both directions along

$s - d$ axes and only one finite boundary along $2d - s$ axes. For example, the range $[\ell, r] \times (-\infty, t]$ is a three-sided two-dimensional query range.

In the *range minimum query (RMQ)* problem, we maintain an array A of n elements from $\mathbb{R}$ so that given an online sequence of queries of the form $(i, j) \in [n] \times [n]$ such that $i \leq j$, we can efficiently compute $\min_{i \leq k \leq j} A[k]$. That is, we compute the minimum element in a given query subarray. RMQ can be applied to solve the lowest common ancestor problem in trees and it is often used in solutions to textual pattern matching problems. In the dynamic variant of this problem, an update $(i, v) \in [n] \times \mathbb{R}$ sets $A[i]$ to v. In the *decremental* variant of the problem, the update may not increase $A[i]$. That is, if $A[i]$ contains u prior to the update, then it must be that $v \leq u$.

We consider two- and three-sided two-dimensional orthogonal range searching problems. In particular, we consider dynamic reporting as well as incremental emptiness. We also consider RMQ due to its close connection to three-sided emptiness.

Model. Our model of computation is the w-bit word RAM model. We assume that our points have integer coordinates that each fit in a single word, so $P \subseteq [U]^d$, where $w \geq \log U$. Similarly, the bounds of each query range are from $[U]$. We make the standard assumption that $w \geq \log n$ so that an index into our input array fits in a single word.

Previous results. In the pointer machine model, the best known bounds for dynamic orthogonal range reporting are $O(n \log n)$ space, and $O(\log n \log \log n)$ update time, and $O(\log n \log \log n + k)$ query time. These bounds are achieved by augmenting range trees [Ben80] with dynamic fractional cascading [MN90]. The $O(\log \log n)$ factors can be eliminated for the partially dynamic variants of the problem. The priority search tree [McC85] solves three-sided dynamic orthogonal range reporting optimally in the pointer machine model with only linear space, $O(\log n)$ update time, and $O(\log n + k)$ query time.

In the word RAM model, sublogarithmic time operations were first obtained for three-sided queries. Willard [Wil00] reduces both the update time and the query time of the priority search tree to $O(\log n / \log \log n)$. Let T_u be the time for an update operation and let T_q be the time for a query operation. Alstrup et al. [AHR98] give a cell probe lower bound for two-dimensional orthogonal range emptiness, showing that $T_q = \Omega(\log n / \log(T_u \log n))$. This lower bound implies a lower bound of $\Omega(\log n / \log \log n)$ on the time per operation (i.e., $\max\{T_u, T_q\}$). These lower bounds hold even when restricted to two-sided queries. Crucially, they hold for the amortized bounds of fully dynamic data structures as well as for the worst-case bounds of partially dynamic data structures, but not for the amortized bounds of partially dynamic data structures.

Despite mounting evidence that the true update and query times for two-dimensional orthogonal range reporting might be $\Theta(\log n/\log\log n)$, Mortensen [Mor06] showed that updates could in fact be made faster. For three-sided queries, his data structure supports updates in $O(\log^{5/6+\epsilon} n)$ time. He gives a worst-case deterministic fully dynamic data structure for four-sided queries that requires $O(n\log^{7/8+\epsilon} n)$ space, $O(\log^{7/8+\epsilon} n)$ update time, and optimal $O(\log n/\log\log n)$ query time. The speed up in update time is achieved by reducing the number of bits required to specify points, packing multiple points into a word, and using techniques analogous to external memory buffering on these packed words. Importantly, Mortensen shows that update time can be reduced convincingly (i.e., by an $\Omega(\text{polylog}\, n)$ factor) below $O(\log n)$ while maintaining optimal $O(\log n/\log\log n)$ query time. In the process, he leaves behind an interesting and fundamental open problem: how fast, then, can we support updates?

Our results. We make an initial effort to determine the complexity of updates by first considering the two- and three-sided special cases and allowing amortization and randomization. All of our update and query bounds are amortized. The only source of randomization in our data structures originates from our use of randomized dynamic predecessor search data structures [Wil83]. It is possible to eliminate all randomization from our data structures by instead using the deterministic predecessor search data structure of Andersson and Thorup [AT07], at the expense of doubly logarithmic factors. For simplicity of exposition we choose not to describe the resulting deterministic bounds.

First, we improve and simplify the techniques of Mortensen [Mor06] at the expense of amortization. Instead of re-inventing external memory techniques to handle the case where we can pack many points into a single word, we explicitly apply techniques already developed for the external memory model and, in fact, begin with the design of an external memory range searching data structure. Our first two results are data structures with improved update times for two- and three-sided reporting. In the following theorems, $\epsilon > 0$ is an arbitrarily small constant.

Theorem 2.1. *There exists a data structure for two-sided two-dimensional dynamic orthogonal range reporting that requires $O(n)$ space, $O(\log^{1/2+\epsilon} n)$ update time, and $O(\log n/\log\log n + k)$ query time. Alternatively, $O((\log n\log\log n)^{1/2})$ update time and $O(\log n + k)$ query time.*

Theorem 2.2. *There exists a data structure for three-sided two-dimensional dynamic orthogonal range reporting that requires $O(n)$ space, $O(\log^{2/3+\epsilon} n)$ update time, and $O(\log n/\log\log n + k)$ query time. Alternatively, $O((\log n\log\log n)^{2/3})$ update time and $O(\log n + k)$ query time.*

Note that $k = |P \cap Q|$ is the output size of a query. Each of our reporting data structures with a query time of the form $O(T(n) + k)$ can be easily adapted to decide

emptiness in $O(T(n))$ time. The bounds for emptiness of Theorem 2.2 also hold for dynamic RMQ.

Theorem 2.3. *There exists a data structure for dynamic RMQ that requires $O(n)$ space, $O(\log^{2/3+\epsilon} n)$ update time, and $O(\log n / \log\log n)$ query time. Alternatively, $O((\log n \log\log n)^{2/3})$ update time and $O(\log n)$ query time.*

Next, we circumvent the lower bound of Alstrup et al. [AHR98] by considering amortized solutions to partially dynamic problems. We first give a simple two-sided data structure.

Theorem 2.4. *There exists a data structure for two-sided two-dimensional incremental orthogonal range emptiness that requires $O(n)$ space, $O(\text{pred} + \log\log n)$ update time, and $O(\text{pred} + \log\log n)$ query time.*

Here, pred is the cost of an update or query to a linear-space dynamic predecessor search data structure containing at most n elements from a universe of size U. These updates and queries can be performed in $O(\log\log U)$ [Wil83] time, $O(\log_w n)$ time [FW90], or $O(\sqrt{\log n / \log\log n})$ time [AT07]. We finally give a three-sided incremental data structure and its decremental RMQ equivalent. These data structures achieve query times up to exponentially faster than can be achieved by their fully dynamic counterparts.

Theorem 2.5. *There exists a data structure for three-sided two-dimensional incremental orthogonal range emptiness that requires $O(n)$ space, $O(\text{pred} + (\log n \log\log n)^{2/3})$ update time, and $O(\text{pred} + (\log n \log\log n)^{1/3})$ query time. Alternatively, $O(\text{pred} + \log n / \log\log n)$ update time and $O(\text{pred} + \log\log n)$ query time.*

Theorem 2.6. *There exists a data structure for decremental RMQ that requires $O(n)$ space, $O((\log n \log\log n)^{2/3})$ update time, and $O((\log n \log\log n)^{1/3})$ query time. Alternatively, $O(\log n / \log\log n)$ update time and $O(\log\log n)$ query time.*

2.1 Preliminaries

Dynamic and static axes. An *axis* is a set from which points and query ranges obtain their coordinates along some dimension. Axes may also specify restrictions on how points are assigned coordinates. A *standard axis* is an axis with coordinates from $[U]$ that accommodates at most n points with distinct coordinates. Our geometric problems have two standard axes.

A standard technique in static orthogonal range searching is rank space reduction. Instead of using the full coordinates of points, we use their x- and y-ranks. Thus,

each point is specified by ranks from $[n]$ instead of coordinates from $[U]$. To handle a query, we must perform a predecessor search for each side of the query range in order to reduce the query range to rank space. In the dynamic case, rank space reduction as such does not work since inserting a new point might require updating the ranks of all of the other points. To encapsulate this problem, Mortensen [Mor06] introduces the concept of a *dynamic axis*. A dynamic axis has coordinates from the set of nodes of a linked list such that one node v is less than another node v' if v occurs prior to v' in the linked list. There is a bijection between points and linked list nodes. Initially, a new point has no node, so an update specifies its position along the axis with a pointer to its predecessor node. A new node is inserted to the linked list after the predecessor node. Query coordinates are specified by pointers to linked list nodes. A dynamic axis of size u can accommodate at most u points/nodes.

It is straightforward to reduce a standard axis to a dynamic axis via predecessor search. This is why many of our results include pred terms. In some cases, in choosing $\text{pred} = O(\sqrt{\log n / \log\log n})$, the pred terms are dominated by the other terms of the update/query times. In these cases, we have omitted the pred terms. Henceforth, we do not consider standard axes and consider instead dynamic axes.

In solving the problem encapsulated by dynamic axes, we will end up with some form of reduction of coordinates to a smaller set of integers (though not to the set of ranks as in rank space reduction). A *static axis* of size u has coordinates from $[u]$ and accommodates at most u points with distinct coordinates.

Three-sided emptiness and RMQ. For any set of points P, a three-sided range of the form $[\ell, r] \times (-\infty, t]$ is empty if and only if the lowest point of P in the slab $[\ell, r] \times (-\infty, +\infty)$ has y-coordinate less than or equal to t. Assume we have an array A of n elements from $[U]$. We construct a specific set P of points such that for each $i \in [n]$, there is a point $(i, A[i]) \in P$. Then, the result of an RMQ (i, j) on A is the y-coordinate of the lowest point of P in the slab $[i, j] \times (-\infty, +\infty)$. So, both types of queries can be solved by finding the lowest point in a query vertical slab. This is precisely how our emptiness data structures operate. Consider an update (i, v) which sets $A[i]$, initially containing u, to v. We update P by deleting (i, u) and inserting (i, v). So, an RMQ update is at most twice as expensive as a three-sided emptiness update. In the case of decremental RMQ, we are guaranteed that $v \leq u$. Assuming we have a three-sided incremental emptiness data structure that can handle multiple points with the same x-coordinate, it is sufficient to update P by only inserting (i, v) without first deleting (i, u). It is easy to modify our incremental emptiness data structures to handle multiple points with the same x-coordinate by, in this case, implicitly deleting (i, u). Thus, (decremental) RMQ can thus be solved by a three-sided (incremental) emptiness data structure with a static x-axis of size n and a standard y-axis. In order to reduce

the standard y-axis to a dynamic axis, we require predecessor search along the y-axis. However, this is only necessary for updates, since an RMQ is specified with only two x-coordinates and no y-coordinates. Henceforth, we do not directly consider RMQ and instead consider only our geometric problems.

Notation. We borrow and extend the notation of Mortensen [Mor06] to specify various different two-dimensional range searching problems. A dynamic problem is specified by $T^s(t_x : u_x, t_y : u_y)$ where $T \in \{\mathrm{R}, \mathrm{R_I}, \mathrm{E}, \mathrm{E_I}\}$; $d \leq s \leq 2d$; $t_x, t_y \in \{\mathrm{d}, \mathrm{s}\}$; and $1 \leq u_x, u_y \leq n$. Depending on T, the problem is either s-sided dynamic reporting (R), s-sided incremental reporting ($\mathrm{R_I}$), s-sided dynamic emptiness (E), or s-sided incremental emptiness ($\mathrm{E_I}$). For each axis $a \in \{x, y\}$, depending on t_a, the axis is either dynamic (d) or static (s). The axis has size u_a. We assume without loss of generality that two-sided queries are of the form $(-\infty, r] \times (-\infty, t]$ and 3-sided queries are of the form $[\ell, r] \times (-\infty, t]$.

We say a data structure has performance (Update : T_u, Query : T_q, Space : S) if it requires $O(T_u)$ update time, $O(T_q)$ query time (or $O(T_q + k)$ query time for reporting problems), and $O(S)$ words of space. We originally defined n as $|P|$, but we redefine it now to an upper bound on $|P|$ that we know in advance and for which we have the guarantee that $w \geq \log n$. We give the performances of data structures as functions of the sizes of their axes rather than of $|P|$. Our final data structures have axes of size n. We can eliminate the requirement of knowing n in advance and simultaneously ensure that performance is a function of $|P|$ rather than n by initially building a data structure with axes of constant size, rebuilding the data structure with axes twice as large whenever the data structure fills up, and rebuilding the data structure with axes half as large whenever the data structure is only a quarter full (except when the axes are already at their initial constant size).

When considering external memory data structures, we denote the input size by N and the output size by K, in keeping with the conventions of the external memory literature. A block can hold B elements and internal memory can hold M elements. In the external memory setting, we say a data structure has performance (Update : T_u, Query : T_q, Space : S) if it requires $O(T_u)$ I/Os for an update, $O(T_q)$ I/Os for a query (or $O(T_q + K/B)$ I/Os for a reporting query), and $O(S)$ blocks of space.

2.2 Data Structures

2.2.1 Dynamic Reporting

An important technique we will apply involves performing some form of reduction of the coordinates of small sets of $u \leq n$ points in such a way that we can describe each point using only $O(\log u)$ bits. Thus, since $w \geq \log n$, we can pack $O(\log n / \log u)$ of these points into a single word. Using standard word operations and table lookups, we can then operate on multiple points at unit cost. A similar situation arises in the external memory model: multiple points fit in a block and we can operate on all of the points in all of the blocks in internal memory at no cost. In fact, it is possible to simulate external memory algorithms to solve problems on packed words. For this reason, we begin with the design of an external memory reporting data structure, which we intend to simulate in the word RAM model.

Lemma 2.7. *For any $f \in [2, B]$, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{s} : N)$ *with performance* $(\mathrm{Update} : (f/B)\log_f N, \mathrm{Query} : \log_f N + K, \mathrm{Space} : N/B)$.

Proof. The data structure is a modified I/O tournament tree (I/O-TT) [KS96]. The I/O-TT is an I/O-efficient priority queue data structure, but we adapt it to answer three-sided range reporting queries. The I/O-TT stores elements, each of which consists of a key and a priority. We map a point to an element so that the point's x-coordinate is the element's key and the point's y-coordinate is the element's priority.

The I/O-TT is a static binary tree on x-coordinates where each node stores a set of at most B points and an update buffer of size B[1]. The sets of points are in heap order by y-coordinate and a non-root node may only contain points if its parent is full. Updates are initially sent to the root node. An update is inserted into a node's update buffer if it cannot be resolved in the node directly. When a node's update buffer is filled, the updates are flushed down to the node's children (by x-coordinate) in $O(1/B)$ amortized I/Os per update. The total cost of an update is thus $O((1/B)h)$ I/Os, where h is the height of the I/O-TT. A query to the I/O-TT involves finding the point with minimum y-coordinate. This point is in the root and can thus be found in $O(1)$ I/Os. The I/O-TT requires $O(N/B)$ blocks of space.

Our first modification is to use the priority search tree [McC85] query algorithm to handle three-sided reporting queries instead of priority queue queries. Assume we have a query range of the form $[\ell, r] \times (-\infty, t]$. Ignoring the update buffers, the I/O-TT is essentially a priority search tree with B points per node instead of only one. The

[1] In the original description of the I/O-TT these sets have size M instead of B, but such large sets are not necessary to achieve the desired bounds.

priority search tree query algorithm recurses down the tree as long as nodes correspond to x-intervals that overlap $[\ell, r]$ and the lowest point stored in a node is not above the line $y = t$. In this way, it visits $O(h + K)$ nodes, where K is the number of output points. For each node that it visits, it outputs all points stored in the node that lie in the query range. We can do so in $O(1)$ I/Os, since there are at most B points stored in each node. This query algorithm is only correct if the updates in the buffers of all of these nodes have been flushed. We flush these $O(h+K)$ buffers as the query algorithm proceeds. These flushing operations may cascade to descendant nodes, but all I/Os performed while cascading are charged to update operations. Therefore, there are only $O(h + K)$ additional I/Os that cannot be charged to update operations.

Our second modification is to increase the fanout of the I/O-TT to $f \in [2, B]$. To ensure that we can still flush updates from a node to its children efficiently, we must reduce the sizes of the sets of points and update buffers to B/f. In this way, we can store these sets for all of a node's children in $O(1)$ blocks and thus flush an update buffer in $O(f/B)$ amortized I/Os per update. Reducing the sizes of sets of points also allows the query algorithm to continue to be able to decide into which children to recurse in $O(1)$ I/Os. Since the height of the I/O-TT is now $O(\log_f N)$, we are done. □

We note that the data structure of Lemma 2.7 is not a typical external memory data structure because, for low values of f, the running time of our queries does not depend on B or M. Thus, our data structure would not be considered efficient by a researcher of external memory data structures. For example, for reporting problems, the goal is typically to report each output point in $O(1/B)$ I/Os rather than $O(1)$ I/Os. We will see that the trade-off we obtain with fast updates and slow queries will be useful in the design of word RAM data structures. Let $m \geq 2$ be the smallest constant such that any of our external memory data structures can operate with $M = mB$. Let $\delta > 0$ be a sufficiently small constant to be determined later.

Lemma 2.8. *For any $u \in [n^{\delta/2m}]$ and $f \in [2, (\delta/2m) \log n / \log u]$, there exists a data structure for $\mathrm{R}^3(\mathrm{s} : u, \mathrm{s} : u)$ with performance $(\mathrm{Update} : 1 + f \log^2 u / \log n$, $\mathrm{Query} : \log_f u, \mathrm{Space} : u)$.*

Proof. We simulate the data structure of Lemma 2.7 in the word RAM model. Since a point requires at most $2 \log N$ bits, if $N = u$, we can store $(\delta/2) \log n / \log u$ elements in $\delta \log n$ bits in a single word. We designate a single word to act as our simulated main memory containing up to $M = (\delta/2) \log n / \log u$ elements. The rest of our actual main memory acts as our simulated external memory: it is divided into blocks of $B = (\delta/2m) \log n / \log u$ elements such that $M = mB$. Since $u \leq n^{\delta/2m}$, each block can hold at least one element. A constant number of standard word operations can transfer a simulated block into or out of our simulated main memory.

The update and query algorithms of Lemma 2.7 use a constant number of different subroutines to manipulate the elements in main memory between I/Os. Since our simulated main memory can be described in exactly $\delta \log n$ bits, we use a global lookup table containing n^δ entries of $\delta \log n$ bits to support constant-time simulations of each of these subroutines. Since our final data structures reuse these lookup tables for many instances of our intermediate data structures (such as this one), we do not include the space for global lookup tables in the space bounds for our intermediate data structures. The lookup tables can be built in time polynomial in their number of entries. We set δ sufficiently small so that they can be built in $O(n)$ time. Whenever we rebuild our final data structures to handle constant factor changes in n, we can also afford to rebuild our global lookup tables.

Substituting $N = u$ and $B = (\delta/2m) \log n / \log u$ into Lemma 2.7 (and converting from space consumption in blocks to space consumption in words) gives the desired bounds. We also need to add a constant term to the running times of the update and query algorithms, since they may read from and write to simulated main memory a constant number of times without performing any simulated I/Os. In the external memory model, these reads and writes to main memory are free, but they are not free in our simulation. □

We now extract and slightly generalize two techniques of Mortensen [Mor06] and encapsulate them in the following two lemmata. We then see how we can directly apply these lemmata to the data structure of Lemma 2.8 to obtain an efficient data structure for $\mathrm{R}^2(\mathrm{d} : n, \mathrm{d} : n)$. The first technique involves converting a data structure with a static axis to a data structure with a dynamic axis. It can also be used as a space reduction technique.

Lemma 2.9 (Lemma 24 of Mortensen [Mor06])**.** *Given a data structure for* $T^s(t_x : u, t_y : u)$ *for* $u \in [n]$ *with performance* $(\text{Update} : T_u, \text{Query} : T_q, \text{Space} : u \operatorname{polylog} u)$ *where* $t_a = \mathrm{s}$*, for some* $a \in \{x, y\}$*, and queries have only one finite boundary along the other axis, then there exists a data structure for the same problem with* $t_a = \mathrm{d}$ *and performance* $(\text{Update} : \log\log u + T_u, \text{Query} : \log\log u + T_q, \text{Space} : u)$.

The second technique involves converting a data structure that can only handle $u \leq n$ points to a data structure that handles n points. This is achieved using a u-ary priority search tree.

Lemma 2.10 (Lemma 23 of Mortensen [Mor06])**.** *Given a data structure for* $T^s(\mathrm{s} : u, \mathrm{d} : u)$ *for* $u \in [n]$ *with performance* $(\text{Update} : T_u, \text{Query} : T_q, \text{Space} : u)$ *where queries have only one finite boundary along the y-axis, then there exists a data structure for* $T^s(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : (\log n / \log u)(\log\log n + T_u), \text{Query} : (\log n / \log u)(\log\log n + T_q), \text{Space} : n)$.

We note that $\mathrm{R}^2(\mathrm{s}:u,\mathrm{s}:u)$ is a special case of $\mathrm{R}^3(\mathrm{s}:u,\mathrm{s}:u)$, so the data structure of Lemma 2.8 also solves $\mathrm{R}^2(\mathrm{s}:u,\mathrm{s}:u)$. Since two-sided queries have only one finite boundary along the x-axis, we obtain the following result by applying Lemma 2.9 to the data structure of Lemma 2.8.

Lemma 2.11. *For any $u \in [n^{\delta/2m}]$ and $f \in [2, (\delta/2m)\log n/\log u]$, there exists a data structure for $\mathrm{R}^2(\mathrm{s}:u,\mathrm{d}:u)$ with performance* $(\text{Update}:\log\log u + f\log^2 u/\log n, \text{Query}:\log\log u + \log_f u, \text{Space}:u)$.

Next, we apply Lemma 2.10 to the data structure of Lemma 2.11, setting $u = 2^{(\delta/2m)((1/f)\log n\log\log n)^{1/2}}$.

Lemma 2.12. *For any $f \in [2, \log n/\log\log n]$, there exists a data structure for $\mathrm{R}^2(\mathrm{s}:n,\mathrm{d}:n)$ with performance* $(\text{Update}:(f\log n\log\log n)^{1/2}, \text{Query}:(f\log n\log\log n)^{1/2} + \log_f n, \text{Space}:n)$.

Since two-sided queries have only one finite boundary along the y-axis, we can once again apply Lemma 2.9 to the data structure of Lemma 2.12.

Theorem 2.13. *For any $f \in [2, \log n/\log\log n]$, there exists a data structure for $\mathrm{R}^2(\mathrm{d}:n,\mathrm{d}:n)$ with performance* $(\text{Update}:(f\log n\log\log n)^{1/2}, \text{Query}:\log_f n + (f\log n\log\log n)^{1/2}, \text{Space}:n)$.

We obtain the data structures of Theorem 2.1 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 2\epsilon$, or alternatively setting $f = 2$, in Theorem 2.13. All of the techniques we have seen so far carry through for both two- and three-sided queries, except for Lemma 2.11. Therefore, in order to obtain results for three-sided queries we need only some way to support a dynamic y-axis for three-sided queries.

We proceed by designing an external memory data structure for online list labeling which we will use to augment our original external memory data structure with a dynamic y-axis. In the online list labeling problem, we maintain an assignment of labels from some universe of totally ordered labels to linked list nodes so that the labels are monotonically increasing along the list. Assume the linked list has at most n nodes. For a universe of size $O(n)$, an insertion or deletion requires that $\Theta(\log^2 n)$ worst-case nodes are relabeled [BCD+02]. However, for a universe of size $O(n^2)$, an insertion or deletion can be limited to relabeling only $O(\log n)$ amortized nodes (this is a folklore modification of [IKR81]). In the external memory setting, we consider a linked list to be a linked list of blocks, where each block contains an ordered array of elements with unique ids. A pointer to a specific element is then its id along with a pointer to its containing block. An insertion is specified by the element to be inserted and a pointer to its intended predecessor. A deletion is specified by a pointer to the element to be deleted. A relabeling consists of a triple (i, ℓ, ℓ'), where i is the id of the element being relabeled, ℓ is its old label, and ℓ' is its new label.

Lemma 2.14. *There exists an external memory data structure for online list labeling of up to N elements with labels from a universe of size $O(N^2)$ that, upon each update, reports $O(\log N)$ amortized relabellings in $O(1 + (1/B)\log N)$ amortized I/Os.*

Proof. Upon an insertion or deletion, we load the target block in a single I/O and add or remove the element. We maintain block sizes of $\Theta(B)$ elements (except when there are too few elements to fill a single block) by splitting and merging adjacent blocks whenever they become a constant factor too large or too small. Each split or merge can be charged to $\Omega(B)$ updates. Each split or merge causes an insertion to or a deletion from the linked list of blocks. We maintain an online list labeling data structure for a polynomially large universe [IKR81] to assign the $O(N/B)$ blocks labels from a universe of size $O((N/B)^2)$ with $O(\log(N/B))$ amortized relabellings per update. If an element has rank r in the block with label ℓ, then its label is $B\ell + r - 1$, which is bounded by $O(N^2)$. For each block relabeled, we can report all of the relabellings for all of its elements in $O(1)$ I/Os. □

Lemma 2.15. *For any $f \in [2, B]$, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$ *with performance* $(\mathrm{Update} : 1 + (f/B)(\log_f N)\log N, \mathrm{Query} : \log_f N + K, \mathrm{Space} : N/B)$.

Proof. We maintain the online list labeling data structure of Lemma 2.14 on the y-axis list of our data structure of type $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$, using x-coordinates as the unique ids. We build the data structure D of Lemma 2.7 on an asymmetric $N \times N'$ grid, where $N' = O(N^2)$, instead of an $N \times N$ grid. The bounds of the data structure increase by only constant factors, but we obtain the requirement that points must be a constant factor larger to store the larger y-coordinates. Queries and updates to our data structure include y-axis pointers, which we convert to y-coordinates in $[N']$ in constant I/Os using the online list labeling data structure. We then forward the operations on to D, including the given x-coordinates and our computed y-coordinates. Upon an update, the online list labeling data structure reports $O(\log N)$ relabellings in $O(1 + (1/B)\log N)$ I/Os. In a scan through the relabellings, we convert each relabeling of the form (i, ℓ, ℓ') to a deletion of (i, ℓ) from D and an insertion of (i, ℓ') to D. So, our update time increases by an $O(\log N)$ factor. □

Lemma 2.16. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m))\log n/\log u]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$ *with performance* $(\mathrm{Update} : 1 + f\log^3 u/\log n, \mathrm{Query} : \log_f u, \mathrm{Space} : u)$.

Proof. We simulate the data structure of Lemma 2.15 as in Lemma 2.8, except that elements now require a constant factor more bits due to the polynomially large universe used by the online list labeling data structure. In internal memory, the y-axis list is a

standard linked list. To support conversions from pointers to y-axis list nodes to simulated external memory pointers, we store in each y-axis list node the external memory element's unique id (x-coordinate) and a pointer to the simulated block containing the element. Whenever there is a split or a merge, we must update $O(B)$ pointers from the y-axis list to simulated blocks. These updates can be charged to the $\Omega(B)$ updates that are required to cause a split or a merge. When a point is reported by the simulated data structure, its x-coordinate is included. To obtain the associated y-axis list node, we simply maintain (in $O(1)$ time per update) an array of size u containing, for each x-coordinate, the associated y-axis list node. □

Applications of Lemmata 2.10 and 2.9 to the data structure of Lemma 2.16 yield the following theorem by setting $u = 2^{(\delta/O(m))((1/f)\log n \log\log n)^{1/3}}$.

Theorem 2.17. *For any $f \in [2, \log n/\log\log n]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : f^{1/3}(\log n \log\log n)^{2/3}, \text{Query} : \log_f n + f^{1/3}(\log n \log\log n)^{2/3}, \text{Space} : n)$.

We obtain the data structures of Theorems 2.2 and 2.3 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 3\epsilon$, or alternatively setting $f = 2$, in Theorem 2.17.

2.2.2 Incremental Emptiness

We will require a couple of predecessor search data structures for linked lists. The first is a data structure of Mortensen [Mor06] that solves a problem called *colored predecessor search* in a linked list L with colored nodes. The data structure supports the following operations:

- insert(u, v, c): inserts node u with color c after node v
- delete(u): deletes node u
- change(u, c): changes the color of u to c
- predecessor(u, c): returns the last node of color c that is not after u

Lemma 2.18 (Theorem 15 of Mortensen [Mor06])**.** *There exists a linear-space data structure for colored predecessor search in a linked list L that supports all operations in $O(\log\log|L|)$ time.*

We also consider a problem called *subsequence predecessor search* in which we maintain a primary linked list L and a set of secondary linked lists $\mathcal{S}$. For the purposes of this problem, let $n = |L| + \sum_{S\in\mathcal{S}} |S|$ be the total size of all lists. Each node u in a

secondary list $S \in \mathcal{S}$ is associated with a primary node $p(u)$, such that mapping S with function p yields a subsequence of L. For any primary node v, there may be multiple nodes u from different secondary lists for which $p(u) = v$. We require the following operations:

- $\text{insert}_\text{p}(u, v)$: inserts primary node u after v in L
- $\text{delete}_\text{p}(u)$: deletes primary node u from L (only if there is no secondary node v with $p(v) = u$)
- $\text{insert}_\text{s}(u, S)$: inserts a new secondary node v with $p(v) = u$ to secondary list S (preserving the subsequence ordering of S)
- $\text{delete}_\text{s}(u)$: deletes secondary node u from its secondary list
- $\text{predecessor}(u, S)$: returns the last secondary node v in secondary list S such that $p(v)$ is not after primary node u in L
- $\text{secondaries}(u)$: returns all secondary nodes v such that $p(v) = u$ for primary node u
- $\text{primary}(u)$: returns $p(u)$ for secondary node u

Lemma 2.19. *There exists a data structure for subsequence predecessor search that requires $O(n)$ space, $O(\log \log n)$ time for updates and predecessor queries, $O(1 + k)$ time to report the k secondary nodes associated with a primary node, and $O(1)$ time to find the primary node associated with a secondary node.*

Proof. We construct an aggregate doubly linked list A containing, for each primary node u in order, pointers to all secondary nodes v such that $p(v) = u$ followed by a pointer to u. We also store pointers from primary and secondary nodes to their corresponding aggregate nodes. We color each node of A with the list of the node to which it stores a pointer. For each aggregate node pointing to a secondary node u, we also store an extra pointer to $p(u)$ which does not affect the aggregate node's color. We build the colored predecessor search data structure of Lemma 2.18 on A, which has size n. We can then support a subsequence predecessor query in secondary list S with a colored predecessor query in A with color S. Reporting the secondary nodes of a primary node requires a walk in A. Reporting $p(u)$ for secondary node u requires following two pointers. It is a simple exercise to verify that all of the update operations can be supported using a constant number of colored predecessor search operations and a constant amount of pointer rewiring. □

We are now ready to give data structures for incremental emptiness. We begin with a simple data structure for two-sided queries.

Lemma 2.20. *There exists a data structure for* $\mathrm{E}_{\mathrm{I}}^2(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : \log\log n, \text{Query} : \log\log n, \text{Space} : n)$.

Proof. Without loss of generality, we handle two-sided queries of the form $(-\infty, r] \times (-\infty, t]$. It is sufficient to find the lowest point in the query range and compare its y-coordinate to t. Since y-coordinates are linked list nodes, we require the list order data structure of Dietz and Sleator [DS87] to compare them. The lowest point with x-coordinate at most r is the minimal point whose x-coordinate is the predecessor of r. A point $p = (p_x, p_y) \in P$ is minimal if and only if there does not exist another point $(p'_x, p'_y) \in P \setminus \{p\}$ such that $p'_x < p_x$ and $p'_y < p_y$. We maintain the colored predecessor search data structure of Lemma 2.18 on the x-axis list. We color nodes associated with minimal points one color and all other nodes another color. We can thus find the minimal point whose x-coordinate is the predecessor of r in $O(\log\log n)$ time. A newly inserted point may be a minimal point, which can result in many, say k, previously minimal points becoming non-minimal. For each of these k points we must execute a color change operation, which requires a total of $O(k \log\log n)$ time. However, in the incremental setting, a point p can only transition from minimal to non-minimal at most once, so we can charge the $O(\log\log n)$ cost of its color change to the insertion of p. □

The data structure of Lemma 2.20 proves Theorem 2.4. Given that two-sided incremental emptiness queries can be supported very efficiently, we can use an alternative to Lemma 2.10 to handled n points given a data structure for only $u \leq n$ points. In particular, we can use a u-ary range tree instead of a u-ary priority search tree. The range tree requires superlinear space; however, we can reduce space to linear once again with an application of Lemma 2.9.

Lemma 2.21. *For any* $u \in [n^{\delta/O(m)}]$ *and* $f \in [2, (\delta/O(m)) \log n / \log u]$, *there exists a data structure for* $\mathrm{E}_{\mathrm{I}}^3(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : (\log n / \log u)(\log\log n + f \log^3 u / \log n), \text{Query} : \log\log n + \log_f u, \text{Space} : n \log n / \log u)$

Proof. We build a static u-ary range tree on x-coordinates. In each internal node of the range tree, we build two auxiliary data structures on the points stored in the node. Each of these data structures has its own y-axis list corresponding to a subsequence of the original y-axis list. The total size of all y-axis lists is $O(nh)$, where h is the height of the range tree. We build the subsequence predecessor search data structure of Lemma 2.19 using the original y-axis list as the primary list and all other y-axis lists as secondary lists. This data structure requires $O(nh)$ space. Now, given a query or update, we can translate it in $O(\log\log(nh)) = O(\log\log n)$ time into the coordinate space of a specific auxiliary data structure. Also, we can convert a point in the coordinate space of a specific auxiliary data structure to our original coordinate space in $O(1)$ time.

One of our auxiliary data structures is that of Lemma 2.20, which handles two-sided ranges of the form $(-\infty, r] \times (-\infty, t]$ and $[\ell, \infty) \times (-\infty, t]$. The other is a data structure which handles three-sided ranges that are aligned to the boundaries of the x-ranges of the node's children. We call these *aligned* queries. This data structure for aligned queries is the data structure of Lemma 2.16 built on the lowest points in each of the node's children. An aligned query is empty if and only if it contains none of the lowest points in each of the node's children. Since a node has u children, the axes of our data structure for aligned queries have size u. All of these auxiliary data structures require a total of $O(nh)$ space.

Given an insertion of a point p, we insert p into the two-sided data structures of all $O(h)$ nodes of the range tree to which p belongs at a cost of $O(h \log\log n)$. If p becomes the lowest point in a child of some node u, we must delete the old lowest point from the data structure of Lemma 2.16 in u and insert p. This introduces another term of $O(hf \log^3 u / \log n)$ to our update time.

Given a query of the form $[\ell, r] \times (-\infty, t]$, we find the lowest common ancestor (LCA) in the range tree of the leaf corresponding to x-coordinate ℓ and the leaf corresponding to x-coordinate r. Using a linear-space data structure, this LCA operation requires only $O(1)$ time [HT84]. In the resulting node u, we can decompose the query into an aligned query and a pair of two-sided queries in children of u. Our range is empty if and only if all three of these subranges are empty. The three emptiness queries to the auxiliary data structures require $O(\log\log n + \log_f u)$ time. Since the height of the range tree is $O(\log_u n) = O(\log n / \log u)$, we are done. □

An application of Lemma 2.9 to the data structure of Lemma 2.21 yields the following theorem.

Theorem 2.22. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m)) \log n / \log u]$, there exists a data structure for $\mathrm{E}^3_{\mathrm{I}}(\mathrm{d} : n, \mathrm{d} : n)$ with performance* $(\text{Update} : (\log n / \log u)(\log\log n + f \log^3 u / \log n), \text{Query} : \log\log n + \log_f u, \text{Space} : n)$.

We obtain the data structures of Theorems 2.5 and 2.6 by setting $u = 2^{(\log n \log\log n)^{1/3}}$ and $f = 2$, or alternatively setting $u = 2^{(\log\log n)^2}$ and $f = \log^{\epsilon'} n$ for a sufficiently small $\epsilon' > 0$, in Theorem 2.22.

2.3 Concluding Remarks

Let $\widetilde{O}$-notation hide $O(\log\log n)$ factors. We have given dynamic data structures for orthogonal range reporting with update times of the form $\widetilde{O}(\log^c n)$ for positive constants

$c < 1$ and $\widetilde{O}(\log n)$ query time. We have also given incremental data structures for orthogonal range emptiness with time per operation of the form $\widetilde{O}(\log^c n)$ for positive constants $c < 1$.

In the two-sided dynamic reporting case, we achieve $c = 1/2$ and we suspect that this is as far as current techniques can take us. For the two-sided incremental emptiness case, we described a simple optimal data structure that achieves $c = 0$. In the three-sided case, for both dynamic reporting and incremental emptiness, we achieve $c = 2/3$. It is open whether c can be reduced to $1/2$ for this case. To achieve $c = 1/2$, it is sufficient to give a data structure for $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$ with performance (Update : $1 + \log^2 u / \log n$, Query : $\log u$, Space : u). In particular, we need to avoid relying on online list labeling to obtain the dynamic y-axis. There are relatively few techniques for proving lower bounds for word RAM data structures in general and, in particular, there is very little known about proving lower bounds on update time.

Curiously, our approach to obtain an efficient incremental emptiness data structure does not seem to be adaptable to solving the decremental emptiness problem. In the incremental emptiness case, our two-sided data structure is efficient because it only needs to maintain the minimal layer of points. In the decremental case, it is not sufficient to maintain only the minimal layer, because if one of the minimal points is deleted, we may need to replace it with points from the second layer of minima. Whether decremental emptiness can be solved as efficiently as incremental emptiness is open. It would be interesting to explore whether or not there are other range searching problems in which the incremental or decremental variant can be solved more efficiently than the fully dynamic variant.

We have focused on three-sided queries but, of course, four-sided queries are interesting. An effort to efficiently handle four-sided queries should begin with an attempt to simplify the work of Mortensen [Mor06] on four-sided queries, just as our attempt to simplify his work on three-sided queries resulted in the progress outlined in this chapter.

Chapter 3

Colored Range Counting

Background and motivation. Terrain data and other geographic information is often stored in the form of a raster data set. The region of interest is partitioned into a *raster* (a grid of square cells) and for each cell in the raster the data pertaining to the corresponding location is stored. Sometimes the data is numerical; in a digital elevation model (DEM), for instance, each cell stores an elevation value. In other applications the data is *categorical*. For a terrain, for example, one may store information about land usage or soil type. Note that numerical data is sometimes interpreted as categorical by considering different ranges of values as different categories (i.e., low elevation, medium elevation, high elevation). Rasters storing categorical data are not only used frequently in geographic information systems (GIS), but also in various other fields.

In this chapter, we are concerned with the following computational problem. Let $\mathcal{G}$ be a raster of n cells storing categorical data and let r be an integer. Without loss of generality, we assume the cells in $\mathcal{G}$ have unit size. We now wish to compute for each cell c in $\mathcal{G}$ its *categorical richness*, that is, the number of different category values appearing in its neighborhood. We call this the *categorical richness* problem. It comes in two flavors, depending on the shape of the window defining the neighborhood of a cell c: in some applications the window is a square of size $(2r+1) \times (2r+1)$ centered at c, in other applications it is a disk of radius r. We refer to the first variant as *square richness* and to the second variant as *disk richness*.

The categorical richness problem appears in many scientific case studies, under different names. In GIS applications, this problem is usually referred to as computing the *patch richness* for every cell in a raster [HR09, SMH08]. Standard GIS software such as GRASS [NBLM12] and FRAGSTATS [MCE12] provide this functionality, but their implementations are inefficient and cannot handle large data sets.

In ecology, de Albuquerque et al. [dABB+15] encounter the square richness problem when they want to compute the variability of elevation data on a raster terrain. First,

they convert the elevation values of the raster cells into elevation categories, and then they compute for each cell c the number of elevation categories that appear within a square window centered at c. They call this the *topographic heterogeneity* of the cell. The concept also appears in other ecological studies [BABO+14, BCA13].

Related work. A straightforward way to solve the categorical richness problem is to explicitly scan for each cell c in the raster $\mathcal{G}$ its (square of circular) window and record the different category values encountered. Since a window contains $\Theta(r^2)$ cells, this algorithm runs in $O(nr^2)$ time. This is infeasible for large data sets, even for moderate values of r. A more refined algorithm would use that windows of neighboring cells differ in only $\Theta(r)$ cells; this can be exploited to obtain an algorithm with $O(nr)$ time. However, also this approach can still become quite slow for large data sets.

An alternative approach, which avoids the dependency on r, is to use data structures for *colored range counting* [GJS05] from computational geometry. These data structures store a set of n colored points—the colors represent the different categories—such that the number of colors within a query region can be counted efficiently. After preprocessing the set of cell centers for such queries one can solve the categorical richness problem by performing a query for each cell c with the window centered at c. Unfortunately, the known data structures for colored range counting are not very efficient: the best known structure with $O(\text{polylog}\, n)$ query time uses $O(n^2 \log^2 n)$ storage [GJS95], which is clearly infeasible in our application. More efficient solutions exist for reporting (rather than counting) all colors in the range: one can obtain $O(\log n + t)$ query time (where t is the number of reported colors) using $O(n \log n)$ storage [SJ05], but this is still too slow for large data sets. For circular ranges, the results are even worse. Note that our application is *offline*, that is, all queries are known in advance. Kaplan et al. [KRSV08] give an algorithm for the offline problem with n rectangular queries, which runs in $O(n^{1.408})$ time—again too slow for large data sets. They also showed that an $o(n^{1.186})$ solution would imply an improvement of the best running time for the well known Boolean matrix multiplication problem. For the offline version with circular queries, no results are known.

We conclude that general results on (online or offline) colored range searching do not solve the categorical richness problem fast enough. The question is thus: can we exploit the special structure of our problem, where the points form a grid and the ranges are centered at these grid points, to obtain a fast solution for the categorical richness problem?

Our results. We answer the above question affirmatively in Section 3.1, both for square richness and for disk richness. Our algorithm for square richness runs in $O(n)$ time, independent of the parameter r specifying the window size. Our algorithm for disk

richness runs in $O((1 + t/r)n)$ time, where t is the total number of different categories in the data set; in practice, we typically have $t \leq r$, in which case the algorithm runs in $O(n)$ time. Our algorithms are not only efficient in theory, but also in practice: in Section 3.3 we present an experimental evaluation showing that they can handle raster data sets of hundreds of millions of cells.

Our second set of results, presented in Section 3.2, concerns colored range counting for general point sets. We show how to preprocess a set P of n points in the plane into a data structure of size $O(n \operatorname{polylog} n)$ such that a color counting query with a query square (of arbitrary size) can be answered in $O(\operatorname{polylog} n)$ time—much better than the best known solution for rectangular queries. In particular, this is done via a reduction to five-dimensional dominance range counting. We also present a similarly efficient data structure for the three-dimensional version of the problem where the query ranges are fixed-size cubes. Finally, we investigate the hardness of the offline problem for points in $\mathbb{R}^3$ and ranges that are variable-sized cubes by relating its computational complexity to that of Boolean matrix multiplication.

3.1 Categorical Richness

3.1.1 Notation and Terminology

The grid. We begin by restating the categorical richness problem using notation from range searching so that all of our results are presented with a uniform notation. Instead of a raster $\mathcal{G}$ of categorical data, our input is an axis-aligned uniform grid P of n colored points. We assume for simplicity that our grid consists of $\sqrt{n}$ rows and $\sqrt{n}$ columns, although our approach also works for non-square grids. Thus, we denote the grid by $P[1..\sqrt{n}, 1..\sqrt{n}]$. We assume without loss of generality that point $P[i, j]$ has coordinates (i, j) and thus adjacent points of a row or column are at unit distance.

Each point $p \in P$ has a color. We identify the colors by positive integers, and let $\mathcal{C} = [t]$ denote the set of all colors that appear in P. We use $\chi(p)$ to denote the color of a point p.

Square richness. Let $r >= 1$ be a real number and let p be a point of P. We define $W_{\text{sq}}(p)$, the *square window* of p, to be the square with side lengths $2r$ centered at p. We consider the square $W_{\text{sq}}(p)$ to be closed, so p' is in $W_{\text{sq}}(p)$ even if p' lies on the boundary of $W_{\text{sq}}(p)$. We define $\mathcal{C}_{\text{sq}}(p) \subseteq \mathcal{C}$ to be the set of all colors that occur in the window $W_{\text{sq}}(p)$, that is, $\mathcal{C}_{\text{sq}}(p)$ is the set of all $c \in \mathcal{C}$ such that there is a point $p' \in W_{\text{sq}}(p)$ with $\chi(p') = c$. Finally, we define $\text{rich}_{\text{sq}}(p)$, the *square richness* of a point p, as the number

of distinct colors in $W_{\text{sq}}(p)$. In other words, $\text{rich}_{\text{sq}}(p) = |\mathcal{C}_{\text{sq}}(p)|$. The square richness problem is now to compute $\text{rich}_{\text{sq}}(p)$ for all points $p \in P$.

Disk richness. The disk richness problem is defined in a similar way. We now use a window $W_{\text{d}}(p)$, which is a disk of radius r centered at p, define $\mathcal{C}_{\text{d}}(p)$ to be the set of colors occurring in $W_{\text{d}}(p)$, and define $\text{rich}_{\text{d}}(p) = |\mathcal{C}_{\text{d}}(p)|$ to be the disk richness of p.

3.1.2 Square Categorical Richness

Next we describe an algorithm that solves the square richness problem in $O(n)$ time, which is optimal. The algorithm is based on the following simple observation.

Observation 3.1. *Let p and p' be two points of P. Then $p \in W_{\text{sq}}(p')$ if and only if $p' \in W_{\text{sq}}(p)$. Hence, $c \in \mathcal{C}_{\text{sq}}(p)$ if and only if $p \in W_{\text{sq}}(p')$ for at least one point p' with $\chi(p') = c$.*

This observation implies that $c \in \mathcal{C}_{\text{sq}}(p)$ if and only if p falls inside the union of all windows $W_{\text{sq}}(p')$ such that $\chi(p') = c$. We call the region covered by this union the *influence region* of c, and we denote it by $A(c)$.

The global algorithm. Observation 3.1 suggests the following algorithm. First, for each color $c \in \mathcal{C}$, extract all points p with $\chi(p) = c$ and compute the influence region $A(c)$. Next, scan all points in P and calculate for each point p the number of influence regions containing p. To do this efficiently, we need to refine the algorithm in the following way: instead of processing P as a whole, we partition it into horizontal *strips* of height $2r$, each consisting of $\Theta(r)$ consecutive rows from P (except maybe the top strip, which can have fewer rows).

Each strip $\mathcal{S}$ is handled as follows. Define $\mathcal{C}_{\mathcal{S}} \subseteq \mathcal{C}$ to be the set of colors c such that there is a window $W_{\text{sq}}(p)$ intersecting $\mathcal{S}$ and with $\chi(p) = c$. Define $A_{\mathcal{S}}(c)$ to be the part of $A(c)$ that falls within $\mathcal{S}$. We start by computing $A_{\mathcal{S}}(c)$ for all $c \in \mathcal{C}_{\mathcal{S}}$. To this end, we first determine $\mathcal{K}(\mathcal{S}, c)$, the collection of points $p \in P$ such that $\chi(p) = c$ and $W_{\text{sq}}(p)$ intersects $\mathcal{S}$. Note that any point $p \in \mathcal{K}(\mathcal{S}, c)$ is either a point of the strip $\mathcal{S}$ itself, or a point of the strip immediately above or below $\mathcal{S}$. Hence, by scanning these (at most) three strips we can generate the sets $\mathcal{K}(\mathcal{S}, c)$ in $O(r\sqrt{n})$ time. Note that by scanning the strips column by column, we can make sure the points in $\mathcal{K}(\mathcal{S}, c)$ are ordered from left to right.

Let $\mathcal{W}(\mathcal{S}, c)$ denote the set of windows corresponding to the points in $\mathcal{K}(\mathcal{S}, c)$. To compute $A_{\mathcal{S}}(c)$ we run a subroutine *UnionInStrip* on $\mathcal{W}(\mathcal{S}, c)$. After we compute all

Algorithm *SquareRichness*(P)
1. Partition P into strips of height $2r$.
2. **for** every strip $\mathcal{S}$ in P
3. **do** Determine the set $\mathcal{C}_\mathcal{S}$.
4. Generate the sets $\mathcal{K}(\mathcal{S}, c)$ for all $c \in \mathcal{C}_\mathcal{S}$.
5. **for** all $c \in \mathcal{C}_\mathcal{S}$
6. **do** $A_\mathcal{S}(c) \leftarrow$ *UnionInStrip*$(\mathcal{S}, \mathcal{W}(\mathcal{S}, c))$
7. Run *RichnessInStrip* on $A_\mathcal{S}(c)$ for $c \in \mathcal{C}_\mathcal{S}$.

Algorithm 1: Algorithm *SquareRichness* that computes the square richness of all points in a grid P.

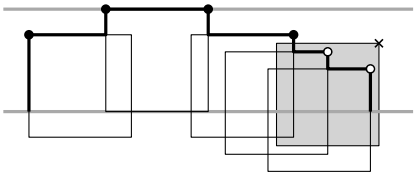


Figure 3.1: Adding a square to the envelope. The fat polyline indicates the current envelope. When the grey window W is added, the non-solid corners are replaced by the corner of W indicated by a cross.

regions $A_\mathcal{S}(c)$, we feed them to a subroutine *RichnessInStrip*. This subroutine computes for every point p in $\mathcal{S}$ the number of regions $A_\mathcal{S}(c)$ containing p, which is equal to the square richness for this point. The complete algorithm, which we call *SquareRichness*, is summarized in Algorithm 1.

It remains to describe the subroutines *UnionInStrip* and *RichnessInStrip*. For the rest of this section, we use $\mathcal{S}[i, j]$ to denote the point that appears at the i-th column and j-th row of strip $\mathcal{S}$. Without loss of generality, we assume that this point has coordinates (i, j).

Subroutine *UnionInStrip*. Consider a color $c \in \mathcal{C}$ and a strip $\mathcal{S}$. Let Union$(\mathcal{X})$ for a set $\mathcal{X}$ of sets denote $\cup_{X \in \mathcal{X}} X$. Our goal is to compute $A_\mathcal{S}(c)$, which is equal to Union$(\mathcal{W}(\mathcal{S}, c)) \cap \mathcal{S}$.

To this end we first partition $\mathcal{K}(\mathcal{S}, c)$ into two sets, $\mathcal{K}(\mathcal{S}, c)^b$ and $\mathcal{K}(\mathcal{S}, c)^t$, which contain the windows intersecting the bottom and top boundary of $\mathcal{S}$, respectively. Note that every window in $\mathcal{W}(\mathcal{S}, c)$ must intersect at least one of the two boundaries. A window that intersects both boundaries is arbitrarily assigned to one of the two sets. We describe how to compute Union$(\mathcal{W}(\mathcal{S}, c)^b) \cap \mathcal{S}$; we can compute Union$(\mathcal{W}(\mathcal{S}, c)^t) \cap \mathcal{S}$ in a similar way.

The portion of the boundary of Union($\mathcal{W}(\mathcal{S}, c)^b$) above the bottom edge of $\mathcal{S}$ is the upper envelope of $\mathcal{W}(\mathcal{S}, c)^b$. We incrementally build this upper envelope, adding the windows of $\mathcal{W}(\mathcal{S}, c)^b$ in order from left to right. (If we have several windows whose centers have the same x-coordinate, we only need to process the highest one.) Our representation of the upper envelope is simply the sorted list of window corners that appear on the upper envelope, from which it is straightforward to extract a full description of the envelope in linear time. Suppose we are currently inserting window W. As long as the rightmost corner of the current upper envelope is contained in W, we delete the corner; see Figure 3.1. This process will terminate due to one of two possible reasons.

One possibility is that we encounter a corner v which is above W. Then, v must be the top-right corner of a window W'. Since W and W' have the same size, the top side of W' is above the top side of W to the left of v. Hence, no further changes are required to the upper envelope to the left of v. We finish this case by adding the top-right vertex of W to the upper envelope.

Another possibility is that we reach a corner v that is to the left of the left side of W (or there are no remaining corners). Then, we simply add the top-left and top-right corners of W to the upper envelope.

After computing the lower envelope of $\mathcal{W}(\mathcal{S}, c)^t$ in a similar way, we have the boundaries of Union($\mathcal{W}(\mathcal{S}, c)^b$) $\cap\, \mathcal{S}$ and Union($\mathcal{W}(\mathcal{S}, c)^t$) $\cap\, \mathcal{S}$ available. Computing $A_{\mathcal{S}}(c)$ can now easily be done by a parallel left-to-right scan of the just computed envelopes. We build $A_{\mathcal{S}}(c)$ by initially including the edges of both envelopes as we scan from left to right. Every time the envelopes cross, we toggle whether or not we include current edges of the envelopes to $A_{\mathcal{S}}(c)$. This leads to the following lemma.

Lemma 3.2. *Given a set $\mathcal{K}(\mathcal{S}, c)$ whose points are sorted from left to right, we can compute $A_{\mathcal{S}}(c)$ in $O(|\mathcal{K}(\mathcal{S}, c)|)$ time. Hence, we can compute all regions $A_{\mathcal{S}}(c)$ for $c \in \mathcal{C}_{\mathcal{S}}$ in $O(r\sqrt{n})$ time in total.*

Proof. Consider the algorithm described above for constructing the upper envelope of $\mathcal{W}(\mathcal{S}, c)^b$. Inserting the windows of $\mathcal{W}(\mathcal{S}, c)^b$ takes $O(|\mathcal{W}(\mathcal{S}, c)^b|)$ time in total, because each window corner can be inserted into and deleted from the upper envelope at most once. Similarly, we compute the lower envelope of $\mathcal{W}(\mathcal{S}, c)^t$ in $O(|\mathcal{W}(\mathcal{S}, c)^t|)$ time, which sums to $O(|\mathcal{W}(\mathcal{S}, c)|) = O(|\mathcal{K}(\mathcal{S}, c)|)$ time for computing both envelopes. The time to compute $A_{\mathcal{S}}(c)$ by a parallel scan of the two envelopes is linear in the size of those envelopes and the number of crossings between the two envelopes, which is easily seen to be linear in the size of the envelopes. (Indeed, a vertical segment of one envelope intersects at most one horizontal segment of the other envelope.)

To prove the second part of the lemma, we note that (as already described) we can generate the sorted sets $\mathcal{K}(\mathcal{S}, c)$ in $O(r\sqrt{n})$ time. This implies that $\sum_{c \in \mathcal{C}_{\mathcal{S}}} |\mathcal{K}(\mathcal{S}, c)| = O(r\sqrt{n})$, which finishes the proof. □

Subroutine *RichnessInStrip*. Given a strip $\mathcal{S}$ and the set of influence regions $A_{\mathcal{S}}(c)$ for $c \in \mathcal{C}_{\mathcal{S}}$, we want to compute the square richness $|\mathcal{C}_{\text{sq}}(p)|$ of each point in $\mathcal{S}$. Recall that $c \in \mathcal{C}_{\text{sq}}(p)$ if and only if $p \in A_{\mathcal{S}}(c)$. We can decide if $p \in A_{\mathcal{S}}(c)$ by splitting the set of vertices of $A_{\mathcal{S}}(c)$ into two subsets and counting the number of vertices of each subset that are dominated by p, as explained next.

Let $V(c)$ be the set of vertices of $A_{\mathcal{S}}(c)$. We split $V(c)$ into subsets $V_{\text{tl+br}}(c)$ and $V_{\text{bl+tr}}(c)$, as follows. Each vertex $v \in V(c)$ is an endpoint of exactly one vertical edge, which can either be left-bounding (meaning $A_{\mathcal{S}}(c)$ lies locally to the right of the edge) or right-bounding (meaning $A_{\mathcal{S}}(c)$ lies locally to the left). If v is the top vertex of a left-bounding edge or the bottom vertex of a right-bounding edge, we add v to $V_{\text{tl+br}}(c)$; otherwise we add v to $V_{\text{bl+tr}}(c)$.

A point (x_1, y_1) *dominates* another point (x_2, y_2) if and only if $x_1 \geq x_2$ and $y_1 \geq y_2$. Given a point p and a set or multiset V of points, let $\text{Dom}(V, p)$ be the subset of V dominated by p. Given a point p, we define $\mu_c(p) = |\,\text{Dom}(V_{\text{bl+tr}}(c), p)| - |\,\text{Dom}(V_{\text{tl+br}}(c), p)|$.

In order to avoid degenerate cases in the next lemma, we assume without loss of generality that r is not an integer multiple of the unit distance. Indeed, if the assumption does not hold, we can instead use a window size of $r' = r + 1/2$. Then for any two points $p, p' \in P$, we have that p' is in the window with side lengths of $2r'$ centered on p if and only if p' is in the window with side lengths of size $2r$ centered on p. So, the square richness of every window remains unchanged. By our assumption, we know that the vertices of $V(c)$ and the points of P do not align vertically or horizontally.

Lemma 3.3. *For a point $p \in \mathcal{S}$, if $p \in A_{\mathcal{S}}(c)$ then $\mu_c(p) = 1$, otherwise $\mu_c(p) = 0$.*

Proof. Let ρ be the horizontal ray starting at p and extending to the left. If $p \in A_{\mathcal{S}}(c)$, then ρ intersects one more left-bounding edge than it intersects right-bounding edges; if $p \notin A_{\mathcal{S}}(c)$, then ρ intersects as many left-bounding as right-bounding edges. A left-bounding edge intersecting ρ contributes one vertex to $V_{\text{bl+tr}}(c)$ and no vertices to $V_{\text{tl+br}}(c)$, while a right-bounding edge intersecting ρ contributes one vertex to $V_{\text{tl+br}}(c)$ and no vertices to $V_{\text{bl+tr}}(c)$. Hence, the total contribution to $\mu_c(p)$ of the edges intersecting ρ is $+1$ if $p \in A_{\mathcal{S}}(c)$ and zero otherwise.

The vertical edges not intersecting ρ have both endpoints dominated by p or neither endpoint. Hence, their contribution to $\mu_c(p)$ is zero. (For an edge with neither endpoint being dominated this is trivial; for an edge with both endpoint dominated this follows because the contributions of the two endpoints of a vertical edge cancel each other.) Thus $\mu_c(p) = 1$ if $p \in A_{\mathcal{S}}(c)$ and $\mu_c(p) = 0$ otherwise. □

According to Lemma 3.3 we can decide if $p \in A_{\mathcal{S}}(c)$ by counting the number of points in $V_{\text{tl+br}}(c)$ and $V_{\text{bl+tr}}(c)$ dominated by p. Instead of doing this separately for each $c \in \mathcal{C}$, we combine these computations. To this end, define $V_{\text{tl+br}}$ and $V_{\text{bl+tr}}$ to be

the multisets obtained by merging all sets $V_{\text{tl+br}}(c)$ and $V_{\text{bl+tr}}(c)$, respectively. Given a point p, we define $\mu(p) = |\operatorname{Dom}(V_{\text{bl+tr}}, p)| - |\operatorname{Dom}(V_{\text{tl+br}}, p)|$. We have the following lemma, which follows readily from Lemma 3.3.

Lemma 3.4. *For a point $p \in \mathcal{S}$, the number of influence regions containing p is equal to $\mu(p)$.*

We compute the values $\mu(p)$ for all points $p \in \mathcal{S}$ in a batched manner, as follows. Let M be a matrix that has the same number of rows and columns as the subset of grid P that lies in $\mathcal{S}$. Each entry in M is an integer, and initially all entries are set to zero. We now go over the points in the multisets $V_{\text{bl+tr}}$ and $V_{\text{tl+br}}$ one by one. When handling a point v we update an entry of M as follows. Let $p' = (i', j')$ be the point in $\mathcal{S}$ that dominates v and is closest to v. In this way we "snap" v to point p' in the grid P. If such a point does not exist, we don't do anything for v since no point in $\mathcal{S}$ dominates it. If $v \in V_{\text{bl+tr}}$ then we increment $M[i', j']$, and if $v \in V_{\text{tl+br}}$ then we decrement $M[i', j']$.

Note that $p = (i, j)$ dominates a point $v \in V_{\text{bl+tr}} \cup V_{\text{tl+br}}$ if and only if we snap v to a point $p' = (i', j')$ such that $i' \leq i$ and $j' \leq j$. Hence,

$$\mu((i, j)) = \sum_{i' \leq i} \sum_{j' \leq j} M[i', j'].$$

This is the so-called *prefix sum* of entry $M[i, j]$. Given a matrix M, the prefix sums of all entries in M can be computed in linear time with a simple algorithm that scans the matrix row by row and uses that $\mu((i, j)) = \mu((i-1, j)) + \sum_{j' \leq j} M[i, j']$. This leads to the following lemma.

Lemma 3.5. *Let $\mathcal{S}$ be a strip of height $2r$. Given the influence regions $A_{\mathcal{S}}(c)$ for all $c \in \mathcal{C}_{\mathcal{S}}$, we can compute the value $\text{rich}_{\text{sq}}(p)$ for every point $p \in \mathcal{S}$ in $O(r\sqrt{n})$ time in total.*

Putting it together. Since we have $O(\sqrt{n}/r)$ strips in total, Lemmata 3.2 and 3.5 and the fact that $|\mathcal{C}_{\mathcal{S}}| = O(r\sqrt{n})$ imply the following result.

Theorem 3.6. *Let $\mathcal{C}$ be a set of colors. Let P be a grid of n points, each associated with a color in $\mathcal{C}$. Algorithm SquareRichness computes the square richness for every point of P in a total of $O(n)$ time.*

3.1.3 Disk Categorical Richness

Our algorithm for the disk richness problem is based on a similar observation as we used for square windows.

Observation 3.7. *Let p and p' be two points of P. Then $p \in W_{\mathrm{d}}(p')$ if and only if $p' \in W_{\mathrm{d}}(p)$. Hence, $c \in \mathcal{C}_{\mathrm{d}}(p)$ if and only if $p \in W_{\mathrm{d}}(p')$ for at least one point p' with $\chi(p') = c$.*

Thus we still have that $c \in \mathcal{C}_{\mathrm{d}}(p)$ if and only if p lies inside the union of all windows $W_{\mathrm{d}}(p')$ with $\chi(p') = c$. However, computing this union is significantly more complicated for disks, and we need to adapt our algorithm in several ways.

The global algorithm. Instead of partitioning the grid P into strips we now partition it into $O(n/r)$ *tiles*, where each tile consists of (at most) $\lfloor r/\sqrt{2} \rfloor \times \lfloor r/\sqrt{2} \rfloor$ points. The tiles are processed as follows.

Consider a tile T and a window $W_{\mathrm{d}}(p)$ of a point $p \in T$. Because the diameter of T—that is, the distance between the top-left and bottom-right points in T—is at most the radius of our windows, we know that $W_{\mathrm{d}}(p)$ covers all the points in T. Based on this observation, we scan all points in T to determine the subset $\mathcal{C}_{\mathrm{in}}$ of colors associated with the points $p \in T$. For each point $p \in T$, we initialize $\mathrm{rich}_{\mathrm{d}}(p)$ to $|\mathcal{C}_{\mathrm{in}}|$. Let $\mathcal{C}_{\mathrm{out}}$ denote the colors in $\mathcal{C} \setminus \mathcal{C}_{\mathrm{in}}$ whose influence regions overlap with T. We need to compute for all points $p \in T$ the number of influence regions of colors in $\mathcal{C}_{\mathrm{out}}$ that contain p, and add this number to $\mathrm{rich}_{\mathrm{d}}(p)$. This is done as follows.

Let ℓ_{top}, ℓ_{bot}, ℓ_{left}, and ℓ_{right} denote the lines containing the top, bottom, left, and right edge of T, respectively. For a color $c \in \mathcal{C}_{\mathrm{out}}$, let $\mathcal{D}_{\mathrm{out}}(c)$ be the collection of windows $W_{\mathrm{d}}(p)$ that intersect T and have $\chi(p) = c$. Note that $\sum_c |\mathcal{D}_{\mathrm{out}}(c)| = O(r^2)$ and the sets $\mathcal{D}_{\mathrm{out}}(c)$ can be generated in $O(r^2)$ time. We partition $\mathcal{D}_{\mathrm{out}}(c)$ into four subsets $\mathcal{D}_{\mathrm{top}}(c)$, $\mathcal{D}_{\mathrm{bot}}(c)$, $\mathcal{D}_{\mathrm{left}}(c)$, and $\mathcal{D}_{\mathrm{right}}(c)$. The set $\mathcal{D}_{\mathrm{top}}(c)$ consists of all windows in $\mathcal{D}_{\mathrm{out}}(c)$ whose centers lie above ℓ_{top}, and the set $\mathcal{D}_{\mathrm{bot}}(c)$ consists of all windows whose centers lie below ℓ_{bot}. The set $\mathcal{D}_{\mathrm{left}}(k)$ contains all windows whose centers lie to the left of ℓ_{left}, and which do not belong to $\mathcal{D}_{\mathrm{top}}(k)$ or $\mathcal{D}_{\mathrm{bot}}(k)$. The set $\mathcal{D}_{\mathrm{right}}(k)$ contains the rest of the windows in $\mathcal{D}_{\mathrm{out}}(k)$; note that these windows have their center to the right of ℓ_{right}.

For each of these four sets we will compute the part of their union inside T using a subroutine *EnvelopeInTile*. Each part is bounded by a (lower, upper, left, or right) envelope, which will allow us to compute them efficiently. Then we will process the collection of these envelopes using a subroutine *RichnessInTile* to determine the richness of each grid point $p \in T$. Algorithm 2 summarizes our global algorithm. It remains to describe the subroutines *EnvelopeInTile* and *RichnessInTile*.

Subroutine *EnvelopeInTile*. Consider a tile T and a collection $\mathcal{D}_{\mathrm{bot}}(c)$ of disks whose centers lie below ℓ_{bot}, the line containing the bottom edge of T. We describe

Algorithm *DiskRichness*(P)

1. Partition P into tiles of at most $\lfloor r/\sqrt{2} \rfloor \times \lfloor r/\sqrt{2} \rfloor$ points.
2. **for** each tile T in P
3. **do** Determine the set $\mathcal{C}_{\text{in}}$ of all colors stored in T, and set $\text{rich}_{\text{d}}(p) = |\mathcal{C}_{\text{in}}|$ for every point $p \in T$.
4. Generate the sets $\mathcal{D}_{\text{top}}(c)$, $\mathcal{D}_{\text{bot}}(c)$, $\mathcal{D}_{\text{left}}(c)$, and $\mathcal{D}_{\text{right}}(c)$ for all $c \in \mathcal{C}_{\text{out}}$.
5. **for** every $c \in \mathcal{C}_{\text{out}}$
6. **do** $\mathcal{E}_{\text{top}}(c) \leftarrow$ *EnvelopeInTile*$(\mathcal{D}_{\text{top}}(c))$
7. $\mathcal{E}_{\text{bot}}(c) \leftarrow$ *EnvelopeInTile*$(\mathcal{D}_{\text{bot}}(c))$
8. $\mathcal{E}_{\text{left}}(c) \leftarrow$ *EnvelopeInTile*$(\mathcal{D}_{\text{left}}(c))$
9. $\mathcal{E}_{\text{right}}(c) \leftarrow$ *EnvelopeInTile*$(\mathcal{D}_{\text{right}}(c))$
10. Run *RichnessInTile* on the collection of envelopes computed in Steps 5–9.

Algorithm 2: Algorithm *DiskRichness* that computes the disk richness of all points in a grid P.

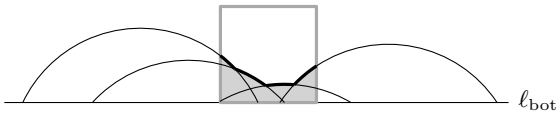


Figure 3.2: The part of Union$(\mathcal{D}_{\text{bot}}(c))$ inside T is shown in grey. It is bounded from above by the upper envelope of the arcs $\alpha(p)$.

a subroutine *EnvelopeInTile* that computes the part of Union$(\mathcal{D}_{\text{bot}}(c))$ inside T. The parts of Union$(\mathcal{D}_{\text{top}}(c))$, Union$(\mathcal{D}_{\text{left}}(c))$, and Union$(\mathcal{D}_{\text{right}}(c))$ inside T can be computed similarly.

As in the case of square windows, we may assume that we generated $\mathcal{D}_{\text{bot}}(c)$ such that the disks in $\mathcal{D}_{\text{bot}}(c)$ are sorted from left to right. If we have multiple disks whose centers have the same x-coordinate we only need to keep the highest one. Thus we can also assume that $\mathcal{D}_{\text{bot}}(c)$ (and similarly $\mathcal{D}_{\text{top}}(c)$, $\mathcal{D}_{\text{left}}(c)$, and $\mathcal{D}_{\text{right}}(c)$) have size $O(r)$.

Let ℓ^+_{bot} denote the halfplane above ℓ_{bot}. For each disk $W_{\text{d}}(p) \in \mathcal{D}_{\text{bot}}(c)$, let $\alpha(p) = \partial W_{\text{d}}(p) \cap \ell^+_{\text{bot}}$ denote the part of the boundary of $W_{\text{d}}(p)$ above ℓ_{bot}. Note that Union$(\mathcal{D}_{\text{bot}}(c))$ in T is bounded from above by (a part of) the upper envelope of the arcs $\alpha(p)$; see Figure 3.2. Thus computing Union$(\mathcal{D}_{\text{bot}}(c))$ in T boils down to computing this envelope. To do this we need the following lemma.

Lemma 3.8. *Let $\mathcal{D}$ be a set of disks intersecting a horizontal line ℓ whose centers lie below ℓ. Let D be the disk in $\mathcal{D}$ whose center has maximum x-coordinate. If D contributes to the upper envelope of the disks above ℓ, then its contribution is a single arc that is the rightmost arc of the envelope.*

Proof. Let p_1 be the rightmost intersection of the boundary of D with ℓ. Let $\mathcal{E}$ be the upper envelope of all the windows in $\mathcal{D}$ except for D. We consider two cases: whether or not p_1 lies under $\mathcal{E}$.

In the first case, p_1 is not under $\mathcal{E}$. Consider traversing the boundary of D counterclockwise starting just to the left of p_1. During this traversal, let p_2 be the first intersection point that we encounter between the boundary of D with $\mathcal{E}$ or ℓ. If p_2 lies on ℓ then we are done since D contributes to the upper envelope the arc from p_1 to p_2, and the rest of the boundary of D lies under ℓ. Otherwise, p_2 lies on the boundary of a window $D' \in \mathcal{D}$. Note that p_2 is one of the two intersection points between the boundaries of D' and D. We now claim that the other intersection point p_2' must lie below ℓ, which implies that D cannot contribute anything other than the arc from p_1 to p_2 to the upper envelope.

Consider the centers q and q' of D and D', respectively. Both intersection points p_2 and p_2' lie on the perpendicular bisector of $\overline{qq'}$. Furthermore they lie in opposite directions from the midpoint of $\overline{qq'}$. Since both q and q' lie under ℓ, so must the midpoint of $\overline{qq'}$. Since p_2 lies over ℓ it must be that p_2', which is in the opposite direction of the midpoint of $\overline{qq'}$, must lie under ℓ.

In the second case, p_1 lies under $\mathcal{E}$. Thus, it is inside another window $D' \in \mathcal{D}$. Let I' and I be the intervals formed by intersecting ℓ with D' and D respectively. We first claim I' contains I. Again, let q' and q be the centers of D' and D, respectively. Notice that the midpoint of I has the same x-coordinate as q, and the same holds for the midpoint of I' and q'. Since q is to the right of q', we have that the midpoint of I is to the right of the midpoint of I'. Thus, more than the right half of I is contained in the right half of I'. Therefore, also the left half of I is contained in I'. Given this configuration, and since the two disks are not identical, if the boundary of D were to intersect with the boundary of D' above ℓ, it would need to do so twice. However, we have already argued that at most one intersection between the two boundaries can occur above ℓ. So, window D does not contribute to the upper envelope of $\mathcal{D}$, and the lemma follows. □

Let $\text{Arcs}(\mathcal{D}_{\text{bot}}(c)) = \{\alpha(p) : W_{\text{d}}(p) \in \mathcal{D}_{\text{bot}}(c)\}$ be the set of arcs of which we want to compute the upper envelope, sorted from left to right. (More precisely, sorted according to the x-coordinate of the centers of the corresponding disks $W_{\text{d}}(p)$.) Our algorithm to compute the upper envelope of $\text{Arcs}(\mathcal{D}_{\text{bot}}(c))$ is similar to the algorithm we used for constructing the upper envelope of a set of squares: we go over the arcs in order, and when adding an arc $\alpha(p)$ we remove the part of the current envelope below $\alpha(p)$ and we add the relevant part of $\alpha(p)$. The details are as follows.

We maintain a list $\mathcal{L}$ that stores the portions of the arcs $\alpha(p)$ appearing on the current envelope, ordered from left to right. To process the next arc $\alpha(p)$, we first

check if the right endpoint of $\alpha(p)$ lies to the right of the right endpoint of the last arc in the list. If this is not the case, then $\alpha(p)$ does not contribute to the envelope (by Lemma 3.8) and we are done with $\alpha(p)$. Otherwise we start walking back along $\mathcal{L}$ as long as we encounter envelope arcs that lie entirely below $\alpha(p)$, and we remove these arcs from $\mathcal{L}$. The walk continues until either (i) we encounter an arc β that is intersected by $\alpha(p)$, or (ii) we encounter an arc that lies fully to the left of $\alpha(p)$, or (iii) the list $\mathcal{L}$ becomes empty.

In case (i) we shrink β by removing the part of β below $\alpha(p)$ and we append the part of $\alpha(p)$ to the right of the intersection point $q = \alpha(p) \cap \beta$ to the list $\mathcal{L}$. By Lemma 3.8 the arc $\alpha(p)$ does not contribute anything to the left of q, and so we can stop. In cases (ii) and (iii) we simply append the entire arc $\alpha(p)$ to $\mathcal{L}$, and we can stop as well.

The running time of the algorithm described above is linear in the number of arcs it processes, since each arc is inserted and deleted at most once. This leads to the following lemma. Let $\mathcal{E}_{\text{bot}}(c)$ denote the envelope that forms the upper boundary of $\text{Union}(\mathcal{D}_{\text{bot}}(c)) \cap T$. Define $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, and $\mathcal{E}_{\text{right}}(c)$ similarly for $\mathcal{D}_{\text{top}}(c)$, $\mathcal{D}_{\text{left}}(c)$, and $\mathcal{D}_{\text{right}}(c)$.

Lemma 3.9. *EnvelopeInTile can compute $\mathcal{E}_{\text{bot}}(c)$, $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, and $\mathcal{E}_{\text{right}}(c)$, in $O(r)$ time.*

We now present a property of the envelopes that is useful for the rest of our analysis.

Lemma 3.10. *Each envelope $\mathcal{E}_{\text{bot}}(c)$, $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, and $\mathcal{E}_{\text{right}}(c)$ has complexity $O(r)$ and length $O(r)$.*

Proof. We prove the lemma for the envelope $\mathcal{E}_{\text{bot}}(c)$; the same arguments apply to the other envelopes. The complexity bound follows from the fact that for every x-coordinate, only one disk centered at the x-coordinate can contribute to $\mathcal{E}_{\text{bot}}(c)$.

We call a point $q \in \mathcal{E}_{\text{bot}}(c)$ an *extremum* if q is a local maximum or a local minimum on $\mathcal{E}_{\text{bot}}(c)$, and q is not the leftmost or rightmost point of $\mathcal{E}_{\text{bot}}(c)$. Note that a local maximum is the top point of a window; a local minimum is a vertex of $\mathcal{E}_{\text{bot}}(c)$, that is, an intersection between two window boundaries, or between a window boundary and ℓ_{bot}. We split $\mathcal{E}_{\text{bot}}(c)$ into pieces at the extrema. As we traverse such a piece from left to right, the y-coordinate is monotonically increasing or decreasing. This means that each piece has Euclidean length at most $2r$. In particular, the first and last pieces have total length at most $4r$. All other pieces are bounded by two extrema. Consider such a piece, which lies between successive extrema b and b'. Then the length of the envelope between b and b' is at most $|x(b) - x(b')| + |y(b) - y(b')|$. We claim that $|x(b) - x(b')| \geq |y(b) - y(b')|$. This implies that the length of $\mathcal{E}_{\text{bot}}(c)$ between the first and last extremum is bounded by $2r$, and thus that the total length of $\mathcal{E}_{\text{bot}}(c)$ is at most $6r$.

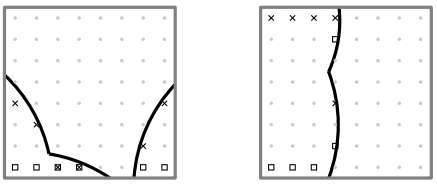


Figure 3.3: Entry points are indicated by small squares, exit points are indicated by crosses. On the left the entry and exit points for $\mathcal{E}_{\text{bot}}(c)$ are shown, on the right for $\mathcal{E}_{\text{left}}(c)$.

It remains to prove the claim. Assume without loss of generality that b is a local maximum and b' is a local minimum, and let $D \in \mathcal{D}_{\text{bot}}(c)$ be the window whose top is b. Consider the rectangle B whose opposite corners are b and b'. Let q be the intersection between the boundary of D with the bottom edge of B. Then we have $|x(b) - x(q)| \geq |y(b) - y(q)| \geq |y(b) - y(b')|$, where the first inequality follows from the fact that the center of D lies below ℓ_{bot}. □

Subroutine *RichnessInTile*. Subroutine *RichnessInTile* takes as input the four envelopes $\mathcal{E}_{\text{bot}}(c)$, $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, and $\mathcal{E}_{\text{right}}(c)$ for each of the colors $c \in \mathcal{C}_{\text{out}}$. Its task is to determine for each point $p \in T$ the number of colors $c \in \mathcal{C}_{\text{out}}$ such that p is covered by a window in $\mathcal{D}_{\text{out}}(c)$. The latter is the case if p lies below $\mathcal{E}_{\text{bot}}(c)$, above $\mathcal{E}_{\text{top}}(c)$, to the left of $\mathcal{E}_{\text{left}}(c)$, or to the right of $\mathcal{E}_{\text{right}}(c)$. *RichnessInTile* is based on this observation, and consists of two steps.

Step I: Computing entry and exit points in each column. Consider an envelope $\mathcal{E}$ (which is one of the four envelopes $\mathcal{E}_{\text{bot}}(c)$, $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, or $\mathcal{E}_{\text{right}}(c)$, for some c) and a column $T[i, 1..\lfloor r/\sqrt{2} \rfloor]$ of the tile T. Let $A(\mathcal{E})$ denote the area enclosed by $\mathcal{E}$ and the corresponding edge of T. The area $A(\mathcal{E}_{\text{bot}}(c))$, for example is enclosed by $\mathcal{E}_{\text{bot}}(c)$ and the bottom edge of T. As we traverse the column from bottom to top, we may enter and exit the area $A(\mathcal{E})$ one or more times. We call the points where this happens *entry points* and *exit points* (for the given envelope) where the first point in a column is also considered an entry point if it lies inside $A(\mathcal{E})$; see Figure 3.3. The envelopes $\mathcal{E}_{\text{bot}}(c)$ and $\mathcal{E}_{\text{top}}(c)$ have at most one entry and exit point per row, while $\mathcal{E}_{\text{left}}(c)$ or $\mathcal{E}_{\text{right}}(c)$ may have multiple entry and exit points. Note that entry and exit points can coincide if we have a point whose center is inside $A(\mathcal{E})$ while the centers of both adjacent points are outside.

In Step I we compute, for each c and each envelope $\mathcal{E}_{\text{bot}}(c)$, $\mathcal{E}_{\text{top}}(c)$, $\mathcal{E}_{\text{left}}(c)$, and $\mathcal{E}_{\text{right}}(c)$, all entry and exit points. This is done as follows.

First consider $\mathcal{E}_{\text{bot}}(c)$. Here a point $T[i, j]$ is an exit point if $\mathcal{E}_{\text{bot}}(c)$ crosses the segment connecting (i, j) and $(i, j + 1)$ or if $T[i, j]$ is the top point in a column and

$\mathcal{E}_{\text{bot}}(c)$ passes above (i, j). Thus we can compute all exit points by tracing $\mathcal{E}_{\text{bot}}(c)$ through T, visiting each column crossing. In each column where $\mathcal{E}_{\text{bot}}(c)$ has an exit point, the bottom point of the column is an entry point. By Lemma 3.10, we can compute all entry and exit points in $O(r)$ time, since every column crossing can be charged to either the complexity or the length of $\mathcal{E}_{\text{bot}}(c)$. The entry and exit points for $\mathcal{E}_{\text{top}}(c)$ can be computed in a similar way.

Now consider $\mathcal{E}_{\text{left}}(c)$; the envelope $\mathcal{E}_{\text{right}}(c)$ is handled similarly. We first compute, in a similar way as above, the *row exit points* with respect to $\mathcal{E}_{\text{left}}(c)$; these are the points $T[i, j]$ such that (i, j) lies to the left of $\mathcal{E}_{\text{left}}(c)$ while $(i + 1, j)$ lies to the right. We can now decide which points in a row are column entry and exit points by looking at the row exit points in adjacent columns. Indeed, suppose $T[i, j]$ is the row exit point in the i-th row, and $T[i + 1, j']$ is the row exit point in the $(i + 1)$-th column. Then if $j' > j$ then all points $T[i+1, j'']$ with $j < j'' \leq j'$ are entry points, and if $j' < j$ then all points $T[i, j'']$ with $j' < j'' \leq j$ are exit points. Because the total number of entry and exit points is $O(r)$—this follows from the $O(r)$ upper bound on the number of column crossings which also applies to $\mathcal{E}_{\text{left}}(c)$—we spend $O(r)$ time in total to compute the entry and exit points for $\mathcal{E}_{\text{left}}(c)$.

Step II: Computing the richness values. With the set of all entry and exit points available, we can compute the richness of the points in T column by column. To process a column, we need a counter $count[c]$ for each $c \in \mathcal{C}_{\text{out}}$ and a global counter $count_{\text{g}}$, all initialized to zero. We use $count[c]$ to count how many of the regions $A(\mathcal{E}_{\text{bot}}(c))$, $A(\mathcal{E}_{\text{top}}(c))$, $A(\mathcal{E}_{\text{left}}(c))$, and $A(\mathcal{E}_{\text{right}}(c))$ contain the current point; thus $0 \leq count[c] \leq 4$. The global counter $count_{\text{g}}$ counts how many of the counters $count[c]$ are positive for the current point. Thus $count_{\text{g}}$ indicates the number of colors $c \in \mathcal{C}_{\text{out}}$ such that the current point is covered by a window in $\mathcal{D}_{\text{out}}(c)$.

Note that a point can be an entry and/or exit point for several different colors c. As we traverse a column we handle each encountered point as follows: for all colors c for which the current point is an entry point we increment $count[c]$ and if $count[c]$ was zero we also increment $count_{\text{g}}$. Next we add $count_{\text{g}}$ to the richness of the current point and output this final richness. Finally, for all colors c for which the current point is an entry point we decrement $count[c]$ and if $count[c]$ becomes zero we also decrement $count_{\text{g}}$.

As explained above, Step I of the subroutine spends $O(r)$ time for each $c \in \mathcal{C}_{\text{out}}$. Thus the total time for Step I, and also the total number of entry and exit points generated, is $O(r \cdot |\mathcal{C}_{\text{out}}|) = O(rt)$. Step II spends $O(1)$ time at each point in T, plus $O(1)$ time for each entry and exit point computed in Step I. We get the following lemma.

Lemma 3.11. *RichnessInTile runs in $O(r^2 + rt)$ time.*

Putting it together. Since we have $O(n/r^2)$ tiles in total, Lemmata 3.9 and 3.11 together imply the following result.

Theorem 3.12. *Let $\mathcal{C}$ be a set of colors. Let P be a grid of n points, each associated with a color in $\mathcal{C}$. Algorithm DiskRichness computes the disk richness for every point of P in $O((1+t/r)n)$ time in total.*

3.2 Hypercube Colored Range Counting

Let P be a set of n colored points in $\mathbb{R}^d$—note that we do not require the points in P to form a grid—and let $\mathcal{C} = [t]$ denote the set of colors associated to the points of P. In this section we study data structures for *colored range counting*: given a query range R, compute the number of distinct colors associated with the points of P inside R.

Variants of the problem exist for different types of query ranges. We consider cases in which query ranges are restricted to be hypercubes. For square colored range counting, we show that there is an efficient data structure with $O(n \operatorname{polylog} n)$ storage and $O(\operatorname{polylog} n)$ query time via a reduction to five-dimensional dominance (uncolored) range counting. For offline cube colored range counting, we give a conditional hardness result via a reduction from Boolean matrix multiplication. We also consider the case in which all queries have the same fixed size. We reduce fixed-size cube colored range counting to three-dimensional dominance (uncolored) range counting. Finally, for completeness, we give a simple reduction from Boolean matrix multiplication to offline fixed-size four-dimensional hypercube colored range counting.

3.2.1 Square Ranges

We start by presenting a data structure for colored range counting in $\mathbb{R}^2$, for the case where query ranges are (variable-size) squares. Our solution in Section 3.1.2 hinged on the duality property of Observation 3.1. In the current setting, we can no longer use this duality property, because different query squares may have different sizes. In order to obtain a duality property for variable-size queries, we need to move into three dimensions.

Definitions. We define the *radius* of a square be half of its side length. For a point p, let $W_r(p)$ be the square window of radius r centered at p, and let $\mathcal{C}_r(p) \subseteq \mathcal{C}$ be the set of all colors that occur in the window $W_r(p)$. The *richness* of $W_r(p)$ is $\operatorname{rich}_r(p) = |\mathcal{C}_r(p)|$. For a color $c \in \mathcal{C}$, let $P_c \subseteq P$ be the set of points with color c.

If p is a point on the plane $z = 0$, let $p \uparrow z'$ denote the same point lifted to the plane $z = z'$. This notation extends to sets of points. Let Pyramid(p) be an infinitely large upside-down three-dimensional pyramid with its apex at some point p in the $z = 0$ plane. Thus Pyramid$(p) = \bigcup_{r \in \mathbb{R}^+} (W_r(p) \uparrow r)$.

Main idea. The following lemma is a duality property that is useful for variable-size queries. The lemma is well known, but for completeness we give a proof.

Lemma 3.13. *For any two points p and q we have $p \in W_r(q)$ if and only if $q \uparrow r \in$* Pyramid(p).

Proof. The intersection of Pyramid(p) with the plane $z = r$ is an elevated window $W_r(p) \uparrow r$. Thus, $q \uparrow r \in$ Pyramid(p) if and only if $q \uparrow r \in W_r(p) \uparrow r$. Clearly, $q \uparrow r \in W_r(p) \uparrow r$ if and only if $q \in W_r(p)$. By Observation 3.1, $q \in W_r(p)$ if and only if $p \in W_r(q)$. □

Lemma 3.13 implies that $c \in \mathcal{C}_r(q)$ if and only if $q \uparrow r$ lies in the union of all pyramids Pyramid(p) where $p \in P_c$. Similarly to Section 3.1.2, we therefore define the influence region $A(c)$ as the union of pyramids of the points in P_c.

Our data structure now works as follows. We construct for each color $c \in \mathcal{C}$ the influence region $A(c)$. For a query range with radius r_q and center q, the richness $\text{rich}_{r_q}(q)$ is the number of influence regions containing $q \uparrow r_q$. To compute this number we decompose the influence regions into simpler parts, and count how many of these simpler parts contain point $q \uparrow r_q$.

Complexity of influence regions. To be able to analyze the efficiency of our data structure we need to bound the complexity of the influence regions.

Lemma 3.14. *For each color $c \in \mathcal{C}$, the complexity of the influence region $A(c)$ is $O(|P_c|)$.*

Proof. An edge of an influence region can either be an edge of one of the pyramids or an intersection of faces of pyramids. We call the former *pyramid edges* and the latter *intersection edges*. There are $4|P_c|$ pyramid edges, because when a pyramid edge enters another pyramid Pyramid(p), it stays inside Pyramid(p). To bound the number of intersection edges, we note that the projection of these edges onto the plane $z = 0$ is equal to the L_∞-Voronoi diagram of the points in P_c. (The relation between Voronoi diagrams in the plane and the lower envelope of certain cones is well known. For the Euclidean distance the cones are circular; for the L_∞-distance the cones are as defined above.) A linear bound on the number of intersection edges thus follows from the fact that L_∞-Voronoi diagrams have linear complexity [Aur91]. □

Decomposition of influence regions. Consider the map $\mathcal{M}_c$ formed by projecting the faces of $A(c)$ to the plane $z = 0$. We know that M_c has $O(|P_c|)$ faces due to Lemma 3.14. The map is octilinear since L_∞-Voronoi diagrams are octilinear and the projections of the pyramid edges align with four of these eight directions. We partition each face of $\mathcal{M}_c$ into smaller faces with constant complexity by shooting vertical rays up and down from each vertex. The resulting faces are vertical slabs that may be cut at the ends either horizontally or diagonally. Let the new map with these simpler faces be $\mathcal{M}'_c$. It also has $O(|P_c|)$ faces, each with at most 4 edges.

We decompose $A(c)$ into $O(|P_c|)$ pieces $\mathcal{F}_c$, such that each piece is the subset of $A(c)$ that lies above some face of $\mathcal{M}'_c$. We call these pieces *towers*. Thus, $\text{rich}_{r_q}(q)$ is the number of towers (across the influence regions of all colors) that contain point $q \uparrow r_q$. The following lemma describes a data structure that can efficiently compute the number of towers stabbed by an arbitrary query point.

Lemma 3.15. *Counting the number of towers amongst a set of n towers stabbed by a query point reduces to five-dimensional dominance range counting in a set of n points.*

Proof. Every tower is the intersection of at most five halfspaces. Each halfspace has one of $O(1)$ orientations. We partition the input towers into types based on number and orientation of their defining halfspaces. There are a constant number of types of towers. We handle each type of tower separately and to answer a query we simply add the counts for each type. Fix a type of tower formed by b halfspaces with b different orientations. For each orientation of a halfspace, we create a coordinate axis that is normal to the bounding plane of the halfspace and that increases towards the interior of the halfspace. We transform each tower F into a b-dimensional point f. For each defining halfspace of F, the coordinate of f on the associated coordinate axis is the value on the axis that the halfspace's bounding plane intersects. We also transform q into a b-dimensional point q'. For each coordinate axis, the coordinate value of q' is the value of the projection of q onto the axis. In this way, $q \in F$ if and only if q' dominates f. Computing the number of b-dimensional points dominated by a b-dimensional point is an instance of b-dimensional dominance range counting. □

We conclude with the following theorem.

Theorem 3.16. *Square colored range counting in a set of n points reduces to five-dimensional dominance range counting in a set of $O(n)$ points.*

3.2.2 Cube Ranges

We now consider the three-dimensional generalization of the problem studied in the previous section. Thus P is a set of n colored points in $\mathbb{R}^3$ and the query ranges are

(arbitrarily-sized) cubes. We give evidence that it may not be possible to obtain a data structure of similar efficiency—$O(n \operatorname{polylog} n)$ storage and $O(\operatorname{polylog} n)$ query time—as in the square case. Our negative result holds in an *offline* setting, that is, when all query ranges are known in advance.

We begin with a simple generalization of a technique of Kaplan et al. [KRSV08] which is able to reduce Boolean matrix multiplication to offline colored range counting problems. The proof of this lemma is similar to the proof given by Kaplan et al.

Lemma 3.17. *Let $\mathfrak{R}$ be a family of ranges in $\mathbb{R}^d$. Suppose that there exist two point sets P, Q in $\mathbb{R}^d$ and a set of ranges $\mathcal{R} \subseteq \mathfrak{R}$ such that:*

- *$|P| = |Q| = \sqrt{n}$ and $|\mathcal{R}| = n$,*
- *for every $(p, q) \in P \times Q$, there is a range $R \in \mathcal{R}$ such that $R \cap (P \cup Q) = \{p, q\}$.*

Then we can multiply two Boolean matrices of size $\sqrt{n} \times \sqrt{n}$ by performing n colored range counting queries with ranges from $\mathfrak{R}$ on a colored point set containing at most $2n$ colored points in $\mathbb{R}^d$.

Proof. We are given two $\sqrt{n} \times \sqrt{n}$ Boolean matrices $A = \{a_{i,j}\}$ and $B = \{b_{i,j}\}$. We wish to compute $C = \{c_{i,j}\}$ where $c_{i,j} = \bigvee_k (a_{k,i} \wedge b_{j,k})$.

We assign each row k of A to a distinct point $p_k \in P$. For each true entry $a_{k,i}$, we construct a point of color i at position p_k. We also store a count row_k with the number of true entries in the row. Similarly, we assign each column k of B to a distinct point $q_k \in Q$. For each true entry $b_{j,k}$, we construct a point of color j at position q_k. We also store a count col_k of the number of true entries in the column. We thus construct a set P_A of at most n points for the rows of A and a set P_B of at most n points for the columns of B.

Note that $c_{i,j}$ is true if and only if the range $R \in \mathcal{R}$ that contains only p_i and q_j contains a duplicate color k, because then there is a k with $a_{k,i} = \text{TRUE}$ and $b_{j,k} = \text{TRUE}$. To decide whether or not there is a duplicate color, we count the number of colors in R and compare it to $\text{row}_i + \text{col}_j$. We can thus evaluate all n cells of C by performing n colored range counting queries on the set $P_A \cup P_B$. □

The proof of Lemma 3.17 exploits the ability to create point sets with many differently colored points at the same position. (Thus the point set is actually a multiset.) In our setting, where the ranges are cubes in $\mathbb{R}^3$, it is straightforward to perturb the points and ranges so that they do not share any coordinate values.

The reduction in the proof of the lemma above yields an instance of an offline range searching problem, since all queries can be generated in advance. Using this lemma, we obtain the following theorem.

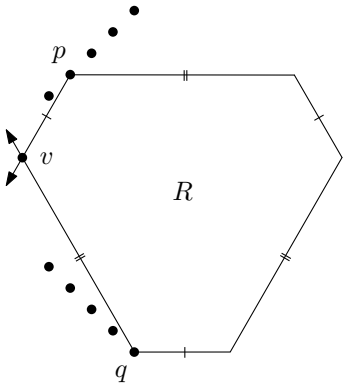


Figure 3.4: The cubic section R that hits p and q.

Theorem 3.18. *Suppose there is an algorithm for offline cube colored range counting that runs in* $\mathrm{T}(n, m)$ *time where* n *is the number of points and* m *is the number of query ranges. Then, we can multiply two Boolean matrices of size* $\sqrt{n} \times \sqrt{n}$ *in* $\mathrm{T}(2n, n) + O(n)$ *time.*

Proof. We intend to invoke Lemma 3.17 where $\mathfrak{R}$ is the set of all axis-aligned cubes in $\mathbb{R}^3$. All points of the sets P and Q that we construct will lie on the plane $z = x + y$. Let a *cubic section* be any two-dimensional shape formed by the intersection of an axis-aligned cube with the plane $z = x + y$. We choose a new coordinate system so that the plane $z = x + y$ in the old coordinate system becomes the plane $z = 0$ in the new coordinate system and each cubic section has a side parallel to the new x-axis. Then it is sufficient to consider the case where $\mathfrak{R}$ is the set of all cubic sections in $\mathbb{R}^2$.

A cubic section is the intersection of two equilateral triangles with the same centers such that one has a bottom side that is parallel to the x-axis and the other has a top side that is parallel to the x-axis. (These triangles are the intersections of opposing octants with the plane $z = 0$. The intersection of the opposing octants is the cube from which the cubic section arises.) A cubic section has either three or six sides. If it has three sides, then it is an equilateral triangle. Otherwise, its opposite sides are parallel and side lengths alternate between two values around the cubic section.

Let P be $\sqrt{n}$ points evenly distributed along the segment from $(0, 1)$ to $(1, 2)$. Let Q be $\sqrt{n}$ points evenly distributed along the segment from $(0, -1)$ to $(1, -2)$. For each pair $(p, q) \in P \times Q$, we construct a cubic section R which contains only points p and q. From p, we shoot a ray at angle $-(2/3)\pi$ with the positive x-axis. From q, we shoot a ray at angle $(2/3)\pi$ with the positive x-axis. Let v be the point of intersection of these two rays. By our construction, v must exist. Let R be the unique cubic section including the two sides $\overline{pv}$ and $\overline{qv}$; see Figure 3.4.

Since p and q are vertices of R, then R clearly contains p and q. The other points of P lie on a tangent of vertex p of R and so they do not lie in R, which is convex. The same is true of the other points of Q.

It is straightforward to map our points back to the plane $z = x + y$ and our cubic sections to the cubes that realize them. Therefore, applying Lemma 3.17 completes the proof of the theorem. □

3.2.3 Fixed-Size Cube Ranges

So far, the best known algorithm to multiply $\sqrt{n} \times \sqrt{n}$ Boolean matrices runs in time $O(n^{1.19})$ [LG14]. Theorem 3.18 suggests that it is unlikely that we can obtain a data structure with $O(n \operatorname{polylog} n)$ space and $O(\operatorname{polylog} n)$ query time for cube colored range counting. Such performance can be achieved, however, in the special case where the query cubes are restricted to be of a fixed size, as we describe next.

Theorem 3.19. *Fixed-size cube colored range counting in a set of n points reduces to three-dimensional dominance range counting in a set of $O(n)$ points.*

We sketch the proof of this theorem, which reuses techniques of the previous sections. Observe that the duality property stated as Observation 3.1 for the square case still applies: a point $p \in P$ is in a query cube R of "radius" r if and only if the center of R is in the cube of radius r centered at p. Thus, the influence region for color $c \in \mathcal{C}$ is the union of the radius-r cubes centered at the points in P_c. Boissonnat *et al.* [BSTY98] show that the union of fixed-sized axis-aligned cubes has linear complexity. It is then straightforward to generalize Lemmata 3.3 and 3.4, which allows us to reduce counting the number of influence regions stabbed by the center of a query cube to a constant number of three-dimensional dominance range counting queries on the $O(n)$ vertices of the influence regions.

3.2.4 Fixed-Size Four-Dimensional Hypercube Ranges

In Section 3.2.3 we showed that we can build an efficient data structure for fixed-size cube colored range counting. It would be interesting to see if we can derive a similar result for even higher dimensions. However, we give evidence that it may not be possible; even for four dimensions, there is a reduction from Boolean matrix multiplication to the offline variant of the problem.

Theorem 3.20. *Suppose there is an algorithm for offline four-dimensional hypercube colored range counting that runs in $\mathrm{T}(n, m)$ time where n is the number of points and m is the number of query ranges. Then, we can multiply two Boolean matrices of size $\sqrt{n} \times \sqrt{n}$ in $\mathrm{T}(2n, n) + O(n)$ time.*

Proof. We prove the theorem indirectly, by showing that rectangle colored range counting reduces to four-dimensional hypercube colored range counting. The rest of the reduction follows from the reduction of Kaplan et al. [KRSV08] from Boolean matrix multiplication to rectangle colored range counting.

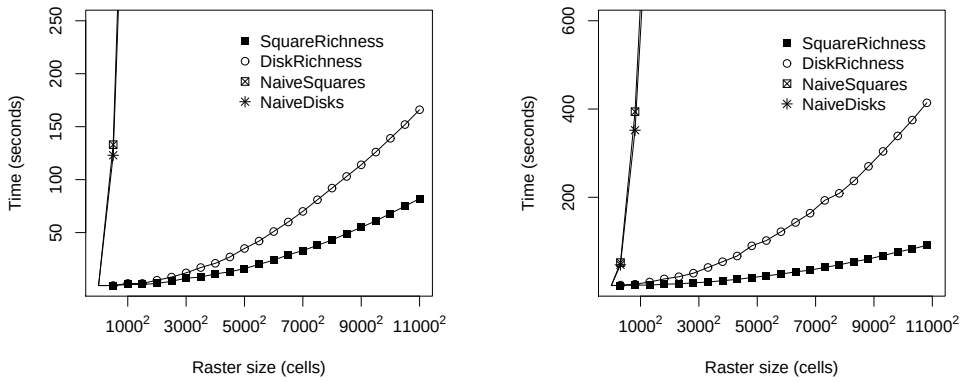
Consider a point $p = (p_x, p_y)$ and a rectangle R at the intersection of the halfspaces $x \geq \ell$, $x \leq r$, $y \geq b$, and $y \leq t$ for $\ell \leq r$ and $b \leq t$. Then $p \in R$ if and only if $p_x \geq \ell$, $p_x \leq r$, $p_y \geq b$, and $p_y \leq t$. We map p to the four-dimensional point $p' = (-p_x, p_x, -p_y, p_y)$ and map R to the four-dimensional orthant R' at the intersection of the halfspaces $x_1 \leq -\ell$, $x_2 \leq r$, $x_3 \leq -b$, and $x_4 \leq t$. Then $p \in R$ if and only if $p' \in R'$. We choose the fixed size of our query hypercube large enough so that we can simulate a four-dimensional orthant using a corner of a hypercube. □

3.3 Experimental Evaluation

We implemented algorithms *StripRichness* and *TileRichness* and we conducted experiments in order to measure their performance in practice. The implementations were developed in C++, and all experiments were run on a workstation with an Intel core i5-2430M CPU. This is a four-core processor with 2.40GHz per core. The main memory of the workstation was 7.8 GB. Our implementations ran on a Linux Ubuntu operating system, release 12.04.

We used two raster datasets. The first raster stores categorical data and was extracted from the Harmonized World Soil Database (version 1.2), a raster that maps soil types over the entire planet [FII+12]. The original raster consists of 43,200 × 21,600 cells, and each cell stores an integer in the range $[0, 32000]$ representing a soil type. From this dataset we extracted a smaller raster of 11,000 × 11,000 cells, representing an area that includes regions from Europe, Northern Africa, and Western Asia. We refer to the extracted raster as `soils`. The number of distinct category values in `soils` is 7,639. The second dataset we used is a DEM that represents the landscape around mountain Glacier Peak, Washington state. This dataset was acquired from the U.S. Geological Survey (USGS) online server [USGS] and consists of 10,812 × 10,812 cells. Each cell in this raster stores a floating-point value in the range $[25.46, 3284.32]$. We processed the raster by rounding the cell values to integers, yielding a raster with 3,259 different values. We refer to the resulting raster as `peak`.

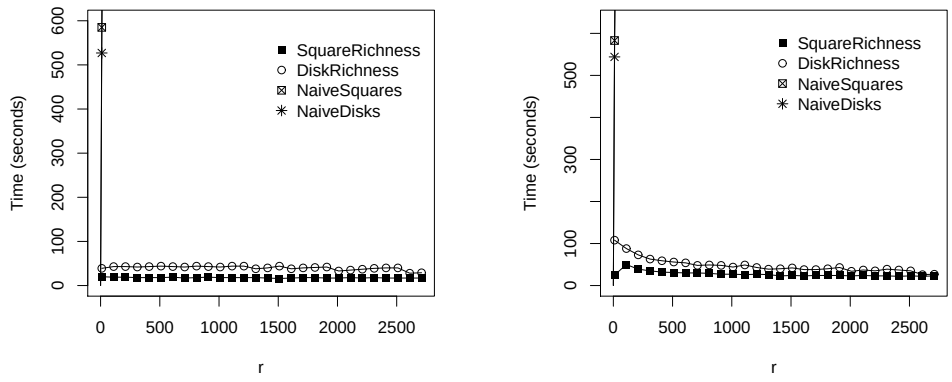
In the first experiment, we measured the running time of our implementations with respect to the size of the input raster, while using a fixed window size and a fixed maximum number of categories. In particular, for every integer $m \in [1, 22]$ we extracted from `soils` the raster that starts from the top-left corner of the dataset and consists of $(500m) \times (500m)$ cells. We then fixed the maximum number of distinct categories that



(a) (b)

Figure 3.5: The running times of our implementations on rasters of variable size, using fixed values for parameters $r = 50$ and $t = 100$. (a) The running times for rasters extracted from `soils` dataset. (b) The running times for rasters extracted from `peak` dataset.

appear in each extracted raster; we measured the minimum and the maximum value found in the raster, and we partitioned the interval defined by these two values into t smaller intervals of equal size. Then, each cell c was assigned a category value $i \in [t]$, if the original value of c belonged to the i-th of these intervals. In this experiment, we fixed the value of t to 100. We then ran our implementation of *SquareRichness* on each of these rasters with square windows of $(2r+1) \times (2r+1)$ cells, and *DiskRichness* with disks of radius r, using $r = 50$. Similarly, from dataset `peak` we extracted rasters of $(312 + 500m) \times (312 + 500m)$ cells for integer $m \in [0, 21]$. We ran our implementations also on these rasters, using the same values for parameters t and r. As reference, we also ran the experiments using two programs (one for square richness and one for disk richness) that compute the richness values in a naive manner in $O(nr^2)$ time. We refer to these two programs as *NaiveSquares* and *NaiveDisks*. The results for this experiment are illustrated in Figure 3.5. We observe that both *SquareRichness* and *DiskRichness* are outstandingly faster than the naive programs. We also see that *DiskRichness* performs worse on `peak` than on `soils`. This is possibly due to the distribution of category values in `soils`. Recall that *DiskRichness* spends $O(|\mathcal{C}_{\text{out}}|r+r^2)$ time to process a tile of $r/\sqrt{2} \times r/\sqrt{2}$ cells in the raster. Set $\mathcal{C}_{\text{out}}$ consists of the category values whose influence regions overlap with the tile, but no cell in the tile stores any of these values. Therefore, if the average size of $\mathcal{C}_{\text{out}}$ is very small among all tiles in the raster then the running time converges to $O(n)$. Indeed, for $r = 50$ the average size of $\mathcal{C}_{\text{out}}$ among the tiles in the entire `soils` raster is roughly five, while the corresponding



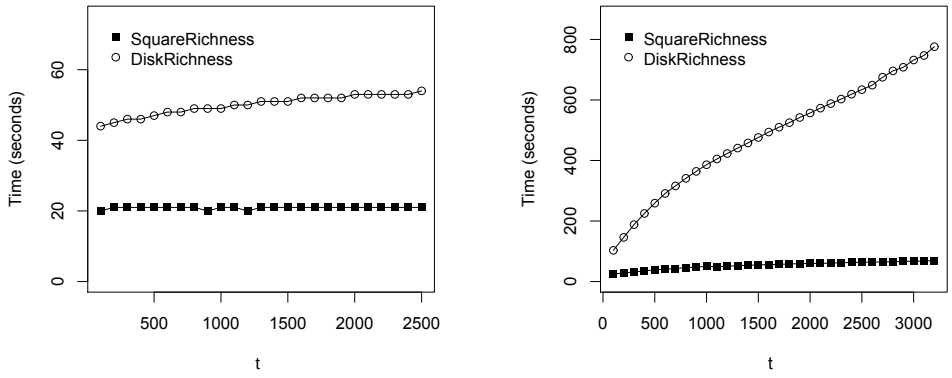
(a) (b)

Figure 3.6: The running times of our two implementations using windows of variable size on a raster of 5,500 × 5,500 cells and with $t = 100$. (a) The running times for a raster extracted from `soils` dataset. (b) The running times for a raster extracted from `peak` dataset.

number for `peak` is roughly 384.5.

In the second experiment, we measured the performance of our implementations using windows of variable size, while keeping fixed the size of the input raster and the maximum number of categories. More specifically, from each of our datasets we extracted a raster of 5,500 × 5,500 cells and we fixed the maximum number of categories in this raster to $t = 100$. We then ran our implementations on the resulting rasters for $r = 10 + 100 \times m$ for integer $m \in [0, 26]$. Figure 3.6 shows the results for this experiment. Again, *SquareRichness* and *DiskRichness* exhibit remarkable performance compared to the naive programs, even for the case where $r = 10$. For larger values of this parameter, each of the naive programs takes several hours to execute. On the other hand, the running times of our algorithms decrease as r becomes larger, especially for *DiskRichness*. This comes to no surprise since the time complexity of *DiskRichness* is $O(n(1 + t/r))$ in theory. The decrease is more evident for the `peak` dataset than for `soils`. This is possibly again due to the fact that in `soils` there is a much smaller number of categories per tile for which the algorithm computes the partial envelopes, leading to performance which depends almost entirely on n even for small values of r.

For the last experiment, we ran our implementations using different values of parameter t, and fixed values for n and r. We ran each of our algorithms on a raster of 5,500 × 5,500 cells extracted from `soils` with fixed $r = 50$ and $t = 100m$ for integer $m \in [1, 25]$. We also ran the algorithms on a raster of the same dimensions extracted



(a) (b)

Figure 3.7: The running times of our implementations for different values of parameter t, on a raster of size $n = 5{,}500 \times 5{,}500$ and with $r = 50$. (a) The running times for a raster extracted from `soils` dataset. (b) The running times for a raster extracted from `peak` dataset.

from `peak`, with $r = 50$ and $t = 100m$ for integer $m \in [1, 32]$ (the number of category values observed in the two rasters is different; this number is 2,569 for the raster taken from `soils` and 3,259 for the raster taken from `peak`). Figure 3.7 shows the results for this experiment. This figure does not include any benchmarks for the naive programs as each program took more than an hour to execute on a single input. We see that for the rasters produced from `peak` the performance of *DiskRichness* is affected by t significantly, although this algorithm still has very good performance. We see also a slight increase in the running time of *SquareRichness* for these datasets. For `soils`, the increase in t induces a small increase in the running time of *DiskRichness* and does not really affect *SquareRichness*.

We conclude that our algorithms for categorical richness are practically efficient and behave as expected by theory.

3.4 Concluding Remarks

We have given efficient algorithms for both square and disk categorical richness. Our square categorical richness algorithm runs in optimal $O(n)$ time. Our disk categorical richness algorithm runs in $O((1 + t/r)n)$ time. Whether or not it is possible to solve disk categorical richness in $O(n)$ time is open. In the disk algorithm, we compute

all of the partial envelopes for a tile in linear time. It is the rest of the algorithm—counting the number of colors of partial envelopes that each grid point stabs—that can spend more than linear time. There is potential for an efficient algorithm for the three-dimensional generalization of square categorical richness, which we might call cube categorical richness. In this problem, the input is a three-dimensional uniform grid of colored points and the output consists of the richness of the cube window around each point. Our techniques are sufficient to handle most of the problem in linear time, except for computing influence regions. In this case, an influence region is the union of axis-aligned fixed-size cubes.

We have also given data structures and conditional lower bounds for hypercube colored range counting. We show that square colored range counting reduces to five-dimensional dominance uncolored range counting. Since our goal was only to show that the problem can be solved in $O(n \operatorname{polylog} n)$ space and $O(\operatorname{polylog} n)$ query time, our reduction is sufficient. It is certainly possible to optimize the exponents of the $O(\operatorname{polylog} n)$ factors by using a more involved reduction; we presented a particularly simple reduction. Also, some of the $O(\log n)$ factors can be converted to $O(\log t)$ factors by applying shallow cuttings for three-dimensional dominance ranges [Afs08]. Our lower bounds are weak for two reasons: 1) they are conditional on the hardness of Boolean matrix multiplication, which is unsettled, and 2) they only apply to offline variants of our problems. It would be more desirable to, for example, prove a lower bound in the form of a trade-off between space and query time for the online variants of our problems.

By stretching the plane appropriately, our square color range counting data structure can be applied to handle ranges with any fixed aspect ratio. Mapping applications on mobile devices are prominent examples in which orthogonal range searching queries have fixed aspect ratios, due to the fixed viewports of mobile devices. Colored range counting is a problem in which the complexity seems to differ greatly depending on whether or not queries are restricted to a fixed aspect ratio. It is of interest to find other such problems that exhibit this behavior.

Chapter 4

Concurrent Range Reporting

Problems. We consider *concurrent range reporting* (CRR) in two-dimensional space. The dataset consists of t sets $P_1, ..., P_t$ of points in $\mathbb{R}^d$, where t is potentially $\omega(1)$. We want to preprocess the sets into a structure such that, given a query range Q and an arbitrary set $C \subseteq \{1, ..., t\}$ such that $r = |C|$, we can efficiently report all the points in $P_i \cap Q$ for each $i \in C$. See Figure 4.1. This is equivalent to performing traditional range searching on r sets of points *simultaneously*, thus raising the hope of improving query efficiency (compared to searching each set separately) by sharing some of the common computation. We consider axis-parallel rectangular ranges (i.e., orthogonal ranges) as well as halfspace ranges. Special versions of the orthogonal problem are *three-sided CRR* and *two-sided CRR* where Q is a rectangle of the form $[a_1, b_1] \times (-\infty, b_2]$ and $(-\infty, b_1] \times (-\infty, b_2]$, respectively.

The CRR problem is useful in several domains. An important example is *geographic information system* (GIS), which organizes a massive volume of spatial entities of different varieties, such as residential buildings, factories, hotels, restaurants, gas stations,

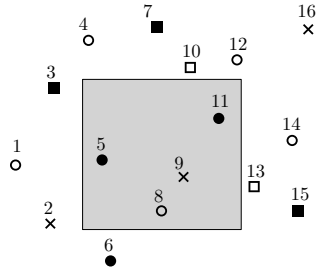


Figure 4.1: The dataset has $t = 5$ sets of points indicated with different types of markers. Let Q be the shaded rectangle and $C = \{\times, \bullet\}$. Only points 5, 9, and 11 should be reported.

schools, etc. One major use of a GIS is to offer a convenient way for a user to explore a map, by rendering the entities in the current navigation window of the user. A user is often given the option of turning on only some *thematic layers*. One such layer, for instance, can be the set of hotels whereas another layer may include all the restaurants, and so on. It is exactly a CRR query to find all and only the entities of the requested layers. An analogous application is a variant of *spatial keyword search*. Imagine that, in Figure 4.1, the points with an identical marker represent restaurants of the same company (e.g., $\times, \bullet, \circ, ...$ correspond to McDonald's, Pizza Hut, Burger King, ... respectively). A typical query is like "*find all restaurants of McDonald's and Pizza Hut in my neighborhood.*" This is exactly a CRR query with $Q = \{\times, \bullet\}$.

Define P as the union of $P_1, ..., P_t$. Set $n = \sum_{i=1}^{t} |P_i|$. In practice, n is often far greater than t. By standard tie-breaking techniques, we consider that no two points in P have the same x- (or y-) coordinate. To facilitate discussion, let us associate every point in P_i with a *color* i. Accordingly, C can be interpreted as a set of colors. Note, however, that CRR should not be confused with *colored range reporting*[1] [AGM02].

Previous results. The one-dimensional CRR problem was studied first by Chazelle and Guibas [CG86b] in their seminal paper on fractional cascading where they obtained a linear-size data structure with $O(\log n + r \log(t/r) + k)$ query time. Here, k is the number of reported points. They also proved a $\Omega(r \log(t/r))$ lower bound in a pointer machine model, although a peculiar aspect of the lower bound is that their representation of the query colors is not the same as the one used in their data structure so their lower bound does not in fact apply to their algorithm (see Section 4.1 for more details). By applying dynamic fractional cascading [CG86a, MN90], this method supports an update in $O(\log n)$ time, but the query cost becomes $O(\log n + r \log(t/r) \log \log n + k)$.

In two dimensions, a trivial solution is to issue r individual range reporting queries, one on each color in C. Using a structure of Chazelle [Cha86], this gives a data structure for the two-dimensional CRR problem that uses $O(n \log n / \log \log n)$ space and answers a query in $O(r \log(n/r) + k)$ time. If all queries are three-sided, the space can be lowered to linear by resorting to the priority search tree [McC85], while the query cost remains the same. The only non-trivial solutions for CRR in dimensions two and higher were given by Afshani et al. [AAL09] (see also [AAL10]). They considered a special instance of three-sided CRR, which has an additional constraint that C can be selected from only a number $m \leq 2^t$ of possible choices. They gave a linear-space structure that

[1] In colored range reporting, the underlying dataset is the same as in the CRR problem. Given a rectangle Q, a query retrieves the distinct *colors* of the points covered by Q. In other words, even if a color has multiple points in Q, the color should be reported only once. The CRR problem differs in that, in addition to a rectangle Q, a query also specifies a color set C of interest. Accordingly, the output of the query is also substantially different from that in colored range reporting, as can be verified easily.

answers a three-sided query in $O(mt + \log n + k)$ time. While this adequately serves the purposes of [AAL09], in our context where C can be any subset of $[t]$, the query time becomes $O(\log n + t \cdot 2^t + k)$.

Our results. Our main contribution is to show that almost optimal results can also be obtained for two-dimensional CRR problems. Our bounds involve some sublogarithmic functions which we now define. Let $\alpha_p(x)$ for $p \geq 1$ be a family of functions that are defined for $x \geq 1$. In the base case $p = 1$, $\alpha_1(x) = x - 2$. For $p > 1$, we let $\alpha_p(x) = \min\{i \mid i \geq 1 \wedge \alpha_{p-1}^{(i-1)}(x) \leq 2\}$. Here, $\alpha_{p-1}^{(i-1)}$ is the identity function composed with $i - 1$ instances of the α_{p-1} function. In this way, $\alpha_2(x) = \lceil n/2 \rceil$, $\alpha_3(x) = \max\{1, \lceil \log x \rceil\}$, $\alpha_4(x) = \max\{1, \lceil \log^* x \rceil\}$ and so on. Let $\alpha(x)$ be $\min\{p \mid \alpha_p(x) \leq 3\}$ so that it is asymptotically equivalent to the inverse Ackermann function [CLRS01].

Theorem 4.1. *Given n points in $\mathbb{R}^2$ divided into t disjoint sets, there exists a pointer machine data structure that answers three-sided CRR queries and has one of the following pairs of bounds (k is the size of the query result):*

1. *$O(n\alpha_p(n/t))$ space and $O(\log n + r \log(t/r) + k)$ query time, for any constant $p \geq 3$;*
2. *$O(n)$ space and $O(\log n + r(\log(t/r) + \log \alpha_p(n/t)) + k)$ query time, for any constant $p \geq 3$; or*
3. *$O(n)$ space and $O(\log n + r \log(t/r)\alpha(n/t) + k)$ query time.*

Each of these data structures require $O(\mathrm{S}(n) \log n)$ preprocessing time, where $\mathrm{S}(n)$ is the data structure's space bound.

Our data structure makes extensive use of a space-saving technique for three-sided range searching appearing as early as the work of [Ove88] that involves dividing the point set into slabs and building data structures for the points within each slab and a data structure on representative points from each slab.

Additionally, we obtain a number of other results that enhance our understanding of CRR problems: we consider two different methods of specifying the query colors as the previous papers had not done this in a satisfactory way (e.g., consider the peculiar aspect of Chazelle and Guibas's lower bound [CG86a]). We prove a significantly more involved lower bound and provide some other reductions that ultimately show the different formulations of the query colors are in fact equivalent. We also observe that the $\Omega(\log n + r \log(t/r) + k)$ query time lower bound also holds in the comparison model, which means our results are almost optimal in other fundamental models such as the real RAM model.

The data structure of Afshani et al. [AAL09] achieves linear space and $O(\log n + k)$ query time for the case $t \leq \log\log n - \log\log\log n$. If we use this data structure for small settings of t and the data structure of Theorem 4.1 for larger t, then we can rewrite the bounds of Theorem 4.1 so that, for $p \geq 4$, $\alpha_p(n/t)$ becomes $\alpha_p(t)$ and $\alpha(n/t)$ becomes $\alpha(t)$. This is because, for $p \geq 4$ and $t > \log\log n - \log\log\log n$, $\alpha_p(n/t) = O(\alpha_p(t))$ and $\alpha(n/t) = O(\alpha(t))$.

By standard techniques (range trees), Theorem 4.1 implies that a general (four-sided) CRR query can be settled with the same query cost, after increasing space by a factor of $O(\log n)$. When restricted to two-sided ranges, and also for halfspace ranges, we obtain optimal data structures with linear space and $O(\log n + r\log(t/r) + k)$ query time.

We sometimes need to perform a *concurrent lookup* on a perfectly balanced binary search tree (BST) $\mathcal{B}$ indexing t keys. Given a set C of sorted keys, each of which is indexed by $\mathcal{B}$, this operation finds the nodes in $\mathcal{B}$ corresponding to the keys in C. Chazelle and Guibas [CG86b] show that the minimum spanning tree of the $|C|$ output nodes in $\mathcal{B}$ has size $O(|C|\log(t/|C|))$. It is then straightforward to perform a concurrent lookup in $O(|C|\log(t/|C|))$ time by augmenting the tree with parent pointers and performing an Euler tour of this minimum spanning tree: at each step we progress towards the least query key that we have not yet found. Since the query keys are sorted, we always know the next least key to progress towards.

4.1 Query Representation and Lower Bounds

In this section, we discuss the representation of query colors as there is a potential for ambiguity when studying CRR problems in the pointer machine model. In fact this ambiguity has already revealed itself in the previous work of Chazelle and Guibas [CG86b]: when employing the fractional cascading technique, they assume that the query colors are given as pointers to special nodes of the pointer machine, but when proving the optimality of their technique, they assume that the query colors are given instead by indices. This means in their data structure, it is assumed that the query begins by having access to r pointers (thus r entry points into the data structure) but when proving the lower bound, they assume the query algorithm begins by having access to one entry point. The latter is a much more restrictive starting setup for the query algorithm which means their data structure does not in fact fit their lower bound framework!

We explicitly define these two different (but as it turns out equivalent) query representations. In the first representation, the set of query colors is a sorted set of indices. We consider this the *index query representation*. In an alternate representation, each color i is associated with some specific node c_i in the pointer machine structure called a

color node. Let $\mathcal{C} = \{c_1, c_2, \ldots, c_t\}$ be the set of all the color nodes. A set of query colors is specified by a set $C \subseteq \mathcal{C}$ (more precisely, C is a set of pointers to the color nodes) that is not necessarily sorted in any way. We call this the *pointer query representation*.

We first give $O(t)$-space and $O(r \log(t/r))$-time reductions between the index and pointer query representations. We then proceed to show that there is a lower bound of $\Omega(r \log(t/r))$ for both query representations. Thus, these representations are equivalent in the pointer machine model, and any data structure for CRR with query time $O(\log n + r \log(t/r) + k)$ is optimal in the pointer machine model.

4.1.1 Query Representation Reductions

Lemma 4.2. *Given a data structure D for CRR with the pointer query representation that requires T_D query time and S_D space, there exists a data structure for CRR with the index query representation that requires $O(S_D + t)$ space and $O(T_D + r \log(t/r))$ query time.*

Proof. We build a BST $\mathcal{B}$ on the t color nodes of D based on the sorted order of the colors. Then, given a sorted set of query colors C, we find their associated color nodes in $O(r \log(t/r))$ time via a concurrent lookup in $\mathcal{B}$. We then query D with the color nodes. □

Lemma 4.3. *Given a data structure D for CRR with the index query representation that requires T_D query time and S_D space, there exists a data structure for CRR with the pointer query representation that requires $O(S_D + t)$ space and $O(T_D + r \log(t/r))$ query time.*

Proof. We construct a color node for each color and store them in a BST $\mathcal{B}$ based on the sorted order of the colors. As shown by Chazelle and Guibas [CG86b], if we augment $\mathcal{B}$ with parent pointers, then we can construct the minimum spanning tree of a set of query color nodes in $O(r \log(t/r))$ time. In the same amount of time, we perform an in-order traversal of the minimum spanning tree to recover the query colors in sorted order. We then query D with the sorted set of query colors. □

4.1.2 Lower Bounds

For the index query representation, obtaining a comparison-based lower bound is simple: consider a dataset where each P_i has only one point and a query range that covers all of these points. For any specific r, there are $\binom{t}{r}$ size-r subsets of $[t]$, each of which defines a unique result. Since any query algorithm must be able to distinguish at least

as many results, it follows that $\Omega(\log \binom{t}{r}) = \Omega(r \log(t/r))$ comparisons are needed. This is essentially a lower bound for a problem we call the *concurrent lookup problem*: given a set K of keys, store them in a data structure such that given a set C of sorted query keys, each of which is in K, find the keys in K that correspond to the keys in C.

We now consider the pointer machine model. In this model, the memory is a directed graph with out degree of two (the model generalizes easily to any constant out degree as well) and each node of the graph can store one input element. To output an element, the query algorithm must navigate the pointers to reach a cell that contains the element. While Chazelle and Guibas proved a pointer machine lower bound [CG86b], their lower bound assumes the colors are given as indices rather than pointers. In this way, the CRR problem inherits the difficulty of the concurrent lookup problem, which makes proving a lower bound easier. Instead, we consider the pointer query representation, which means the query algorithm begins with access to r entry points to the data structure; this is in contrast to the traditional pointer machine data structures in which the query algorithm begins from the root of the data structure. In fact, because of this, we cannot rely only on the difficulty of the concurrent lookup problem, and we must additionally consider the geometry of the problem. Essentially, we space out many instances of the concurrent lookup problem and show that the color nodes cannot adequately help all instances of the problem. While simple counting arguments work to establish a lower bound when starting from one pointer, they seem to be ineffective when starting from $|Q|$ pointers. When the query algorithm begins from one pointer, it can only access $2^{O(x)}$ different subsets of the memory cells by making x pointer jumps so to be able to access $\binom{t}{r}$ sets we must have $2^{O(x)} \geq \binom{t}{r}$ or $x = \Omega(r \log(t/r))$. On the other hand, if we give the query algorithm access to r pointers, there are $\binom{t}{r}$ ways just to pick our entry points and so the counting argument is rendered completely ineffective. To establish a lower bound for the pointer query representation, we have to work harder.

Theorem 4.4. *Any pointer machine data structure for CRR with the pointer query representation that uses $(n/t)^{O(1)}$ space, requires $\Omega(\log n + r \log(t/r) + k)$ query time.*

Proof. Our bad input instance is a one-dimensional point set composed of n/t sets of size t. Each set only contains one point of each color and the points in the i-th set are laid consecutively in an *interval* I_i; the intervals are disjoint and they are also used as geometric ranges for queries. Thus, for a set $C \subseteq \mathcal{C}$ of color nodes, the output size for the query range I_i is exactly $r = |C|$. In our proof, the number of colors for all queries will be fixed but to avoid introducing extra variables, we will continue to use r to refer to this fixed value.

If $\log n \geq r \log(t/r)$ then there is nothing left to prove since there is a trivial $\Omega(\log n)$ lower bound in the pointer machine model. In the rest of this proof, we assume otherwise. Let G be the directed graph that corresponds to the memory layout of the

data structure: the vertices of G are memory cells and edges correspond to the pointers between the memory cells. G has t cells that correspond to the color nodes; the query algorithm begins the search from these cells. We say a memory cell (in G) is *shallow*, if it lies at a depth less than $\log(n/t) - 2$ of a color node. We say an interval I is *shallow*, if there are at least $t/2$ shallow cells that store points in I. Since the number of shallow cells is less than $t \cdot 2^{\log(n/t)-2} = n/4$, there are less than $n/(2t)$ shallow intervals. We will only use non-shallow intervals for our queries so one can safely ignore the shallow intervals in the rest of this proof.

Given a non-shallow interval I, consider the query algorithm: it starts with access to r pointers and then continues to make pointer jumps in graph G until all the cells containing the output have been accessed. We ignore the part of the output that is stored in shallow cells so for the rest of this proof, by output we mean only the points that are not stored in shallow cells (in other words, we only require the query algorithm to output elements stored in non-shallow cells).

Consider the subgraph H of G composed of memory cells that are explored at the query time and the edges that are used to visit these cells for the first time. Except for the color nodes, each vertex of H has in-degree of exactly one (each color node has in-degree zero). So, H is a forest with r arborescences[2] such that each color node in C is the root of one arborescence. H contains at most r cells that store the output points of the query. We call these the *output cells*. We denote the number of output cells in an arborescence $\mathcal{T}$ with $\mathrm{k}(\mathcal{T})$. Let αr be the size of H which is also a lower bound on the query time. We call a subgraph of G a β-heavy hub of size s if it is an arborescence of size s with β output cells. Intuitively, our proof idea is to combine two things: one, that for every query, the subgraph H contains at least one β-heavy hub of size $O(\alpha\beta)$ and two, that the number of β-heavy hubs of size $O(\alpha\beta)$ that the graph G can use at the query time depends on and grows as a function of α.

We set $\beta = c\log(n/t)/\log(t/r)$, where c is a constant to be determined later. Consider an arborescence $\mathcal{T}$ in H with $\mathrm{k}(\mathcal{T}) > 0$. Observe that $\mathcal{T}$ must have at least $\log(n/t) - 2$ vertices (since its output cells are not shallow). If a fraction (say a quarter) of all the output elements lie in arborescences $\mathcal{T}$ such that $\mathrm{k}(\mathcal{T}) \leq 2\beta$, then we are good: each such arborescence uses at least an average of $\log(t/r)/(2c)$ pointer jumps to output one element and thus the query time is already $\Omega(r\log(t/r))$. Thus, assume otherwise which means we can afford to remove any arborescences $\mathcal{T}$ with $\mathrm{k}(\mathcal{T}) \leq 2\beta$ and still have three quarters of the output elements left. After this, since the total size of all the arborescences were αr, it easily follows that there exists at least one arborescence $\mathcal{T}$ such that $|T| \leq 2\alpha\,\mathrm{k}(\mathcal{T})$ with $\mathrm{k}(\mathcal{T}) \geq 2\beta$. In this case, by using the same technique as in [Afs12], we can find at least one β-heavy hub of size $O(\alpha\beta)$.

[2] An arborescence is a directed tree in which there is a directed path from the root to every other node, or informally, a directed tree in which the edges are directed away from the root.

We now look at the overall structure of graph G. While the definition of a β-heavy hub only makes sense with respect to a given query, the structure of the hub must still be embedded in graph G. This means that G has only a limited number of different β-heavy hubs of size $O(\alpha\beta)$. To bound this number, we can pick a cell v in graph G, consider $O(\alpha\beta)$ pointer jumps that visit $O(\alpha\beta)$ other memory cells in G and then pick β cells out of all the visited cells to be output cells. After considering all the different possible pointer jumps, and all the different ways to pick β cells out of $O(\alpha\beta)$ cells, it is a simple exercise (for more details see [Afs12]) to show that the number of different possible β-heavy hubs of size $O(\alpha\beta)$ in G is $\mathrm{S}(n)2^{O(\alpha\beta)}$ where $\mathrm{S}(n)$ is the total space used by the data structure ($\mathrm{S}(n)$ is in fact the number vertices in graph G). Now, remember that we have at least $n/(2t)$ non-shallow intervals so there exists a non-shallow interval I such that there are at most $\mathrm{S}(n)2^{O(\alpha\beta)}t/n$ possible β-heavy hubs of size $O(\alpha\beta)$ that involve cells that store points of I.

We fix I as the query range and it remains to define the color set. Since I has only one point of each color, output points and query colors are interchangeable (i.e., we can determine C by a query's set of output points). To pick the set of output points, we consider the points of I that are not stored in shallow cells; by the non-shallowness of I, there are at least $t/2$ such points. C is picked by randomly sampling each such point with probability $2r/t$ (remember that r is a fixed parameter). The number of points that we consider is at least $t/2$, so we expect to sample r colors. Now, consider a possible β-heavy hub h. If the point stored at one of the output cells of h is not sampled, then h is not useful since it cannot become a β-heavy hub for our query. Thus, the probability that h is useful for our query is at most $(2r/t)^\beta$. This implies that the expected number of β-heavy hubs that can be useful for the query is at most

$$\left(\frac{2r}{t}\right)^{\beta} \mathrm{S}(n)2^{O(\alpha\beta)}\frac{t}{n}.$$

Since $\beta = c\log(n/t)/\log(t/r)$, we have $(r/t)^\beta = 2^{-c\log(n/t)} = (n/t)^{-c}$. To be able to answer the query, the query should have at least one such β-heavy hub, implying

$$\left(\frac{t}{n}\right)^{c-1} \mathrm{S}(n)\left(\frac{n}{t}\right)^{O\left(\frac{c\alpha}{\log(t/r)}\right)} \geq 1.$$

Since $\mathrm{S}(n) = (n/t)^{O(1)}$, by setting c to a large enough constant, it follows that we must have $\alpha = \Omega(\log(t/r))$. □

Corollary 4.5. *For $t \leq \sqrt{n}$, any pointer machine data structure for CRR with the pointer query representation that uses polynomial space, requires $\Omega(\log n + r\log(t/r) + k)$ query time.*

4.2 Two-Sided and Halfspace CRR

Before looking at two-dimensional CRR, consider the concurrent version of predecessor search. In this problem we must preprocess t sets $P_1, \ldots, P_t$ of elements so that we can efficiently find the predecessor of a query element q in P_i for each i in a set of query colors $C \subseteq [t]$ such that $r = |C|$. The solution to one-dimensional CRR of Chazelle and Guibas [CG86b] also solves concurrent predecessor search in linear space and optimal $O(\log n + r \log(t/r))$ time.

Assume we have a data structure for traditional range reporting that, in order to answer a query, performs a single one-dimensional predecessor search and then uses $O(k)$ additional time to report all necessary points. Supposed further that the query element used in the one-dimensional predecessor search depends only on the original query range (i.e., it does not depend on the points stored in the data structure). Observe that we can then build an efficient data structure for CRR by building this traditional data structure for each color, but performing the predecessor searches for each query color concurrently. This concurrent predecessor search requires $O(\log n + r \log(t/r))$ time, and all subsequent work is bounded by $O(k)$.

Lemma 4.6. *There exists a linear-space data structure for two-sided range reporting that requires, for any query, a single predecessor search that depends on only the query and $O(k)$ additional time. The data structure requires $O(n \log n)$ preprocessing time.*

Proof. Given a point set P, let $\ell_1, \ldots, \ell_m$ be all m non-empty layers of minima of P, such that ℓ_i contains the minima of $P \setminus \bigcup_{j=1}^{i-1} \ell_j$. We store the points of each layer in a linked list in order of increasing x-coordinate (or equivalently, decreasing y-coordinate). For each layer ℓ_i, we build a catalog S_i containing the points of ℓ_i keyed by their x-coordinates. We connect the catalogs into a path ordered by layer index to create a catalog graph that supports fractional cascading. Preprocessing time is dominated by the construction of the layers of minima in $O(n \log n)$ time by a simple sweep-line algorithm.

Given a query with range $Q = (-\infty, x_0] \times (-\infty, y_0]$, we report points layer by layer, starting at ℓ_1, until we reach a layer ℓ_i such that $\ell_i \cap Q = \emptyset$. Since the points of ℓ_i dominate the points of all further layers, for any further layer ℓ_j, it must also be that $\ell_j \cap Q = \emptyset$. When we consider some layer ℓ_i, we must report $\ell_i \cap Q$. In order to do so, we search for the point p associated with the predecessor of x_0 in S_i. If there is no such point, then $\ell_i \cap Q = \emptyset$ and we are done. Otherwise, p is the lowest point of ℓ_i that is to the left of x_0. We report points of the linked list for ℓ_i, starting from p, until we reach a point with y-coordinate greater than y_0.

We need to search for the predecessor of x_0 in a path of at most $O(k)$ catalogs in our catalog graph. By fractional cascading, this requires a single predecessor search for x_0 in the augmented catalog for S_1, followed by $O(k)$ additional work. □

Lemma 4.6 implies an optimal two-sided CRR pointer machine structure with linear space and $O(\log n + r\log(t/r) + k)$ query time that can be built in $O(n \log n)$ time. There is also a linear-space data structure for two-dimensional halfspace range reporting that performs a single predecessor search followed by $O(k)$ additional work. It is similar to the data structure of Lemma 4.6, but uses convex layers instead of layers of minima. Convex layers can also be constructed in $O(n \log n)$ time [Cha85]. Finding a point on a convex layer in a given halfspace reduces to predecessor search over the slopes of the edges of the convex layer, using the slope of the halfspace boundary line as the query element. Therefore, there is a linear-space data structure for two-dimensional halfspace CRR with $O(\log n + r\log(t/r) + k)$ query time that can be built in $O(n \log n)$ time.

4.3 Three-Sided CRR

This section concerns the proof of Theorem 4.1. Recall that the dataset P has n points in $\mathbb{R}^2$, each of which is associated with a color $i \in [t]$. A three-sided CRR query specifies a rectangle $Q = [a_1, b_1] \times (-\infty, b_2]$ and a color set $C \subseteq [t]$, and retrieves $P_i \cap Q$ for each $i \in C$. As mentioned at the beginning of this chapter, there is a simple linear-space data structure for three-sided CRR based on priority search trees that requires suboptimal $O(r\log(n/r) + k)$ query time. Priority search trees can be built in $O(n \log n)$ time, which is thus the data structure's preprocessing time. Note, that in the special case where $n = O(t)$, this naive data structure is actually optimal. We will use this data structure as the base case for a few recursive structures. We will also use the two-sided solution of Section 4.2 as a black box. For each new data structure in this section, the analysis of the data structure's preprocessing time is identical to that of its space consumption, except that the preprocessing bound has an extra multiplicative $O(\log n)$ factor. This extra factor is simply propagated from the naive three-sided data structures and optimal two-sided data structures that we use as black boxes.

Lemma 4.7. *There exists a data structure for three-sided CRR requiring $O(n\log(n/t))$ space and $O(\log n + r\log(t/r) + k)$ query time.*

Proof. As mentioned in Section 4.2, using a range tree and our optimal two-sided solution, we can create an $O(n \log n)$-space data structure for three-sided queries with optimal query time. By modifying the tree so that it has fat leaves of size $O(t)$, and building the naive data structure for three-sided CRR in these leaves, we reduce space to $O(n\log(n/t))$ and retain optimal query time. □

Note that the data structure of Lemma 4.7 requires $O(n\alpha_3(n/t))$ space. We now give a technique that reduces space from $O(n\alpha_p(n/t))$ to $O(n\alpha_{p+1}(n/t))$.

Theorem 4.8. *There exists a data structure for three-sided CRR that requires* $O(np \cdot \alpha_p(n/t))$ *space and* $O(\log n + rp \log(t/r) + k)$ *query time, for any* $p \geq 3$.

The data structure of Theorem 4.8 for parameter $p = q$ recursively uses the data structure for $p = q - 1$. Let $\mathrm{S}_p(n)$ and $\mathrm{T}_p(n)$ be the space and query time, respectively, required by our data structure for parameter p. In the base case $p = 3$, we simply use the data structure of Lemma 4.7, so we have $\mathrm{S}_3(n) = O(n\alpha_3(n/t))$ and $\mathrm{T}_3(n) = O(\log n + r \log(t/r) + k)$, as required. We now wish to prove the theorem for $p = q > 3$.

We construct a range tree $\mathcal{T}$ on the x-coordinates of points so that each node u is associated with an x-range $I(u)$ and the x-ranges of the children of u partition $I(u)$ such that an equal number of points of P lie in each child x-range. The children of u are ordered based on the natural ordering of their x-ranges. Let P_u be the points of P which lie in $I(u)$. Our range tree $\mathcal{T}$ has a different fanout parameter at each level. If the root of $\mathcal{T}$ is at level 1, then a node u at level i is associated with an x-range $I(u)$ that contains $t\alpha_{q-1}^{(i-1)}(n/t)$ points. Our range tree $\mathcal{T}$ has fat leaves of size at most $2t$ at level $\alpha_q(n/t)$. This range tree can also be viewed as a recursive construction: we divide the n points into columns of size $t\alpha_{q-1}(n/t)$ and recurse in each column.

In each fat leaf u, we store the naive three-sided CRR data structure on P_u, which is optimal for $|P_u| = O(t)$ points. In each non-root node u we build a two-sided CRR data structure D_u^- on P_u for queries of the form $(-\infty, b_1] \times (-\infty, b_2]$ and (by mirroring P_u) another data structure D_u^+ for queries of the form $[a_1, \infty) \times (-\infty, b_2]$. Now, consider an internal node u. We build a BST $\mathcal{B}_u$ on the set of left boundaries of the x-ranges of the children of u. Finally, we build a data structure $D_u^{||}$ on P_u for three-sided ranges of the form $[a_1, b_1) \times (-\infty, b_2]$ where $a_1, b_1 \in \mathcal{B}_u$. We call these *aligned* queries.

Lemma 4.9. *Given a data structure* D *that uses* $\mathrm{S}_D(n)$ *space and answers three-sided CRR queries in* $\mathrm{T}_D(n)$ *time, there is a data structure that supports aligned queries in* n/s *columns of* s *points and uses at most* $\mathrm{S}_D(nt/s) + O(n)$ *space and answers queries in* $\mathrm{T}_D(nt/s) + O(k)$ *time.*

Proof. In each column i, we store all points of the same color j in a linked list $L_i(j)$ so that the points are in ascending order of their y-coordinates. Since we do not store lists for colors which are not represented in a column, this requires $O(n)$ space. We call the first point (i.e., the one with the smallest y-coordinate) of each list $L_i(j)$ a head point. See Figure 4.2. We store all head points in data structure D, which requires at most $\mathrm{S}_D(nt/s)$ space, since each column has at most t head points.

Given an aligned query with range $Q = [a_1, b_1] \times (-\infty, b_2]$ and color set C, we forward the query to the data structure containing only head points, so that in $\mathrm{T}_D(nt/s)$ time we have reported all appropriate head points. Consider any non-head point that should be reported. Then, every prior point in its linked list $L_i(j)$ must also be reported. The

points of $L_i(j)$ that lie in Q thus form a prefix of $L_i(j)$. We know that the head point for $L_i(j)$ was reported. So, for every head point that was reported, we scan its linked list and report every point we see until we reach a point that is outside of Q. We charge this additional work to the output size. □

The data structure $D_u^{||}$ is that of Lemma 4.9, where $s = t\alpha_{q-1}^{(i)}(n/t)$ and D is the data structure of Theorem 4.8 with parameter $p = q - 1$.

For the space analysis, we strengthen our claimed space bound to $\mathrm{S}_p(n) \leq cnp \cdot \alpha_p(n/t)$ for a sufficiently large constant c. Our induction hypothesis gives that $\mathrm{S}_{q-1}(n) \leq cn(q-1)\alpha_{q-1}(n/t)$. Consider a single level i of $\mathcal{T}$. Each node u in level i requires at most $\mathrm{S}_{q-1}(|P_u|t/s) + c|P_u|$ space. Here the $c|P_u|$ term includes the linked lists of Lemma 4.9, as well as the data structures D_u^-, D_u^+, and $\mathcal{B}_u$. By the induction hypothesis and simple arithmetic, the total space for u is less than or equal to $c|P_u|q$. The total space required at level i is thus at most cnq, since every point in P lies in the x-range of exactly one node at level i. Since $\mathcal{T}$ has $\alpha_q(n/t)$ levels, the total space for $\mathcal{T}$ is $cnq\alpha_q(n/t)$, as required.

We now show how queries are answered. Given a query with range $Q = [a_1, b_1] \times (-\infty, b_2]$ and color set C, we first find the lowest node u in $\mathcal{T}$ such that $a_1, b_1 \in I(u)$. We do so by predecessor search for a_1 and b_1 in $\mathcal{B}_v$ for each ancestor v of u, starting from the root of $\mathcal{T}$. This process takes $O(\sum_v (1 + \log(|\mathcal{B}_v|))) = O(\alpha_q(n/t) + \log n) = O(\log n)$ time.

If u is a leaf, we query the naive three-sided data structure stored at u with range Q and color set C and we are done in $O(r \log(t/r) + \log n + k)$ time. If u is an internal node, then we know that u has two different children v_a and v_b such that $a_1 \in I(v_a)$ and $b_1 \in I(v_b)$. Let a_1' be the left boundary of the x-range of the right sibling of v_a and let b_1' be the left boundary of $I(v_b)$. We decompose Q into three subranges $Q^+ = [a_1, a_1') \times (-\infty, b_2]$, $Q^{||} = [a_1', b_1') \times (-\infty, b_2]$, and $Q^- = [b_1', b_1] \times (-\infty, b_2]$, as shown in Figure 4.2.

Amongst the points of P_{v_a}, the range Q^+ contains the same points as $[a_1, \infty) \times (-\infty, b_2]$, thus we handle this subrange in $O(\log n + r \log(t/r) + k)$ time via a two-sided query to $D_{v_a}^+$. We handle Q^- symmetrically with a two-sided query to $D_{v_b}^-$. Finally, we handle $Q^{||}$ by recursing in $D_u^{||}$ with the range $Q^{||}$. At each level of recursion we perform $O(\log n + r \log(t/r))$ work that cannot be charged to the output size. So, the total query time is $O(p(\log n + r \log(t/r)) + k)$.

We can reduce the $O(p \log n)$ term to $O(\log n)$ by fractional cascading. This term comes from two $O(\log n)$ time algorithms that we invoke at each of the $O(p)$ levels of recursion: first, finding the node u in $\mathcal{T}$ at which we can decompose Q into an aligned query and two-sided queries, and second, the two-sided CRR queries. Both of these algorithms amount to predecessor search for a_1 and b_1 in different sets of elements. In

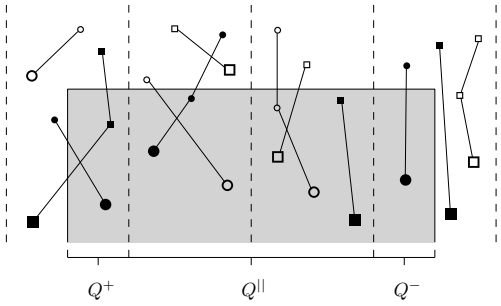


Figure 4.2: The points of P_u are partitioned into columns. Head points (depicted large) are the lowest points of each linked list. The query range is decomposed into two-sided queries in the outer columns and a single aligned query.

finding node u we search for a_1 and b_1 in $\mathcal{B}_v$ for each ancestor v of u. The two-sided CRR query algorithms search for a_1 or b_1 in layers of minima.

We construct a global catalog graph for all $O(p)$ levels of recursion. Each node u in a tree $\mathcal{T}$ at recursion level q is associated with a catalog S_u that contains the elements of $\mathcal{B}_u$. If u is the root of $\mathcal{T}$, then S_u is linked to S_v for some node v in recursion level $q+1$ that uses $\mathcal{T}$ in $D_v^{||}$. If u is not the root of $\mathcal{T}$, then S_u is instead linked to S_v where v is the parent of u. If u is an internal node, then S_u is linked to S_v where v is the root of a tree at recursion level $q-1$ in $D_u^{||}$ as well as S_w for each child w of u. In this case, S_u is also linked to the catalogs for the first layers of minima in D_u^+ and D_u^-. Our catalog graph thus has locally bounded degree of 5: although there are a non-constant number of links to children, they only contribute to an increase of the locally bounded degree by one since the ranges of the catalogs of the children are all disjoint.

An issue that we have thus far ignored is that, when we recurse, we change the query x-coordinates from a_1 and b_1 to a_1' and b_1'. Typically, in fractional cascading, one searches for the same query elements in every catalog. However, it is straightforward to extend fractional cascading without penalty so that it is possible to switch the query value to the result of a previous search in an adjacent catalog. When we recurse, this is precisely what we need to do: a_1' and b_1' are the results of searches in $\mathcal{B}_u$ (now S_u) and these values become the new query values in the root of the tree for $D_u^{||}$.

Now, we must count the number of catalogs that we visit to give a bound on running time. At each level of the tree at each recursion level, there are a constant number of catalogs that we search and cannot charge to the output size. Thus, the $O(p \log n)$ term is reduced by fractional cascading to $O(\log n) + \sum_{q=4}^{p} O(\alpha_q(n/t)) = O(\log n)$. This completes the proof of Theorem 4.8.

By fixing p to a constant in Theorem 4.8, we obtain the first trade-off of Theorem 4.1. To get the other two trade-offs, we need one final step to reduce space to $O(n)$.

Theorem 4.10. *There exists a data structure for three-sided CRR that requires $O(n)$ space and $O(\log n + r(p\log(t/r) + \log(p\alpha_p(n/t))) + k)$ query time, for any $p \geq 3$.*

Proof. We divide our set of points P into columns of $O(tp\alpha_p(n/t))$ points. To handle queries that lie entirely within one of these columns, we build the naive data structure for three-sided queries in each column. These data structures require linear space and $O(r\log(tp\alpha_p(n/t)/r) + k) = O(r(\log(t/r) + \log(p\alpha_p(n/t))) + k)$ query time. To handle queries that do not lie entirely within one column, we reuse the technique of Theorem 4.8 to decompose the query into two-sided queries in two columns and an aligned query. By Lemma 4.9, where $s = O(tp\alpha_p(n/t))$ and D is the data structure of Theorem 4.8, we handle aligned queries in linear space and $O(\log n + rp\log(n/t) + k)$ query time. □

By fixing p to a constant in Theorem 4.10, we obtain the second trade-off of Theorem 4.1. Finally, by setting $p = \alpha(n/t)$, we obtain the third trade-off. This completes the proof of Theorem 4.1.

4.4 Concluding Remarks

We have given a pointer machine data structure for three-sided CRR that requires slightly superlinear space and optimal query time. It is open whether a linear-space data structure can obtain optimal $O(r\log(t/r) + \log n + k)$ query time. The extra factor in our space bound is tiny and it may be an artifact of our specific technique. Our solution does not apply shallow cuttings and, since they are such a powerful tool for three-sided queries, it may be that there is some application of shallow cuttings that yields a linear-space data structure with optimal query time.

The only technique we have so far to support general four-sided CRR queries involves using a range tree to reduce four-sided queries to a pair of three-sided queries at the cost of a multiplicative $O(\log n)$ factor in space. The tight bound on space for traditional range reporting is $\Theta(n\log n/\log\log n)$ [Cha90]. By a simple reduction, the $\Omega(n\log n/\log\log n)$ lower bound on space carries over to CRR. So, the use of a binary range tree to support four-sided CRR brings the data structure's space bound a multiplicative $\Omega(\log\log n)$ factor further from the lower bound. To reduce space, we can increase the fanout of the range tree, say to $\log^\epsilon n$, but it is not clear how to support CRR queries efficiently in a $(\log^\epsilon n)$-ary range tree. The exact same problem arises in colored range reporting, which gives evidence that it might only be possible to save the $\Omega(\log\log n)$ factor in space for uncolored range reporting.

Another problem for which a concurrent variant is interesting is point location. In concurrent point location we must preprocess t subdivisions of the plane $S_1, S_2, \ldots, S_t$. Given a query point q and a set of colors $C \subseteq [t]$, we must locate q in S_i for each

$i \in C$. This problem is closely related to a two-dimensional generalization of fractional cascading studied by Chazelle and Liu [CL04]. Chazelle and Liu show that no efficient bounds can be achieved for the two-dimensional generalization of fractional cascading without using at least quadratic space. However, the hard instance in their lower bound uses non-orthogonal subdivisions of the plane. In the special case of concurrent orthogonal point location, there is a simple linear-space data structure that requires $O(\log n + t + k)$ query time: decompose the subdivisions into rectangles and store these rectangles in the rectangle-stabbing data structure of Chazelle [Cha86]. It is open whether the t term in this query time can be reduced to $O(r \log(t/r))$ as in our CRR data structures. If we can develop a space-saving technique for concurrent point location similar to the technique of Theorem 4.8 for CRR, we can at least match the bounds for three-sided CRR.

In the word RAM model, there are still one-dimensional concurrent problems that are open. Since traditional one-dimensional range reporting can be solved in linear space and $O(1 + k)$ query time one-dimensional CRR can be solved in linear space and optimal $O(r + k)$ query time [ABR01], there is a trivial linear-space data structure for one-dimensional CRR with optimal $O(r + k)$ query time. However, the same cannot be said for the concurrent predecessor search problem, since traditional predecessor search in the RAM model using polynomial space requires super-constant query time [PT07]. A trivial linear-space data structure obtains $O(r \cdot \text{pred})$ query time, where pred is the cost of traditional predecessor search in the word RAM model. It is open whether concurrent predecessor search can be solved in linear space and $O(\text{pred} + r)$ query time.

References

[AAL09] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

[AAL10] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting: Query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

[ABR00] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[ABR01] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[Afs08] Peyman Afshani. On dominance reporting in 3D. In *European Symposium on Algorithms (ESA)*, pages 41–51, 2008.

[Afs12] Peyman Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Symposium on Computational Geometry (SoCG)*, pages 339–346, 2012.

[AGM02] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *European Symposium on Algorithms (ESA)*, pages 17–28, 2002.

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Symposium on Foundations of Computer Science (FOCS)*, pages 534–544, 1998.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[AT07] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM (JACM)*, 54(3):1–40, 2007.

[Aur91] Franz Aurenhammer. Voronoi diagrams—A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[BABO+14] José Miguel Barea-Azcón, Blas M. Benito, Francisco J. Olivares, Helena Ruiz, Javier Martín, Antonio L. García, and Rogelio López. Distribution and conservation of the relict interaction between the butterfly Agriades zullichi and its larval foodplant (Androsace vitaliana nevadensis). *Biodiversity and Conservation*, 23(4):927–944, 2014.

[BCA13] Blas M. Benito, Luis Cayuela, and Fabio S. Albuquerque. The impact of modelling choices in the predictive performance of richness maps derived from species-distribution models: Guidelines to build better diversity models. *Methods in Ecology and Evolution*, 4(4):327–335, 2013.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, 1975.

[Ben80] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM (CACM)*, 23(4):214–229, 1980.

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences (JCSS)*, 65(1):38–72, 2002.

[BM72] Rudolf Bayer and Edward M. McCreight. Efficient worst-case data structures for range searching. *Acta Informatica*, 1(3):173–189, 1972.

[BSTY98] Jean-Daniel Boissonnat, Micha Sharir, Boaz Tagansky, and Mariette Yvinec. Voronoi diagrams in higher dimensions under certain polyhedral distance functions. *Discrete & Computational Geometry*, 19(4):485–519, 1998.

[CG86a] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[CG86b] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.

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

[Cha86] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing (SICOMP)*, 15(3):703–724, 1986.

[Cha88] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing (SICOMP)*, 17(3):427–462, 1988.

[Cha90] Bernard Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[CL04] Bernard Chazelle and Ding Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *Journal of Computer and System Sciences (JCSS)*, 68(2):269–284, 2004.

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

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[dABB+15] Fábio Suzart de Albuquerque, Blas Benito, Paul Beier, Maria José Assunção-Albuquerque, and Luis Cayuela. Supporting underrepresented forests in Mesoamerica. *Natureza & Conservação*, 2015. doi: 10.1016/j.ncon.2015.02.001.

[DS87] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[FII+12] FAO, IIASA, ISRIC, ISSCAS, and JRC. Harmonized World Soil Database (version 1.2), 2012. FAO, Rome, Italy and IIASA, Laxenburg, Austria.

[FW90] Michael L. Fredman and Dan E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[GJS95] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, 1995.

[GJS05] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Computational geometry: Generalized intersection searching. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2005.

[Han04] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[HR09] James J. Hayes and Scott M. Robeson. Spatial variability of landscape pattern change following a ponderosa pine wildfire in northeastern New Mexico, USA. *Physical Geography*, 30(5):410–429, 2009.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing (SICOMP)*, 13(2):338–355, 1984.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 417–431, 1981.

[JL93] Ravi Janardan and Mario Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry & Applications*, 3(1):39–69, 1993.

[JMS05] Joseph JaJa, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 558–568, 2005.

[KRSV08] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008.

[KS96] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Symposium on Parallel and Distributed Processing (SPDP)*, pages 169–176, 1996.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

[LvW13] Kasper Green Larsen and Freek van Walderveen. Near-optimal range reporting structures for categorical data. In *Symposium on Discrete Algorithms (SODA)*, pages 265–277, 2013.

[LW77] Der-Tsai Lee and C.K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing (SICOMP)*, 14(2):257–276, 1985.

[MCE12] Kevin McGarigal, Sam A. Cushman, and Eduard Ene. *FRAGSTATS v4: Spatial Pattern Analysis Program for Categorical and Continuous Maps*. University of Massachusetts, Amherst, 2012. Available at: `http://www.umass.edu/landeco/research/fragstats/fragstats.html`.

[MN90] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[MNA87] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound for the complexity of the union-split-find problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 479–488, 1987.

[Mor06] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM Journal on Computing (SICOMP)*, 35(6):1494–1525, 2006.

[NBLM12] Markus Neteler, M. Hamish Bowman, Martin Landa, and Markus Metz. GRASS GIS: A multi-purpose open source GIS. *Environmental Modelling & Software*, 31:124–130, 2012.

[Nek09] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

[Ove88] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, 1988.

[Păt07] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *Symposium on Theory of Computing (STOC)*, pages 40–46, 2007.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Symposium on Discrete Algorithms (SODA)*, pages 555–564, 2007.

[SJ05] Qingmin Shi and Joseph JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Information Processing Letters (IPL)*, 95(3):382–388, 2005.

[SMH08] T. G. Smolinski, M. G. Milanova, and A. E. Hassanien. *Applications of Computational Intelligence in Biology: Current Trends and Open Problems*, volume 122 of *Studies in Computational Intelligence*. Springer-Verlag Berlin Heidelberg, 2008.

[USGS] United States Geological Survey. The National Map Viewer and Download Platform. Available at: `http://nationalmap.gov/viewer.html`. Accessed: 2015-06-28.

[Tar79] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences (JCSS)*, 18(2):110–127, 1979.

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters (IPL)*, 6(3):80–82, 1977.

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

[Wil85] Dan E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing (SICOMP)*, 14(1):232–253, 1985.

[Wil00] Dan E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing (SICOMP)*, 29(3):1030–1049, 2000.