DATA STRUCTURES FOR COLORED COUNTING IN GRIDS AND TREES

by

Younan Gao

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
August 2023

Table of Contents

List of Tables . . . . . **v**

List of Figures . . . . . **vi**

Abstract . . . . . **viii**

Acknowledgements . . . . . **ix**

Chapter 1 Introduction **1**

1.1 Organization of the Thesis 3

Chapter 2 Preliminaries **5**

2.1 Notation 5

2.2 Models of Computation 5

2.3 The Ball Inheritance Problem 6

2.4 Ordinal Trees and Labeled Trees 7
2.4.1 Navigation in Colored Ordinal Trees 8
2.4.2 Colored Path Emptiness Queries 9
2.4.3 Centroid Decomposition 9

Chapter 3 Colored 2D Orthogonal Range Counting **11**

3.1 Introduction 11
3.1.1 Previous Work 12
3.1.2 Our Results 15

3.2 Preliminaries 16
3.2.1 Notation 16
3.2.2 Colored 2D Orthogonal Range Emptiness 17
3.2.3 Segment Trees and Interval Trees 17
3.2.4 Orthogonal Stabbing Queries over 3D Canonical Boxes 18
3.2.5 Reducing Colored 2D 3-Sided Range Counting to 3D Stabbing Counting over Canonical Boxes 23

3.3 A New Framework of Achieving Time-Space Tradeoffs 25
3.3.1 Overview of the Data Structure Framework 25
3.3.2 Computing Intersections between Color Sets 28

3.4 Two More Solutions with Better Space Efficiency 34
3.4.1 Achieving $O(n \lg^2 n)$-Word Space 34
3.4.2 Achieving $O(n \lg n)$-Word Space 40

3.5 Conclusion 45

Chapter 4 Faster Path Queries in Colored Trees 47

4.1 Introduction 47
4.1.1 Previous Work 49
4.1.2 Our Contributions 50

4.2 Preliminaries 51
4.2.1 Counting Colors for All Root-to-node Paths 52
4.2.2 Node Sampling 52
4.2.3 Rectangular Matrix Multiplication 52
4.2.4 Min-Plus Product and Min-Plus-Query-Witness Problem . . . 53

4.3 Batched Colored Path Counting 54
4.3.1 Color Counting over Paths Containing the Root 54
4.3.2 Faster Preprocessing via Sparse Matrix Multiplication 56
4.3.3 Color Counting on an Arbitrary Path 60

4.4 Batched Path Mode Queries 65
4.4.1 Queries for Infrequent Colors 66
4.4.2 Marking $O(n^{1-t_2})$ Nodes 69
4.4.3 Queries for Frequent Colors over Predefined Paths 70
4.4.4 Mode Queries on an Arbitrary Path 87

4.5 Conclusion 91

Chapter 5 Approximate Colored Path Counting 93

5.1 Introduction 93
5.1.1 Previous Work 94
5.1.2 Our Results 95

5.2 Preliminaries 97
5.2.1 Partial Rank 97
5.2.2 Tree extraction 98
5.2.3 The k-nearest distinct ancestor queries. 99
5.2.4 Colored Path Counting over All Paths 99
5.2.5 The Chernoff Bound 99

5.3 2-Approximate Colored Path Counting 99
5.3.1 Counting over a Path that Contains a Fixed Node 100

5.3.2 Counting over Arbitrary Paths 101
5.3.3 Speeding up the Query 105

5.4 $(\mathbf{1 \pm \epsilon})$-Approximate Colored Path Counting 108
5.4.1 Colored Type-2 Path Counting in Optimal Time 109
5.4.2 Random Sampling 113
5.4.3 Approximate Colored Path Counting over Canonical Paths . . 114
5.4.4 Approximate Colored Path Counting over Arbitrary Paths . . 117

5.5 Conclusion 118

Chapter 6 Conclusion and Future Work 120

6.1 Results 120

6.2 Future Work 122

Bibliography 125

List of Tables

3.1 Bounds of colored 2D orthogonal range counting structures. . 14

5.1 A summary of our results on approximate colored path counting and colored type-2 path counting. 96

List of Figures

1.1 The examples of colored path counting and path mode queries. 2

2.1 An example of a wavelet tree constructed upon a set of 16 points in 2D rank space. 7

2.2 An example of finding and removing a centroid of a tree. . . . 9

2.3 An example of a centroid decomposition of a tree. 10

3.1 An example of the two layers of segment trees. 19

3.2 An example of a second-layer segment tree. 21

3.3 An example of dividing region $U(A)$. 24

3.4 An example of matrix $M(v)$. 30

3.5 An example of a binary range tree data structure that solves 2D dominance range reporting and counting. 35

3.6 An example of a second layer interval tree. 37

3.7 Illustrating how the endpoints of a box projected on the yz-plane are assigned into different sets $S_{\ell,\ell}, S_{\ell,r}, S_{u,\ell}$ and $S_{u,r}$ . . 42

4.1 An example of marking nodes using the method of Lemma 13 and its corresponding matrix M. 55

4.2 An example of computing matrix M using matrix A and A^T. . 57

4.3 An example of matrices $\hat{M}$ and $\hat{A}$. 58

4.4 Illustrating the construction of matrix M' provided that matrix $\hat{M}$ is available as shown in Figure 4.3. 59

4.5 Illustrating the construction of matrix M provided that matrix M' is available as shown in Figure 4.4. 60

4.6 Illustrating the recursive data structure using the centroid decomposition. 61

4.7 An example of the weighted tree T'. 62

4.8 An example of identifying the component in the recursive data structure through the path minimum queries. 63

4.9 Three different cases in which $P_{u,v} \subseteq P_{s,t}$. 67

4.10 An example of finding maximum frequency of a frequent color in each predefined path. 71

4.11 Illustrating the proof of Lemma 21 with an example. 73

5.1 Examples of colored type-1 and type-2 path counting. 94

5.2 An example of tree extraction. 98

5.3 The data structure for 2-approximate queries. 100

5.4 The input tree T and its transformed binary tree $\tilde{T}$. 102

5.5 An example of a component tree. 104

5.6 The SP's constructed for the component tree presented in Figure 5.5. 105

5.7 An example of the first traversal strategy. 107

5.8 An example of the second traversal strategy. 108

Abstract

In this thesis, we design data structures for colored counting queries. Our goal is to preprocess a set, P, of n colored objects into a data structure such that given a query Q, the number of distinct colors in $P \cap Q$ can be computed efficiently. The problem settings studied primarily fall in two categories: *colored two-dimensional (2D) orthogonal range counting* where the input is a set of n colored points on the plane and the query is an axis-aligned rectangle, and *colored path counting* where the input is an ordinal tree on n colored nodes and the query is an arbitrary path. First, we consider the colored 2D orthogonal range counting problem. We design three new data structures, and the bounds of each can be expressed in some form of time-space tradeoff. By setting appropriate parameter values for these solutions, we achieve four time-space tradeoffs, which are $O(n \lg^3 n)$ space and $O(\sqrt{n} \lg^{5/2} n \lg \lg n)$ query time, $O(n \lg^2 n)$ space and $O(\sqrt{n} \lg^{4+\lambda} n)$ query time, $O(n \frac{\lg^2 n}{\lg \lg n})$ space and $O(\sqrt{n} \lg^{5+\lambda} n)$ query time and $O(n \lg n)$ space and $O(n^{1/2+\lambda})$ query time, respectively, for any constant $0 < \lambda < 1$. Second, we consider the *batched colored path counting* problem, in which the interest lies in the overall running time for n query paths, including the preprocessing time. By reducing the problem to *sparse matrix multiplication*, we design a solution that answers n colored path counting queries in $O(n^{1.4071})$ time. Another related problem, *batched path mode queries*, is also considered, in which given n query paths and we are asked to find a color of maximum multiplicity in each path. We present a solution that answers n queries in $O(n^{1.483814})$ time using the fast computation of *min-plus products* over structured matrices. Third, we extend the study to *approximate colored path counting*. We design data structures for *2-approximate colored path counting* with $O(n)$ space and $O(\lg^\lambda n)$ query time, $O(n \lg \lg n)$ space and $O(\lg \lg n)$ time and $O(n \lg^\lambda n)$ space and $O(1)$ time, respectively, for any constant $0 < \lambda < 1$. We also present a linear space data structure that supports $(1 \pm \epsilon)$-*approximate colored path counting* in $O(\epsilon^{-2} \lg n)$ time, for any $0 < \epsilon < 1$.

Acknowledgements

I would like to thank my supervisor, Dr. Meng He. Dr. He recruited me as a graduate student 5 years ago and offered me the opportunity of doing research. Since then, Dr. He has taught me many invaluable problem-solving and academic writing skills, as well as a lot of knowledge on data structures. He encouraged me to look for and work on the research topics I like, making me feel free to make choices. Doing a Ph.D. is difficult and stressful. Dr. He, being a great mentor, has offered me numerous advice on both technical and non-technical matters, helping me address concerns in studies and life. Dr. He has also spent much time on proofreading this thesis and helping me improve the writings in many places.

I thank the committee members, Dr. David Bremner, Dr. Norbert Zeh and Dr. Travis Gagie for volunteering their precious time and offering thoughtful comments and constructive criticisms.

I am very grateful for the writing advisors, Janice MacDonald Eddington, Dr. Adam Auch and Dr. Vanessa Lent for sharing their expertise regarding the presentation and helping me prepare the thesis defense.

Lastly, I would like to offer my gratitude to my parents for their support and encouragement over the years, especially under the lock-downs due to the pandemic.

Chapter 1

Introduction

The number of distinct elements in a multiset is one of the most fundamental statistics that arise in many applications, such as data mining, databases and information retrieval. Let A denote an array of n elements, i.e, $\{a_1, a_2, \cdots, a_n\}$. Any sub-array, $\{a_i, a_{i+1}, \cdots, a_j\}$, can be seen as a typical example of a multiset. The problem of preprocessing A into a data structure, such that the number of distinct elements in any sub-array can be computed efficiently, has been well studied [63, 27]. Data elements can be organized in more complicated structures than arrays, such as points in multi-dimensional space and nodes on trees, and the problem, counting the number of distinct elements, emerges therein. If each distinct element in the data set is assigned a unique color, then counting the number of distinct elements is generalized to counting the number of distinct colors.

In the *colored orthogonal range counting* problem, each point in a dataset is assigned a color that represents the data it stores and a multiset contains the points in a multi-dimensional axis-aligned rectangle. Accordingly, colored counting in a multiset is of interest, since the problem has applications in databases. For example, the information of an athlete, including his/her country, age and weight, might be stored as a record in the database, e.g., (country=Canada, age=18, weight=85kg). And a typical query could be as follows: How many different countries are there represented by the athletes with an age between 18 and 25 and a weight between 80kg and 90kg? To answer the query, we represent each athlete by a point such that the age and the weight of the athlete are stored as x- and y-coordinates of the point, and the country that the athlete is from is encoded as the point color. Then the query is two-dimensional axis-aligned rectangle, i.e., $[18, 25] \times [80, 90]$. Thereby, the number of distinct colors assigned to the points in the query range is the number of countries that one is looking for. Henceforth, we focus on the two-dimensional space where the input points reside on the grid.

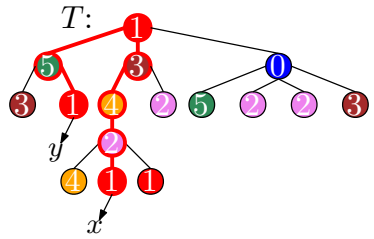


Figure 1.1: The examples of colored path counting and path mode queries. The input is a tree T on 16 nodes, each assigned a color drawn from $\{0, 1, 2, 3, 4, 5\}$. Let $\mathrm{P}_{x,y}$ be the query path. There are 5 distinct colors and a unique mode, labeled by 1, in $\mathrm{P}_{x,y}$.

Trees, as a versatile structure, can store data information on their nodes. In the *colored path counting* problem [44], each tree node stores a color (or a label) as a data element and a multiset contains the nodes in an arbitrary path. Given a query path, this problem asks to count the number of distinct colors that are assigned to the nodes in the path. Another fundamental statistic in paths is called *path mode*, which is a node label that appears at least as frequently as other labels in a query path. See Figure 1.1 for the examples of both path queries. Both of them have applications in *Extensible Markup Language (XML)*. In XML documents, the hierarchical structure of the texts can be modeled as ordinal trees. An ordinal tree is a rooted tree in which the children of a node are ordered. Each tag in an XML text carries a name and attributes, and both can be regarded as node labels. The problems, colored path counting and path mode queries, can be used to count the number of the distinct tag names and find a most frequent tag name in the paths from a tag to its ancestors.

The common difficulty in solving colored counting in grids or trees, as well as path mode queries is that the query is not decomposable. In grids, query ranges are of the form $[a, b] \times [c, d]$. After dividing a query range into disjoint subranges, simply adding up the number of distinct colors in each subrange might not give the correct answer to the query range. Similarly, suppose that query path $\mathrm{P}_{x,y}$ is partitioned into two subpaths $\mathrm{P}_{x,v}$ and $\mathrm{P}_{v,y}$ by a node v in $\mathrm{P}_{x,y}$. Being aware of the answers to $\mathrm{P}_{x,v}$ and $\mathrm{P}_{v,y}$ does not give much help in finding the answer to $\mathrm{P}_{x,y}$. Especially, the sets of modes in $\mathrm{P}_{x,v}$, $\mathrm{P}_{v,y}$ and $\mathrm{P}_{x,y}$, respectively, could be pair-wise disjoint.

Interestingly, both colored counting queries in grids or trees, as well as path mode queries share the same conditional lower bound: For any of these three problems, i.e., colored path counting, colored 2D orthogonal range counting and path mode queries,

with current knowledge, one might not be able to obtain a data structure that can be built in $o(n^{\omega/2})$ time and meanwhile support each query in $o(n^{\omega/2-1})$ time, save for polylogarithmic speedups, where $\omega < 2.37286$ is the exponent of matrix multiplication [1]. When only pure combinatorial approaches are allowed in solving any of these three problems, it might be infeasible to design a data structure that can be built in $o(n^{3/2})$ time and meanwhile support each query in $o(\sqrt{n})$ time, save for polylogarithmic speedup, following from the bounds on the best combinatorial approach for Boolean matrix multiplication [70]. Due to the difficulty of all three problems, their approximate versions have been brought to attention [25, 66, 44], in which the requirement that finds the precise number of distinct colors or a mode in a multiset is relaxed and instead, a query asks for a value that is within a predefined factor of the precise number of distinct colors in a multiset or a color whose frequency in a multiset is at most a predefined factor smaller than a true mode's. These problems are of interest in theoretical computer science, as the design and analysis of these approximate algorithms always involves mathematical proof, which helps us comprehend with provable guarantees how closely we are able to approximate the optimal answers in polylogarithmic time and near linear working space.

In this thesis, we study three aspects of colored counting queries. First, we revisit the colored two-dimensional orthogonal range counting problem and design data structures that improve the query times and space costs asymptotically. Second, we study the problem of answering n colored path counting queries, in which the preprocessing time matters, since it is included in the overall query time. In addition, we also consider its relevant problem, *batched path mode queries*. Third, we study the approximate version of the colored path counting problem. We aim at maintaining the linear space cost of the data structure, while improving the query time asymptotically.

1.1 Organization of the Thesis

The rest of this thesis is structured as follows.

Chapter 2 describes the notation and the preliminary knowledge that is commonly used throughout this thesis. The model of computation that is applied can be found there as well.

In Chapter 3, we conduct a study on colored two-dimensional orthogonal range counting. Obviously, a 4-sided query range can be decomposed into two subranges, each of which is bounded by 3 sides. We propose a new framework that computes the number of distinct colors that exist in both 3-sided subranges. Based on the new framework, we design three data structures for the problem. The attention is given to two important metrics of a data structure: query time and the space cost. We achieve three different data structure solutions. This chapter is based on the joint work with Meng He [34].

In Chapter 4, we consider batched colored path counting and batched path mode queries. We concentrate on the efficient preprocessing and query time of the data structures. We show that by applying the centroid decomposition of trees, an arbitrary path can be dealt with as a path that contains a centroid of some subtree. This technique is used to solve both path query problems. Given that query paths are through a fixed node of the input tree, say, a centroid, we design for colored path counting a data structure whose construction can be facilitated by sparse matrix multiplication and for path mode queries a data structure whose construction can be facilitated by a min-plus product. Not only can our data structures be built in $o(n^{1.5})$ time, but they also support each query in $o(n^{0.5})$ time, faster than any existing combinatorial method. The results of Chapter 4 have been published in [35].

In Chapter 5, we design data structures for approximate colored path counting. We consider this problem under two different approximate measures, i.e., 2- and $(1 \pm \epsilon)$-approximate. By applying the technique, *random sampling*, we present a reduction from the latter to the prior. For the prior, we present a data structure with different time-space tradeoffs. To this end, we revisit the technique, centroid decomposition, and design a space-efficient data structure such that the preorder rank of a tree node in any recursive level can be computed efficiently. This chapter is based on the joint work with Meng He [36].

In Chapter 6, we conclude this thesis by summarizing the technical details used in different chapters, as well as the results we have achieved, and pointing out the future research directions, including relevant open problems.

Chapter 2

Preliminaries

In this chapter, we introduce the preliminary knowledge and notation used throughout this thesis.

2.1 Notation

We use $\lg n$ to represent $\log_2 n$ for short, i.e., the base-2 logarithm of n, and we denote by $[n]$ the set $\{0, 1, 2, \cdots, n-1\}$, for any integer $n > 0$. The $\tilde{O}$ notation hides the polylog(n) factors, e.g., $O(n \lg^2 n) = \tilde{O}(n)$, and we write down *iff* as a shortened form of *if and only if*.

2.2 Models of Computation

Unless otherwise specified, all results presented throughout this thesis are valid under the word random-access machine (word RAM) model [28]. In this model, each cell in the memory is a word of w bits. We usually assume that $w = \Theta(\lg n)$, where n denotes the input size, e.g., n elements in an array, so that the address of any cell in the memory can be encoded and stored in a constant number of memory cells. Hence, if a data structure occupies s cells, then its space cost can also be stated as $s \cdot w = \Theta(s \lg n)$ bits.

Operations that are performed on a constant number of memory cells are called *word operations*. The basic instruction set supported in word RAM consists of reading (i.e., random access to a memory cell), writing (i.e., writing data into a memory cell), arithmetic operations (including addition, subtraction, multiplication, and division), and bitwise logical operations (including AND, OR, SHIFT, XOR, and NEGATION). Each word operation in the basic instruction set takes constant time. Any other word operation can be simulated by the technique, *table lookup*, taking constant time as well. The running time of an algorithm is measured by the number of word operations

performed, and the space cost is measured by the number of memory cells used by the algorithm.

2.3 The Ball Inheritance Problem

In two-dimensional *rank space* [29], the point coordinates are drawn from the universe $\{0, 1, \cdots, n-1\}^2$, and no two points have the same x- or y-coordinates. The wavelet tree [37] constructed over the y-coordinates of n points in 2D rank space is a binary balanced tree with $\lg n$ levels. Without loss of generality, we assume that n is a power of 2. Each node of the tree represents a range of y-coordinates as follows: The i-th leaf from the left represents the y-range $[i, i]$ for each $i \in [0..n-1]$; and the range represented by an internal node is the union of the ranges represented by its children. Hence, the root represents y-range $[0..n-1]$. Each node v is associated with a sequence of points, $P(v)$, that includes all the input points whose y-coordinates are in the y-range of node v, and points in $P(v)$ are sorted by their x-coordinates. For example, the root node is associated with n points increasingly sorted by x-coordinates. Each internal node v explicitly stores a bit vector, $\mathcal{B}(v)$, such that if the point $P(v)[i]$ is a leaf descendant of the left child of v, then $\mathcal{B}(v)[i]$ is set to 0; otherwise $\mathcal{B}(v)[i]$ is set to 1. All the $\mathcal{B}(v)$'s built for internal nodes occupy $O(n \lg n)$ bits of space in total. An example of a wavelet tree is shown in Figure 2.1.

The *ball inheritance problem* [14] can be defined over a binary wavelet tree. In this problem, given an arbitrary internal node v and an integer $0 \leq i < |P(v)|$, we are asked to find the coordinates of $P(v)[i]$. Henceforth, we denote this query by point(v, i) for simplicity. This problem was formally defined by Chan et al. and was used in solving *orthogonal range searching* [14]. We will see its applications on colored counting in Chapters 3 and 5. If sequence $P(v)$ is explicitly stored at each internal node v, then operation point(v, i) can be answered trivially in constant time. However, storing all $P(v)$'s for internal nodes overall requires $O(n \lg n)$ words of space, which is not affordable sometimes in our linear space data structure. Instead, the support for point(v, i) addressed in Lemma 1 provides a space-efficient solution:

Lemma 1 *[14, Lemma 2.3] Let $0 < \lambda < 1$ be any small positive constant. A binary wavelet tree over n points in 2D rank space can be augmented with a data structure in $O(nf(n) \lg n)$ bits of space to support* point(v, i) *in $g(n)$ time, where*

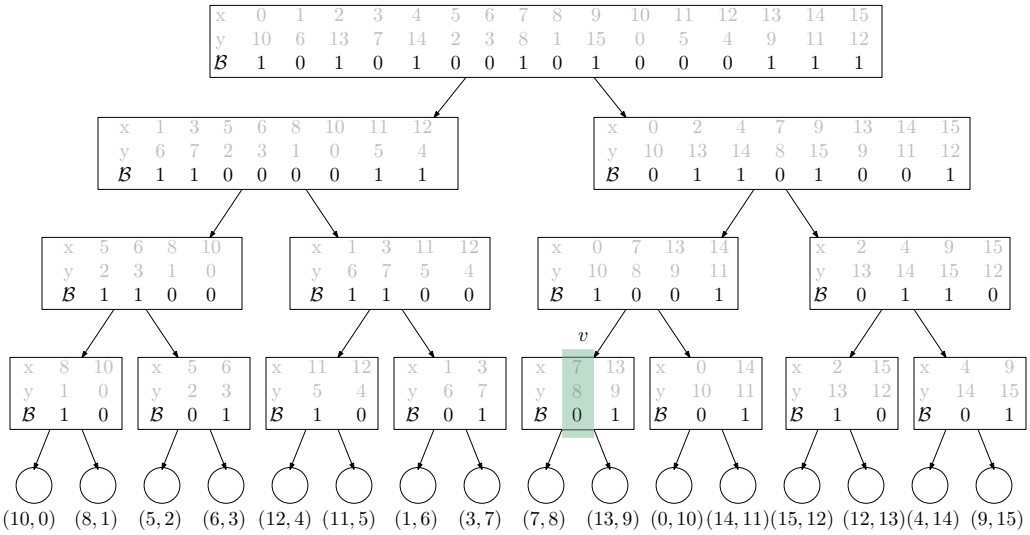


Figure 2.1: An example of a wavelet tree constructed upon a set of 16 points in 2D rank space. The sequence $P(v)$ that associates with node v in the figure stores points $(7, 8)$ and $(13, 9)$, sorted by their x-coordinates. Function point$(v, 0)$ returns point $(7, 8)$, following its definition. Given that points $(7, 8)$ and $(13, 9)$ are stored at the left and right child of v, respectively, $\mathcal{B}(v)$ stores 0 and 1 from left to right.

a) $f(n) = O(1)$ *and* $g(n) = O(\lg^{\lambda} n)$,

b) $f(n) = O(\lg \lg n)$ *and* $g(n) = O(\lg \lg n)$ *or*

c) $f(n) = O(\lg^{\lambda} n)$ *and* $g(n) = O(1/\lambda)$.

2.4 Ordinal Trees and Labeled Trees

Regarding each tree node color as an integer label, the input tree studied in this thesis is both an ordinal tree[1] and a labeled tree.

Let $|T|$ denote the number of nodes in the input tree T. The topmost node of the tree is called the *root*, denoted by $\perp$. For any two nodes $x, y \in T$, we use $\mathrm{P}_{x,y}$ to represent the path whose endpoints are x and y. Thus, $\mathrm{P}_{x,\perp}$ is a root-to-node path. Each non-root node x has a single *parent*, denoted by parent(x), which is the node incident to x in path $\mathrm{P}_{x,\perp}$. Symmetrically, node x is a *child* of parent(x). A *leaf* in the tree is a node that has no children. The ancestors of a node can be defined in

[1]The path query problems studied in this thesis can also be defined over free trees. However, we follow the definitions given in previous work [23, 44] and assume that the input tree is ordinal, as this allows us to directly apply previous solutions to the problems defined over ordinal trees.

an inductive way: Node x is an ancestor of itself; and the ancestors of parent(x) are ancestors of x as well. If node y is an ancestor of x and y is different from x, then we call y a *proper ancestor* of x. The *descendants* or the *proper descendants* of a node can be defined symmetrically. Given that y is an ancestor of x, we define $\mathrm{P}'_{x,y}$ to be the path whose endpoints are x and the child of y that is an ancestor of x, i.e., $\mathrm{P}'_{x,y} = \mathrm{P}_{x,y} \setminus \{y\}$. We denote by T_x the subtree of T rooted at node x. We define the *depth* or *level* of a node x to be the number of edges in path $\mathrm{P}_{x,\perp}$. Hence, the depth of $\perp$ is always 0. To identify each node in the tree, we refer to the i-th node visited in a preorder traversal as the *i-th node* for short, where i starts from 0. We define the degree of a tree node to be the number of edges incident to this node.

As a labeled ordinal tree, each node is labeled by a color that is encoded by an integer. For a node x, its color is indicated by $c(x)$, where $c(x) \in [C]$ and $C \leq n$ denotes the total number of distinct colors in the tree. And we use $C(\mathrm{P}_{x,y})$ to represent the set of distinct colors that appear in $\mathrm{P}_{x,y}$.

2.4.1 Navigation in Colored Ordinal Trees

To support the basic navigational operations over the input tree, we apply the succinct representation of ordinal trees by [45] and the result of He et al. [47] on labeled tree representations. The following lemma summarizes the operations used in our solution and their respective complexity. Following their notation, we call a node (resp. ancestor) assigned color α an α-node (resp. α-ancestor).

Lemma 2 ([45, 47]) *Let T be an ordinal tree on n nodes with each node assigned a color from $[C]$, where $C \leq n$. A data structure occupying $n \lg C + 2n + o(n \lg C)$ bits can be built over T in $O(n)$ time to support*

- *counting the number,* $\mathrm{depth}_\alpha(x)$*, of α-ancestors of node x in $O(\lg \lg C)$ time*[2]*,*
- *counting the number,* $\mathrm{depth}(x)$*, of ancestors of x in $O(1)$ time,*
- *finding the lowest common ancestor,* $\mathrm{LCA}(x, y)$*, of nodes x and y in $O(1)$ time,*

[2]Note that the structure of He et al. [47] can support $\mathrm{depth}_\alpha(x, i)$ in $O(\lg \frac{\lg C}{\lg w})$ time, faster than what is stated in Lemma 2. However, this requires a string representation with support for rank and select [5] which uses perfect hashing, and it is not known how to construct this structure in $O(n)$ deterministic time. Therefore, we swap it with the string representation of Belazzougui et al. [4] which can be constructed in linear deterministic time and achieve the bounds in Lemma 2.

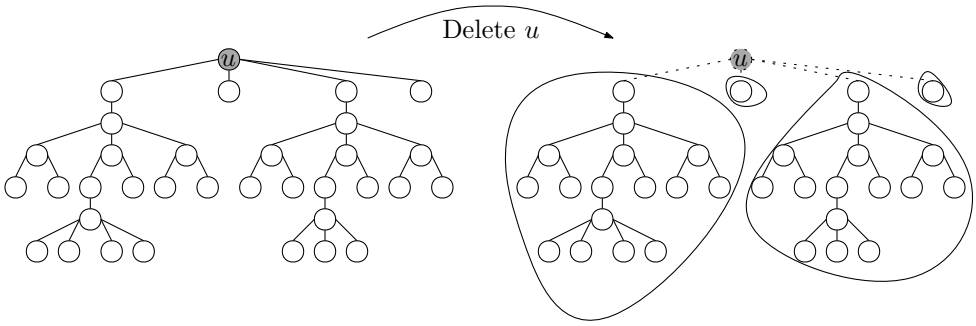


Figure 2.2: An example of finding and removing a centroid of a tree. In this figure, the centroid, denoted by u, of the tree happens to be the root. After removing u, each connected component contains at most $n/2$ nodes.

- *counting the number,* $\text{nbdesc}(x)$*, of descendants of node* x *in* $O(1)$ *time,*
- *finding the parent node,* $\text{parent}(x)$*, of non-root node* x *in* $O(1)$ *time and*
- *finding the ancestor,* $\text{level\_anc}(v, d)$*, of* v *at depth* $\text{depth}(v) - d$ *in* $O(1)$ *time.*

2.4.2 Colored Path Emptiness Queries

Given a query path $\mathrm{P}_{x,y}$ in a tree and a color α, a *colored path emptiness* query determines whether color α appears in $\mathrm{P}_{x,y}$. He and Kazi [44] showed how to use depth_α and LCA to compute the number of appearances of α in $\mathrm{P}_{x,y}$ in $O(\lg \lg C)$ time. Via counting the number of appearances of color α in $\mathrm{P}_{x,y}$, one can figure out whether or not α appears in $\mathrm{P}_{x,y}$. Therefore,

Lemma 3 *Let* T *denote a labeled ordinal tree on* n *nodes, each of which is labeled by a color drawn from* $[C]$*, where* $C \leq n$*. A data structure occupying* $O(n \lg C)$ *bits can be built over* T *in* $O(n)$ *time to support the colored path emptiness queries in* $O(\lg \lg C)$ *time.*

Especially, when C is a constant, so is the bound of the running time.

2.4.3 Centroid Decomposition

A *centroid* of an n-node tree is a node whose removal splits the tree into connected components each containing at most $\lfloor n/2 \rfloor$ nodes. See Figure 2.2 for an example. As proved by Jordan in 1869 [54], any tree of n nodes has at least one centroid. A centroid can be found in $O(n)$ time.

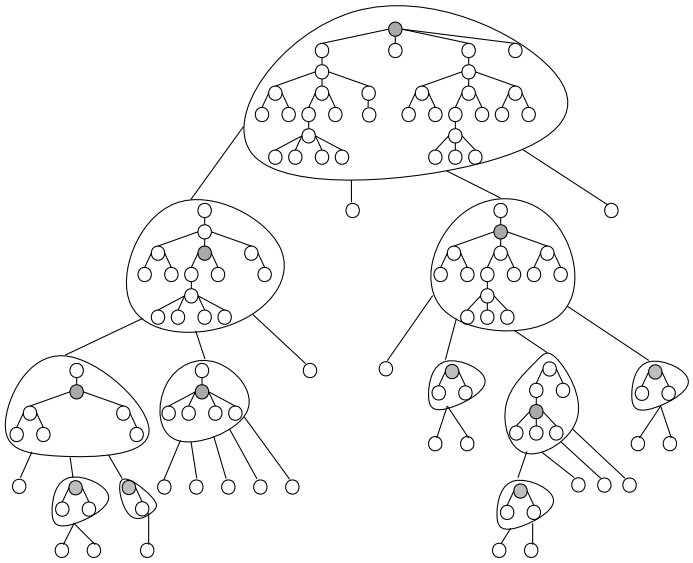


Figure 2.3: An example of a centroid decomposition of a tree. In the figure, the splinegons represent the connected components after removing centroids at different recursive levels. The gray node within each component indicates the centroid to be removed. In the last recursive level, each component to be generated contains one node of the tree.

Centroid decomposition is a technique applied on trees, and it works in a recursive way. A tree itself can be regarded as a connected component. In the first recursive level, we identify a centroid of the input tree, and by removing it, the tree is divided into connected components. Then we recursively apply this procedure on each of these connected components and finally obtain a centroid decomposition of the input tree. Given that the input tree contains n nodes, the recursion contains $O(\lg n)$ levels. See Figure 2.3 for an example. The centroid decomposition of any n-node tree can be computed in $O(n)$ time [21].

Chapter 3

Colored 2D Orthogonal Range Counting

3.1 Introduction

In computational geometry, there have been extensive studies on problems over points associated with information represented as colors [40, 42, 57, 55, 59, 38, 63, 27, 41, 15, 13, 44, 66]. Among them, the *colored 2D orthogonal range counting* query problem is one of the most fundamental. In this problem, we preprocess a set, P, of n points on the plane, each colored in one of C different colors such that given an orthogonal query rectangle, the number of distinct colors of the points contained in this rectangle can be computed efficiently.

This problem is important in both theory and practice. Theoretically, it has connections to matrix multiplication: The ability to answer m colored range counting queries offline over n points on the plane in $o(\min\{n, m\}^{\omega/2})$ time, where ω is the best current exponent of the running time of matrix multiplication, would yield a faster algorithm for Boolean matrix multiplication [55]. In practice, the records in database systems and many other applications are often associated with categorical information which can be modeled as colors. For example, in the Structured Query Language (SQL), keyword `DISTINCT` is used for computing information about the distinct categories of the records within a query range, which can be modeled using colored range query problems, and these queries have also been used in database query optimization [16].

One challenge in solving the colored 2D orthogonal range counting problem is that the queries are not easily decomposable: If we partition the query range into two or more subranges, we cannot simply obtain the number of distinct colors in the query range by adding up the number of distinct colors in each subrange. Furthermore, the conditional lower bound based on matrix multiplication as described above gives theoretical evidence on the hardness of this problem. Indeed, if polylogarithmic query times are desired, the solution with the best space efficiency [61] uses $O(n^2 \lg n / \lg \lg n)$

words of space and answers queries in $O((\lg n/\lg\lg n)^2)$ time. There is a big gap between the complexities of this solution and those of the optimal solution to *2D orthogonal range counting* for which we do not have color information and are only interested in computing the number of points in a 2D orthogonal query range. The latter can be solved in merely linear space and $O(\lg n/\lg\lg n)$ query time [51].

Applications that process a significant amount of data would typically require structures whose space costs are much lower than quadratic. As the running time of the best known combinatorial algorithm of multiplying two $n \times n$ Boolean matrices is $\Theta(n^3/\operatorname{polylog}(n))$ [3, 9, 70], the conditional lower bound of colored 2D orthogonal range counting implies that no solution can simultaneously have preprocessing time better than $\Omega(n^{3/2})$ and query time better than $\Omega(\sqrt{n})$, by purely combinatorial methods with current knowledge, save for polylogarithmic speed-ups. To match this query time within polylogarithmic factors, the most space-efficient solution uses $O(n\lg^4 n)$ words of space to answer queries in $O(\sqrt{n}\lg^8 n)$ time [55]. Despite this breakthrough, the exponents in the polylogarithmic factors in the time and space bounds leave much room for potential improvements. Hence, in this chapter, we aim at decreasing these polylogarithmic factors in both space and time costs and designing solutions that are more desirable for applications that manage large data sets.

3.1.1 Previous Work

Gupta et al. [40] showed how to reduce the colored orthogonal range searching problem in 1D to orthogonal range searching over uncolored points in 2D, thus achieving a linear-space solution with $O(\lg n/\lg\lg n)$ query time. Later, the query time was improved to $O(\lg k/\lg\lg n)$ by Nekrich [63] with an adaptive data structure, where k is the number of distinct colors of the points in the query range.

To solve 2D colored orthogonal range counting, Gupta et al. [40] used persistent data structures to extend their 1D solution to 2D and designed a data structure of $O(n^2\lg^2 n)$ words that supports queries in $O(\lg^2 n)$ time. Kaplan et al. [55] achieved the same bounds by decomposing the input points into disjoint boxes in 3D and reducing the problem to 3D stabbing counting queries. Recently, Munro et al. [61] showed that colored 3D 3-sided range counting can be answered in $O((\lg n/\lg\lg n)^2)$ time using a data structure of $O(n(\lg n/\lg\lg n))$ words of space, which implies a

solution to colored 2D 3-sided range counting with the same time and space bounds. For each distinct x-coordinate x_i of the points in the point set, if we use the strategy of Gupta et al. [40] and Kaplan et al. [55] to build a data structure supporting 2D 3-sided queries upon the points whose x-coordinates are greater than or equal to x_i, then this set of data structures constructed can be used to answer a 4-sided query. This yields a solution to the colored 2D orthogonal range counting problem with $O(n^2 \lg n / \lg\lg n)$ words of space and $O((\lg n / \lg\lg n)^2)$ query time. Kaplan et al. [55] further showed how to achieve time-space tradeoffs by designing a solution with $O(X \lg^7 n)$ query time that uses $O((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$ words of space. Setting $X = \lceil \sqrt{n} \lg n \rceil$ minimizes space usage, achieving an $O(n \lg^4 n)$-word solution with $O(\sqrt{n} \lg^8 n)$ query time. When only linear space is allowed, Grossi and Vind [38] showed how to answer a query in $O(n/\operatorname{polylog}(n))$ time. Though not explicitly stated anywhere, by combining an approach that Kaplan et al. [55] presented for dimensions of 3 or higher (which also works for 2D) and a linear space solution to 2D orthogonal range emptiness [14], the query time can be improved to $O(n^{3/4} \lg^{\lambda} n)$ for any constant $0 < \lambda < 1$ using a linear space structure. Finally, Kaplan et al. also considered the offline version of this problem and showed that n colored 2D orthogonal range counting queries can be answered in $O(n^{1.4071})$ time.

Researchers have also studied the approximate colored range counting problem. In the $(1+\epsilon)$-approximate colored range counting problem, instead of computing the exact number, k, of distinct colors in the query range, the solution returns a value k', such that $k \leq k' \leq (1+\epsilon) \cdot k$, where $0 < \epsilon < 1$ is a pre-specified parameter. El-Zein et al. [27] considered this problem in 1D space, and they designed a succinct data structure to answer a $(1+\epsilon)$-approximate colored counting query in constant time. Another relaxed version, called $(1 \pm \epsilon)$-approximate colored range counting, has also been studied, in which the answer returned is in the range $[(1-\epsilon) \cdot k, (1+\epsilon) \cdot k]$. In 2D, Rahul [66] provided a reduction from $(1 \pm \epsilon)$-approximate colored orthogonal range counting to 2D colored orthogonal range reporting (which reports each distinct color in a 2D orthogonal query range). Based on this reduction, Rahul gave an $O(n \lg n)$-word data structure supporting $(1 \pm \epsilon)$-approximate colored orthogonal range counting in $O(\epsilon^{-2} \lg n)$ time.

Table 3.1: Bounds of colored 2D orthogonal range counting structures. The results in the form of time-space tradeoffs are listed in the top portion, in which X and γ are integer parameters in $[1, n]$ and $[2, n]$, respectively. The bottom portion presents results with specific bounds, among which marked with a † are those obtained from the top portion by setting appropriate parameter values.

Source	Model	Query Time	Space Usage in Words
[55]	PM	$O(X \lg^7 n)$	$O((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$
Cor. 1	PM	$O(\lg^5 n + X \lg^3 n)$	$O((\frac{n}{X})^2 \lg^4 n + n \lg^3 n)$
Thm. 1	RAM	$O(\lg^4 n + X \lg^2 n \lg\lg n)$	$O((\frac{n}{X})^2 \lg^4 n + n \lg^3 n)$
Thm. 2	RAM	$O(\lg^6 n + X \lg^{3+\lambda} n)$	$O((\frac{n}{X})^2 \lg^4 n + n \lg^2 n)$
Thm. 3	RAM	$O(\gamma^2 \lg^6 n \log_\gamma^2 n + X \lg^{3+\lambda} n\gamma \log_\gamma n)$	$O((\frac{n}{X})^2 \lg^2 n \log_\gamma^2 n$ $+n \lg n \log_\gamma n)$
[40, 55]	PM	$O(\lg^2 n)$	$O(n^2 \lg^2 n)$
[61]	RAM	$O((\lg n / \lg\lg n)^2)$	$O(n^2 \lg n / \lg\lg n)$
[55]†	PM	$O(\sqrt{n} \lg^8 n)$	$O(n \lg^4 n)$
Cor. 1†	PM	$O(\sqrt{n} \lg^{7/2} n)$	$O(n \lg^3 n)$
Thm. 1†	RAM	$O(\sqrt{n} \lg^{5/2} n \lg\lg n)$	$O(n \lg^3 n)$
Thm. 2†	RAM	$O(\sqrt{n} \lg^{4+\lambda} n)$	$O(n \lg^2 n)$
Thm. 3†	RAM	$O(\sqrt{n} \lg^{5+\lambda} n)$	$O(n \frac{\lg^2 n}{\lg\lg n})$
Thm. 3†	RAM	$O(n^{1/2+\lambda})$	$O(n \lg n)$
[55]	PM	$O(n^{3/4} \lg n)$	$O(n \lg n)$
[38]	RAM	$O(n / \text{polylog}(n))$	$O(n)$
[55, 14]	RAM	$O(n^{3/4} \lg^\lambda n)$	$O(n)$

The colored orthogonal range counting problem has also been studied in higher dimensions [59, 55, 66, 44]. Furthermore, He and Kazi [44] generalized it to colored path counting by replacing the first dimension with a tree topology. One of their solutions to a path query problem generalized from colored 2D orthogonal range counting also uses linear space and provides $O(n^{3/4} \lg^\lambda n)$ query time. We end this brief survey by commenting that after a series of work [52, 40, 55, 63, 38, 41], Chan and Nekrich [15] solved the related 2D orthogonal range reporting problem in $O(n \lg^{3/4+\lambda} n)$ words of space and $O(\lg\lg n + k)$ query time for points in rank space, where k is the output size. This almost matches the bounds of the optimal solution to (uncolored) 2D orthogonal range reporting over points in rank space, which uses $O(n \lg^\lambda n)$ words to answer queries in $O(\lg\lg n + k)$ time [14].

3.1.2 Our Results

Under the word RAM model, we present three results, all in the form of time-space tradeoffs, for colored two-dimensional orthogonal range counting. Specifically, for an integer parameter $X \in [1, n]$, we propose solutions (all space costs are in words),

- with $O((\frac{n}{X})^2 \lg^4 n + n \lg^3 n)$ space and $O(\lg^4 n + X \lg^2 n \lg\lg n)$ query time; setting $X = \lceil \sqrt{n \lg n} \rceil$ achieves $O(n \lg^3 n)$ space and $O(\sqrt{n} \lg^{5/2} \lg\lg n)$ query time;
- with $O((\frac{n}{X})^2 \lg^4 n + n \lg^2 n)$ space and $O(\lg^6 n + X \lg^{3+\lambda} n)$ query time for any constant $\lambda \in (0, 1)$; setting $X = \lceil \sqrt{n} \lg n \rceil$ achieves $O(n \lg^2 n)$ space and $O(\sqrt{n} \lg^{4+\lambda})$ query time;
- with $O((\frac{n}{X})^2 \lg^2 n \cdot \log_\gamma^2 n + n \lg n \cdot \log_\gamma n)$ space and $O(\gamma^2 \cdot \lg^6 n \cdot \log_\gamma^2 n + X \cdot \lg^{3+\lambda} n \cdot \gamma \log_\gamma n)$ query time for an integer parameter $\gamma \in [2, n]$; setting $X = \lceil \sqrt{n \lg n \log_\gamma n} \rceil$ and $\gamma = \lceil \lg^\lambda n \rceil$ achieves $O(n \frac{\lg^2 n}{\lg\lg n})$ space and $O(\sqrt{n} \lg^{5+\lambda'} n)$ query time for any $\lambda' > 2\lambda$, while setting $X = \lceil \sqrt{n \lg n} \rceil$ and $\gamma = \lceil n^{\lambda/5} \rceil$ achieves $O(n \lg n)$ space and $O(n^{1/2+\lambda})$ query time.

When presenting each result, we also gave the bounds of the most space-efficient tradeoff that can be achieved by setting appropriate parameter values. The conditional lower bound based on Boolean matrix multiplication that we discussed before gives some evidence on the difficulty of achieving query time better than $\sqrt{n}$ by more than a polylogarithmic factor using combinatorial approaches without increasing these space costs polynomially.

Prior to our work, only the time-space tradeoff presented by Kaplan et al. [55] can achieve near-linear space and $O(\sqrt{n}\,\text{polylog}(n))$ query time. More specifically, their solution uses $O((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$ words of space and achieves $O(X \lg^7 n)$ query time. The most space-efficient tradeoff that can be obtained from it is an $O(n \lg^4 n)$-word structure with $O(\sqrt{n} \lg^8 n)$ query time. Thus we indeed achieve the goal of improving significantly the polylogarithmic factors in both the query time and the space cost.

It is worthwhile to mention that the result of Kaplan et al. works under the pointer machine (PM) model. Thus, for an absolutely fair comparison, we show how our first result can be adapted to the same model of computation to achieve

$O((\frac{n}{X})^2 \lg^4 n + n \lg^3 n)$ space and $O(\lg^5 n + X \lg^3 n)$ query time. Thus under PM, we have an $O(n \lg^3 n)$-word structure with $O(\sqrt{n} \lg^{7/2} n)$ query time. This still improves the query time and the space bounds of previous work. In the rest of this chapter, however, we assume the word RAM model of computation unless otherwise specified, since most of our results are designed under it. See Table 3.1 for a comparison of our results to all previous results.

To achieve these results, we use the standard technique of decomposing a 4-sided query range to two 3-sided subranges with a range tree. Then the answer can be obtained by adding up the numbers of distinct colors assigned to points in each subrange and then subtracting the number of distinct colors that exist in both. We still use an approach of Kaplan et al. to reduce colored 2D 3-sided range counting to 3D stabbing queries over a set of boxes. What is new is our scheme of achieving time-space tradeoffs when computing the number of colors that exist in both subranges. Compared to the original scheme designed by Kaplan et al. for the same purpose, ours gives more flexibility in the design of the 3D stabbing query structures that can work with the new scheme. This extra flexibility further allows us to use and design different stabbing query structures to achieve new results.

3.2 Preliminaries

In this section, we introduce some notation and previous results used in this chapter.

3.2.1 Notation

Throughout this chapter, we assume points are in general positions unless otherwise specified. In three-dimensional space, we call a box B *canonical* if it is of the form $[x_1, +\infty) \times [y_1, y_2) \times [z_1, z_2)$, where $x_1, y_1, z_1 \in \mathbb{R}$ and $y_2, z_2 \in \mathbb{R} \cup \{+\infty\}$. We use $B.x_1$ to refer to the lower bound of the x-range of B, $B.y_1$ and $B.y_2$ to respectively refer to the lower and upper bounds of the y-range of B, and so on. Let $(p.x, p.y, p.z)$ denote the coordinates of a point P. We say a point $q \in \mathbb{R}^3$ *dominates* another point $p \in \mathbb{R}^3$ if $q.x \geq p.x$, $q.y \geq p.y$ and $q.z \geq p.z$ hold simultaneously.

3.2.2 Colored 2D Orthogonal Range Emptiness

An *orthogonal range emptiness* query determines whether an axis-aligned query rectangle contains at least one point in the point set P. Observe that a solution to this query problem directly leads to a solution to the colored version of this problem called *colored orthogonal range emptiness*, in which each point in P is colored in one of C different colors, and given a color c and an axis-aligned rectangle, the query asks whether the query range contains at least one point colored in c. The reduction works as follows: For each color $1 \leq c \leq C$, let P_c denote the subset of P containing all points colored in c. If we construct an orthogonal range emptiness structure over P_c for each color c, then we can answer a colored orthogonal range emptiness query by querying the structure constructed over the points with the query color. The following lemma thus directly follows from the work of Chan et al. [14] on 2D orthogonal range emptiness queries:

Lemma 4 *Let $0 < \lambda < 1$ be any small constant. Given n colored points in 2-dimensional rank space, there is a data structure occupying $n \cdot f(n)$ words that answers colored 2D orthogonal range emptiness queries in $g(n)$ time, where*

a) $f(n) = O(1)$ and $g(n) = O(\lg^{\lambda} n)$ or

b) $f(n) = O(\lg \lg n)$ and $g(n) = O(\lg \lg n)$.

3.2.3 Segment Trees and Interval Trees

The *segment tree* [6] employed in this chapter is a balanced binary tree. Given a set S of n segments on the real line, the segment tree T constructed over these segments has $O(n)$ leaves. Let $\{p_0, p_1, p_2, ..., p_m, p_{m+1}\}$ be the increasingly sorted list of distinct endpoints of the segments in S, where $m \leq 2n$, and let p_0 and p_{m+1} denote $-\infty$ and $+\infty$, respectively. The real line is then partitioned into *elementary intervals*, $(p_0, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \cdots, (p_{m-1}, p_m), [p_m, p_m]$, and (p_m, p_{m+1}), in the order from left to right. The segment tree, as a balanced binary tree, has the same number of leaves as the number of elementary intervals. Each node v of T corresponds to an interval $I(v)$: The i-th leaf of T from left corresponds to the i-th elementary interval listed before, and the interval $I(v)$ corresponding to an internal node v is the union

of the elementary intervals corresponding to the leaves in the subtree rooted at v. A segment $[l, r]$ *spans* a node $v \in T$ if interval $I(v) \subseteq [l, r]$, and for each segment $s \in S$, we say that s is *relevant* to node v if s spans node v but does not span the parent of v. In this way, any segment is relevant to at most two nodes of T at the same tree level. We store segments in S at the nodes that they are relevant to. Given that the height of T is $O(\lg n)$, each segment can be stored at $O(\lg n)$ different nodes of the tree, and thus the segment tree constructed over S uses $O(n \lg n)$ words of space. The examples in Figures 3.1 and 3.2 use segment trees to organize data.

An *interval tree*, as a binary tree, can be defined in a recursive way. Let S denote a set of n intervals that the interval tree is constructed upon. If S has merely a single interval, then the interval tree is a leaf storing S. Otherwise, let v denote the root node and let $m(v)$ be the median of the $2n$ endpoints of the intervals from S. We store a set, $I(v)$, of intervals at node v such that $I(v)$ consists of all the intervals from S that cover median $m(v)$. Let $I_\ell(v)$ and $I_r(v)$ denote the sets of intervals from S that stay completely to the left and the right of $m(v)$, respectively. The left (resp. right) subtree of v is an interval tree for the set $I_\ell(v)$ (resp. $I_r(v)$). Note that given that $m(v)$ is the median, we have $|I_\ell(v)| \leq |S|/2$ and $|I_r(v)| \leq |S|/2$. After at most $\lg n$ recursive levels, both sets $I_\ell(v)$ and $I_r(v)$ are empty. Hence, the interval tree for the interval set S has at most $\lg n$ tree levels. Since each interval from S is stored in exactly one node of the tree, the interval tree is a linear space data structure. Figure 3.6 gives an example. In this example, the median of all interval endpoints is 7. Since only intervals $[3, 13)$ and $[2, 12)$ contain the median, they are assigned to the set $I(v')$, where v' is the root node. The set $I_\ell(v')$ contains intervals $[1, 4)$ and $[5, 7)$, as they completely stay on the left side of median. Symmetrically, the remaining intervals $[8, 15)$ and $[10, 14)$ are on the right side of the median, so we put them into the set $I_r(v')$. Then constructing the subtrees rooted at the left and right children of v' recursively upon interval sets $I_\ell(v')$ and $I_r(v')$ results in the interval tree shown in the figure.

3.2.4 Orthogonal Stabbing Queries over 3D Canonical Boxes

In the *3D stabbing counting* problem, we preprocess a set of 3D boxes, such that given a query point q, we can compute the number of boxes containing q efficiently, while in

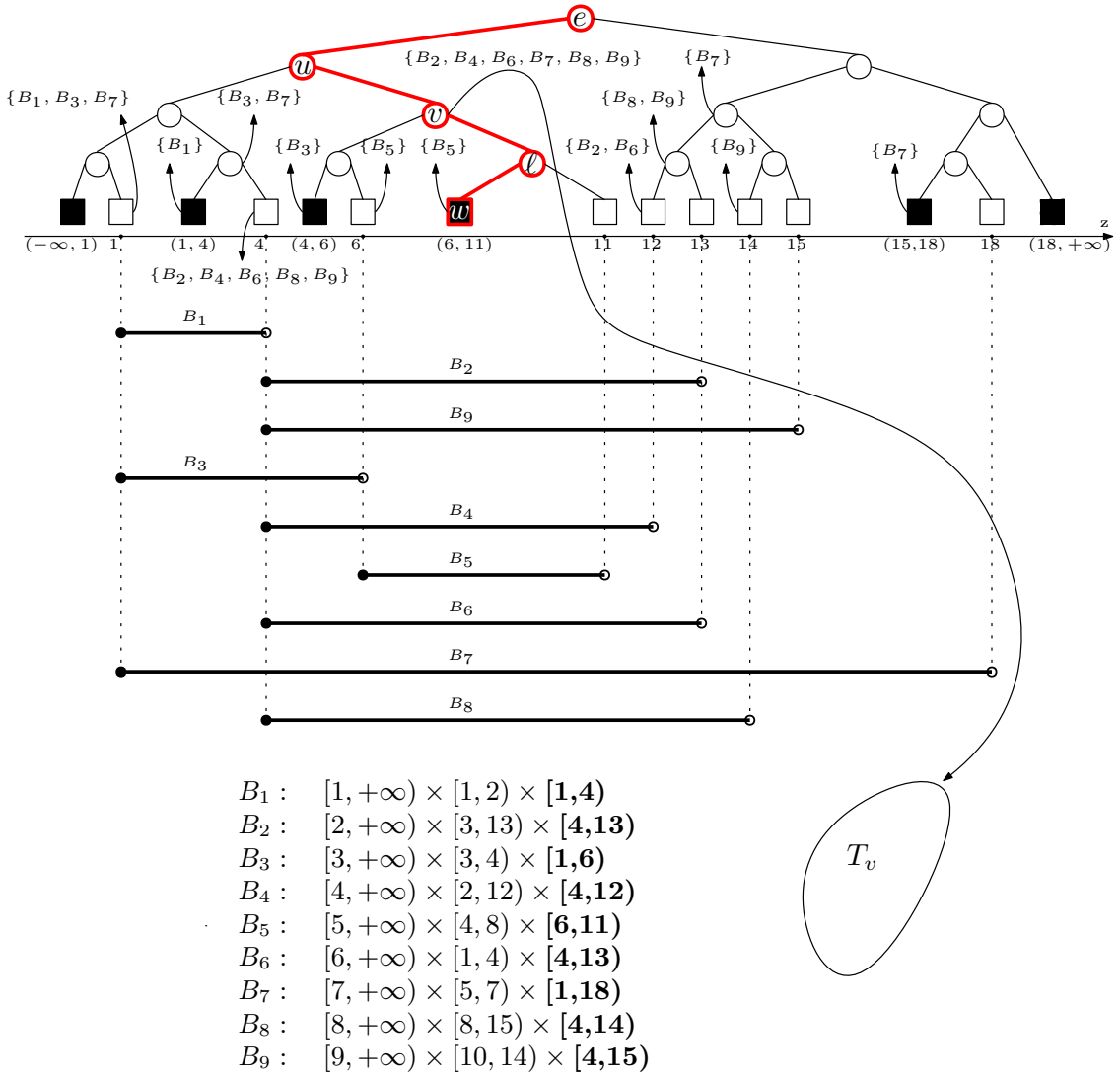


Figure 3.1: An example of the two layers of segment trees constructed upon a set of 9 canonical boxes, $B_1, B_2, \cdots, B_9$. All the empty elementary intervals, such as $(11, 12)$ and $(12, 13)$, are omitted. The path colored in red shows the query procedure for the query point $(q.x = 5, q.y = 10, q.z = 7)$. The second-layer segment tree T_v stored at node v is shown in Figure 3.2.

the *3D stabbing reporting* query problem, we report these boxes. In both our solution and the solution of Kaplan et al. [55], a special case of the 3D stabbing searching problem is considered, in which each box is a canonical box.

Here we introduce the data structure of Kaplan et al. [55] that solves the stabbing query problems over a set of n canonical boxes in 3D, as our first solution to the colored range counting problem augments this data structure and uses it as a component.

The data structure consists of two layers of segment trees. The structure at the top-layer is a segment tree constructed over the z-segments of the boxes. More precisely, we project each box onto the z-axis to obtain a segment, which we refer to as the *z-segment* of this box, and the top-layer segment tree is constructed over the z-segments of all the boxes[1]. A box is assigned to a node v in the segment tree if the z-segment of the box is relevant to v. Let $B(v)$ denote the set of boxes that are assigned to each node v in the top-layer segment tree. Then we construct a segment tree over the y-segments of the boxes in $B(v)$, and refer to the newly constructed tree as the bottom-layer segment tree. Again, to each node v' of this bottom-layer segment tree, we assign a subset of boxes in $B(v)$ in which the y-segment of each box contains interval $I(v')$ (corresponding to node v'). Specifically, we sort these boxes assigned to v' by their x_1-coordinates, i.e., the left endpoints of their projections on the x-axis, and store them in a list, denoted by $sList(v')$. All the bottom-layer segment trees with a list $sList(v')$ of boxes being stored at each node together form the bottom-layer structure. Observe that in the bottom-layer, each internal node has at most two children, and the boxes stored there are sorted by their x_1-coordinates. So we can use the fractional cascading [19] data structure to link together the box lists stored at each non-root node and its parent to speed up the query time by a logarithmic factor.

Figures 3.1 and 3.2 together give an example of the two layers of segment trees constructed upon 9 canonical boxes, $B_1, B_2, \cdots, B_9$. In particular, the segment tree in Figure 3.1 shows the top-layer segment tree constructed upon the z-segments of the boxes. The segments immediately below z-axis indicate these z-segments. The real line along the z-axis is divided into 15 elementary intervals by the segment endpoints. In the tree, leaves in white correspond to segment endpoints, while each black leaf represents a non-empty interval between two consecutive endpoints. All the empty elementary intervals are omitted due to space limitation. For example, the interval $I(v)$ of node v in the top-layer tree is set to be $(4, 11]$, which is the union of the elementary intervals corresponding to the leaf descendants of node v, and set $B(v)$ contains boxes B_2, B_4, B_6, B_7, B_8, and B_9, as the z-segments of these boxes are relevant to node v. Note that all the boxes that are assigned to the nodes along a path

[1]Later, we also use *y-segments* of the boxes to refer to the segments obtained by projecting these boxes onto the y-axis.

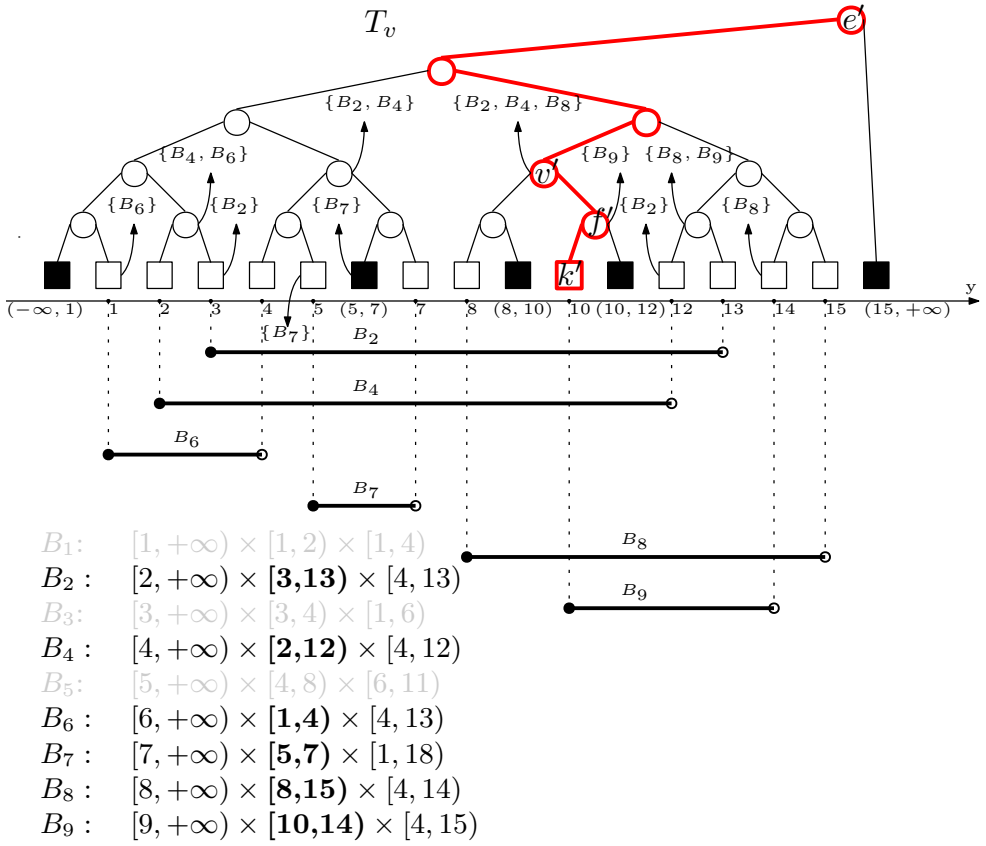


Figure 3.2: An example of a second-layer segment tree constructed over y-segments of the boxes B_2, B_4, B_6, B_7, B_8 and B_9 given in Figure 3.1. All the empty elementary intervals are omitted. The path colored in red shows the query procedure for the query point $(q.x = 5, q.y = 10, q.z = 7)$.

from the root to a leaf are pairwise distinct. The segment tree in Figure 3.2 represents the bottom-layer segment tree constructed upon the y-segments of the boxes in $B(v)$. The boxes stored at each node of the bottom-layer segment tree are sorted by their x-coordinates. In the figure, $sList(v')$ stores boxes B_2, B_4 and B_8 in the order.

Let $(q.x, q.y, q.z)$ be the query point. The query algorithm first performs a search in the top-layer segment tree to locate the $O(\lg n)$ nodes such that the z-segments of the boxes stored on these nodes contain $q.z$ (the z-segments of a canonical box, $[x_1, +\infty) \times [y_1, y_2) \times [z_1, z_2)$, contains $q.z$ if $z_1 \leq q.z < z_2$). Among these boxes, we need to identify the ones whose projections on the xy-plane contain point $(q.x, q.y)$. To achieve that, we query the bottom-layer segment tree constructed for each of the $O(\lg n)$ nodes returned before. The searches within the bottom-layer structures end at $O(\lg^2 n)$ nodes of the bottom-layer trees, and each of these nodes stores a box list

sList, in which these boxes are unbounded in the positive x direction and sorted by their x_1-coordinates. As a result, in the same *sList*, the boxes that contain the query point forms a prefix of that *sList*. Observe that in the same bottom-layer segment tree, i) the nodes visited by a query lie in a path from the root of the bottom-layer segment tree to a leaf, ii) and we always search for $q.x$ at these nodes. So we apply a binary search over the *sList* stored at the root of the bottom-layer segment tree to find the prefix size and then apply the fractional cascading data structure to find the prefix sizes for other nodes in the root-to-leaf path. In a bottom-layer segment tree, we spend $O(\lg n + \lg n)$ time instead of $O(\lg n \cdot \lg n)$ time, thereby saving a logarithmic factor, due to the factional cascading data structure.

For example, let $(5, 10, 7)$ be the query point. First, the search is performed in the top-layer segment tree. As shown in Figure 3.1, the element interval of the leaf w, which is $(6, 11)$, in the top-layer tree contains $q.z$, and thus the query visits $O(\lg n)$ nodes in the path from the root e to the leaf w. The boxes stored in these nodes consist of $B_2, B_4, B_6, B_7, B_8, B_9$ and B_5, and the z-segment of each box contains $q.z$. Then, the query proceeds over the bottom-layer segment trees that are constructed over the y-segments of the boxes in the sets, $B(e), B(u), B(v), B(\ell)$ and $B(w)$. Especially, the segment tree in Figure 3.2 shows the bottom-layer segment tree that is constructed for the boxes in $B(v)$. Similarly, the query visits $O(\lg n)$ nodes in the path from the root e' to the leaf k' in the bottom-layer tree, since the y-segments of the boxes stored in the non-empty *sList*'s of these nodes all contain $q.y$. Those non-empty lists include $sList(v')$ and $sList(f')$. As mentioned before, the boxes in $sList(v')$ and $sList(f')$ are disjoint. In each *sList*, the boxes that contains the query point, if there are any, form a prefix of *sList*, as all the boxes in the same list are sorted by their x_1-coordinates. For instance, the boxes that contain the query point in $sList(v')$ are B_2 and B_4, and both boxes together form a prefix of $sList(v')$. Identifying the prefix, as the last step of the query, can be achieved by a binary search over the x_1-coordinates of the boxes.

Lemma 5 ([55]) *Given a set of n canonical boxes in three dimensions, the above data structure occupies $O(n \lg^2 n)$ words and answers stabbing counting queries in $O(\lg^2 n)$ time and stabbing reporting queries in $O(\lg^2 n + k)$ time, where k denotes the number of boxes reported. Furthermore, the output of the reporting query is the union of $O(\lg^2 n)$ disjoint subsets, each containing the boxes stored in a nonempty prefix of*

a bottom-layer sorted list. The preprocessing time is $O(n \lg^2 n)$.

3.2.5 Reducing Colored 2D 3-Sided Range Counting to 3D Stabbing Counting over Canonical Boxes

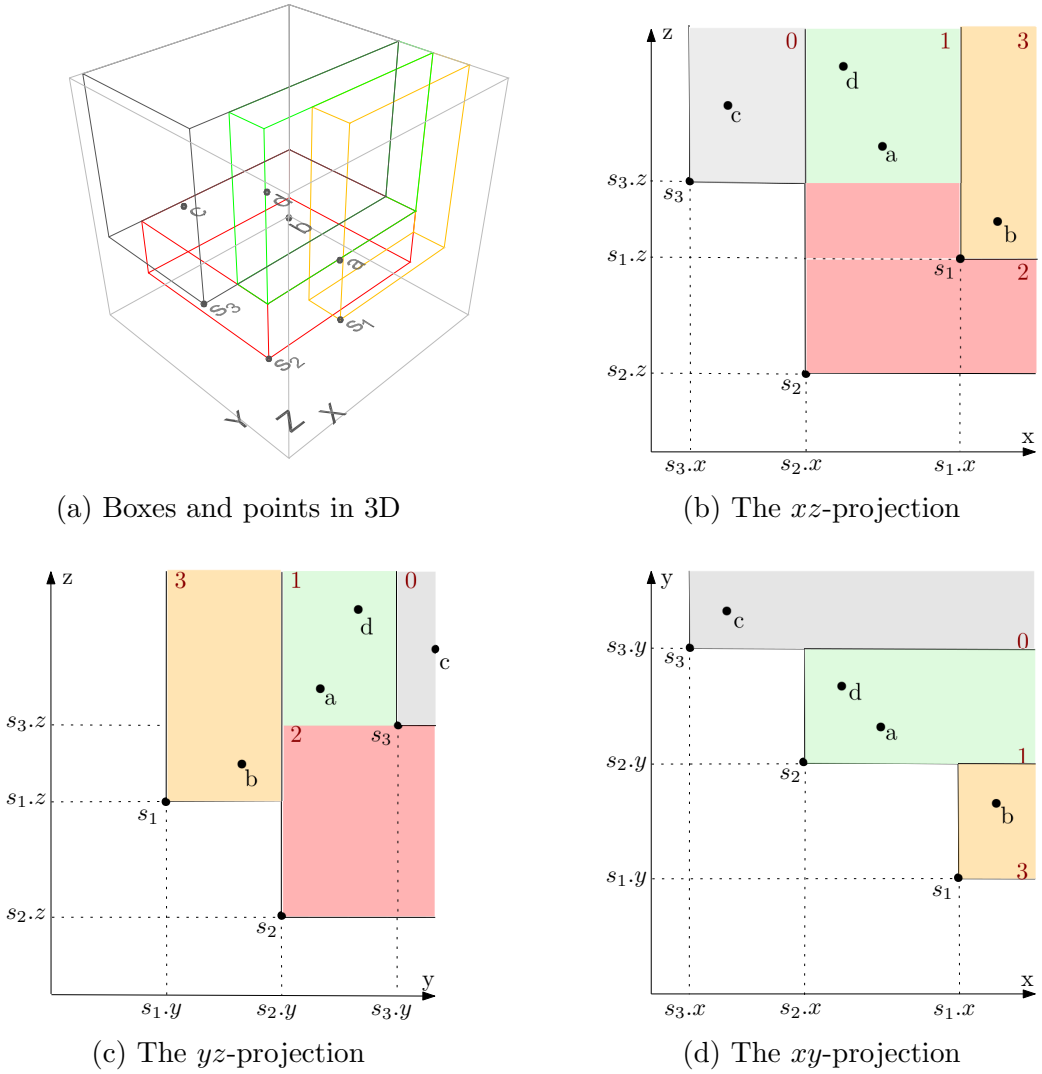
A key technique used in both the solutions of Kaplan et al. [55] and ours is a reduction from colored 2D 3-sided range counting queries, in which each query range is of the form $[a, b] \times [c, +\infty)$ for some $a, b, c \in \mathbb{R}$, to 3D orthogonal stabbing queries over canonical boxes[2]. The reduction is performed in two steps. First we reduce the colored 2D 3-sided range counting query problem to the 3D colored dominance counting problem, in which we preprocess a set, P, of colored points in $\mathbb{R}^3$ such that given a query point q in $\mathbb{R}^3$, one can report the number of distinct colors in $P \cap (-\infty, q.x] \times (-\infty, q.y] \times (-\infty, q.z]$ efficiently. This reduction works as follows: For each point, $p = (p.x, p.y)$, we create a point, $p' = (-p.x, p.x, -p.y)$, in $\mathbb{R}^3$, and assign to it the color of P. Then a colored 2D 3-sided range counting query over the original points in which the query range is $[a, b] \times [c, +\infty)$ can be answered by performing a 3D colored dominance query over the newly created points, using $(-\infty, -a] \times (-\infty, b] \times (-\infty, -c]$ as the query range.

To further reduce the 3D colored dominance counting problem to 3D orthogonal stabbing counting over canonical boxes, we need some additional notation: Given a point P in 3D, let Q_p^+ denote region $[p.x, +\infty) \times [p.y, +\infty) \times [p.z, +\infty)$, and given a point set A, let $U(A)$ denote the region of $\bigcup_{p \in A} Q_p^+$. Then the following lemma is crucial:

Lemma 6 ([55]) *Given a set, A, of n points in three-dimensional space, a set of $O(n)$ pairwise disjoint 3D canonical boxes can be computed in $O(n \lg^2 n)$ time such that the union of these boxes is the region $U(A)$.*

Originally Kaplan et al. [55, Theorem 2.1] proved Lemma 6 for d-dimensional space, where $d \geq 1$; for a general d, the number of boxes required to cover $U(A)$ is $O(n^{\lfloor d/2 \rfloor})$. To help readers understand this decomposition, Figure 3.3 shows an example in 3D space.

[2]Later, we deal with query ranges of the form $[a, b] \times (-\infty, d]$ for some $a, b, d \in \mathbb{R}$, and a similar reduction also works.



(a) Boxes and points in 3D

(b) The xz-projection

(c) The yz-projection

(d) The xy-projection

Figure 3.3: An example of dividing $U(A)$, where $A = \{s_1, s_2, s_3, a, b, c, d\}$ is a set of points in 3D, into 4 canonical disjoint boxes, including $[s_1.x, +\infty) \times [s_1.y, s_2.y) \times [s_1.z, +\infty)$ in orange (encoded by 3), $[s_2.x, +\infty) \times [s_2.y, +\infty) \times [s_2.z, s_3.z)$ in red (encoded by 2), $[s_2.x, +\infty) \times [s_2.y, s_3.y) \times [s_3.z, +\infty)$ in green (encoded by 1), and $[s_3.x, +\infty) \times [s_3.y, +\infty) \times [s_3.z, +\infty)$ in gray (encoded by 0). Figures 2(b), 2(c) and 2(d) show the projections of the boxes on xy-, xz-, and yz-planes, respectively. Points a, b, c and d are in the interior of the boxes, while points s_1, s_2 and s_3 are on the box boundaries.

With this lemma, the reduction works as follows. Let P denote the input colored point set of the 3D colored dominance counting problem, and let C denote the number of colors. Then for each color $c \in [C]$, we apply Lemma 6 to partition $U(P_c)$, where P_c is the set of all the points in P that are colored c, into a set, B_c, of $O(|U(P_c)|)$ disjoint 3D canonical boxes. We then construct a 3D stabbing counting structure

over $B = \bigcup_{c=0}^{C-1} B_c$. Note that this data structure is constructed over $O(|P|)$ canonical boxes, as $\sum_{c=0}^{C-1} |B_c| = \sum_{c=0}^{C-1} O(|P_c|) = O(|P|)$. To answer a 3D colored dominance counting query over P, for which the query range is the region dominated by a point q, observe that if q dominates at least one point in P_c, then it must be located within $U(P_c)$. Since $U(P_c)$ is partitioned into the boxes in B_c, we conclude that q dominates at least one point in P_c iff q is contained in a box in B_c. Furthermore, since the boxes in B_c are pairwise disjoint, q is either contained in exactly one box in B_c or outside $U(P_c)$. Hence, the number of distinct colors in the region dominated by q is equal to the number of boxes in B that contains q, which can be computed by performing a stabbing counting query in B using q as the query point.

3.3 A New Framework of Achieving Time-Space Tradeoffs

We present three new solutions to colored 2D orthogonal range counting in this section and Section 3.4. They follow the same framework, of which we give an overview in Section 3.3.1. One key component of this framework is a novel scheme of computing the sizes of the intersections between the color sets assigned to different subsets of points that lie within the query range; Section 3.3.2 describes this scheme and shows how to combine it with Lemma 5 to immediately achieve a new time-space tradeoff for colored 2D orthogonal range counting.

3.3.1 Overview of the Data Structure Framework

Let P denote a set of n points on the plane, each assigned a color identified by an integer in $[C]$. To support colored orthogonal range counting over P, we construct a binary range tree T over the y-coordinates of the points in P such that each leaf of T stores a point of P, and from left to right, the points stored in the leaves are increasingly sorted by their y-coordinates. For each internal node v of T, we construct the following data structures:

- $P(v)$, a list that contains the points stored at the leaf descendants of v, sorted by x-coordinate;
- $P_y(v)$, a sorted list that contains the y-coordinates of the points in $P(v)$;

- $S(v_l)$ (resp. $S(v_r)$), the data structure for colored 2D 3-sided range counting that is constructed over $P(v_l)$ (resp. $P(v_r)$), where v_l (resp. v_r) is the left (resp. right) child of v; $S(v_l)$ (resp. $S(v_r)$) requires query ranges to be open at the top (resp. bottom), e.g., $[a, b] \times [c, +\infty)$ (resp. $[a, b] \times (-\infty, d]$); the specific data structures to be adopted will be decided later for different tradeoffs; and

- $E(v_l)$ (resp. $E(v_r)$), the data structure for colored 2D orthogonal range emptiness query that is constructed by the data structure of Lemma 4 over the point set in rank space converted from $P(v_l)$(resp. $P(v_r)$).

Let $Q = [a, b] \times [c, d]$ be the query rectangle. Given an internal node v, let $C_Q(v)$ denote the set of distinct colors assigned to points in $P(v) \cap Q$. We first find the leftmost and rightmost leaves in which the y-coordinate of the point stored is no less than c and no more than d, respectively. Then we locate the lowest common ancestor, u, of both leaves on T. As all the points from P that are in the query range Q must be in $P(u)$, $|C_Q(u)|$ is the answer to the query. To compute $|C_Q(u)|$, let u_l and u_r denote the left and right children of u, respectively. By the inclusion-exclusion principle, we know that $|C_Q(u)| = |C_Q(u_l)| + |C_Q(u_r)| - |C_Q(u_l) \cap C_Q(u_r)|$. Among the terms on the right hand side of this equation, $|C_Q(u_l)|$ and $|C_Q(u_r)|$ can be computed by performing colored 2D 3-sided range counting queries over $S(v_l)$ and $S(v_r)$, using $[a, b] \times [c, +\infty)$ and $[a, b] \times (-\infty, d]$ as query ranges, respectively. What remains is to compute $|C_Q(u_l) \cap C_Q(u_r)|$.

This idea of decomposing a 4-sided query range into two 3-sided query ranges has been used before for both colored 2D orthogonal range reporting [41] and counting [55]. Furthermore, to support colored 2D 3-sided range counting, we reduce it to 3D stabbing queries over canonical boxes, applying the reduction of Kaplan et al. [55] (summarized in Section 3.2). Thus, the techniques summarized so far have been used in previous work (without the construction of range emptiness structures). Our main contribution is a new scheme that computes $|C_Q(u_l) \cap C_Q(u_r)|$ and achieves time-space tradeoffs. This new scheme gives us more *flexibility* in the design of 3D stabbing query structures, thus allowing us to achieve new results.

Next we describe the conditions that a 3D stabbing query structure must meet so that we can combine it with our scheme of computing $|C_Q(u_l) \cap C_Q(u_r)|$, while

deferring the details of our scheme to Section 3.3.2. The stabbing query structure consists of multiple layers of trees of some kind: The top-layer tree is constructed over the entire set of canonical boxes, and each of its nodes is assigned a subset of boxes; the second layer consists of a set of trees, each constructed over the boxes assigned to a node in the top-layer tree, and so on; in the end, each bottom-layer tree node is assigned a list of boxes in a certain order as well. Henceforth, we refer to the complete box list assigned to a node of the bottom-layer tree as a *bottom list* for convenience, e.g., the $sList$ in the data structure for Lemma 5. We require that the query algorithm locates a set S of bottom-layer tree nodes: For each node $v \in S$, there exists a nonempty prefix of the bottom list assigned to v such that these prefixes over all the nodes in S form a partition of the set of boxes containing the query point q; furthermore, the size of each of these prefixes can be computed efficiently and that each box in a prefix can also be reported efficiently.

For example, Lemma 5 satisfies these conditions and can be used in our framework. In Lemma 5, the data structure contains two layers of segment trees. In the top-layer tree, a set, $B(u)$, of boxes is assigned to each node u, and we construct a new segment tree upon $B(u)$ at the bottom-layer. As a result, the bottom-layer consists of $O(n)$ segment trees. At each bottom-layer tree node v, we explicitly store the boxes in a bottom list, $sList(v)$, sorted by their single x_1-coordinates. Due to this, if there are boxes in $sList(v)$ that contain the query point, then these boxes form a prefix of $sList(v)$, and the size of the prefix can be identified by a predecessor search, and once the prefix is identified, each box that contains the query point can be reported in constant time.

Even though Kaplan et al. proved Lemma 5 and used it successfully in their $O(n^2 \lg^2 n)$ space solution, they cannot directly use it with their scheme of achieving time-space tradeoffs. Indeed, in the scheme designed by Kaplan et al., the list of boxes containing the query point are also decomposed into a bounded number of sub-lists. However, their scheme requires each of the decomposed sub-list to be equal to a bottom list stored at a bottom-layer tree node. With respect to Lemma 5, one must makes sure that in each nonempty $sList$ in the bottom-layer segment trees, either all boxes contain the query point or there is no box that contains the query point. Obviously, the data structure, two layers of segment trees, fails the requirement of

their scheme: As shown in Figure 3.1, only the first two boxes of $sList(v')$ in the bottom-layer segment tree contain the query point $(5, 10, 7)$, while the third box does not. Instead, Kaplan et al. expand this structure with a third layer which is a segment tree constructed over x-coordinates of the boxes, increasing both time and space costs. On the other hand, our scheme relaxes this restriction. More specifically, if a $sList$ has some boxes that contain the query point, instead of requiring that the query point must be contained by all the boxes in the list, we only need to make sure that the query point is contained in the boxes in a prefix of that $sList$. For example, as mentioned before, although not all the boxes in $sList(v')$ contain the query point, the boxes that do contain the query point, B_2 and B_4, form a prefix of $sList(v')$. This extra flexibility allows us to use Lemma 5 and alternative 3D stabbing query structures (to be shown later in Section 3.4) in our framework.

3.3.2 Computing Intersections between Color Sets

We now introduce our new scheme of computing $|C_Q(u_l) \cap C_Q(u_r)|$ and then combine it with the stabbing query structure from Lemma 5 to achieve a new time-space tradeoff for colored 2D orthogonal range counting. Since our scheme works with some other stabbing query structures, we describe it assuming a stabbing query structure satisfying the conditions described in Section 3.3.1 is used. To understand this scheme more easily, it may be advisable for readers to think about how it applies to the stabbing query structure of Lemma 5.

Recall that the problems of computing $|C_Q(u_l)|$ and $|C_Q(u_r)|$ have each been reduced to a 3D stabbing query. Furthermore, for each stabbing query, all reported boxes are distributed into a number of disjoint bottom lists, and the boxes in the same bottom list stored at a bottom-layer tree node form a nonempty prefix. For the stabbing query that is performed to compute $|C_Q(u_l)|$, we define D_Q to be a list in which each element is one of the non-empty disjoint prefixes, and the union of all these prefixes forms the list of reported boxes. U_Q is defined in a similar way for $|C_Q(u_r)|$. For instance, if we use the data structure for Lemma 5 to answer these stabbing queries, D_Q and U_Q therein each contain $O(\lg^2 n)$ prefixes, and thus both $|D_Q|$ and $|U_Q|$ are upper bounded by $O(\lg^2 n)$. As shown in Section 3.2.5, when reducing 2D 3-sided colored counting to 3D stabbing queries over canonical boxes,

we decompose the region $U(P_c)$ constructed by the points colored by c into pairwise disjoint canonical boxes. For convenience, we assign to each canonical box color c, if the box is part of $U(P_c)$, for each $c \in [C]$. Note that at most one canonical box colored in c contains the query point, which implies that each box in $\bigcup_{s \in D_Q} s$ (resp. $\bigcup_{t \in U_Q} t$) has a distinct color. For each set $s \in D_Q$ and $t \in U_Q$, let $C(s)$ and $C(t)$ denote the set of colors associated with the boxes in s and t, respectively. Then we have $|C_Q(u_l)| = \sum_{s \in D_Q} |C(s)|$, $|C_Q(u_r)| = \sum_{t \in U_Q} |C(t)|$, and $|C_Q(u_l) \cap C_Q(u_r)| = \sum_{s \in D_Q, t \in U_Q} |C(s) \cap C(t)|$.

To compute $\sum_{s \in D_Q, t \in U_Q} |C(s) \cap C(t)|$, extra preprocessing steps are required. For each node v in the binary range tree T, we construct a matrix $M(v)$ as follows: Let $X \in [1, n]$ be an integer parameter to be chosen later. If the length, m, of a bottom list in the stabbing query structure $S(v_l)$ or $S(v_r)$ is greater than X, we divide the list into $\lceil m/X \rceil$ blocks such that each block is of length X, with the possible exception of the last block. If $m \leq X$, then the entire bottom list is seen as a single block. If a block is of length X, we call it a *full block*. Let $b_l(v)$ and $b_r(v)$ denote the total numbers of full blocks over all bottom lists in $S(v_l)$ and $S(v_r)$, respectively. Then $M(v)$ is a $b_l(v) \times b_r(v)$ matrix, in which each row (or column) corresponds to a nonempty prefix of a bottom list in $S(v_l)$ (or $S(v_r)$) that ends with the last entry of a full block, and each entry $M(v)[i, j]$ stores the number of colors that exist in both the set of colors assigned to the boxes in the prefix corresponding to row i and the set of colors assigned to the boxes in the prefix corresponding to column j. See Figure 3.4 for an example.

To bound the size of $M(v)$, we define the *duplication factor*, $\delta(n)$, of a stabbing query structure that satisfies the conditions in Section 3.3.1 to be the maximum number of bottom lists that any canonical box can be contained in. For example, the duplication factor of the structure for Lemma 5 is $O(\lg^2 n)$. (To see this, observe that in a segment tree constructed over n segments, each segment is stored in $O(\lg n)$ tree nodes. Since segment trees are used in both layers of the data structure for Lemma 5, each box can be stored in $O(\lg^2 n)$ different bottom lists.) As each full block contains X boxes, the numbers of full blocks in the bottom lists of $S(v_l)$ and $S(v_r)$ are at most $\lfloor \delta(n)|P(v_l)|/X \rfloor$ and $\lfloor \delta(n)|P(v_r)|/X \rfloor$, respectively. Therefore, $M(v)$ occupies $O(\delta(n)^2|P(v_l)||P(v_r)|/X^2) = O((\delta(n)|P(v)|/X)^2)$ words.

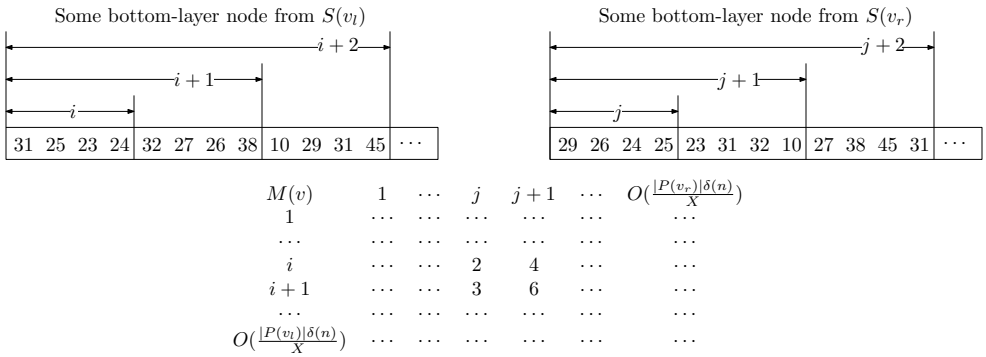


Figure 3.4: An example of matrix $M(v)$. Matrix $M(v)$ is an $O(\frac{|P(v_l)|\delta(n)}{X}) \times O(\frac{|P(v_r)|\delta(n)}{X})$ matrix. The integers in the left and right rectangles represent the colors of boxes in the *sList*'s that are stored at some bottom-layer nodes of $S(v_l)$ and $S(v_r)$, respectively. The box colors that are in the same *sList* are distinct, and they are divided into blocks. Especially, in this example, the block size is 4. Each row (resp. column) of $M(v)$ corresponds to a nonempty prefix of a bottom list in $S(v_l)$ (resp. $S(v_r)$) that ends with the last entry of a full block. In the figure, the i-th row and the j-th column of $M(v)$ correspond to the prefix of $S(v_l)$ marked by number-i and the prefix of $S(v_r)$ marked by number-j on the top, respectively. In particular, the prefix marked by $i+1$ in $S(v_l)$ and the prefix marked by $j+1$ in $S(v_r)$ share 6 distinct colors, which are $\{31, 25, 23, 24, 32, 26\}$; hence, $M(v)[i+1, j+1]$ is set to 6.

With these matrices, the computation of $\sum_{s\in D_Q, t\in U_Q} |C(s) \cap C(t)|$ can proceed as follows. Given that each $s \in D_Q$ is a prefix of a bottom list, s can be split into two parts: s_h which is the (possibly empty) prefix of s that consists of all the full blocks entirely contained in s, and s_l which contains the remaining (less than X) boxes of s. For each $t \in U_Q$, we split t into t_h and t_l in a similar way. Thus we have $C(s_h) \cup C(s_l) = C(s)$ and $C(t_h) \cup C(t_l) = C(t)$. Since no two boxes in s have the same color and the same applies to the boxes in t, $C(s_h) \cap C(s_l) = C(t_h) \cap C(t_l) = \emptyset$ also holds. Thus, we have

$$\begin{aligned}
&\sum_{s\in D_Q, t\in U_Q} |C(s) \cap C(t)| \\
&= \sum_{s\in D_Q, t\in U_Q} (|C(s_h) \cap C(t_h)| + |C(s_h) \cap C(t_l)| + |C(s_l) \cap C(t)|) \\
&= \sum_{s\in D_Q, t\in U_Q} |C(s_h) \cap C(t_h)| + \sum_{s\in D_Q, t\in U_Q} |C(s_h) \cap C(t_l)| \\
&+ |(\bigcup_{s\in D_Q} C(s_l)) \cap \bigcup_{t\in U_Q} C(t)|.
\end{aligned} \tag{3.1}$$

For the first term in the last line of Equation 3.1, we can retrieve $|C(s_h) \cap C(t_h)|$ from the matrix $M(v)$ for each possible pair of s_h and t_h and sum them up. Therefore, the first term can be computed in $O(|D_Q| \cdot |U_Q|)$ time. For the third term, observe that $\bigcup_{t \in U_Q} C(t) = C_Q(u_r)$; thus, the third term can be computed by performing a colored 2D orthogonal range emptiness query over $P(u_r)$ with c as the query color for each color $c \in \bigcup_{s \in D_Q} C(s_l)$ and with Q as the query range. Note that the range emptiness query data structure $E(v_r)$ defined in Section 3.3.1 is built upon the points $\hat{P}(v_r)$ in rank space. We need to reduce Q into rank space with respect to $\hat{P}(v_r)$ before performing colored range emptiness queries. Note that all these queries share the same query range, and thus we need only convert Q into rank space once. This can be done by performing binary searches in $P(v)$ and $P_y(v)$ in $O(\lg n)$ time. As $|\bigcup_{s \in D_Q} C(s_l)| = |\bigcup_{s \in D_Q} s_l| = O(|D_Q| \cdot X)$, it requires $O(\lg n + X|D_Q| \cdot (g(n) + \tau(n)))$ time to compute these colors and then answer all these queries, where $g(n)$ denotes the query time of each range emptiness query in Lemma 4 and $\tau(n)$ denotes the query time of reporting a box and its color in the query range.

Finally, to compute the second term in the last line of Equation 3.1, observe that,

$$\sum_{s \in D_Q, t \in U_Q} |C(s_h) \cap C(t_l)| = \\ |(\bigcup_{s \in D_Q} C(s)) \cap (\bigcup_{t \in U_Q} C(t_l))| - |(\bigcup_{s \in D_Q} C(s_l)) \cap (\bigcup_{t \in U_Q} C(t_l))|. \tag{3.2}$$

The first term of the right hand side of Equation 3.2 can be computed in $O(\lg n + X|U_Q| \cdot (g(n) + \tau(n)))$ time, again by performing range emptiness queries, but this time we use data structure $E(v_l)$. For the second term, we retrieve and sort the colors in $\bigcup_{s \in D_Q} C(s_l)$ and those in $\bigcup_{t \in U_Q} C(t_l)$, and by scanning colors in both sorted lists we can compute the intersection of the color sets. Given that $|(\bigcup_{s \in D_Q} C(s_l))|$ (resp. $|(\bigcup_{t \in U_Q} C(t_l))|$) are bounded by $O(X|D_Q|)$ (resp. $O(X|U_Q|)$), the two sets of colors can be retrieved in $O(X|D_Q|\tau(n))$ and $O(X|U_Q|\tau(n))$ time and then sorted using Han's sorting algorithm [43] in $O(X|D_Q| \lg\lg n)$ and $O(X|U_Q| \lg\lg n)$ time. Thus the second term in the last line of Equation 3.1 can be computed in $O(X(|U_Q| + |D_Q|)(\lg\lg n + \tau(n)))$ time. Overall, computing $|C_Q(u_l) \cap C_Q(u_r)|$ requires $O(\lg n + |D_Q| \cdot |U_Q| + X(|U_Q| + |D_Q|)(\lg\lg n + g(n) + \tau(n)))$ time. Lemma 7 summarizes the complexities of our framework.

Lemma 7 *Suppose that the 3D stabbing query structure of $S(v_l)$ (or $S(v_r)$) for each node $v \in T$ has duplication factor $\delta(n)$, occupies $O(|P(v)|h(n))$ words, and, given a query point q, it can compute $\phi(n)$ disjoint sets of boxes whose union is the set of boxes containing q in $O(\phi(n))$ time. Furthermore, each subset is a nonempty prefix of a bottom list, and after this prefix is located, its length can be computed in $O(1)$ time and each box in it can be reported in $O(\tau(n))$ time. Let $f(n)$ and $g(n)$ be the functions set in Lemma 4 to implement $E(v_l)$ and $E(v_r)$. Then the structures in our framework occupy $O((n\delta(n)/X)^2 + n \lg n(f(n) + h(n)))$ words and answer a colored 2D orthogonal range counting query in $O(\phi^2(n) + X\phi(n)(\lg\lg n + g(n) + \tau(n)) + \lg n)$ time, where X is an integer parameter in $[1, n]$.*

Proof. At each node v of T, we store

i) a pair of point lists $P(v)$ and $P_y(v)$,

ii) the colored range emptiness query structures $E(v_l)$ and $E(v_r)$,

iii) the stabbing query data structures $S(v_l)$ and $S(v_r)$,

iv) and the matrix $M(v)$.

Among them, both $P(v)$ and $P_y(v)$ use $|P(v)|$ words of space; $E(v_l)$ and $E(v_r)$ use $O(|P(v)|f(n))$ words of space by Lemma 4; $S(v_l)$ and $S(v_r)$ use $O(|P(v)|h(n))$ words of space; and the matrix $M(v)$ uses $O((\delta(n)|P(v)|/X)^2)$ words of space. Summing the space costs over all internal nodes of the range tree T, the overall space cost is at most $\sum_{v\in T} O((\delta(n)|P(v)|/X)^2+|P(v)|(f(n)+h(n)))$. To simplify this expression, first observe that $\sum_{v\in T} |P(v)| = O(n \lg n)$. Furthermore, we can calculate $\sum_{v\in T} |P(v)|^2$ as follows: At the i-th level of T, there are 2^i nodes, and each node stores a point list of length $n/2^i$. Therefore, the sum of the squares of the lengths of the point lists at the ith level is $n^2/2^i$. Summing up over all levels, we have $\sum_{v\in T} |P(v)|^2 = O(n^2)$. Therefore, the overall space cost simplifies to $O((n\delta(n)/X)^2 + n \lg n(f(n) + h(n)))$.

Given a query range Q, both $|C_Q(u_l)|$ and $|C_Q(u_r)|$ can be computed in $O(\phi(n))$ time, as both $C_Q(u_l)$ and $C_Q(u_r)$ are partitioned into $\phi(n)$ disjoint sets of boxes and the number of boxes in each set can be computed in constant time. As shown before, computing $|C_Q(u_l) \cap C_Q(u_r)|$ can be reduced to computing $\sum_{s\in D_Q, t\in U_Q} |C(s) \cap C(t)|$,

which requires $O(\lg n + |D_Q| \cdot |U_Q| + X(|U_Q| + |D_Q|)(\lg\lg n + g(n) + \tau(n)))$ time. Since both $|D_Q|$ and $|U_Q|$ are upper bounded by $\phi(n)$, the overall query time is $O(\lg n + \phi^2(n) + X\phi(n)(\lg\lg n + g(n) + \tau(n)))$. ■

Now, we are ready to present a new time-space tradeoff by combining the stabbing query data structure from Lemma 5 and our framework summarized by Lemma 7. Following both lemmas, we have $h(n) = O(\lg^2 n)$, $\phi(n) = O(\lg^2 n)$, $\tau(n) = O(1)$, and $\delta(n) = O(\lg^2 n)$. To implement $E(v_l)$ and $E(v_r)$, we use part b) of Lemma 4, so $f(n) = O(\lg\lg n)$ and $g(n) = O(\lg\lg n)$. Hence:

Theorem 1 *There is a data structure of $O((\frac{n}{X})^2 \lg^4 n + n\lg^3 n)$ words of space that answers colored 2D orthogonal range counting queries in $O(\lg^4 n + X\lg^2 n \lg\lg n)$ time, where X is an integer parameter in $[1, n]$. In particular, setting $X = \lceil\sqrt{n\lg n}\rceil$ yields an $O(n\lg^3 n)$-word structure with $O(\sqrt{n}\lg^{5/2} n \cdot \lg\lg n)$ query time.*

Unlike our result in Theorem 1, the solution of Kaplan et al. with $O((\frac{n}{X})^2 \lg^6 n + n\lg^4 n)$ words of space and $O(X\lg^7 n)$ query time works under the pointer machine model. Nevertheless, with some modifications, our solution can also be made to work under this same model. First, Lemma 4 requires the word RAM model, we can replace it by the optimal solution to the 2D orthogonal range emptiness query problem by Chazelle [17] with $O(n\lg n/\lg\lg n)$ words of space and $O(\lg n)$ query time. Thus, $g(n) = O(\lg n)$, while the overall space cost of the data structure remains unchanged. Second, when computing $|(\bigcup_{s\in D_Q} C(s_l)) \cap (\bigcup_{t\in U_Q} C(t_l))|$, we cannot use Han's sorting algorithm [43] which requires the word RAM. Instead, using mergesort, we can compute this value in $O(X(|U_Q| + |D_Q|) \cdot \lg n)$ time. Finally, to simulate a matrix $M(v)$, we can use lists indexed by binary search trees, so that we can retrieve each entry in $O(\lg n)$ time. Thus, we achieve the following result:

Corollary 1 *Under the arithmetic pointer machine model, given n colored points on the plane, there is a data structure of $O((\frac{n}{X})^2 \lg^4 n + n\lg^3 n)$ words of space that answers colored orthogonal range counting queries in $O(\lg^5 n + X\lg^3 n)$ time, where X is an integer parameter in $[1, n]$. In particular, setting $X = \lceil\sqrt{n\lg n}\rceil$ yields an $O(n\lg^3 n)$-word structure with $O(\sqrt{n}\lg^{7/2} n)$ query time.*

3.4 Two More Solutions with Better Space Efficiency

In the previous section, we show a solution whose space cost is $O(n \lg^3 n)$ for colored 2D orthogonal range counting. In this section, we design another two data structures with potentially better space efficiency.

3.4.1 Achieving $O(n \lg^2 n)$-Word Space

We design an alternative solution for 3D stabbing queries over canonical boxes whose space cost is a logarithmic factor less than that in Lemma 5, and it also satisfies the conditions described in Section 3.3.1 and can thus be applied in our framework. This leads to another time-space tradeoff for colored 2D orthogonal range counting, whose space cost can be as little as $O(n \lg^2 n)$ by setting appropriate parameter values.

This new 3D stabbing query solution requires a data structure for 2D dominance counting and reporting. To achieve that, we augment a binary range tree, $\hat{T}$, which is constructed over the y-coordinates of the input points, and each node v of $\hat{T}$ is conceptually associated with a list, $P(v)$, of points that are leaf descendants of v, sorted by x-coordinate, but the coordinates of points in $P(v)$ are not explicitly stored. Lemma 8 presents this data structure. We are aware of other solutions better than the result shown in Lemma 8 for 2D dominance counting and reporting [51, 14]. However, Lemma 8 gives us additional range tree functionalities that are required for our next two solutions to colored 2D orthogonal range counting.

Lemma 8 *Consider a binary range tree $\hat{T}$ constructed over a set P of n points on the plane as described above. The binary tree can be augmented using $O(n)$ additional words to support 2D dominance range reporting and counting queries. Specifically, given a query range Q, which is the region dominated by a point q, a set S of $O(\lg n)$ nodes can be located in $O(\lg n)$ time such that*

- *a nonempty prefix $L(v)$ of $P(v)$ contains a subset of points in $P \cap Q$ for each node $v \in S$,*
- *these $|S|$ prefixes are pairwise disjoint,*
- *the union of these $|S|$ prefixes contains all the points in $P \cap Q$,*

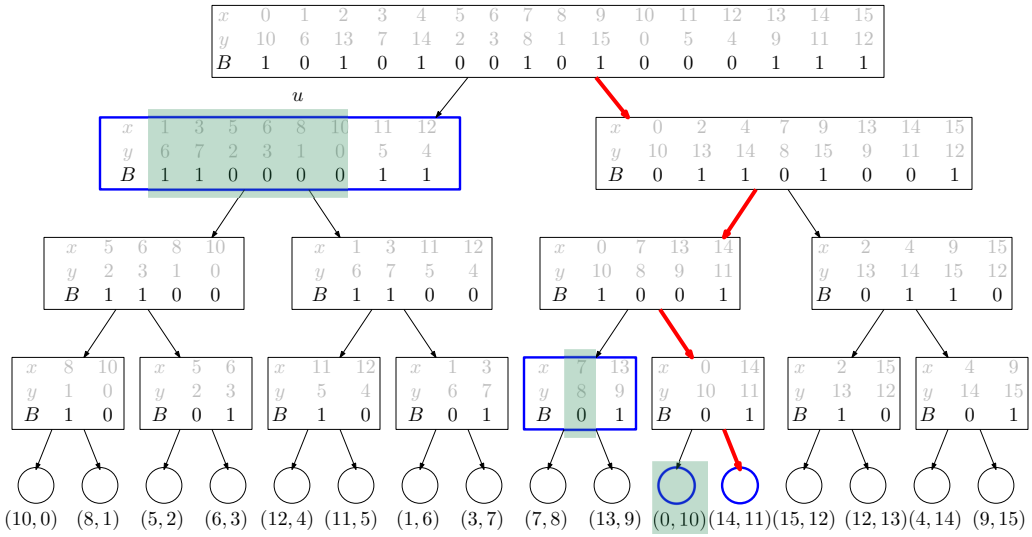


Figure 3.5: An example of a binary range tree data structure that solves 2D dominance range reporting and counting. In the query range $(-\infty, 10] \times (-\infty, 11]$, we find points $(1, 6)$, $(3, 7)$, $(5, 2)$, $(6, 3)$, $(8, 1)$, $(10, 0)$, $(7, 8)$ and $(0, 10)$.

- *the individual sizes of these $|S|$ prefixes can be computed in $O(\lg n)$ time and*
- *each point in a prefix can be reported in $O(\lg^{\lambda} n)$ additional time for any constant $0 < \lambda < 1$.*

Proof. For simplicity, we assume that point coordinates are in rank space[3]. Then the i-th leaf from the left represents the range $[i, i]$ for each $i \in [0..n-1]$. The range represented by an internal node v is $[i, j]$ if the leftmost leaf descendant and the rightmost descendant of v is the i- and j-th leaf, respectively.

At each internal node v of $\hat{T}$, we store a bit vector, $B(v)$, such that if the point $P(v)[i]$ is a leaf descendant of the left child of v, then $B(v)[i]$ is set to 0; otherwise $B(v)[i]$ is set to 1. Obviously, $|B(v)|$ equals to $|P(v)|$. We construct an $O(|P(v)|)$-bit space data structure, shown in [20], upon $B(v)$ to support $\text{rank}(v, k)$ queries in constant time, for any $0 \leq k < |B(v)|$, such that $\text{rank}(v, k) = \sum_{i \leq k} B(v)[i]$. These bit vectors over all internal nodes v of $\hat{T}$ use $\sum_v O(|B(v)|) = O(n \lg n)$ bits, which is $O(n)$ words of space. An example of the data structure is shown in Figure 3.5. The x- and y-coordinates of the points in $P(v)$ and the bit vector $B(v)$ are all presented

[3]By duplicating points in P and storing them in two sorted sequences, one ordered by x-coordinate and the other by y-coordinate, the time required to perform the conversion between original coordinates and coordinates in rank space does not affect the claimed query time.

at each node v of the tree. The point coordinates are in gray color, indicating that they are not explicitly stored in the data structure.

Given a query range $Q = (-\infty, q.x] \times (-\infty, q.y]$, we find the path, π, from the root node of $\hat{T}$ to the $q.y$-th leaf. The node set S can be constructed as follows: For each node u in π, if it is the right child of its parent, we add its left sibling into S; otherwise, we skip u. The $q.y$-th leaf is added into S as well. As a result, the ranges represented by the nodes in S form a partition of the query y-range $(-\infty, q.y]$. In addition, for each node u of $\hat{T}$, all points in $P(u)$ are increasingly sorted by their x-coordinates. Thus, a prefix, $L(u)$, of $P(u)$ contains points in the query x-range $(-\infty, q.x]$. Precisely, the size of prefix is exactly the same as the index, i, of the leftmost point of $P(u)$ whose x-coordinate is more than $q.x$, i.e., $L(u) = P(u)[0..i-1]$. To compute $|L(v)|$ for all $v \in S$, the following observation is crucial: Given that s and t are two nodes of $\hat{T}$, where s is the parent of t, if t is the left child of s, then $|L(t)|$ is $|L(s)| - \text{rank}(B(s), |L(s)|)$; otherwise, it is $\text{rank}(B(s), |L(s)|)$. Clearly, $|L(r)|$ is $q.x + 1$ at the root node r. By traversing the nodes of π downwards and performing one rank operation at each level, we can compute $|L(v)|$ for each $v \in S$. The total running time is $O(\lg n)$.

To support reporting, if the points in $P(v)$ were explicitly stored at each node v, then $P(v)[i]$ can be trivially answered in constant time. However, storing the point lists for all the nodes of $\hat{T}$ would require $O(n \lg n)$ words, which is not affordable. Instead, we augment $\hat{T}$ with the data structure that solves the ball inheritance problem, as stated as part a) of Lemma 1. This costs us $O(n)$ additional words of space and allows us to find the coordinates of $P(v)[i]$ in $O(\lg^{\lambda} n)$ time. As a result, a point of $P(v)$ in the query range can be reported in $O(\lg^{\lambda} n)$ time provided that $|L(v)|$ is known. ■

Figure 3.5 gives an example to show how the query proceeds in a binary range tree. As shown in the figure, the query range is $(-\infty, 10] \times (-\infty, 11]$, and the query path π starts from the root and ends at the leaf that stores the point $(14, 11)$. The four nodes highlighted with bold blue borders in the figure, including two internal nodes and two leaves, form the set S; each of them except the leaf representing point $(14, 11)$ is the left child of some node in π. At each node v of these four nodes, the y-coordinates of the points are in the range $(-\infty, 11]$, and the points whose x-coordinates are in the

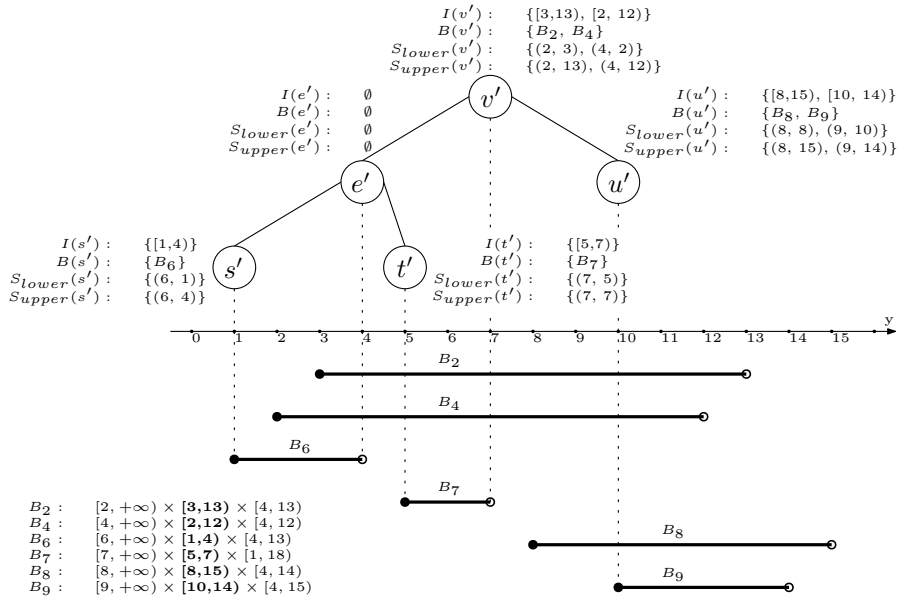


Figure 3.6: The interval tree in the second layer constructed over the y-segments of the boxes in $B(v)$ (where $B(v)$ represents the set of boxes that are assigned to node v in the top-layer segment tree shown in Figure 3.1). All the boxes in $B(v)$ are listed in the bottom-left corner of the figure and their y-segments are highlighted in bold and also drawn below the y-axis; the vertical dotted lines indicate the medians in different recursive levels. Given that the y-segments of boxes B_2 and B_4 contain the median, which is 7, of the endpoints of all y-segments in $B(v)$, both boxes are assigned to set $B(v')$, where v' is the root of the interval tree. Following our definitions, the lower points of B_2 and B_4 are $(2, 3)$ and $(4, 2)$, and the upper points are $(2, 13)$ and $(4, 12)$.

range $(-\infty, 10]$ form a prefix of $P(v)$. To compute the size of each prefix, we take node u as an example. We can tell from Figure 3.5 that u is the left child of the root node r and the first 11 nodes in $P(r)$ are in the range $(-\infty, 10]$ along the x-axis. As argued before, the number of 0-bits in $B[0..10]$ is the same as the size of the prefix $L(u)$. Therefore, by calling $11 - \text{rank}(B(r), 10)$, we find the size of $L(u)$, which is 6. The prefixes of the other nodes in S can be computed in a similar way, and we highlight these prefixes using dark rectangles in the figure.

Now we are ready to show the second stabbing query data structure. To answer stabbing queries over a set of n canonical boxes in three-dimensional space, we construct a data structure consisting of three layers of trees, but a different tree structure is adopted in each layer. The top-layer tree, denoted by T_1, is a segment tree built

upon the z-segments of the boxes[4]. A box is assigned to a node in T_1 if its z-segment is relevant to this node. Henceforth, let $B(v)$ denote the set of boxes that are assigned to node v of T_1. Then, we construct an interval tree, $T_2(v)$, over the y-segments of the boxes in $B(v)$. The interval trees constructed for all the nodes of the top-layer tree form the middle-layer structure. Recall that $T_2(v)$, as an interval tree, stores a list, $I(v')$ of intervals at each tree node v'. To each node v' of $T_2(v)$, we assign a set $B(v')$ of boxes as a subset of $B(v)$ as follows: For each box b in $B(v)$, b is assigned to $B(v')$ iff the y-segment of b is contained in $I(v')$. Furthermore, we use the boxes in $B(v')$ to define two point sets, $S_{lower}(v')$ and $S_{upper}(v')$, on the plane as follows: By projecting all the boxes in $B(v')$ onto the xy-plane, we get a set of right-open rectangles, each of the form $[x_1, +\infty] \times [y_1, y_2]$. Then, $S_{lower}(v')$ is the set of the lower left vertices of these rectangles (henceforth called *lower points*), i.e., $\{(B.x_1, B.y_1) | B \in B(v')\}$, and $S_{upper}(v')$ is the set of the upper left vertices (henceforth called *upper points*), i.e., $\{(B.x_1, B.y_2) | B \in B(v')\}$. See Figure 3.6 for an illustration. Next, we build binary range trees, $T_{lower}(v')$ and $T_{upper}(v')$, over $S_{lower}(v')$ and $S_{upper}(v')$, respectively, applying Lemma 8. These range trees constructed for all the interval tree nodes form the bottom-layer structure.

Following Lemma 8, each node v'' of a binary range tree in the bottom-layer is conceptually associated with a list, $P(v'')$, of lower or upper points, which are the points stored in the leaf descendants of v'', sorted by x-coordinate. Each point of $P(v'')$ represents a box. Since there is a one-to-one correspondence between a point in $P(v'')$ and the box it represents, we may abuse notation and use $P(v'')$ to refer to the list of boxes that these points represent when the context is clear. Hence $P(v'')$ is regarded as the bottom list when applying the new stabbing query data structure in our framework. Lemma 9 summarizes the solution.

Lemma 9 *Given a set of n canonical boxes in three dimension, the data structure above occupies $O(n \lg n)$ words and answers stabbing counting queries in $O(\lg^3 n)$ time and stabbing reporting queries in $O(\lg^3 n + k \cdot \lg^\lambda n)$ time, where k denotes the number of boxes reported. Furthermore, the set of reported boxes is the union of $O(\lg^3 n)$ different disjoint sets, each of which is a nonempty prefix of some bottom list in the*

[4]More precisely, the segment tree is constructed over the segments that are generated by projecting each box onto the z-axis.

data structure.

Proof. The top-layer segment tree occupies $O(n \lg n)$ words, while both interval trees and binary range trees from Lemma 8 are linear-space data structures. Therefore, the overall space cost is $O(n \lg n)$ words.

To show how to answer a query, let q be the query point. Our query algorithm first searches for $q.z$ in the top-layer segment tree. This locates $O(\lg n)$ nodes of the top-layer tree. Each node v located in this phase stores a list, $B(v)$, of boxes whose z-segments contain $q.z$. It now suffices to show how to count and report the boxes in $B(v)$ whose projections on the xy-plane contain point $(q.x, q.y)$. To do this, we use the interval tree in the middle-layer that is constructed over $B(v)$.

Recall that at each node v' of an interval tree, we keep the median $m(v')$ of the endpoints of the y-segments of the boxes that are stored at the descendants of v' (including itself), and we also store a list, $B(v')$, of boxes whose y-segments contain median $m(v')$. Furthermore, the lower endpoint of a y-segment corresponds to the y-coordinate of a point in $S_{lower}(v')$, and the upper endpoint of a y-segment corresponds to the y-coordinate of a point in $S_{upper}(v')$.

The next phase of our algorithm starts from the root, r', of this interval tree. We consider two cases: either $q.y \leq m(r')$ or $q.y > m(r')$. If $q.y \leq m(r')$, a y-segment of a box in $B(r')$ contains $q.y$ iff its lower endpoint is less than or equal to $q.y$. This means, among the points in $S_{lower}(r')$, those lying in the range $(-\infty, q.x] \times (-\infty, q.y]$ correspond to the boxes of $B(r')$ whose projections on the xy-plane contain point $(q.x, q.y)$. Since the z-segment of each of these boxes already contains $q.z$, these boxes contain q in the three-dimensional space. Hence, by performing a dominance query over $T_{lower}(r')$ using $(-\infty, q.x] \times (-\infty, q.y]$ as the query range, we can find these boxes. The lower endpoints of the y-segments of the boxes stored in the right subtree of r' are all higher than $m(r')$, and thus none of these boxes can possibly contain the point $(q.x, q.y, q.z)$. As a result, we descend to the left child of r afterwards and repeat this process. Otherwise, if $q.y > m(r')$ instead, then we perform a dominance query over $T_{upper}(r')$ using $(-\infty, q.x] \times (q.y, +\infty)$ as the query range to find the boxes of $B(r')$ that contain q, descend to the right child of r, and then repeat.

To analyze the running time, observe that this algorithm locates $O(\lg n)$ nodes

in the top-layer segment tree, and for each of these nodes, it further locates $O(\lg n)$ nodes in the middle-layer interval trees. Hence, we perform a 2D dominance counting and reporting query using Lemma 8 for each of these $O(\lg^2 n)$ interval tree nodes, and the proof completes. ■

In an interval tree, each segment is stored at exactly one node, while in a segment tree or a binary range tree, each segment or point can be stored at $O(\lg n)$ nodes. Therefore, this data structure has duplication factor $\delta = O(\lg^2 n)$. By combining Lemmas 5 and 7, we have $h(n) = O(\lg n)$, $\phi(n) = O(\lg^3 n)$ and $\tau(n) = O(\lg^\lambda n)$. We again use part b) of Lemma 4 to implement $E(v_l)$ and $E(v_r)$, so $f(n) = O(\lg \lg n)$ and $g(n) = O(\lg \lg n)$. Hence:

Theorem 2 *There is a data structure of $O((\frac{n}{X})^2 \lg^4 n + n \lg^2 n)$ words of space that answers colored 2D orthogonal range counting queries in $O(\lg^6 n + X \lg^{3+\lambda} n)$ time, where X is an integer parameter in $[1, n]$ and $0 < \lambda < 1$ is an arbitrary constant. In particular, setting $X = \lceil \sqrt{n} \lg n \rceil$ yields an $O(n \lg^2 n)$-word structure with $O(\sqrt{n} \lg^{4+\lambda} n)$ query time.*

3.4.2 Achieving $O(n \lg n)$-Word Space

In this section, we further improve the space cost of the data structure for 3D stabbing queries over canonical boxes. The new solution also satisfies the conditions described in Section 3.3.1 and can thus be applied in our framework. This leads to our third time-space tradeoff for colored 2D orthogonal range counting, whose space cost can be as efficient as $O(n \lg n)$ words by setting appropriate parameter values.

Our solution will adopt a space-efficient data structure for 3D orthogonal dominance range searching. Using a range tree with node degree $\gamma \in [2, n]$, we can transform a linear space data structure for 2D dominance range counting and reporting shown in Lemma 8 into a new data structure that supports dominance range queries in 3D and uses $O(n \log_\gamma n)$ words.

Lemma 10 *There is a data structure of $O(n \log_\gamma n)$ words of space that answers 3D dominance counting queries in $O(\gamma \lg n \cdot \log_\gamma n)$ time and 3D dominance reporting queries in $O(\gamma \lg n \cdot \log_\gamma n + k \lg^\lambda n)$ time, where k is the number of reported points and γ is an integer parameter in $[2, n]$.*

Proof. The data structure is a balanced γ-ary range tree, $\tilde{T}$, constructed upon z-coordinates of the points. Each leaf of $\tilde{T}$ stores an input point, and all leaves from left to right are increasingly sorted by the z-coordinates of the points. At each internal node v, we explicitly store a list, $P(v)$, of points that are leaf descendants of v. Given $P(v)$, we construct a 2D dominance range searching data structure upon the x- and y-coordinates of the points by applying Lemma 8. Since $\tilde{T}$ has $O(\log_\gamma n)$ tree levels and each level stores a linear space data structure, the overall space cost is $O(n \log_\gamma n)$ words.

Given a query range $Q = (-\infty, q.x] \times (-\infty, q.y] \times (-\infty, q.z]$, we find the path, π, from the root node of $\tilde{T}$ to the leaf that stores the point whose z-coordinate is a predecessor of $q.z$. Path π contains $O(\log_\gamma n)$ nodes, and the leaf it contains can be found in $O(\lg n)$ time by a binary search over all the leaf points. We construct a node set S such that for each node $u \in \pi$, we append all u's siblings on its left into S, and finally the leaf node in π is appended into S as well. As a result, the z-coordinates of points in $P(v)$ for each $v \in S$ are within $(-\infty, q.z]$, i.e., the query range along z-axis. Thus, we can find the points $P(v) \cap Q$ and compute $|P(v) \cap Q|$ by 2D dominance searching upon $P(v)$ by applying Lemma 8. As π has $O(\log_\gamma n)$ nodes and each node v of π has $O(\gamma)$ siblings, a 3D dominance counting (resp. reporting) query takes $O(\gamma \lg n \cdot \log_\gamma n)$ (resp. $O(\gamma \lg n \cdot \log_\gamma n + k \lg^\lambda n)$) time. ∎

Now we are ready to show the third data structure for stabbing queries over 3D canonical boxes. It again consists of three layers of trees, with interval trees in the top- and middle- layers, plus the data structures for 3D dominance range searching implemented by the data structure of Lemma 10 in the bottom-layer. More precisely, the structure at the top layer is an interval tree, T_1, constructed over the z-segments of the boxes. To each node v in the top-layer tree, we assign a list, $B(v)$ of boxes, whose z-segments are contained in the segment set $I(v)$ (See Section 3.2.3 for the definition of $I(v)$), and we further construct an interval tree, $T_2(v)$, over the y-segments of the boxes in $B(v)$. These interval trees constructed for all the nodes of the top-layer tree form the middle-layer structure. At each node v' of an interval tree in the middle-layer, we store a list, $B(v')$, of boxes whose y-segments are contained in the segment set $I(v')$. Given the box list, $B(v')$, we define four point sets, $S_{\ell,\ell}(v')$, $S_{\ell,r}(v')$, $S_{u,\ell}(v')$ and $S_{u,r}(v')$ in 3D such that $S_{\ell,\ell}(v') = \{(B.x_1, B.y_1, B.z_1) | B \in B(v')\}$,

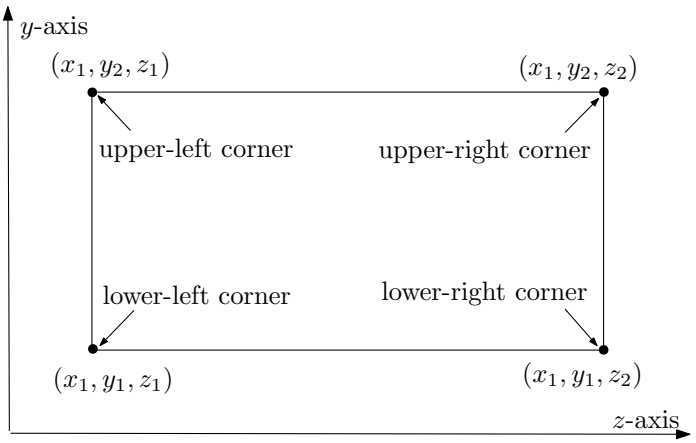


Figure 3.7: Illustrating how the endpoints of a box projected on the yz-plane are assigned into different sets $S_{\ell,\ell}, S_{\ell,r}, S_{u,\ell}$ and $S_{u,r}$

$S_{\ell,r}(v') = \{(B.x_1, B.y_1, B.z_2) | B \in B(v')\}$, $S_{u,\ell}(v') = \{(B.x_1, B.y_2, B.z_1) | B \in B(v')\}$, and $S_{u,r}(v') = \{(B.x_1, B.y_2, B.z_2) | B \in B(v')\}$. Then, we build a set of 3D dominance range searching structures $T_{\ell,\ell}(v')$, $T_{\ell,r}(v')$, $T_{u,\ell}(v')$ and $T_{u,r}(v')$ over $S_{\ell,\ell}(v')$, $S_{\ell,r}(v')$, $S_{u,\ell}(v')$ and $S_{u,r}(v')$, respectively, applying Lemma 10. See Figure 3.7 for an illustration. The 3D dominance range searching structures constructed for the nodes of all middle-layer interval trees form the bottom-layer structure.

As shown in the previous section, we need to identify the bottom lists from the data structure described above, which are crucial when applying the data structure in our framework. Without loss of generality, we take the 3D dominance range searching structure $T_{\ell,\ell}(v')$ built in the bottom-layer as an example. Observe that $T_{\ell,\ell}(v')$ is a γ-ary range tree, of which each internal node, v'', stores a binary range tree $\hat{T}(v'')$ implemented by the data structure of Lemma 8 for 2D dominance range searching. Recall that each node $\hat{v}$ of a binary range tree $\hat{T}(v'')$ is conceptually associated with a list, $P(\hat{v})$, of points from the set $S_{\ell,\ell}(v')$, which are the points stored in the leaf descendants of $\hat{v}$, sorted by x-coordinate. Each point of $P(\hat{v})$ represents a box. Since there is a one-to-one correspondence between a point in $P(\hat{v})$ and the box it represents, we again use $P(\hat{v})$ to refer to the list of boxes that these points represent. Hence, $P(\hat{v})$ is regarded as the bottom list when applying the data structure in our framework. The following lemma summarizes our third solution to stabbing queries over 3D canonical boxes:

Lemma 11 *Given a set of n canonical boxes in three dimension, the data structure*

described above occupies $O(n \log_\gamma n)$ *words of space and answers stabbing counting queries in* $O(\lg^2 n \cdot \gamma \lg n \cdot \log_\gamma n)$ *time and stabbing reporting queries in* $O(\lg^2 n \cdot \gamma \lg n \cdot \log_\gamma n + k \cdot \lg^\lambda n)$ *time, where* k *denotes the number of boxes reported and* γ *is an integer parameter in* $[2, n]$. *Furthermore, the set of reported boxes is the union of* $O(\lg^2 n \cdot \gamma \lg n \cdot \log_\gamma n)$ *different disjoint sets, each of which is a nonempty prefix of some bottom list in the data structure.*

Proof. Both the top- and middle- layer interval trees are linear-space data structures, while the 3D dominance range searching structures from Lemma 10 occupy $O(n \log_\gamma n)$ words of space in total. Therefore, the overall space cost is $O(n \log_\gamma n)$ words.

Before showing the query algorithm, we review the data structure of an interval tree. At each node v of the top-layer interval tree T_1, we keep i) the median, $m_z(v)$, of the endpoints of the segments that are stored at v's descendants (including itself), ii) a list, $I_z(v)$, of the segments that contain $m_z(v)$ and iii) a middle-layer interval tree, $T_2(v)$, that is constructed over the y-segments of the boxes in $B(v)$. And we use $m_y(v')$ and $I_y(v')$ to represent the median and the segment list that are stored at tree node v' of $T_2(v)$, respectively. Note that a segment in $I_z(v)$ (resp. $I_y(v')$) corresponds to the z-segment (resp. y-segment) of a box in $B(v)$ (resp. $B(v')$), and thus if a segment in $I_z(v)$ (resp. $I_y(v')$) does not contain the projection of the query point onto the z-axis (resp. y-axis), then it means that the box that this segment corresponds to cannot contain the query point.

Let $q = (q.x, q.y, q.z)$ be the query point. The query visits $O(\lg n)$ nodes v of the interval tree in the top-layer and $O(\lg^2 n)$ nodes v' of the interval trees in the middle-layer. By comparing $q.z$ and $m_z(v)$ (resp. $q.y$ and $m_y(v')$), one can decide which child of v (resp. v') to be visited next. For example, let r denote the root of T_1 and assume that $q.z \leq m_z(r)$. Since the nodes in the subtree rooted at the right child of r stores the z-segments of the boxes whose minimum endpoints are greater than $m_z(r)$, none of these segments could possibly contain $q.z$. As a result, after traversing the middle-layer interval tree $T_2(r)$ that is constructed over the y-segments of the boxes in $B(r)$, we descend to the left child of r. The procedure of traversing $T_2(r)$ is the same as we have seen in the proof of Lemma 9.

The comparison between $q.z$ and $m_z(v)$ in the top-layer and the comparison between $q.y$ and $m_y(v')$ in the middle-layer result in four different cases. To find the boxes that contain the query point under each different case, we apply the corresponding 3D dominance range searching data structure that is built in the bottom-layer:

- When $q.z \leq m_z(v)$ and $q.y \leq m_y(v')$, we search $T_{\ell,\ell}(v')$ for the points of $S_{\ell,\ell}(v')$ in the query range $(-\infty, q.x] \times (-\infty, q.y] \times (-\infty, q.z]$.
- When $q.z \leq m_z(v)$ and $q.y > m_y(v')$, we search $T_{u,\ell}(v')$ for the points of $S_{u,\ell}(v')$ in the query range $(-\infty, q.x] \times (q.y, +\infty) \times (-\infty, q.z]$.
- When $q.z > m_z(v)$ and $q.y \leq m_y(v')$, we search $T_{\ell,r}(v')$ for the points of $S_{\ell,r}(v')$ in the query range $(-\infty, q.x] \times (-\infty, q.y] \times (q.z, +\infty)$.
- When $q.z > m_z(v)$ and $q.y > m_y(v')$, we search $T_{u,r}(v')$ for the points of $S_{u,r}(v')$ in the query range $(-\infty, q.x] \times (q.y, +\infty) \times (q.z, +\infty)$.

We give the reasoning of the first case, and similar arguments apply to the other three cases. Given that $q.z \leq m_z(v)$, a segment in $I_z(v)$ contains $q.z$ iff its minimum endpoint is less than or equal to $q.z$. Given that $q.y \leq m_y(v')$, a segment in $I_y(v')$ contains $q.y$ iff its lower endpoint is less than or equal to $q.y$. Therefore, among the points in $S_{\ell,\ell}(v')$, those lie in $(-\infty, q.x] \times (-\infty, q.y] \times (-\infty, q.z]$ correspond to the boxes containing $(q.x, q.y, q.z)$. Hence, by performing a dominance query over $T_{\ell,\ell}(v')$ using $(-\infty, q.x] \times (-\infty, q.y] \times (-\infty, q.z]$ as the query range, we find these boxes.

To analyze the running time, observe that this algorithm locates $O(\lg n)$ nodes in the top-layer interval tree, and for each of these nodes, it further locates $O(\lg n)$ nodes in the middle-layer interval trees. Hence, we perform a 3D dominance counting and reporting query using Lemma 10 for each of these $O(\lg^2 n)$ interval tree nodes, and the proof completes. ■

In an interval tree, each interval is stored in exactly one node, while in a 3D dominance range searching structure as shown in Lemma 10, each point can be associated with $O(\lg n \cdot \log_\gamma n)$ nodes. Therefore, this data structure has duplication factor $\delta(n) = O(\lg n \cdot \log_\gamma n)$. By combining Lemmas 11 and 7, we have $h(n) = O(\log_\gamma n)$, $\phi(n) = O(\lg^2 n \cdot \gamma \lg n \cdot \log_\gamma n)$ and $\tau(n) = O(\lg^\lambda n)$. This time, we use part a) of Lemma 4 to implement $E(v_l)$ and $E(v_r)$, so $f(n) = O(1)$ and $g(n) = O(\lg^\lambda n)$. Hence:

Theorem 3 *There is a data structure of* $O((\frac{n}{X})^2 \lg^2 n \cdot \log_\gamma^2 n + n \lg n \cdot \log_\gamma n)$ *words of space that answers colored 2D orthogonal range counting queries in* $O(\gamma^2 \cdot \lg^6 n \cdot \log_\gamma^2 n + X \cdot \lg^{3+\lambda} n \cdot \gamma \log_\gamma n)$ *time, where* X *is an integer parameter in* $[1, n]$, γ *is an integer parameter in* $[2, n]$, *and* λ *is any constant in* $(0, 1)$. *Setting* $X = \lceil \sqrt{n \lg n \log_\gamma n} \rceil$ *and* $\gamma = \lceil \lg^\lambda n \rceil$ *yields an* $O(n \frac{\lg^2 n}{\lg \lg n})$*-word structure with* $O(\sqrt{n} \lg^{5+\lambda'} n)$ *query time for any constant* $\lambda' > 2\lambda$. *Alternatively, setting* $X = \lceil \sqrt{n \lg n} \rceil$ *and* $\gamma = \lceil n^{\lambda/5} \rceil$ *yields an* $O(n \lg n)$*-word structure with* $O(n^{1/2+\lambda})$ *query time.*

3.5 Conclusion

In this chapter, we design space-efficient data structures to solve colored 2D orthogonal range counting. We apply the standard technique of decomposing a 4-sided query range to two 3-sided subranges using a binary range tree. To compute the number of colors in each subrange, we apply the same reduction from colored 2D 3-sided range counting to stabbing queries over a set of 3D 5-sided boxes, developed by Kaplan et al. [57], but we use three different data structures for solving the latter problem: The first one consists of two layers of segment tree, occupying $O(n \lg^2 n)$ words of space and supporting stabbing queries in $O(\lg^2 n + k)$ time and stabbing counting in $O(\lg^2 n)$ time. The second one contains a segment tree, an interval tree and a binary range tree at different layers, occupying $O(n \lg n)$ words of space and supporting stabbing queries in $O(\lg^3 n + k \lg^\lambda n)$ time and stabbing counting in $O(\lg^3 n)$ time. The third one is formed by a two layers of interval trees plus a linear space data structure for 3D dominance range queries, occupying $O(n)$ words of space and supporting stabbing queries in $O(n^\lambda + k \lg^\lambda n)$ time and stabbing counting in $O(n^\lambda)$ time. To compute the number of colors that appear in both subranges, we design a novel scheme. Combining our scheme and the different data structures for 3D stabbing queries, we achieve three new solutions to colored 2D orthogonal range counting such that each of them can be expressed in the form of time-space tradeoff. Comparing to the tradeoff solution given by Kaplan et al. [55], each of our solutions improves the space usage by polylog(n) factors. Specially, by setting the parameters in the tradeoff solutions to be appropriate values, our data structures only use $O(n \lg^3 n)$, $O(n \lg^2 n)$, $O(n \frac{\lg^2 n}{\lg \lg n})$, and $O(n \lg n)$ words, and support colored 2D orthogonal range

counting in $O(\sqrt{n}\lg^{5/2} n \lg\lg n)$, $O(\sqrt{n}\lg^{4+\lambda} n)$, $O(\sqrt{n}\lg^{5+\lambda} n)$, and $O(n^{1/2+\lambda})$ time, respectively.

Finally, we conclude this chapter with one open problem. As mentioned in Section 3.1.1, one of the data structures by Kaplan et al. can be modified to achieve a linear space solution, but its query time is $\omega(n^{3/4})$. However, the most space efficient data structure we achieved uses $O(n \lg n)$ words of space and supports each query in $O(n^{1/2+\lambda})$ time. There is a gap in the query time between our solution and the linear space solution. The open problem is whether there is a linear space data structure that supports colored 2D orthogonal range counting in $o(n^{3/4})$ time.

Chapter 4

Faster Path Queries in Colored Trees

4.1 Introduction

Trees are used to represent information in many areas of computer science. In tree-structured data, additional properties, such as categorical information, are often encoded as colors of tree nodes. To facilitate the retrieval of color information, researchers have defined the following queries over an ordinal tree T on n nodes with each node assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$: Given a path in T, a *colored path counting query* returns the number of distinct colors assigned to the nodes in this path, while a *path mode query* returns a *mode* of the path, i.e, a most frequent color among the multiset of colors assigned to nodes in this path. These queries can be used to compute fundamental statistic information over tree-structured data.

Researchers have studied these query problems and designed data structure solutions [58, 23, 44]. Different time-space tradeoffs have been achieved, and among the best linear-space solutions under word RAM, the structure of He and Kazi [44] can answer a colored path counting query in $O(\sqrt{n} \lg \lg C)$ time, and the solution of Durocher et al. [23] can answer a path mode query in $O(\sqrt{n/w} \lg \lg n)$ time, where w denotes the number of bits stored in a word. The support for these queries is thus much slower than the support for many other path queries in trees, such as path minimum [18, 2, 56, 22, 8, 12], path medium [58, 64, 48, 49], path counting [18, 58, 64, 48, 49] and path majority [23, 30], for which linear-space solutions with sublogarithmic or even constant query times exist.

However, researchers have given evidence to show that these solutions to colored path counting and path mode queries are efficient, by proving conditional lower bounds. It has been shown that the multiplication of two $\sqrt{n} \times \sqrt{n}$ Boolean matrices can be performed by answering n colored path counting queries or n path mode queries. This reduction was explicitly given for colored path counting [44], while for path mode queries, it follows from the same conditional lower bound on range

mode queries [10] in arrays, for which we preprocessing an array A such that given a range $[i, j]$, we can find a most frequent element in $A[i, j]$ efficiently. Note that when the input tree has a single path only, path mode queries becomes range mode queries. This reduction means, with current knowledge, the total running time of answering n of these paths or range queries, including preprocessing, cannot be faster than $n^{\omega/2}$, save for polylogarithmic speedups, where $\omega < 2.37286$ denotes the exponent of matrix multiplication [1]. Furthermore, since the best known combinatorial approach of multiplying two $n \times n$ Boolean matrices under the word RAM model requires $\Theta(n^3/\operatorname{polylog}(n))$ time [70], the total time of answering n of these queries cannot be faster than $n^{1.5}$, save for polylogarithmic speedups, using pure combinatorial methods with current knowledge. Since the structures of He and Kazi [44] and Durocher et al. [23] can be built in $\tilde{O}(n^{1.5})$ time, they can be used to answer n colored path counting or path mode queries in $\tilde{O}(n^{1.5})$ time, matching this conditional lower bound on pure combinatorial methods within polylogarithmic factors.

The problem of answering n queries given offline is the batched version of these query problems. To achieve $O(n^{1.5-\epsilon})$-time solutions for some positive constant ϵ, Williams and Xu [69] reduced batched range mode to the min-plus product of a pair of matrices of special structures, which makes it possible to answer a batch of n range mode queries over an array of length n in $O(n^{1.4854})$ time. Gu et al. [39] further improved the running time to $O(n^{1.4805})$. Similar ideas have also yielded dynamic range mode structures with $O(n^{0.655994})$ query and update times [39]. This is surprising as Jin and Xu [53] showed that dynamic range mode structure cannot simultaneously support update and query in $O(n^{2/3-\epsilon})$ time for any positive constant ϵ using purely combinatorial method with current knowledge. Even before that, Kaplan et al. [57] used sparse matrix multiplication to answer n colored orthogonal range counting queries over n colored points on the plane in $O(n^{1.4071})$ total time. This query counts the number of distinct colors assigned to points in each of n axis-aligned query rectangles and is related to colored path counting in the sense that they both generalize colored 1D range counting [31, 63] and have the same conditional lower bound.

Despite all these exciting works which beat the conditional lower bounds for combinatorial methods, no previous work was done to solve batched colored path counting

or batched path mode queries in $O(n^{1.5-\epsilon})$-time. In this chapter, we use matrix multiplication and min-plus product to solve these problems.

4.1.1 Previous Work

The study on colored range counting started in the 1D case, for which Gupta et al. [42] showed a reduction to 2D orthogonal range counting over uncolored points, hence achieving a linear space solution with $O(\lg n/\lg\lg n)$ query time. This problem has been studied extensively in 2D [40, 57, 38, 61, 34], for which Kaplan et al. [55] proved the conditional lower bound based on Boolean matrix multiplication which we discussed previously. They further designed a data structure occupying $O((\frac{n}{t})^2\lg^6 n + n\lg^4 n)$ words that supports colored 2D range counting in $O(t\lg^7 n)$ time for any $0 < t \leq n$, which was later improved by Gao and He [34] by shaving off several $\log n$-factors in time/space bounds. See also Chapter 3 for the details of the improvements. Kaplan et al. also showed how to solve batched colored 2D orthogonal range counting in $\tilde{O}(n^{\frac{2\omega}{\omega+1}}) = O(n^{1.4071})$ time by reducing it to sparse matrix multiplication. Colored range counting has also been studied in high dimensions [40, 57, 38]. Finally, He and Kazi [44] considered colored path counting in trees and proved a conditional lower bound which is also based on Boolean matrix multiplication. They designed an $O(n + \frac{n^2}{t^2})$-word structure that answers queries in $O(t\lg\lg C)$ time for any $t \in [1, n]$, and it can be constructed in $O(\frac{n^2}{t}\lg\lg C)$ time. Hence it implies an $O(n^{3/2}\lg\lg C)$-time solution to batched colored path counting.

Since Krizanc et al. [58] proposed range and path mode query problems, a long series of papers have been published on these and related problems [58, 10, 11, 26, 69, 68, 39, 53]. The best linear-space solutions include the structure of Chan et al. [10] that answers range mode queries in arrays in $O(\sqrt{n/w})$ time and the structure of Durocher et al. [23] that answers path mode queries in trees in $O(\sqrt{n/w}\lg\lg n)$ time. Chan et al. [10] also studied dynamic range mode queries in arrays, and their solution was later improved by El-Zein et al. [26], whose linear-space structure supports both queries and updates in $O(n^{2/3})$ time. It is worth mentioning that all the results summarized in this paragraph use purely combinatorial approaches. They match the conditional lower bounds [10, 11, 53] within polylogarithmic factors.

Recently, more efficient solutions to the batched range mode problem in arrays [68,

69, 39] have been found. Williams and Xu [69] reduced this problem to the min-plus product of a pair of matrices. The second matrix has the property that the entries at each row are non-decreasing, which allows designing a truly subcubic time algorithm for the min-plus product of two $n \times n$ matrices. With it, they can solve batched range mode in $O(n^{1.4854})$ time. Later, the query time was improved by Gu et al. [39] to $O(n^{1.4805})$. Sandlund and Xu [68] broke the $O(n^{2/3})$ per-operation time barrier for dynamic range mode in arrays; they reduced the problem to the *min-plus-query-witness problem*, and achieved a dynamic data structure that supports both queries and updates in $O(n^{0.655994})$ time. Later, Gu et al. [39] further improved the time for each operation to $O(n^{0.6524})$.

4.1.2 Our Contributions

In this section, we summarize our results and give a brief overview of our methods.

Our Results. We have achieved the following results:

- an $\tilde{O}(n^{\frac{2\omega}{\omega+1}}) = O(n^{1.4071})$-time algorithm for batched colored path counting in trees, improving the previous best approach which solves this problem in $\tilde{O}(n^{1.5})$ time [44];
- an $\tilde{O}(n^{\frac{24+2\omega}{17+\omega}}) = O(n^{1.483814})$-time algorithm for batched path mode queries in trees, improving the previous best result with $\tilde{O}(n^{1.5})$ running time [23].

Overview of Our Approach. To achieve these results, we develop new algorithmic ideas to address the challenges we encounter due to the tree topology. The first challenge is how to apply a divide-and-conquer approach to batched path queries. The solution of Williams and Xu [69] to batched range mode recursively divides the input array into halves, and at each level of recursion, they build data structures that answer queries whose ranges straddle the midpoint. This ensures that the set of possible queries considered share subranges instead of being disjoint, facilitating preprocessing. The solution of Kaplan et al. [57] to batched colored 2D orthogonal range counting is based on a similar idea in 2D. To adapt to tree topology, we apply the centroid decomposition of trees instead. Then, in each component obtained as a

result of the decomposition, we preprocess for queries whose paths cross the centroid. This decomposition scheme helps us solve batched path queries.

When preprocessing for query paths that contain the centroid in a component, we mark a subset of nodes and attempt to use either sparse matrix multiplication or min-plus product as in previous work. However, more twists to previous approaches are needed. In the solution of Kaplan et al. [57] to batched colored 2D orthogonal counting, the matrices that they need to multiply during preprocessing are already sparse. This is however not the case in our solution to batched colored path counting. To resolve this, we use the properties of our node marking scheme to carefully reduce the problem of multiplying these matrices to the multiplication of two different but related matrices that are sparse. There is a similar challenge for batched path mode. Previous solutions to batched range mode in [68, 39] reduce the preprocessing for each set of query ranges to the min-plus-query-witness problem over two matrices of which the second matrix is monotone, i.e., entries in the same row are non-decreasing. This allows applying strategies such as dividing each entry by a carefully chosen integer and rounding down the result to decrease the total number of different entries in the matrix. Note that this measure, i.e., the total number of different entries, plays an important role in the fast three-phase algorithm presented by Gu et al. [39]. Inspired by previous work [68, 39], we find a reduction from the preprocessing for each set of query paths to the min-plus-query-witness problem as well. However, due to the tree topology, the second matrix in our case is not monotone, so this way of applying integer division does not apply. To overcome this difficulty, we propose a two-level marking scheme. By applying it, we manage to create a related matrix in which the total number of different entries between consecutive columns can be smaller by a polynomial factor, comparing to the original matrix generated directly from the tree using the one-level marking scheme, so that we are able to adopt the fast three-phase algorithm presented by Gu et al. [39] mentioned earlier and achieve the fast preprocessing time.

4.2 Preliminaries

This section introduces the previous results used in this chapter, including basic path queries, tree node sampling technique and sparse rectangular matrix multiplication.

4.2.1 Counting Colors for All Root-to-node Paths

Lemma 3 in Section 2.4.2 shows a data structure that supports each colored path emptiness queries in $O(\lg \lg C)$ time. We can further use it to compute $\{|C(\mathrm{P}_{x,\perp})| : x \in T)\}$, i.e., the numbers of distinct colors on all root-to-node paths. To do this, perform a preorder traversal of T, and each time we visit a node x, we compute $|C(\mathrm{P}_{x,\perp})|$ as follows: If x is the root, then $|C(\mathrm{P}_{x,\perp})|$ is 1. Otherwise, locate $x' = \mathrm{parent}(x)$, and answer a colored path emptiness query to find out whether $c(x)$ appears in $\mathrm{P}_{x',\perp}$. If it does, set $|C(\mathrm{P}_{x,\perp})| = |C(\mathrm{P}_{x',\perp})|$; otherwise, $|C(\mathrm{P}_{x,\perp})| = |C(\mathrm{P}_{x',\perp})| + 1$. Overall, n emptiness queries are called. Hence, this process uses $O(n \lg \lg C)$ time.

Lemma 12 *Let T be a colored ordinal tree on n nodes with each node assigned a color drawn from $[C]$, where $C \leq n$. The numbers of distinct colors on all root-to-node paths on T can be computed in overall $O(n \lg \lg C)$ time using $O(n)$ words of working space.*

4.2.2 Node Sampling

In our solutions, we use the following lemma based on the pigeonhole principle to select a subset of tree nodes and precompute information for them.

Lemma 13 ([50]) *Let T be a tree on n nodes. Given an integer $t \in [1, n]$, an integer $\ell \in [0, t)$ can be found in $O(n)$ time such that, if one marks nodes at every t levels of T starting from level ℓ, at most n/t nodes will be marked.*

Throughout this chapter, we call the nodes selected by the method of Lemma 13 *marked nodes*, and we refer to the i-th marked node visited in a preorder traversal as the *i-th marked node* for short, where i starts from 0, and this node is denoted by x_i.

4.2.3 Rectangular Matrix Multiplication

Matrix multiplication is one of the most important problems in theoretical computer science. It can be defined as follows: Given an $m \times n$ matrix A and an $n \times p$ matrix B, the product of A and B, denoted by AB, is an $m \times p$ matrix in which entry $(AB)_{i,j} = \sum_{k=0}^{n-1} A_{i,k} \times B_{k,j}$ for each $i \in [m]$ and $j \in [p]$. Throughout this chapter, the row or column indexes of a matrix always start from 0.

Let $\beta \geq 1$ be a constant. Following [39], we use $\omega(1, \beta, 1)$ to denote the minimum value such that the product of an $n \times n^\beta$ matrix and an $n^\beta \times n$ matrix can be computed in $O(n^{\omega(1,\beta,1)+\lambda})$ time for any small constant $\lambda > 0$. Occasionally, we write down in this chapter ω to represent $\omega(1, 1, 1)$ for short. The best bounds of $\omega(1, \beta, 1)$ to date for different β are given by Le Gall and Urrutia [32], e.g., $\omega(1, 2, 1) \in [2, 3.251640)$, and the current best bound on ω is $[2, 2.37286)$, given by Alman and Williams [1].

All bounds mentioned above apply to general matrices. In this chapter, we are interested in sparse matrices in which the numbers of non-zero entries are bounded. Kaplan et al. [57] design a fast algorithm for sparse matrix multiplication. Their result is shown as follows:

Lemma 14 ([57, Theorem 2.5]) *Let A be an $m \times n$ matrix having at most t non-zero entries, where $t \geq m^{\frac{\omega+1}{2}}$. Then, given the list of non-zero entries of A as the input without storing A verbatim, the product of A and the transpose of A can be computed in $O(tm^{\frac{\omega-1}{2}})$ time.*

4.2.4 Min-Plus Product and Min-Plus-Query-Witness Problem

Given an $m \times n$ matrix A and an $n \times p$ matrix B, the min-plus product of A and B, denoted by $A \star B$, is an $m \times p$ matrix in which entry $(A \star B)_{i,j} = \min_k\{A_{i,k} + B_{k,j}\}$. Following [68], we define Min-Plus-Query-Witness problem as Problem 1.

Problem 1 *(Min-Plus-Query-Witness problem [68]) Build a data structure upon a pair of input matrices A and B such that given two integers i and j as a query, an index k^* with $A_{i,k^*} + B_{k^*,j} = \min_k\{A_{i,k} + B_{k,j}\}$ can be found efficiently.*

The following result that solves the min-plus-query-witness query problem when the entries of the first matrix are bounded will be used in our solution.

Lemma 15 *[68, Lemma 9] Let $\beta \geq 1$ be a constant and let A be an $m \times m^\beta$ integer matrix, in which each entry is drawn from $\{-M, \cdots, M\} \cup \{\infty\}$ for some $M \geq 1$; let B be an $m^\beta \times m$ matrix, in which each entry is a number encoded by* polylog m *bits. Given matrices A and B as the input, there is a data structure of $\tilde{O}(Mm^{2+\beta-\sigma} + m^{1+2\beta-\sigma})$ words of space that can be constructed in $\tilde{O}(Mm^{\omega(1,\beta,1)+\beta-\sigma})$ time and support finding an index k^* such that $A_{i,k^*} + B_{k^*,j} = \min_k\{A_{i,k} + B_{k,j}\}$ in $\tilde{O}(m^\sigma)$ time, where σ denotes any constant such that $0 \leq \sigma \leq \beta$.*

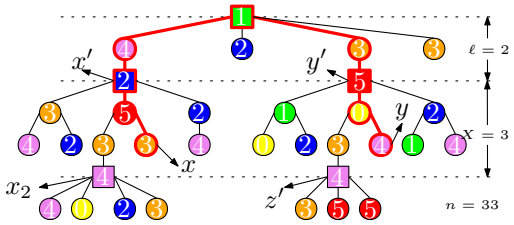
4.3 Batched Colored Path Counting

We first show, in Section 4.3.1, a data structure to answer a restricted version of colored path counting which requires the query path to contain the root, and we defer the analysis of its preprocessing time to Section 4.3.2, in which we present a reduction from colored path counting to sparse matrix multiplication. Finally, in Section 4.3.3, we generalize this solution to be workable for arbitrary paths, which yields a new result for batched colored path counting.

4.3.1 Color Counting over Paths Containing the Root

We represent the tree T using the data structure of Lemma 2. As discussed in Section 2.4.2, this structure supports colored path emptiness. Then, for an integer parameter $0 < X \leq n$ to be chosen later, we select at most n/X nodes of T using the method of Lemma 13 and mark them. This means we mark nodes at every X levels of T, starting from some level $\ell \in [0, X-1]$ determined by the method of Lemma 13. In addition, we mark the root node as well and denote it by x_0. The number, m, of nodes that we mark satisfies $m \leq n/X + 1$. Recall that for each $i \in [m]$, x_i denotes the i-th marked node visited in a preorder traversal as described in Section 4.2.2. For each marked node x, we precompute $r(x)$ which is the rank of x among marked nodes defined this way, where $r(x) \in [m]$, as well as the value $|C(\mathrm{P}_{x,\perp})|$. Furthermore, each node in the tree stores a flag indicating whether it is marked, as well as a pointer to the lowest marked proper ancestor. See Figure 4.1 for an example.

Next, we construct an $m \times m$ matrix M. For every pair of integers i and j in $[0, m-1]$, $M_{i,j}$ stores $|C(\mathrm{P}_{x_i,\perp}) \cap C(\mathrm{P}_{x_j,\perp})|$, i.e., the number of distinct colors that appear in both the path between the i-marked node and the root and the path between j-th marked node and the root. As shown in Figure 4.1, Path $\mathrm{P}_{x',\perp}$ and Path $\mathrm{P}_{y',\perp}$ together share a single common color, which is color 1; therefore, entry $M_{1,3}$ is set to be 1. It is worth mentioning that our query algorithm to be described later only uses entries of M that correspond to two marked nodes whose lowest common ancestor is the root. The other entries are never used, but we precompute them regardless. Indeed, both paths $\mathrm{P}_{z',\perp}$ and $\mathrm{P}_{y',\perp}$ contain colors, 1, 3, and 5, and thus entry $M_{3,3}$ is set to be 3; however, since $\mathrm{LCA}(z', y')$ is different from $\perp$, entry $M_{3,3}$ will never be



M	0	1	2	$3(y')$	4
0	1	1	1	1	1
$1(x')$	1	3	3	1	2
2	1	3	5	3	4
$3(z')$	1	1	3	3	3
4	1	2	4	3	5

Figure 4.1: An example of marking nodes using the method of Lemma 13 and its corresponding matrix M. Apart from the root, the other marked nodes are at levels $\ell + X \times i$ and represented by squares, where ℓ and X are set to be 2 and 3, respectively. Each node is labeled by an integer representing the node color. Since the input tree contains 6 different colors, the integers used for the colors are drawn from [6], i.e., $\{0, 1, 2, 3, 4, 5\}$. The query path is $\mathrm{P}_{x,y}$, through the root of the tree. And nodes labeled by x' and y' are the lowest marked proper ancestors of x and y, respectively.

used in our algorithm.

Overall, our data structures use $O(n + (\frac{n}{X})^2)$ words.

To describe the query algorithm, let $\mathrm{P}_{x,y}$ denote a query path containing the root. Since $\perp \in \mathrm{P}_{x,y}$, we always have $\mathrm{LCA}(x, y) = \perp$. W.l.o.g., we assume that neither x nor y is the root node. Let x' and y' denote the lowest marked ancestors of x and y, respectively, and we divide the query path $\mathrm{P}_{x,y}$ into three disjoint subpaths: $\mathrm{P}'_{x,x'}$, $\mathrm{P}_{x',y'}$ and $\mathrm{P}'_{y,y'}$. Following the inclusion-exclusion principle, we have that $|C(\mathrm{P}_{x',y'})| = |C(\mathrm{P}_{x',\perp})| + |C(\mathrm{P}_{\perp,y'})| - M_{r(x'),r(y')}$. Since the three terms on the right-hand side of this formula have all been precomputed, $|C(\mathrm{P}_{x',y'})|$ can be computed in constant time. Next, we count the number of distinct colors that appear in $\mathrm{P}'_{x,x'}$ but not in $\mathrm{P}_{x',y'}$. This can be done by iterating through each node z in $\mathrm{P}'_{x,x'}$ in the direction towards x and check whether $c(z)$ appears in $\mathrm{P}_{\mathrm{parent}(z),y'}$ by performing a path emptiness query. The number of distinct colors that are in $\mathrm{P}'_{y,y'}$ but not in $\mathrm{P}_{x,y'}$ can be counted in a similar way. Adding these two counts to $|C(\mathrm{P}_{x',y'})|$ yields the answer. Since x (resp. y) and x' (resp. y') are at most $X - 1$ levels apart, the query time is $O(X \lg \lg C)$. This query algorithm is adapted from an algorithm of He and Kazi [44] for arbitrary query paths, though we use a different matrix. As a result, we achieve Lemma 16 that supports queries over any path that contains the root.

Lemma 16 *Let $0 < X \leq n$ be an arbitrary integer. Given that matrix M is available, in which $M_{r(x'),r(y')}$ stores $|C(\mathrm{P}_{x',\perp}) \cap C(\mathrm{P}_{y',\perp})|$ for each pair of $O(n/X)$ marked nodes*

x' and y' selected by the method of Lemma 13, an additional $O(n)$-word space data structure can be constructed in $O(n \lg\lg C)$ time to support finding in $O(X \lg\lg C)$ time $|C(\mathrm{P}_{x,\perp}) \cap C(\mathrm{P}_{y,\perp})|$ for any pair of nodes x and y.

Proof. To see the preprocessing, apart from the matrix M, our data structure consists of three components:

i) an $O(n)$-word data structure for colored path emptiness queries, which can be built in $O(n)$ time, applying Lemma 3,

ii) the numbers of distinct colors on all root-to-node paths, which can be computed in $O(n \lg\lg C)$ time, applying Lemma 12 and

iii) the $O(n/X)$ marked nodes found in $O(n)$ time, applying the method of Lemma 13.

All these data structure components, other than matrix M, can be constructed in overall $O(n \lg\lg C)$ time. Both the space cost of the data structure and its query algorithm have been given, so Lemma 16 follows. ■

4.3.2 Faster Preprocessing via Sparse Matrix Multiplication

To compute matrix M, one way is to define an $m \times C$ matrix A, in which entry $A_{i,\alpha}$ is set to 1 if color $\alpha \in C(\mathrm{P}_{x_i,\perp})$, and it is set to 0 otherwise. Then, we have $M = AA^T$, where A^T denotes the transpose of A. Note that matrix A could have as many as $\Omega(\frac{n}{X} \times C)$ non-zero entries, where C can be as large as n. In the example shown in Figure 4.2, more than half of entries in matrix A are non-zero. As a result, computing M directly by multiplying A and A^T can be costly.

Matrix A might store more non-zero entries than necessary: Given any pair of marked nodes a and b such that b is the lowest marked proper ancestor of a, any color α that appears in $C(\mathrm{P}_{b,\perp})$ appears in $C(\mathrm{P}_{a,\perp})$ as well, so α is recorded multiple times in A. Intuitively, if we only record colors in $C(\mathrm{P}'_{a,b})$ in the matrix row corresponding to node a, then the number of non-zero entries in that row is at most X, and if each row stores no more than X non-zero entries, the new matrix would have $O(n)$ non-zero entries overall, as only m rows exist, where $m \leq n/X + 1$. However, this

A	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	1	1	0	1	0
2	0	1	1	1	1	1
3	0	1	0	1	0	1
4	1	1	0	1	1	1

$\times$

A^T	0	1	2	3	4
0	0	0	0	0	1
1	1	1	1	1	1
2	0	1	1	0	0
3	0	0	1	1	1
4	0	1	1	0	1
5	0	0	1	1	1

$=$

M	0	1	2	3(y')	4
0	1	1	1	1	1
1(x')	1	3	3	1	2
2	1	3	5	3	4
3	1	1	3	3	3
4	1	2	4	3	5

Figure 4.2: An example of computing matrix M using matrix A and A^T. The columns of matrix A (or the rows of matrix A^T) correspond to different node colors, and each different color is encoded by a distinct integer. Note that in this specific example, more than half of the entries in matrix A are non-zero.

leads to a new problem: We cannot compute $|C(\mathrm{P}_{a,\perp})|$ by simply adding up $|C(\mathrm{P}_{b,\perp})|$ and $|C(\mathrm{P}'_{a,b})|$, as $C(\mathrm{P}_{b,\perp}) \cap C(\mathrm{P}'_{a,b})$ might be non-empty. Instead, we propose a new definition, $\hat{C}(a)$, such that $\hat{C}(a) = C(\mathrm{P}'_{a,b}) \setminus C(\mathrm{P}_{b,\perp})$. For example, as shown in Figure 4.1, the lowest marked proper ancestor of marked node x_2 is x', and there are two colors that appear in $\mathrm{P}'_{x_2,x'}$ but not appear in $\mathrm{P}_{x',\perp}$, which are orange (encoded by 3) and red (encoded by 5). The new definition brings us two properties, i.e., $|\hat{C}(a)| \leq X$ and $|C(\mathrm{P}_{a,\perp}) \cap C(\mathrm{P}_{c,\perp})| = |C(\mathrm{P}_{b,\perp}) \cap C(\mathrm{P}_{c,\perp})| + |\hat{C}(a) \cap C(\mathrm{P}_{c,\perp})|$, for any node c, and both of them play an important role in our faster preprocessing method. To achieve faster preprocessing time, we use the computation of two related but different $m \times m$ matrices, $\hat{M}$ and M', as stepping stones, whose definitions will be given later. Due to the first property, matrix $\hat{M}$ can be computed more efficiently using sparse matrix multiplication; due to the second property, turning $\hat{M}$ into M', as well as M' into M, only requires processing $m \times m$ entries in the matrices, and thus the running time can be bounded by $O(n + m^2)$, which could be far less than $O(m \cdot C)$.

Now we are ready to present the faster preprocessing method. We define $\hat{M}$ as an $m \times m$ matrix, in which for each pair of integers $i, j \in [0, m-1]$, $\hat{M}_{i,j}$ stores $|\hat{C}(x_i) \cap \hat{C}(x_j)|$. For example, from Figure 4.1, we know that $\hat{C}(x') = \{$blue:2, pink:4$\}$ and $\hat{C}(y') = \{$orange:3, red:5$\}$, and the intersection of both color sets is empty. Hence, the entry $\hat{M}_{1,3}$ corresponding to the pair of nodes x' and y' is set to be 0. To compute

$\hat{A}$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	0	1	0
2	0	0	0	1	0	1
3	0	0	0	1	0	1
4	1	0	0	0	1	0

$\times$

$\hat{A}^T$	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	1	0	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	0	1	1	0

$=$

$\hat{M}$	0	1	2	$3(y')$	4
0	0	0	0	0	0
$1(x')$	0	2	0	0	1
2	0	0	2	2	0
3	0	0	2	2	0
4	0	1	0	0	2

Figure 4.3: An example of matrices $\hat{M}$ and $\hat{A}$. In matrix $\hat{A}$, each row contains at most 2 non-zero entries, less than X. As a result, there are $O(n)$ non-zero entries in matrix $\hat{A}$.

$\hat{M}_{i,j}$, we define an $m \times C$ matrix $\hat{A}$, and for each integer $i \in [0, m-1]$ and each color $\alpha \in [0, C-1]$, $\hat{A}_{i,\alpha}$ is set to be 1 if $\alpha \in \hat{C}(x_i)$ and 0 otherwise. Then $\hat{M} = \hat{A}\hat{A}^T$. See Figure 4.3 for an example. Each row of $\hat{A}$ indicates whether each color appears in the set $\hat{C}(a)$ for some marked node a and each row has at most X non-zero entries. Since $\hat{A}$ has m rows and $m \leq n/X + 1$, overall $\hat{A}$ has at most $n + X \leq 2n$ non-zero entries only.

To compute the non-zero entries in $\hat{A}$, initialize a bit vector V of C 0 bits. We then perform a preorder traversal of T, and for each marked node x encountered, we compute the non-zero entries in the $r(x)$-th row by iterating over the ancestors of x upward starting from x until we reach the lowest marked proper ancestor, y, of x. Before reaching y, for each color c encountered, if $V[c] = 0$, we set $V[c] = 1$. During this process, we also chain the 1 bits in V using a doubly linked list L. Upon reaching y, L and the positions of the corresponding 1 bits in V give us the non-zero entries in the $r(x)$-th row of A. We then iterate through L to set the corresponding entries of V back to 0 and deallocate L before we continue the traversal of T to locate the next marked node. Since x and y are at most X levels apart, L has at most X items at all times, and hence the non-zero entries at each row of A can be located in $O(X)$ time. Therefore, across all m rows, the non-zero entries of A can be reported in $O(n)$ time. With these non-zero entries as input, we can apply Lemma 14 to compute $\hat{M}$ in $O(n^{(\omega+1)/2}/X^{(\omega-1)/2})$ time for any $X \in [n^{(\omega-1)/(\omega+1)}, n]$.

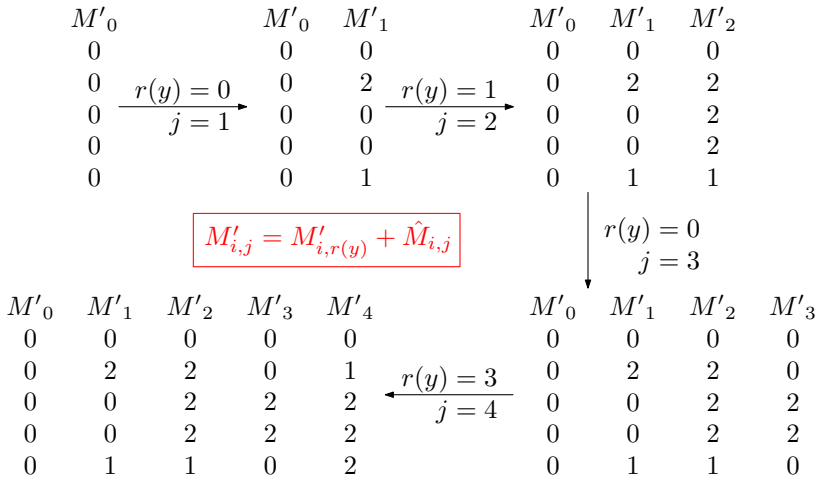


Figure 4.4: Illustrating the construction of matrix M' provided that matrix $\hat{M}$ is available as shown in Figure 4.3. In the figure, node y represents the lowest marked proper ancestor of node x_j, and we use M'_j to represent the j-th column of matrix M'. The figure shows that Matrix M' is constructed from the 0-th column to the $(m-1)$-st column.

The other $m \times m$ matrix that is used to help us compute M is called M'. For each pair of integers $i, j \in [0, m-1]$, entry $M'_{i,j}$ stores $|\hat{C}(x_i) \cap C(\mathrm{P}_{x_j,\perp})|$. To compute M', we perform a preorder traversal of T. Each time we visit a marked node x_j, we compute the j-th column of M' as follows: If $x_j = \perp$, then $j = 0$ and, for each $i \in [0, m-1]$, we set entry $M'_{i,0}$ to be 0, since $c(\perp) \notin \hat{C}(x_i)$. If x_j is not the root, we locate the lowest marked proper ancestor, y, of x_j. Since y is visited before x_j, $M'_{i,r(y)}$ has already been computed, and we set $M'_{i,j} = M'_{i,r(y)} + \hat{M}_{i,j}$. To see the correctness, observe that $M'_{i,j} = |\hat{C}(x_i) \cap C(\mathrm{P}_{x_j,\perp})| = |\hat{C}(x_i) \cap (C(\mathrm{P}_{y,\perp}) \cup \hat{C}(x_j))|$; since $C(\mathrm{P}_{y,\perp}) \cap \hat{C}(x_j) = \emptyset$, this is equal to $|\hat{C}(x_i) \cap C(\mathrm{P}_{y,\perp})| + |\hat{C}(x_i) \cap \hat{C}(x_j)| = M'_{i,r(y)} + \hat{M}_{i,j}$. This way we can compute M' in $O(n + (\frac{n}{X})^2)$ time provided that $\hat{M}$ is available. See Figure 4.4 for an example of constructing matrix M'.

After computing $\hat{M}$ and M', we can compute M by performing another preorder traversal of T. Each time we visit a marked node x_i, we compute the i-th row of M as follows: If $x_i = \perp$, then $M_{i,j} = 1$ for any $j \in [m]$. Otherwise, we locate the lowest marked proper ancestor, y, of x_i. Since y is visited before x_i, $M_{r(y),j}$ has already been computed, and we set $M_{i,j} = M_{r(y),j} + M'_{i,j}$. To see the correctness, observe that $M_{i,j} = |C(\mathrm{P}_{x_i,\perp}) \cap C(\mathrm{P}_{x_j,\perp})| = |(C(\mathrm{P}_{y,\perp}) \cup \hat{C}(x_i)) \cap C(\mathrm{P}_{x_j,\perp})|$; since $C(\mathrm{P}_{y,\perp}) \cap \hat{C}(x_i) = \emptyset$, this is equal to $|C(\mathrm{P}_{y,\perp}) \cap C(\mathrm{P}_{x_j,\perp})| + |\hat{C}(x_i) \cap C(\mathrm{P}_{x_j,\perp})| = M_{r(y),j} + M'_{i,j}$. In this

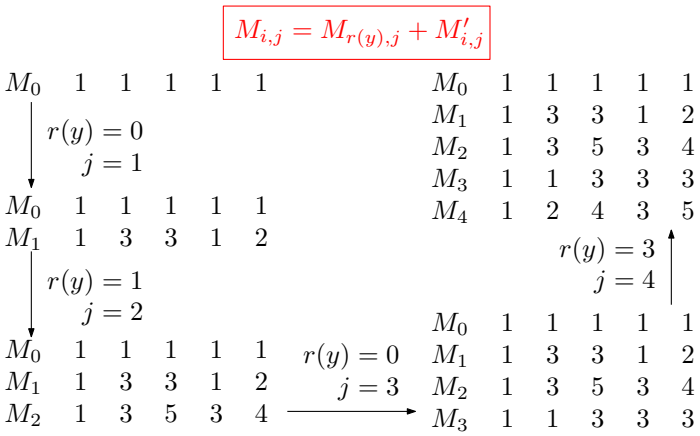


Figure 4.5: Illustrating the construction of matrix M provided that matrix M' is available as shown in Figure 4.4. In the figure, node y represents the lowest marked proper ancestor of node x_j, and we use M_i to represent the i-th row of matrix M. The figure shows that Matrix M is constructed from the 0-th row to the $(m-1)$-st row. After all, one can verify that the matrix M computed in this new method is exactly the same as the one shown in Figure 4.2.

way, we can compute M in $O(n+(\frac{n}{X})^2)$ time after computing $\hat{M}$ and M'. See Figure 4.5 for an example. The total preprocessing time is hence $O(n \lg\lg C+(\frac{n}{X})^2+\frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}})$ for any integer $X \in [n^{\frac{\omega-1}{\omega+1}}, n]$. Our result can be summarized as Lemma 17.

Lemma 17 *Let $X \in [n^{\frac{\omega-1}{\omega+1}}, n]$ be an arbitrary integer. The matrix M, in which $M_{r(x'),r(y')}$ stores $|C(\mathrm{P}_{x',\perp}) \cap C(\mathrm{P}_{y',\perp})|$ for each pair of $O(n/X)$ marked nodes x' and y' selected by the method of Lemma 13, can be constructed in $O(n \lg\lg C+(\frac{n}{X})^2+\frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}})$ time.*

4.3.3 Color Counting on an Arbitrary Path

We now generalize the structure in the previous section to support queries over arbitrary paths. Our strategy is to decompose T using centroid decomposition.

At level 0 of the recursion, the given tree T is a connected component by itself and we call it the level-0 component. We find a centroid, u, of T, and define a new rooted tree T^u by designating u as the root of T, reorienting edges when necessary. Then we build the query structure in Section 4.3.1 over T^u. Afterwards, we remove u from T, and build our data structure recursively over each connected component that has more than X nodes. In general, at the i-th level of the recursion, we have a

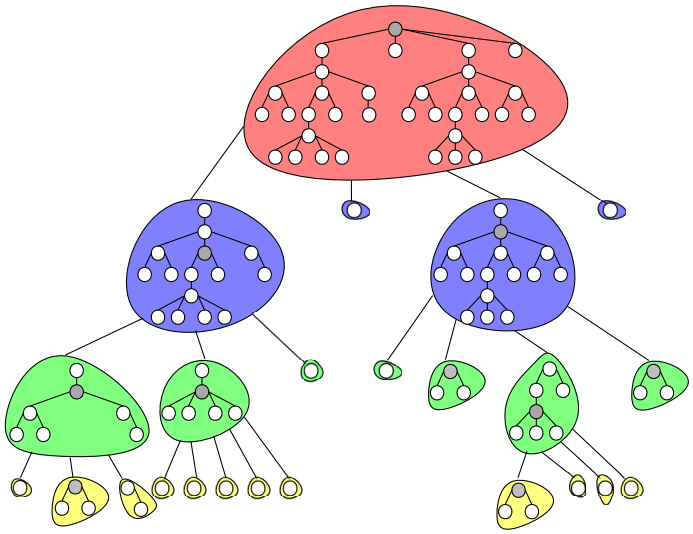


Figure 4.6: Illustrating the recursive data structure using the centroid decomposition. The components in different recursive levels are in different colors. And the centroid in each component that contains more than one node is highlighted in gray color. Each leaf component represents a base component, in which there are at most X nodes, where X is set to be 3 in this particular example.

set of connected components called *level-i components* obtained by removing from T the centroids computed in previous levels of the recursion. For each component, we find its centroid and designate the centroid as the root of this component to build the query structure of Section 4.3.1. One minor detail is that, before building the query structure over a component of size n', we need to ensure that colors are encoded as nonnegative integers less than n'. Thus, when $n' \leq C$, we sort the colors that appear in this component using integer sorting in $O(n' \lg\lg n') = O(n' \lg\lg C)$ time [43] and re-encode these colors using their ranks. Then we remove the centroid of each level-i component to split it into a set of level-$(i+1)$ components and recurse. When a component has at most X nodes, we no longer apply this recursive procedure to it, and we call it a *base component*. Thus, we have $O(\lg(n/X))$ recursion levels. See Figure 4.6 for an example of the recursive data structure.

In addition, for each node $x \in T$, we store a list of $O(\lg(n/X))$ pointers, and the i-th pointer maps x to its copy in a level-i component; this pointer is null if x is removed as a centroid node found in a previous level. Furthermore, we build a weighted tree T' by assigning weights to the nodes of T as follows: If a node x is chosen as the centroid node of a level-i component, its weight is i. If x is never chosen

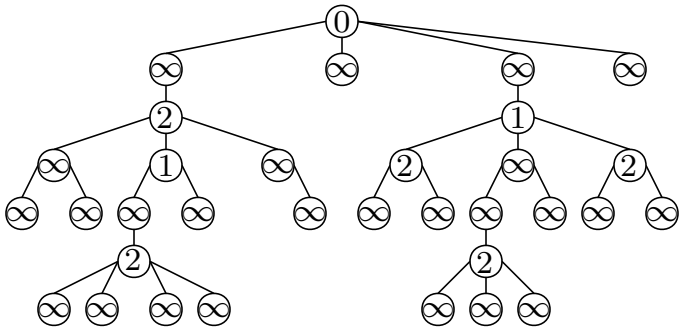


Figure 4.7: An example of the weighted tree T'.

as a centroid, then its weight is ∞. See Figure 4.7 for an example of T'. Then we construct the linear-space data structure of Chan et al. [12, Theorem 1.1] to support *path minimum queries* over T' in constant time; a path minimum query returns the node with minimum weight in a given query path.

Since we have $O(\lg(n/X))$ recursion levels, both the space costs and construction time of this new structure is a factor of $O(\lg(n/X))$ more than those of the structure in Section 4.3.1. Obviously, the space cost is dominated by the storage of the query structures in Section 4.3.1 constructed over the components at all levels. To bound this cost, consider level i of the recursion. Let c_i denote the number of level-i components. We order these components arbitrarily, and let n_j denote the number of nodes in the j-th level-i component. Then $\sum_{j=1}^{c_i} n_j \leq n$. The space cost of the query structures constructed over all level-i components is then $O(\sum_{j=1}^{c_i}(n_j + \frac{n_j^2}{X^2})) \leq O(n + (\sum_{j=1}^{c_i} n_j^2)/X^2) \leq O(n + (\sum_{j=1}^{c_i} n_j \cdot n)/X^2) \leq O(n + \frac{n^2}{X^2})$. Summing up over all $O(\lg \frac{n}{X})$, the total space cost is $O((\lg \frac{n}{X})(n + \frac{n^2}{X^2}))$ words.

Similar to the space cost analysis, it suffices to show that the total time required to construct the query structures in Section 4.3.1 over all the components at any level of recursion is $O(n \lg\lg C + (\frac{n}{X})^2 + \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}})$. To show this, we again use c_i to represent the number of level-i components, and n_j denotes the number of nodes in the j-th level-i component. Then the time needed to construct the query structures over these components is $O(\sum_{j=1}^{c_i}(n_j \lg\lg C + (\frac{n_j}{X})^2 + \frac{n_j^{(\omega+1)/2}}{X^{(\omega-1)/2}})) \leq O(\sum_{j=1}^{c_i}(n_j \lg\lg C + \frac{n_j \cdot n}{X^2} + \frac{n_j \cdot n^{(\omega-1)/2}}{X^{(\omega-1)/2}})) = O(n \lg\lg C + (\frac{n}{X})^2 + \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}})$. Note that the above calculation already includes the time needed to re-encode colors when a component has fewer than C nodes.

To answer a query with $\mathrm{P}_{x,y}$ as the query path, query T' to find the smallest

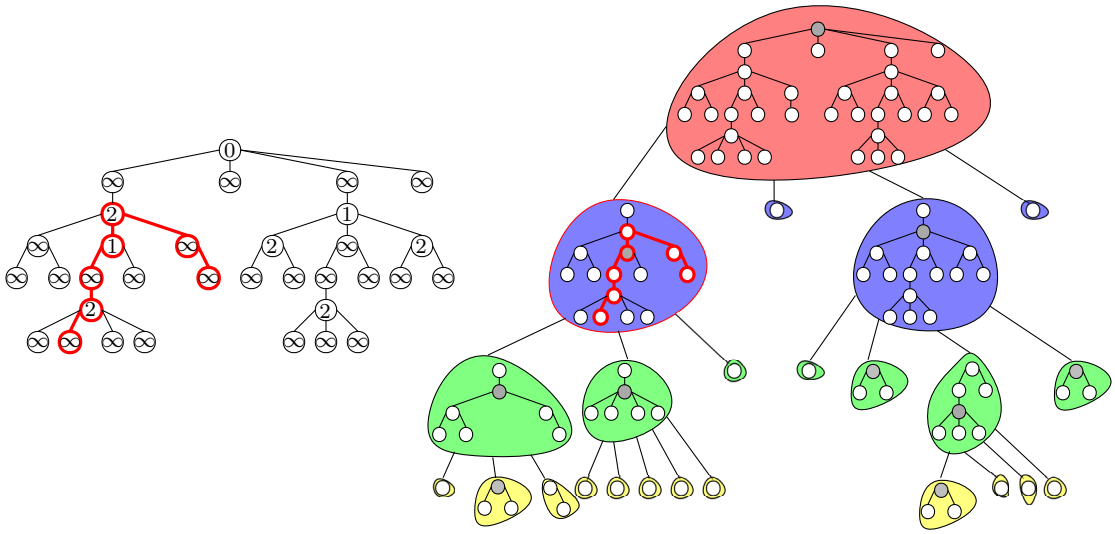


Figure 4.8: An example of identifying the component in the recursive data structure through the path minimum queries. The query path in T' is as shown in the left figure, on which the minimum weight is 1. Therefore, the query proceeds over a level-1 component as shown in the right figure. In this component, the query path contains its centroid.

weight, s, assigned to nodes in $\mathrm{P}_{x,y}$. If $s = \infty$, then x and y are in the same base component, and thus $\mathrm{P}_{x,y}$ has at most X vertices. We can traverse $\mathrm{P}_{x,y}$ in T to count the number of distinct colors; each time we visit a new node, we perform a path emptiness query to determine whether we have already encountered this color. This way we can answer the query in $O(X \lg\lg C)$ time. Otherwise, observe that no nodes in $\mathrm{P}_{x,y}$ are chosen as centroids for recursion level $s-1$ or smaller. Therefore, x and y must reside in a same level-s component S, and $P_{x,y}$ contains the centroid of S. Since this centroid is designated as the root of S before building the query structure of Section 4.3.1 over S, we can use this query structure to answer the query in $O(X \lg\lg C)$ time. See Figure 4.8 for an example. Thus we have:

Lemma 18 *Let T be a colored ordinal tree on n nodes with each node assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$, and let X be an arbitrary integer in $[n^{\frac{\omega-1}{\omega+1}}, n]$. A data structure of $O((\lg \frac{n}{X})(n + \frac{n^2}{X^2}))$ words can be constructed in $O((\lg \frac{n}{X})(n \lg\lg C + (\frac{n}{X})^2 + \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}}))$ time to support colored path counting query in $O(X \lg\lg C)$ time.*

Proof. Combining Lemmas 16 and 17, we achieve an $O(n + (\frac{n}{X})^2)$-word data structure that only supports queries for the paths containing the root. The construction

time is bounded by $O((n \lg \lg C + (\frac{n}{X})^2 + \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}}))$. To generalize this solution to queries over arbitrary paths, we apply the centroid decomposition technique as described in this section. As a result, both the bounds of the space cost and the preprocessing time are increased by an $O(\lg \frac{n}{X})$ factor, while the query time bound is maintained. ■

We finally solve the batched colored path counting problem by first building the query structure of Lemma 18 and then using it to answer n queries. Theorem 4 presents our result:

Theorem 4 *A batch of n colored path counting queries over a colored tree T on n nodes can be answered in $\tilde{O}(n^{\frac{2\omega}{\omega+1}}) = O(n^{1.4071})$ time in total.*

Proof. Recall that the overall running time of batched colored path counting queries includes two parts: the overall query time of n queries, bounded by $O(n \cdot X \lg \lg C)$, and the overall preprocessing time, bounded by $O((\lg \frac{n}{X})(n \lg \lg C + (\frac{n}{X})^2 + \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}}))$. By setting the overall query time to be equal to the preprocessing time, we obtain the proper value for X. The details can be found as follows:

Given that $X \geq n^{\frac{\omega-1}{\omega+1}}$ and $\omega \geq 2$, we notice that

$$\begin{aligned}
(\frac{n}{X})^2 &\leq n^{2(\frac{\omega+1}{\omega+1} - \frac{\omega-1}{\omega+1})} \\
&= n^{\frac{4}{\omega+1}} \\
&\leq n^{\frac{2\omega}{\omega+1}} \\
&= n^{\frac{(\omega+1)+(\omega-1)}{\omega+1}} \\
&= n \cdot n^{\frac{(\omega-1)}{\omega+1}} \\
&\leq n \cdot X.
\end{aligned}$$

And thus, the term $(\frac{n}{X})^2$ in the preprocessing time bound is absorbed by the relatively bigger term $n \cdot X$ in the query time bound, as long as $X \geq n^{\frac{\omega-1}{\omega+1}}$. Observe that the bound of the overall running time is a polynomial of n. In the rest of the analysis, we ignore for simplicity the polylog factors that appear in both the query time and the preprocessing time. By setting the term $n \cdot X$ to be equal to the term $\frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}}$,

we have

$$\begin{aligned} n \cdot X &= \frac{n^{(\omega+1)/2}}{X^{(\omega-1)/2}} \\ X^{(2+\omega-1)/2} &= n^{(\omega+1-2)/2} \\ X^{(1+\omega)/2} &= n^{(\omega-1)/2} \\ X &= n^{\frac{\omega-1}{\omega+1}}. \end{aligned}$$

Therefore, by setting X to be $\lceil n^{\frac{\omega-1}{\omega+1}} \rceil$, the overall running time of batched path colored counting is bounded by $\tilde{O}(n \cdot n^{\frac{\omega-1}{\omega+1}}) = \tilde{O}(n^{\frac{2\omega}{\omega+1}}) = O(n^{1.4071})$. ■

4.4 Batched Path Mode Queries

To solve batched path mode queries over tree T, let t_1 and t_2 be two constant parameters, such that $0 \leq t_2 \leq t_1 \leq 1$. We categorize node colors into two different types, i.e., *infrequent colors* and *frequent colors*. A color α is an infrequent color if it is assigned to at most n^{1-t_1} nodes in T; otherwise, we call it a frequent color. Thereby, a mode of a query path could be either a frequent or an infrequent color.

In Section 4.4.1, we present a data structure for queries over infrequent colors. To achieve that, we define a new problem in trees and name it *path containment*. Given a set of paths as the input and an arbitrary path as the query, the path containment problem is to find out whether any input path is contained by the query path. We will see in Section 4.4.1 two reductions, from finding a most frequent color in the multiset of infrequent colors assigned to the nodes in a query path to the path containment problem and then from the latter to a classic computational geometry problem, *five dimensional dominance range searching*.

To handle frequent colors, we introduce a two-level marking scheme in Section 4.4.2, via which we select $O(n^{1-t_2})$ nodes from the tree as marked nodes.

Let x' and y' be any pair of marked nodes. We prove that finding a most frequent color in the multiset of frequent colors assigned to the nodes in $\mathrm{P}_{x',\perp} \cup \mathrm{P}_{y',\perp}$ can be reduced to min-plus products. In terms of computing the min-plus products, we present a special structure shared by the matrices generated from colored trees, due to which we manage to adjust the three-phase algorithm presented by Gu et al. [39]

and achieve an efficient preprocessing time bound. All these details can be found in Section 4.4.3.

In Section 4.4.4, we assemble all components mentioned above and solve path mode queries for arbitrary paths.

4.4.1 Queries for Infrequent Colors

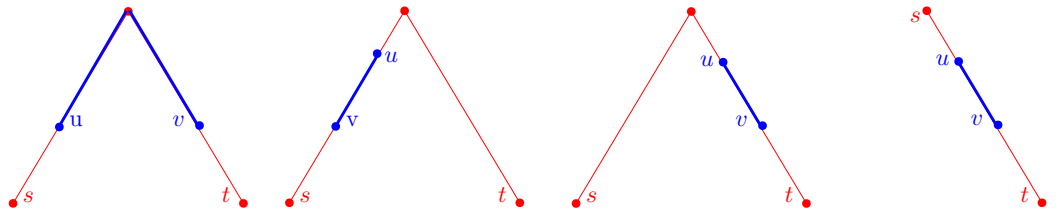
In this section, we first introduce and solve the *path containment* problem in trees. Then we use the solution to this problem as a black box to find one of the most frequent elements in the multiset of infrequent colors assigned to the nodes in the query path.

In the path containment problem, we preprocess a set, S, of paths over an ordinal tree T such that given a query path $\mathrm{P}_{s,t}$, we can determine efficiently whether there exists at least one path in S that is a subpath of $\mathrm{P}_{s,t}$. To solve this problem, we give a reduction from the path containment problem to five-dimensional dominance range emptiness queries. In the latter problem, the dominance range with respect to a query point (a, b, c, d, e) is defined to be $(-\infty, a] \times (-\infty, b] \times (-\infty, c] \times (-\infty, d] \times (-\infty, e]$. Our result is summarized as Lemma 19.

Lemma 19 *Let T denote an ordinal tree and S be a set of $|S|$ paths in T. If T is represented by the data structure of Lemma 2, then a data structure of $O(|S| \lg^4 |S|)$ extra words of space can be constructed in $O(|S| \lg^5 |S|)$ time to answer a path containment query over S in $O(\lg^4 |S|)$ time.*

Proof. Let $\mathrm{P}_{s,t}$ denote the query path and $\mathrm{P}_{u,v}$ denote any path in S. Recall that each node is identified by its preorder rank. W.l.o.g., assume that $u \leq v$ and $s \leq t$.

Observe that $\mathrm{P}_{u,v} \subseteq \mathrm{P}_{s,t}$ iff both nodes u and v are in $\mathrm{P}_{s,t}$. To determine whether both nodes u and v are in $\mathrm{P}_{s,t}$, we can check the ancestor-descendant relationships among nodes u, v, $\mathrm{LCA}(u, v)$, s, t and $\mathrm{LCA}(s, t)$. Especially, given a pair of nodes x and y, node x is an ancestor of node y iff $x \leq y < x + |T_x|$. Our strategy is to map $\mathrm{P}_{u,v}$ to a point, $(u, -u - |T_u| + 1, v, -v - |T_v| + 1, -\mathrm{depth}(\mathrm{LCA}(u, v)))$, in five-dimensional space, so let S' denote the set containing all the points that the paths in S are mapped to. Then, $\mathrm{P}_{u,v} \subseteq \mathrm{P}_{s,t}$ holds in three different cases:



(a) case 1 (b) case-2 (c) case 3-a) (d) case 3-b)

Figure 4.9: Three different cases in which $P_{u,v} \subseteq P_{s,t}$. Note that Case 3 contains two sub-cases.

Case 1: LCA(u, v) is neither u nor v, LCA(s, t) is neither s nor t, and $P_{u,v} \subseteq P_{s,t}$. Observe that if u is an ancestor of s, v is an ancestor of t, and LCA$(u, v) =$ LCA(s, t), then $P_{u,v} \subseteq P_{s,t}$. See Figure 4.9(a) for an example. Given that u is an ancestor of s and v is an ancestor of t, LCA(u, v) is a common ancestor of s and t. If depth(LCA(u, v)) equals to depth(LCA(s, t)), then LCA(u, v) is the same as LCA(s, t). The reason is that at a specific depth, the common ancestor of s and t is unique. Hence, if $u \leq s$, $-u - |T_u| + 1 \leq -s$, $v \leq t$, $-v - |T_v| + 1 \leq -t$ and depth(LCA(u, v)) equals to depth(LCA(s, t)), then $P_{u,v} \subseteq P_{s,t}$. Finding out whether $P_{u,v}$ exits for the first case is equivalent to determining whether in S' there is at least one point dominated by point $(s, -s, t, -t, -\text{depth}(\text{LCA}(s, t)))$. A five-dimensional dominance range emptiness query is sufficient.

Case 2: $P_{u,v} \subseteq P_{\text{LCA}(s,t),s}$. Observe that a path $P_{u,v}$ exists such that $P_{u,v} \subseteq P_{\text{LCA}(s,t),s}$ iff u is an ancestor of s, v is an ancestor of s, and LCA(u, v) is a descendant of LCA(s, t). See Figure 4.9(b) for an example. To find out whether $P_{u,v}$ exists, it is sufficient to perform a five-dimensional dominance range emptiness query over S' using $(s, -s, s, -s, -\text{depth}(\text{LCA}(s, t))$ as the query point.

Case 3: $P_{u,v} \subseteq P_{\text{LCA}(s,t),t}$. Observe that a path $P_{u,v}$ exists such that $P_{u,v} \subseteq P_{\text{LCA}(s,t),t}$ iff u is an ancestor of t, v is an ancestor of t, and LCA(u, v) is a descendant of LCA(s, t). See Figures 4.9(c) and 4.9(d) for examples. This can be determined by performing a five dimensional dominance range emptiness query over S' with query point $(t, -t, t, -t, -\text{depth}(\text{LCA}(s, t)))$.

Given a query path $P_{s,t}$, we map it into three query ranges shown in the three

different cases mentioned earlier and perform 5D dominance range emptiness queries to find out whether at least one of three query ranges contains at least a point. If so, at least one path in S is contained in $\mathrm{P}_{s,t}$; otherwise, no path in S is contained in $\mathrm{P}_{s,t}$.

Since T is represented by the data structure of Lemma 2, operations LCA, nbdesc and depth can be supported in constant time. Therefore, a path from set S can be mapped in constant time into a point in S', and the 5D dominance query range can be computed in constant time as well. To answer a 5D dominance range emptiness query over S', we construct a five-dimensional range tree [7, Section 5.4]. Since S' has $|S|$ points, this data structure occupies $O(|S| \lg^4 |S|)$ words of space, answers a query in $O(\lg^4 |S|)$ time, and can be built in $O(|S| \lg^5 |S|)$ time. With it, a path containment query can be answered in $O(\lg^4 |S|)$ time. ■

The path containment problem described above can be regarded as a generalized version of interval containment queries over intervals on a real line. As shown in [69, Proof of Theorem 6.1], the latter problem can be used for finding a most frequent element in the multi-set of infrequent colors within the sub-array. Similarly, we can apply Lemma 19 to support queries over infrequent colors in trees.

Lemma 20 *An $O(n^{2-t_1} \lg^4 n)$-word structure can be constructed in $O(n^{2-t_1} \lg^5 n)$ time such that given a query path $\mathrm{P}_{x,y}$, the most frequent element and its frequency in the multiset of infrequent colors assigned to the nodes in $\mathrm{P}_{x,y}$ can be computed in $O(n^{1-t_1} \lg^4 n)$ time.*

Proof. First, we represent T using the data structure of Lemma 2. Then, we iterate through each infrequent color α'. For each pair of nodes, u and v, colored in α', such that $u \leq v$, we create path $\mathrm{P}_{u,v}$ with assigned weight $\mathtt{w}$, where $\mathtt{w}$ denotes the number of nodes colored in α' in $\mathrm{P}_{u,v}$. We claim that in this way, the total number of weighted paths to be created for all infrequent colors is at most n^{2-t_1}. To see the correctness of the claim, let $\mathrm{Freq}(\alpha)$ denote the total number of tree nodes that are colored in α. For any infrequent color α', we have $\mathrm{Freq}(\alpha') \leq n^{1-t_1}$; hence, the total number of paths created is at most $\sum_{\alpha'} \binom{\mathrm{Freq}(\alpha')}{2} \leq \sum_{\alpha'} (\mathrm{Freq}(\alpha') \cdot n^{1-t_1}) \leq n^{2-t_1}$, since $\sum_{\alpha'} \mathrm{Freq}(\alpha') \leq n$.

We divide the weighted paths into n^{1-t_1} groups, such that the paths that share the same weight, $\mathtt{w}$, are assigned to group $\mathtt{w}$, for each $1 \leq \mathtt{w} \leq n^{1-t_1}$. Then we construct the data structure of Lemma 19 over the paths within each group. Since at most n^{2-t_1} paths are created, the overall space cost of the data structure for all groups is $O(n^{2-t_1} \lg^4 n)$ words and constructing the data structure of Lemma 19 over $O(n^{2-t_1})$ paths takes $O(n^{2-t_1} \lg^5 n)$ time.

Given a query path $\mathrm{P}_{x,y}$, the most frequent element (along with its frequency) in the multiset of infrequent colors assigned to $\mathrm{P}_{x,y}$ can be found as follows: Follow the order from group n^{1-t_1} to group 1 and perform a path containment query over paths within each group until we find the first path, $\mathrm{P}_{u,v}$, that is contained in $\mathrm{P}_{x,y}$. Finally, return the color of the endpoints of $\mathrm{P}_{u,v}$ and the path weight (as its frequency). Since there are n^{1-t_1} groups and a path containment query for each group can be answered in $O(\lg^4 n)$ time, the overall query time is $O(n^{1-t_1} \lg^4 n)$. ■

In our solution to path mode queries, we divide in the preprocessing stage the color set into two types, i.e., the set of infrequent colors and the set of frequent colors. A mode of a path can be from either of both types. The data structure of Lemma 20 allows us to find a most frequent color in the multiset of the infrequent colors assigned to the nodes in an arbitrary query path. We will see in Section 4.4.4 how the data structure of Lemma 20 is being used in solving path mode queries.

4.4.2 Marking $O(n^{1-t_2})$ Nodes

To solve colored path counting, we apply the node-marking strategy following from Lemma 13 once for each connected component. As a result, we select at most n'/X nodes to be marked for a component with n' nodes. As we have seen in Section 4.3.1, this single-level marking scheme is sufficient for solving colored path counting. When applying the same scheme in path mode queries, we found the matrix generated from the marked nodes does not have the needed structure for faster computation of the min-plus product. The detail of the structure will be explained later in Section 4.4.3. Therefore, we introduce a two-level marking scheme as follows:

Let X be an integer parameter in $[n^{t_2}, n]$ to be determined later. We choose at most n/X nodes of T using the method of Lemma 13, mark these nodes and the root, and call them *tier*-1 *marked nodes*. Since the tier-1 marked nodes form a subset

of levels of T, removing them splits T into a forest of subtrees. Then we use the same lemma to mark nodes at every $\lceil n^{t_2} \rceil$ levels of each subtree starting from some level of the subtree, and these nodes are called *tier-2 marked nodes*. Since the total number of nodes over all subtrees is less than n, there are $O(n^{1-t_2})$ tier-2 marked nodes. We regard both tier-1 and tier-2 marked nodes as *marked nodes*, and there are $O(n/X + n^{1-t_2}) = O(n^{1-t_2})$ marked nodes in total, as $X \geq n^{t_2}$. It follows that a path that connects any non-root node to its lowest proper ancestor that is marked contains no more than $\lceil n^{t_2} \rceil + 1$ nodes. As before, we refer to the i-th marked node, x_i, visited in a preorder traversal as the *i-th marked node* for short, starting from 0, and $r(x_i)$ is the rank of the marked node x_i, where $r(x_i) \in O(n^{1-t_2})$.

4.4.3 Queries for Frequent Colors over Predefined Paths

We refer to a path that contains root $\perp$ and has a pair of marked nodes as its endpoints as a predefined path. Given a predefined path $\mathrm{P}_{x',y'}$, we present in this section how to find a most frequent element and its frequency in the multiset of frequent colors assigned to nodes in $\mathrm{P}_{x',y'} \setminus \{\perp\}$ (or $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$)[1]. The techniques for computing min-plus products are used in our solution.

Reducing Path Queries to Min-Plus-Query-Witness Problem

A frequent color appears more than n^{1-t_1} times in T, so there are only $O(n^{t_1})$ distinct frequent colors. Let μ denote the total number of marked nodes, and let κ denote the total number of distinct frequent colors. Then $\mu = O(n^{1-t_2})$ and $\kappa = O(n^{t_1})$. We number the frequent colors incrementally starting from 0 in an arbitrary order, and we refer to the frequent color numbered by k as color f_k, where $0 \leq k \leq \kappa - 1$. Next, we construct a $\mu \times \kappa$ matrix M. Corresponding to marked node x_i and frequent color f_k, $M_{i,k}$ stores the negation of the frequency of f_k in path $\mathrm{P}'_{x_i,\perp}$. It follows that the min-plus product of M and its transpose, denoted by $M \star M^T$, is a $\mu \times \mu$ matrix, in which entry $(M \star M^T)_{i,j}$ stores the negation of the maximum frequency of a frequent color in $\mathrm{P}'_{x_i,\perp} \cup \mathrm{P}'_{x_j,\perp}$, provided $\perp \in \mathrm{P}_{x_i,x_j}$. As before, some entries of this matrix correspond to a pair of nodes whose lowest common ancestor is not the root

[1]For each color other than $c(\perp)$, its frequencies in $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ and in $\mathrm{P}_{x',y'}$ are exactly the same, while later we consider $c(\perp)$ separately.

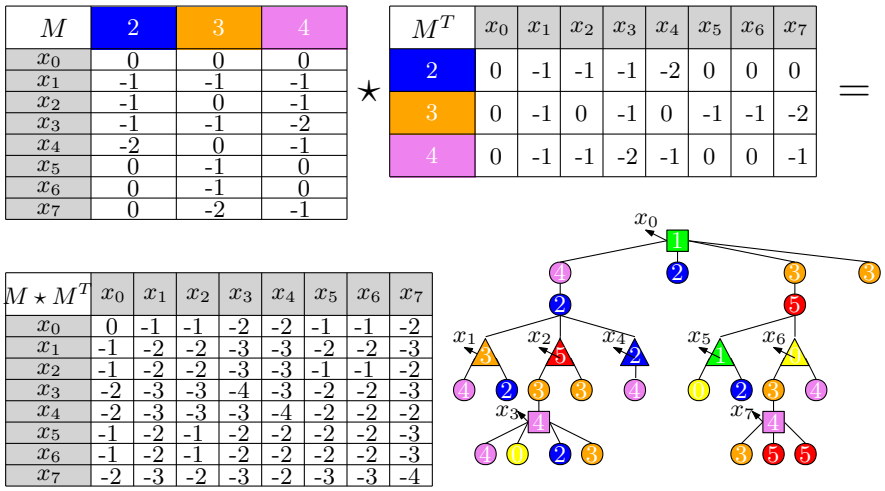


Figure 4.10: An example of finding maximum frequency of a frequent color in each predefined path. The frequent colors, including 2, 3 and 4, each appear more than 5 times in the tree. The tier-1 marked nodes, represented by squares, are selected by the method of Lemma 13, setting parameter t to be 5. Removing all the tier-1 nodes splits the tree into a forest of subtrees, and the tier-2 marked nodes, denoted by triangles, are selected by the method of Lemma 13 to each subtree, setting t to be 3. The matrices M, M^T, and $M \star M^T$ are defined as before. A mode of the predefined path P_{x_3,x_7} is 4, whose frequency is 3, and entry $(M \star M^T)_{3,7}$ stores -3.

and are thus never used, but we compute these entries regardless. See Figure 4.10 for an example. In this way, finding a most frequent element in the multiset of frequent colors assigned to nodes in each predefined path is equivalent to solving Problem 1 using M and M^T as input matrices. In other words, given k^* as the answer for a query (i, j), we know that color f_{k^*} appears at least as frequently as other frequent colors in the predefined path $\mathrm{P}'_{x_i,\perp} \cup \mathrm{P}'_{x_j,\perp}$. Naively, computing the answers for all pairs of (i, j) takes $O(\mu^2\kappa)$ time. We describe a data structure solution, in which both the preprocessing time and the overall running time for n queries are bounded by $o(\mu^2\kappa)$.

Solving Problem 1 with Three Preprocessing Steps

Henceforth, we use k^* to denote an index such that $M_{i,k^*} + M^T_{k^*,j} = \min_k\{M_{i,k} + M^T_{k,j}\}$. Gu et al. provided an efficient solution to Problem 1 when the second input matrix is monotone, i.e., entries in the same row are non-decreasing. In our case, the second matrix, M^T, does not have such a property due to the tree topology. For example,

matrix M^T in Figure 4.10 is obviously not a monotone matrix.

The Total Difference of a Matrix. Our first step is to generalize their definition of *total range* over a monotone matrix to our notion of *total difference*, defined over an arbitrary $a \times b$ matrix A as $\sum_{j=1}^{b-1}(\sum_{k=0}^{a-1}[\![A_{k,j} \neq A_{k,j-1}]\!])$, in which the Iverson notation $[\![A_{k,j} \neq A_{k,j-1}]\!]$ evaluates to 1 if $A_{k,j} \neq A_{k,j-1}$ is true and 0 otherwise. We will see that in our data structure solution, the total difference of M^T decides the preprocessing time. For now, we prove that the total difference of M^T is bounded.

Lemma 21 *The total difference of matrix M^T is bounded by $O(n)$.*

Proof. Recall that we refer to the i-th node visited in a preorder traversal as the *i-th node* for short, where i starts from 0. For each node x, we precompute $\hat{r}(x)$ which is the rank of x among n nodes defined this way. We turn tree T into a directed graph G by replacing each edge (u, v) of T with directed edges (u, v) and (v, u) and construct an empty integer array E. Starting from the root node, for each node x visited in Eulerian tour of G, append $\hat{r}(x)$ into E. Since G has $2(n-1)$ edges, E stores $2(n-1)$ entries in the end, and $\hat{r}(x)$ appears d times in $E[0, 2n-3]$ if the degree of x in T is d. Let $F[0, n-1]$ denote an array such that $F[i]$ stores the leftmost index of integer i in E, and let x_j denote the j-th marked node in a preorder traversal. Observe that i) $F[\hat{r}(x_j)] - F[\hat{r}(x_{j-1})]$ equals to the number of nodes in the walk from the marked node x_{j-1} (not included) to the marked node x_j and ii) each node in this walk contributes at most one value to $\sum_{k=0}^{\kappa-1} |M^T_{k,j} - M^T_{k,j-1}|$. Therefore, we have $\sum_{k=0}^{\kappa-1} |M^T_{k,j} - M^T_{k,j-1}| \leq F[\hat{r}(x_j)] - F[\hat{r}(x_{j-1})]$. Hence, $\sum_{j=1}^{\mu-1}\sum_{k=0}^{\kappa-1}[\![M^T_{k,j} \neq M^T_{k,j-1}]\!] \leq \sum_{j=1}^{\mu-1}\sum_{k=0}^{\kappa-1} |M^T_{k,j} - M^T_{k,j-1}| \leq F[\hat{r}(x_{\mu-1})] = O(n)$. This telescoping sum shows that the total difference of matrix M^T is always bounded by $O(n)$. ∎

To improve the preprocessing time to be $o(\mu^2\kappa)$, the total difference of M^T as shown in Lemma 21 is still not small enough. Gu et al. decreased the total difference of their second input matrix by simply dividing each matrix entry by a carefully chosen integer and then rounding down each resulted quotient. However, the same method would fail to decrease the total difference of M^T in our case as dividing by a nonzero integer won't change equality; $\sum_{k=0}^{\kappa-1}[\![M^T_{k,j} \neq M^T_{k,j-1}]\!]$ might be equal to

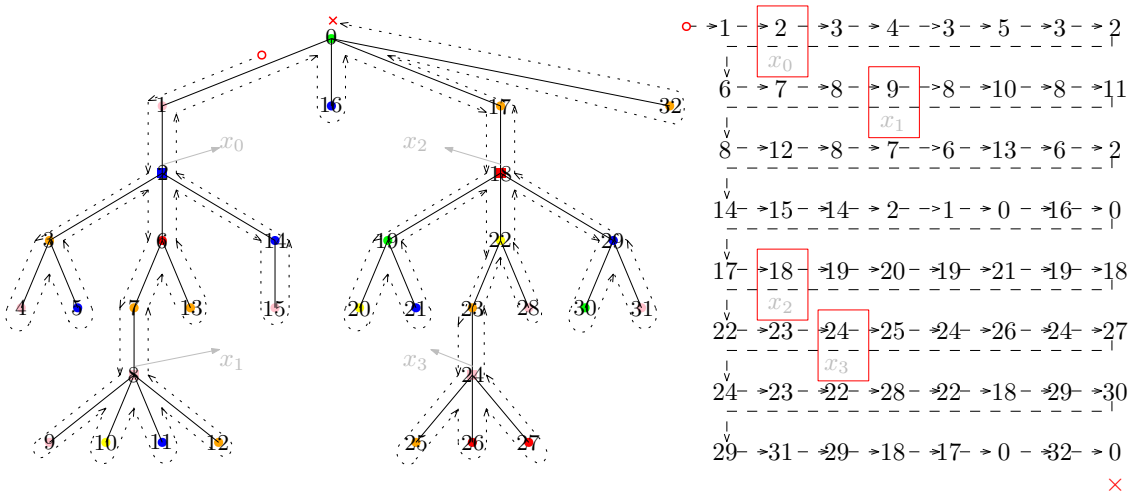


Figure 4.11: Illustrating the proof of Lemma 21 with an example. On the left-hand side, it shows the Euler Tour of a colored tree with 33 nodes. The tour starts with the circle and ends with the cross, visiting 64 ($= 2 \times (33 - 1)$) nodes in total. Each node has an integer on it, representing its preorder rank. The arrays $E[0, 63]$ and $F[0, 32]$ are presented in the right-hand side. Especially, the entries of array E are in black color, and the entries of array F that correspond to the marked nodes are in light color and highlighted by rectangles; namely, $F[\hat{r}(x_0)] = 1$, $F[\hat{r}(x_1)] = 11$, $F[\hat{r}(x_2)] = 33$ and $F[\hat{r}(x_3)] = 42$.

$\sum_{k=0}^{\kappa-1}[\![\lfloor \frac{M^T_{k,j}}{c} \rfloor \neq \lfloor \frac{M^T_{k,j-1}}{c} \rfloor]\!]$ for some integer $c \neq 0$. So we design a new approach to construct a matrix $\tilde{B}$ that has a smaller total difference.

Making the Total Difference Smaller. Henceforth, we define $\text{mparent}_1(v)$ to be node v's lowest proper ancestor that is tier-1 marked and $\text{mparent}(v)$ to be v's lowest proper ancestor that is either tier-1 or tier-2 marked. For a tier-1 marked node $\hat{v}_1$, let $R_k(\hat{v}_1)$ denote the frequency of the frequent color f_k in $\text{P}'_{\hat{v}_1,\perp}$, and for a tier-2 marked node $\hat{v}_2$, let $R'_k(\hat{v}_2)$ denote the frequency of f_k in $\text{P}'_{\hat{v}_2,\text{mparent}(\hat{v}_2)}$ and $\Psi(\hat{v}_2)$ denote the set of tier-2 marked nodes in $\text{P}'_{\hat{v}_2,\text{mparent}_1(\hat{v}_2)}$. We define parameter W to be $\lfloor n^{(1-t_2)\theta} \rfloor$, where θ is a constant with $0 \leq \theta \leq 1$. With the above notation, we construct matrix $\tilde{B}$ which has the same size as M^T. For each pair (k, j), if the j-th column of M^T corresponds to a tier-1 marked node $\hat{v}_1$, then $\tilde{B}_{k,j}$ stores $\lfloor \frac{-R_k(\hat{v}_1)}{W} \rfloor$. Otherwise, this column must correspond to a tier-2 marked node $\hat{v}_2$; let $\hat{v}_1$ denote $\text{mparent}_1(\hat{v}_2)$ and we set $\tilde{B}_{k,j} = \lfloor \frac{-R_k(\hat{v}_1)}{W} \rfloor + \sum_{u \in \psi(\hat{v}_2)} \lfloor \frac{-R'_k(u)}{W} \rfloor$. Observe that

$M^T_{k,j} = -R_k(\hat{v}_1) + \sum_{u\in\psi(\hat{v}_2)}(-R'_k(u))$. It follows that

$$\tilde{B}_{k,j}W \leq M^T_{k,j} \leq \tilde{B}_{k,j}W + (|\psi(\hat{v}_2)|+1)(W-1). \tag{4.1}$$

Furthermore, Lemma 22 shows that the total difference of $\tilde{B}$ is bounded.

Lemma 22 *The total difference of $\tilde{B}$ is $O(\frac{n}{X}\cdot n^{t_1} + \frac{n}{W})$. Setting X to be $\lfloor Wn^{t_1}\rfloor$, the total difference of $\tilde{B}$ is $O(n/W)$.*

Proof. Let $\hat{v}_j$ be any marked node that corresponds to column M^T_j where $j \geq 1$, and let Diff_j denote $\sum_{k=0}^{\kappa-1}[\![\tilde{B}_{k,j} \neq \tilde{B}_{k,j-1}]\!]$. We prove that the total difference, $\sum_{j=1}^{\mu-1}\text{Diff}_j$, is upper bounded by $O(\frac{n}{X}\cdot n^{t_1} + \frac{n}{W})$.

If $\hat{v}_j$ is a tier-1 marked node, Diff_j could be as large as $\kappa = O(n^{t_1})$, since matrix $\tilde{B}$ has κ rows and Diff_j is no more than κ. All $O(n/X)$ tier-1 marked nodes correspond to $O(n/X)$ columns in $\tilde{B}$, and these columns contribute a value of $O(n/X)\cdot\kappa = O(\frac{n}{X}\cdot n^{t_1})$ to the total difference of $\tilde{B}$. If $\hat{v}_j$ is a not tier-1 marked node, then it must be a tier-2 marked node. Let $\hat{u}_{j-1}$ denote the lowest ancestor[2] of $\hat{v}_{j-1}$ that is a tier-1 marked node. We consider two different cases: i) $\hat{u}_{j-1}$ is different from $\text{mparent}_1(\hat{v}_j)$; and ii) $\hat{u}_{j-1}$ is the same as $\text{mparent}_1(\hat{v}_j)$.

We claim that the number of pairs of $\hat{v}_{j-1}$ and $\hat{v}_j$ in the first case is bounded by $O(n/X)$. Since $\hat{u}_{j-1}$ is different from $\text{mparent}_1(\hat{v}_j)$, $\hat{u}_{j-1}$ must be in path $\text{P}'_{\hat{v}_{j-1},\hat{z}_j}$ (not including node $\hat{z}_j$), where $\hat{z}_j$ denotes $\text{LCA}(\hat{v}_{j-1},\hat{v}_j)$. For each $1 \leq j \leq \mu-1$, let S denote the set of paths $\text{P}'_{\hat{v}_{j-1},\hat{z}_j}$ such that $\hat{v}_j$ is a tier-2 marked and $\hat{u}_{j-1}$ is different from $\text{mparent}_1(\hat{v}_j)$. Observe that each tier-1 marked node appears in at most one path in S. Given that there are $O(n/X)$ marked nodes, S stores at most $O(n/X)$ paths, so the claim has been proved. Although Diff_j could be as large as κ, given the limited number of pairs of $\hat{v}_{j-1}$ and $\hat{v}_j$ in the first case, the total difference contributed by the corresponding columns is bounded by $O(n/X)\cdot\kappa = O(\frac{n}{X}\cdot n^{t_1})$.

If $\hat{u}_{j-1}$ is the same as $\text{mparent}_1(\hat{v}_j)$, then both $\hat{v}_{j-1}$ and $\hat{v}_j$ are in the subtree rooted by $\hat{u}_{j-1}$ (or $\text{mparent}_1(\hat{v}_j)$). Consider the simplest case in which node $\hat{v}_{j-1}$ is $\text{mparent}(\hat{v}_j)$. Following the strategy that is used to construct matrix $\tilde{B}$, for any frequent color f_k, entry $\tilde{B}_{k,j-1}$ differs from $\tilde{B}_{k,j}$ iff color f_k appears no less than W times in $\text{P}'_{\hat{v}_j,\text{mparent}(\hat{v}_j)}$. Note that $\text{P}'_{\hat{v}_j,\text{mparent}(\hat{v}_j)}$ contains at most $\lceil n^{t_2}\rceil$ nodes following

[2]Note that $\hat{u}_{j-1}$ could be the same as $\hat{v}_{j-1}$ in the case that $\hat{v}_{j-1}$ is a tier-1 marked node.

from the method of Lemma 13; therefore, $\text{Diff}_j \leq \frac{\lceil n^{t_2} \rceil}{W}$. In general, if $\text{P}_{\hat{v}_j, \hat{v}_{j-1}} \setminus \hat{v}_{j-1}$ contains ℓ marked nodes, then $\text{Diff}_j \leq \ell \cdot \frac{\lceil n^{t_2} \rceil}{W}$. Although the value Diff_j for a specific j might vary according to the value of ℓ, which could be as large as κ, we aim at bounding the sum of Diff_j in Case-ii). We define S to be the set of paths $\text{P}_{\hat{v}_{j-1}, \hat{v}_j}$ such that $\hat{u}_{j-1}$ is the same as $\text{mparent}_1(\hat{v}_j)$ and T_γ to be a *tree extraction* [48] from tree T of the set of all marked nodes (including tier-1 and tier-2 marked nodes). With Eulerian tour technique being applied to T_γ, similar to the proof of Lemma 21, one can prove that all paths in S contains at most $O(n/X + n^{1-t_2}) = O(n^{1-t_2})$ marked nodes in total; therefore, the sum of Diff_j in Case-ii) is bounded by $O(n^{1-t_2}) \cdot \frac{\lceil n^{t_2} \rceil}{W} = O(n/W)$.

By summing up the total differences computed in each case above, we prove that the total difference of $\tilde{B}$ is bounded by $O(\frac{n}{X} \cdot n^{t_1} + \frac{n}{W})$. ∎

New Notations in Our Method. Gu et al. [39] solved Problem 1 with three preprocessing steps. We generalize their solution by adjusting each step. After all, our method is workable for matrices M and M^T, where M^T is even non-monotone but has a bounded total difference. Before introducing the adjusted version of the method by Gu et al., we introduce some notations used in this new method.

We define a matrix $\tilde{A}$, in which entry $\tilde{A}_{i,k} = \lfloor \frac{M_{i,k}}{W} \rfloor$. Let's compare the matrices M and $\tilde{A}$ as well as the matrices M and $\tilde{B}$.

Lemma 23 *(i) $\tilde{A}_{i,k} \cdot W \leq M_{i,k}$; (ii) $M_{i,k} \leq W\tilde{A}_{i,k} + W - 1$; (iii) $\tilde{B}_{k,j} \cdot W \leq M^T_{k,j}$; and (iv) $M^T_{k,j} \leq W \cdot \tilde{B}_{k,j} + (W-1) \cdot (2 + W \cdot n^{(t_1 - t_2)})$.*

Proof. Given that $\tilde{A}_{i,k} = \lfloor \frac{M_{i,k}}{W} \rfloor$, statement (i) and statement (ii) immediately follow, and statement (iii) holds following from Equation 4.1. Let $\hat{v}_j$ be the marked node that corresponds to column M^T_j where $j \geq 1$. As shown as Equation 4.1, if $\text{P}'_{\hat{v}_j, \text{mparent}_1(\hat{v}_j)}$ contains ℓ marked nodes, then we have $M^T_{k,j} \leq \tilde{B}_{k,j} W + (\ell + 1)(W - 1)$. By the method of Lemma 13, $\text{P}'_{\hat{v}_j, \text{mparent}_1(\hat{v}_j)}$ contains at most $\lceil X / \lceil n^{t_2} \rceil \rceil$ nodes that are tier-2 marked. Since X is set to $\lfloor W n^{t_1} \rfloor$, $\ell \leq W \cdot n^{(t_1 - t_2)} + 1$; therefore, $\tilde{B}_{k,j} \cdot W \leq M^T_{k,j} \leq W \cdot \tilde{B}_{k,j} + (W-1) \cdot (2 + W \cdot n^{(t_1 - t_2)})$. ∎

Let matrix $\tilde{C}'$ denote $\tilde{A} \star \tilde{B}$. Following [39], we call a triple (i, k, j)

- *almost relevant* if $0 \leq \tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C}'_{i,j} \leq \frac{(W-1) \cdot (3 + W \cdot n^{(t_1 - t_2)})}{W}$, and

- *weakly relevant* if $0 \leq M_{i,k} + M^T_{k,j} - W \cdot \tilde{C'}_{i,j} \leq 2(W-1) \cdot (3 + W \cdot n^{(t_1 - t_2)})$.

It turns out that the set of almost relevant triples is a subset of the set of weakly relevant triples.

Lemma 24 *Every almost relevant triple is weakly relevant as well.*

Proof. Following from statement (i) and statement (iii) of Lemma 23, we first prove that $M_{i,k} + M^T_{k,j} - W \cdot \tilde{C'}_{i,j}$ is always no less than 0:

$$\begin{aligned} W \cdot \tilde{C'}_{i,j} &\leq W \cdot \tilde{A}_{i,k} + W \cdot \tilde{B}_{k,j} \\ &\leq M_{i,k} + M^T_{k,j}. \end{aligned}$$

It remains to show that if $\tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$, then $M_{i,k} + M^T_{k,j} - W \cdot \tilde{C'}_{i,j} \leq 2(W-1) \cdot (3 + W \cdot n^{(t_1-t_2)})$. Following from statement (ii) and statement (iv) of Lemma 23, one can deduce that

$$\begin{aligned} M_{i,k} + M^T_{k,j} - W\tilde{C'}_{i,j} &\leq (W\tilde{A}_{i,k} + W - 1) \\ &\quad + (W\tilde{B}_{k,j} + (W-1)(2 + Wn^{(t_1-t_2)})) - W\tilde{C'}_{i,j} \\ &= W(\tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j}) + 3(W-1) + (W-1)Wn^{(t_1-t_2)} \\ &\leq (W-1)(3 + Wn^{(t_1-t_2)}) + 3(W-1) + (W-1)Wn^{(t_1-t_2)} \\ &= 2(W-1)(3 + Wn^{(t_1-t_2)}). \end{aligned}$$

Therefore, every almost relevant triple is weakly relevant as well. ∎

Lemma 25 explains why we care about triples that are almost relevant.

Lemma 25 $\tilde{A}_{i,k^*} + \tilde{B}_{k^*,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)(Wn^{t_1-t_2}+3)}{W}$, *where* k^* *denotes an index such that* $M_{i,k^*} + M^T_{k^*,j} = \min_k\{M_{i,k} + M^T_{k,j}\}$

Proof. Let k' be the index such that $M_{i,k'} + M^T_{k',j} = \min_{k \in K'}\{M_{i,k} + M^T_{k,j}\}$, where $K' = \{k : \tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} > \frac{(W-1)(Wn^{t_1-t_2}+3)}{W}\}$; let k'' be the index such that $M_{i,k''} + M^T_{k'',j} = \max_{k \in K''}\{M_{i,k} + M^T_{k,j}\}$, where $K'' = \{k : \tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \leq 0\}$. We claim that $M_{i,k'} + M^T_{k',j} > M_{i,k''} + M^T_{k'',j}$ always holds. The claim can be proved using

Lemma 23 and the definitions of k' and k'':

$$\begin{aligned} M_{i,k'} + M^T_{k',j} &\geq W\tilde{A}_{i,k'} + W\tilde{B}_{k',j} \\ &> W\tilde{C'}_{i,j} + (W-1)(Wn^{(t_1-t_2)}+3) \\ &\geq W\tilde{A}_{i,k''} + W\tilde{B}_{k'',j} + (W-1)(Wn^{(t_1-t_2)}+3) \\ &= W\tilde{A}_{i,k''} + (W-1) + W\tilde{B}_{k'',j} + (W-1)(2+Wn^{(t_1-t_2)}) \\ &\geq M_{i,k''} + M^T_{k'',j}. \end{aligned}$$

As a result, we know that $k^* \notin K'$. Let K^* denote set $\{k : \tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)(Wn^{t_1-t_2}+3)}{W}\}$. Therefore, $k^* \in K^*$. ■

Overview of Our Method. The method contains three preprocessing steps. In the first preprocessing step, we introduce a fast algorithm that computes matrix $\tilde{C}'$. Recall that $\tilde{C}' = \tilde{A} \star \tilde{B}$. Matrix $\tilde{C}'$ will be used in the second preprocessing step.

In the second preprocessing step, we construct $(c+2)\mu^\rho \ln n$ different pairs of matrices $A^{(r)}$ and $B^{(r)}$ using a randomized technique, where $\rho \geq 0$ is a parameter. All finite entries of $A^{(r)}$ are of small absolute values, so we can apply Lemma 15 to compute the min-plus product of $A^{(r)}$ and $B^{(r)}$ efficiently. Especially, the non-infinity entries of $A^{(r)}$ and entries of $B^{(r)}$ are linear combinations of matrices M, M^T and $\tilde{C}'$. Let $C^{(r)} = A^{(r)} \star B^{(r)}$ for each $r \in [(c+2)\mu^\rho \ln n]$. Given a query (i,j), our ultimate goal is to find k^*. We observe that if $A^{(\hat{r})}_{i,k^*} \neq \infty$ for some $\hat{r} \in [(c+2)\mu^\rho \ln n]$, then $M_{i,k^*} + M^T_{k^*,j} = C^{(\hat{r})}_{i,j} + W \cdot \tilde{C'}_{i,j^{(\hat{r})}}$ and $M_{i,k^*} + M^T_{k^*,j} \leq C^{(r)}_{i,j} + W \cdot \tilde{C'}_{i,j^{(r)}}$ for all $r \in [(c+2)\mu^\rho \ln n]$. Of course, we don't know what $\hat{r}$ is beforehand, so we have to iterate through each $r \in [(c+2)\mu^\rho \ln n]$ to figure it out. It turns out that this iteration is affordable.

However, there might be cases such that $A^{(r)}_{i,k^*} = \infty$ for all $r \in (c+2)\mu^\rho$. In the third preprocessing step, we take care of those remaining cases. Especially, Lemma 25 will be used in this preprocessing step to help narrow down the searching scope for k^*. We will see that the number of triples (i,k,j) such that $A^{(r)}_{i,k} = \infty$ for all $r \in (c+2)\mu^\rho$ and $\tilde{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)(Wn^{t_1-t_2}+3)}{W}$ is bounded. By using this property, we can bound the preprocessing time in this step.

The three-step solution is originally from Gu et al. [39] for computing the min-plus product of matrices A and B, in which matrix B is monotone. For simplicity of

presentation, Gu et al. first showed a randomized algorithm and then derandomized it. For the same reason, we first present the adjusted version of their randomized algorithm and defer the derandomization method to the end of this section. In the end, the running time of the adjusted solution is deterministic as well.

Before, we have proved that the total difference of matrix M^T is bounded by $O(n)$, and we made effort to construct a new matrix $\tilde{B}$ with smaller total difference. We will see that matrix $\tilde{B}$ plays a crucial role in the first and the third preprocessing steps, as well as the derandomization step. And due to its small total difference, we can design efficient algorithms in those steps.

Preprocessing Step 1. In the first preprocessing step, the goal is to compute matrix $\tilde{C'}$. We introduce a fast algorithm to achieve that.

Lemma 26 *Computing $\tilde{C'}$ takes $O((n^{1-t_2+t_1} + n^{2-2t_2} + n^{(2-t_2)}/W) \lg n)$ time.*

Proof. A similar proof has been given in [39, Proof of Lemma 17]. Henceforth, given any two entry-pairs (f_1, k_1) and (f_2, k_2), we regard that pair (f_1, k_1) is no more than pair (f_2, k_2) iff $f_1 \leq f_2$.

Fix $i \in [\mu]$; we maintain a red-black tree, T_i, as follows: i) Add entry-pair $(\tilde{A}_{i,k} + \tilde{B}_{k,0}, k)$ into T_i for each $k \in [\kappa]$; ii) then find the minimum entry-pair, (f_m, k_m), in T_i, and set $\tilde{C'}_{i,0}$ to be f_m; iii) iterate through $j \in [1, \mu)$, and each time $\tilde{B}_{k,j-1} \neq \tilde{B}_{k,j}$, replace $(\tilde{A}_{i,k} + \tilde{B}_{k,j-1}, k)$ with $(\tilde{A}_{i,k} + \tilde{B}_{k,j}, k)$ in T_i, and each time j increases, find the new minimum entry-pair in T_i, and update entry $\tilde{C'}_{i,j}$ as in step (ii).

For each $i \in [\mu]$, we make $O(\kappa + n/W)$ updates on T_i and perform $O(\mu)$ queries for minimum entry-pairs, given that the total difference of $\tilde{B}$ is bounded by $O(n/W)$. A red-black tree supports each update and minimum query in $O(\lg n)$ time. The overall computation time is bounded by $O(\mu(\kappa + n/W + \mu) \lg n) = O((n^{1-t_2+t_1} + n^{(2-t_2)}/W + n^{2-2t_2}) \lg n)$. Storing entries of $\tilde{C'}$ uses $O(n^{2-2t_2})$ words. ∎

Preprocessing Step 2. In this step, we take matrices M, M^T and $\tilde{C'}$ as input and partially solve Problem 1 under certain conditions.

Let $\rho \geq 0$ be a parameter to be chosen later; let c be any constant that is no less than 1. For each $r \in [(c+2)\mu^\rho \ln n]$, we sample $j^{(r)}$ uniformly at random from $[\mu]$

and construct matrices $A^{(r)}$ and $B^{(r)}$ as follows:

$$A_{i,k}^{(r)} = \begin{cases} M_{i,k} + M^T_{k,j^{(r)}} - W\tilde{C'}_{i,j^{(r)}} & \textbf{if } 0 \le M_{i,k} + M^T_{k,j^{(r)}} - W\tilde{C'}_{i,j^{(r)}} \\ & \qquad \le 2(W-1)(3+Wn^{(t_1-t_2)}) \\ & \textbf{and } A_{i,k}^{(r')} = \infty \ \forall r' < r, \\ \infty & \textbf{otherwise,} \end{cases} \tag{4.2}$$

$$B_{k,j}^{(r)} = M^T_{k,j} - M^T_{k,j^{(r)}}.$$

Following [39], for a pair (i,k), if $A_{i,k}^{(\hat{r})} \neq \infty$ for some $\hat{r} \in [(c+2)\mu^\rho \ln n]$, we call (i,k) *covered*; otherwise it is *uncovered*. Similarly, we call a triple (i,k,j) covered iff (i,k) is covered. Let $C^{(r)} = A^{(r)} \star B^{(r)}$, for each $r \in [(c+2)\mu^\rho \ln n]$. Next we prove Lemma 27, which will be used later for finding k^*, when (i,k^*) is covered.

Lemma 27 *If (i,k^*) is covered, then*

$$M_{i,k^*} + M^T_{k^*,j} = \min_{r \in [(c+2)\mu^\rho \ln n]} \{C_{i,j}^{(r)} + W \cdot \tilde{C'}_{i,j^{(r)}}\}.$$

Proof. Given that (i,k^*) is covered, there exists an index $\hat{r} \in [(c+2)\mu^\rho \ln n]$ such that $A_{i,k^*}^{(\hat{r})}$ is set to be $M_{i,k^*} + M^T_{k^*,j^{(\hat{r})}} - W\tilde{C'}_{i,j^{(\hat{r})}}$ and $B_{k^*,j}^{(\hat{r})}$ is set to be $M^T_{k^*,j} - M^T_{k^*,j^{(\hat{r})}}$. Thereby,

$$\begin{aligned} A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})} + W\tilde{C'}_{i,j^{(\hat{r})}} &= M_{i,k^*} + M^T_{k^*,j^{(\hat{r})}} - W\tilde{C'}_{i,j^{(\hat{r})}} + M^T_{k^*,j} - M^T_{k^*,j^{(\hat{r})}} + W\tilde{C'}_{i,j^{(\hat{r})}} \\ &= M_{i,k^*} + M^T_{k^*,j}. \end{aligned}$$

We claim that $C_{i,j}^{(\hat{r})}$ equals to $A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})}$. Given that $C_{i,j}^{(\hat{r})} \le A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})}$, it is sufficient to prove that no k' exists such that $k' \neq k^*$ and $A_{i,k'}^{(\hat{r})} + B_{k',j}^{(\hat{r})} < A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})}$. We give the proof by contradiction. If k' exists, then $A_{i,k'}^{(\hat{r})}$ cannot be ∞. Following from Equation 4.2, we know that $A_{i,k'}^{(\hat{r})} + B_{k',j}^{(\hat{r})} + W\tilde{C'}_{i,j^{(\hat{r})}} = M_{i,k'} + M^T_{k',j} < A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})} + W\tilde{C'}_{i,j^{(\hat{r})}} = M_{i,k^*} + M^T_{k^*,j}$. This conflicts with the claim that $M_{i,k^*} + M^T_{k^*,j} = \min_k\{M_{i,k} + M^T_{k,j}\}$. Therefore, k' does not exist, so $C_{i,j}^{(\hat{r})} = A_{i,k^*}^{(\hat{r})} + B_{k^*,j}^{(\hat{r})}$ and $C_{i,j}^{(\hat{r})} + W\tilde{C'}_{i,j^{(\hat{r})}} = M_{i,k^*} + M^T_{k^*,j}$.

Next, we claim that $C_{i,j}^{(r)} + W \cdot \tilde{C'}_{i,j^{(r)}}$ is no less than $M_{i,k^*} + M^T_{k^*,j}$ for all $r \in [(c+2)\mu^\rho \ln n]$. Again, we give the proof by contradiction. Assume that there is an index r'' such that $C_{i,j}^{(r'')} + W \cdot \tilde{C'}_{i,j^{(r'')}} < M_{i,k^*} + M^T_{k^*,j}$. Recall that $C_{i,j}^{(r'')} = \min_k\{A_{i,k}^{(r'')} + B_{k,j}^{(r'')}\}$. Let k'' denote any index such that $A_{i,k''}^{(r'')} + B_{k'',j}^{(r'')} = C_{i,j}^{(r'')}$. Obviously, $A_{i,k''}^{(r'')} \neq \infty$, so

$A^{(r'')}_{i,k''} = M_{i,k''} + M^T_{k'',j^{(r'')}} - W\tilde{C'}_{i,j^{(r'')}}$ and $B^{(r'')}_{k'',j} = M^T_{k'',j} - M^T_{k'',j^{(r'')}}$. As a result, we would find that $C^{(r'')}_{i,j} + W \cdot \tilde{C'}_{i,j^{(r'')}} = A^{(r'')}_{i,k''} + B^{(r'')}_{k'',j} + W \cdot \tilde{C'}_{i,j^{(r'')}} = M_{i,k''} + M^T_{k'',j} < M_{i,k^*} + M^T_{k^*,j}$, which is impossible.

Combining both claims, we prove that if (i, k^*) is covered, then $M_{i,k^*} + M^T_{k^*,j} = \min_{r\in[(c+2)\mu^\rho \ln n]}\{C^{(r)}_{i,j} + W \cdot \tilde{C'}_{i,j^{(r)}}\}$. ■

In the second preprocessing step, we build the data structures over all pairs of matrices A^r and B^r, such that if (i, k^*) happened to be covered, the answer k^* would be found efficiently. Otherwise, we defer the solution to finding k^* to the third preprocessing step. Given that all non-infinity entries of A^r are bounded by $2(W-1)(3+Wn^{(t_1-t_2)})$, we can apply Lemma 15 so that all data structures can be constructed efficiently. The result of the second preprocessing step can be summarized as follows:

Lemma 28 *Given $(c+2)\mu^\rho \ln n$ pairs of matrices $A^{(r)}$ and $B^{(r)}$, a data structure of*

$$\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)} + n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)})$$

words can be built in time

$$\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)\cdot(\rho+\omega(1,t_1/(1-t_2),1)+\frac{t_1}{1-t_2}-\sigma)})$$

to partially solve the min-plus-query-witness query problem upon matrices M and M^T. More precisely, given a query (i, j), if (i, k^*) has been covered, then k^* can be found in $\tilde{O}(n^{(1-t_2)\cdot(\rho+\sigma)})$ time, where ρ denotes any constant such that $\rho \geq 0$ and σ denotes any constant such that $0 \leq \sigma \leq \frac{t_1}{1-t_2}$.

Proof. The data structure part is straightforward. Since the non-infinity entries in $A^{(r)}$'s are bounded, we can apply Lemma 15 to solve Problem 1 with each pair of matrices $A^{(r)}$ and $B^{(r)}$ as the input.

We analyze the space cost and the preprocessing time of the data structure. We substitute M, β, and m in Lemma 15 for the values as follows: $M = 2(W-1)\cdot(3 + W \cdot n^{(t_1-t_2)})$, $m = n^{1-t_2}$, and $\beta = t_1/(1-t_2)$. Then it follows that the space cost for a single pair of matrices $A^{(r)}$ and $B^{(r)}$ is bounded by $\tilde{O}(S^{(r)}(M, m, \beta, \sigma))$ words of

space, where

$$S^{(r)}(M,m,\beta,\sigma) = Mm^{2+\beta-\sigma} + m^{1+2\beta-\sigma}$$
$$= 2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\sigma)} + n^{(1-t_2)(1+\frac{2t_1}{1-t_2}-\sigma)}.$$

Similarly, the time spent on building the data structure for $A^{(r)}$ and $B^{(r)}$ is bounded by $\tilde{O}(P^{(r)}(M,m,\beta,\sigma))$, where

$$P^{(r)}(M,m,\beta,\sigma) = Mm^{\omega(1,\beta,1)+\beta-\sigma}$$
$$= 2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)(\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)}.$$

In the second preprocessing step, we construct $(c+2)\mu^\rho \ln n$ pairs of matrices $A^{(r)}$ and $B^{(r)}$. Recall that μ is bounded by $O(n^{1-t_2})$. The overall space cost of the data structure in words is

$$\tilde{O}(\sum_r S^{(r)}(M,m,\beta,\sigma) = \tilde{O}(n^{(1-t_2)\rho} \cdot S^{(r)}(M,m,\beta,\sigma)))$$
$$= \tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)} + n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)}).$$

And the overall preprocessing time for all $(c+2)\mu^\rho \ln n$ pairs of matrices is

$$\tilde{O}(\sum_r P^{(r)}(M,m,\beta,\sigma) = \tilde{O}(n^{(1-t_2)\rho} \cdot P^{(r)}(M,m,\beta,\sigma)))$$
$$= \tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)}) \cdot n^{(1-t_2)\cdot(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)}).$$

Note that the factor $(c+2)\ln n$ is hidden in the notation $\tilde{O}$.

Next, we give the query algorithm that finds k^*. For each $r \in [(c+2)\mu^\rho \ln n]$, we query over the data structure built upon matrices $A^{(r)}$ and $B^{(r)}$ with (i,j) as the query parameters. Let $k^{(r)}$ denote the index returned for each r. Then, we compute the set $\{(A^{(r)}_{i,k^{(r)}} + B^{(r)}_{k^{(r)},j} + W \cdot \tilde{C'}_{i,j^{(r)}}, k^{(r)}),\ \text{for all } r \in [(c+2)\mu^\rho \ln n]\}$, find the minimum entry-pair from the set, and return the index-entry as the answer. The correctness of the algorithm is implied by Lemma 27.

Finally, we give the query time analysis. Substitute m in Lemma 15 for n^{1-t_2}. Then finding $k^{(r)}$ for each pair of $A^{(r)}$ and $B^{(r)}$ takes $\tilde{O}(n^{(1-t_2)\sigma})$ time. For all $r \in [(c+2)\mu^\rho \ln n]$, the overall query time is bounded by $\tilde{O}(n^{(1-t_2)\cdot(\rho+\sigma)})$ time. ∎

Remarks after the Second Preprocessing Step. Before concluding the second preprocessing step, we bound the number of triples that are almost relevant and uncovered. Lemma 24 tells that the number of triples that are almost relevant is no more than the the number of triples that are weakly relevant. Now it is sufficient to give the upper bound of the number of triples that are weakly relevant and uncovered.

Lemma 29 *([69, Lemma 4.3],[39, Lemma 18]) With probability at least* $1 - n^{-c}$ *for any constant* $c \geq 1$*, the number of uncovered and weakly relevant triples* (i, k, j) *is bounded by* $O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)})$*; therefore, the number of triples that are almost relevant and uncovered is bounded by* $O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)})$.

Proof. We call a pair (i, k) *potentially covered* if the number of triples (i, k, j) that are weakly relevant is at least $\mu^{1-\rho}$. Note that as shown in the proof of Lemma 24, $M_{i,k} + M^T_{k,j} - W \cdot \tilde{C'}_{i,j}$ is always no less than 0. To confirm whether a triple (i, k, j) is weakly relevant or not, we can simply check whether $M_{i,k} + M^T_{k,j} - W \cdot \tilde{C'}_{i,j} \leq 2(W-1) \cdot (3 + W \cdot n^{(t_1-t_2)})$.

Given a potentially covered pair (i, k), after $(c+2)\mu^\rho \ln n$ rounds, we will sample a $j^{(r)}$, such that $(i, k, j^{(r)})$ is weakly relevant with probability at least

$$\begin{aligned}
1 - (1 - \frac{\mu^{1-\rho}}{\mu})^{(c+2)\mu^\rho \ln n} &= 1 - (1 - \frac{1}{\mu^\rho})^{(c+2)\mu^\rho \ln n} \\
&\geq 1 - e^{-\frac{1}{\mu^\rho}(c+2)\mu^\rho \ln n} \quad \text{(Note that } (1 - \frac{1}{\mu^\rho}) \leq e^{-\frac{1}{\mu^\rho}}\text{).} \\
&= 1 - e^{-\ln n^{c+2}} \\
&= 1 - \frac{1}{n^{c+2}}.
\end{aligned}$$

As a result, (i, k, j) will be covered for all j. There are no more than $n^{t_1} \cdot \mu$ pairs of (i, k). All potentially covered pairs (i, k) will be covered with probability at least $(1 - n^{-(c+2)})^{n^{t_1}\cdot\mu} \geq 1 - n^{-c}$, as $n^{t_1} \cdot \mu \leq n^2$.

On the other hand, if a pair (i, k) is not potentially covered, then the number of those triples (i, k, j) that are weakly relevant is less than $\mu^{1-\rho}$; we cannot guarantee that those triples (i, k, j) are covered with high probability. Given that there are only $n^{t_1} \cdot \mu$ pairs of (i, k) in total, the number of the triples (i, k, j) that are weakly relevant and are contributed by all the non-potentially covered pairs (i, k) is less than $\mu^{1-\rho} \cdot n^{t_1} \cdot \mu = O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)})$; therefore, with probability no less than

$1 - n^{-c}$, the number of triples that are weakly relevant and uncovered is bounded by $O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)})$. ■

As shown in [39, Section 3.1, Full Version], the randomized part in this preprocessing step can be derandomized. We defer the details to the end of this section.

Preprocessing Step 3. For each pair (i, j), let $(F_{i,j}, V_{i,j})$ denote the minimum entry-pair in set $\{(M_{i,k} + M^T_{k,j}, k) : (i, k, j)$ is an uncovered and almost relevant$\}$, keyed by the first item of each entry-pair. In this preprocessing step, we enumerate the triples that are uncovered and almost relevant and compute $(F_{i,j}, V_{i,j})$ for all pairs of (i, j).

Lemma 30 *In $O((n^{1-t_2+t_1} + n^{(2-t_2)}/W) \lg n + n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)} + n^{2-2t_2})$ time, one can find all almost relevant and uncovered triples and compute $(F_{i,j}, V_{i,j})$ for all μ^2 pairs of (i, j), where $i, j \in [\mu]$*

Proof. In the third preprocessing step, we only need to consider the uncovered triples. First, we filter out the triples that have been covered in the second preprocessing step by constructing a matrix $\check{A}$, such that $\check{A}_{i,k}$ is set to ∞ if (i, k) is covered; otherwise, $\check{A}_{i,k}$ is set to $\tilde{A}_{i,k}$.

Next, we present in the following paragraph an algorithm that constructs a linked list, $Y_{i,j}$, for each pair of i and j, where $i, j \in [\mu]$. In $Y_{i,j}$, we store the entry-pairs $(M_{i,k} + M^T_{k,j}, k)$ such that $0 \le \check{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \le \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$. As a result, $Y_{i,j}$ stores a multiset, and by iterating through entry-pairs in $Y_{i,j}$, we can find the the minimum pair-entry $(F_{i,j}, V_{i,j})$.

Fix $i \in [\mu]$. When j equals to 0, the algorithm can be described as follows: i) Create a red-black tree, T_i, and add entry-pair $(\check{A}_{i,k} + \tilde{B}_{k,0}, k)$ into T_i for each $k \in [\kappa]$; ii) construct a linked list $Y_{i,0}$; iii) and list, in increasing order, the elements that are being stored in T_i, meanwhile appending entry-pair $(M_{i,k} + M^T_{k,0}, k)$ into $Y_{i,0}$ as long as $0 \le \check{A}_{i,k} + \tilde{B}_{k,0} - \tilde{C'}_{i,0} \le \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$, and stop immediately after an entry-pair $(\check{A}_{i,k} + \tilde{B}_{k,0}, k)$ is spotted, such that $\check{A}_{i,k} + \tilde{B}_{k,0} - \tilde{C'}_{i,0} > \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$. Then we iterate through $j \in [1, \mu)$. Each time j increases, we proceed as follows: i) Replace $(\check{A}_{i,k} + \tilde{B}_{k,j-1}, k)$ with $(\check{A}_{i,k} + \tilde{B}_{k,j}, k)$ in T_i, as long as $\tilde{B}_{k,j-1} \ne \tilde{B}_{k,j}$; ii) construct an empty linked list $Y_{i,j}$; iii) list, in increasing order, the elements that are being

stored in the T_i, meanwhile appending entry-pair $(M_{i,k} + M^T_{k,j}, k)$ into $Y_{i,j}$ as long as $0 \leq \check{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$, and stop immediately after an entry-pair $(\check{A}_{i,k} + \tilde{B}_{k,j}, k)$ is spotted, such that $\check{A}_{i,k} + \tilde{B}_{k,j} - \tilde{C'}_{i,j} > \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$.

After performing the algorithm described above for each $i \in [\mu]$, we know that all $Y_{i,j}$'s store at most $O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)})$ entry-pairs, following from Lemma 29. For each pair (i, j), we iterate through elements in $Y_{i,j}$ to find the the minimum pair-entry $(F_{i,j}, V_{i,j})$.

We conclude the proof by giving the analysis of the running time and the space cost. For each $i \in [\mu]$, we make $O(\kappa + n/W)$ updates on T_i, and each update requires $O(\lg n)$ time. Computing all pair-entries, $(F_{i,j}, V_{i,j})$, need $O(n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)} + n^{2-2t_2})$ time. The overall computation time is bounded by $O(\mu(\kappa + n/W)\lg n + n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)} + n^{2-2t_2}) = O((n^{1-t_2+t_1} + n^{(2-t_2)}/W)\lg n + n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)} + n^{2-2t_2})$. Storing all $F_{i,j}$ and $V_{i,j}$ uses $O(n^{2-2t_2})$ words of space. ∎

The Querying Procedure. Let (i, j) be the query parameters. If (i, k^*) is covered, then we use Lemma 28 to find k^* in $\tilde{O}(n^{(1-t_2)(\rho+\sigma)})$ time. Otherwise, since $0 \leq \tilde{A}_{i,k^*} + \tilde{B}_{k^*,j} - \tilde{C'}_{i,j} \leq \frac{(W-1)(Wn^{t_1-t_2}+3)}{W}$, following Lemma 25, k^* must be stored in $V_{i,j}$, which can be found in constant time.

As a result, we have solved the min-plus-query-witness problem over M and M^T and achieve Lemma 31. Recall that parameter W was set to be $\lfloor n^{(1-t_2)\theta} \rfloor$.

Lemma 31 *A $\tilde{O}(n^{2-2t_2} + n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+2+\frac{t_1}{1-t_2}-\sigma)} + n^{(1-t_2)(\rho+1+\frac{2t_1}{1-t_2}-\sigma)})$-word data structure can be constructed upon matrices M and M^T in $\tilde{O}(n^{1-t_2+t_1} + n^{2-2t_2} + n^{1+(1-t_2)(1-\theta)} + n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)} + n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)})$ time, such that a query defined in Problem 1 can be answered in $\tilde{O}(n^{(1-t_2)(\sigma+\rho)})$ time, where ρ, σ, and θ denote any constants such that $\rho \geq 0$, $0 \leq \sigma \leq \frac{t_1}{1-t_2}$, and $0 \leq \theta \leq 1$.*

Proof. In this proof, we analyze the overall space cost and the construction time of the data structure. The preprocessing contains three steps.

In the first preprocessing step, we construct matrices $\tilde{C}'$ taking $O((n^{1-t_2+t_1} + n^{2-2t_2} + n^{(2-t_2)}/W)\lg n)$ time. Storing entries of $\tilde{C}'$ occupies $O(n^{2-2t_2})$ words of space. In the second preprocessing steps, we construct $(c+2)\mu^\rho \ln n$ pairs of matrices $A^{(r)}$ and $B^{(r)}$ and build data structure applying Lemma 15 upon each pair of $A^{(r)}$ and

$B^{(r)}$. As shown in Lemma 28, the overall space cost of the data structures built in this step is $\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)})\cdot n^{(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)}+n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)})$ words, and the preprocessing time is $\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)})\cdot n^{(1-t_2)\cdot(\rho+\omega(1,t_1/(1-t_2),1)+\frac{t_1}{1-t_2}-\sigma)})$. The third preprocessing step is to construct matrix $V_{i,j}$. Lemma 30 shows that this step takes $O((n^{1-t_2+t_1}+n^{(2-t_2)}/W)\lg n+n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)}+n^{2-2t_2})$ time.

Substitute the parameter W for $\lfloor n^{(1-t_2)\theta}\rfloor$. Summing up the space cost required in each step, the total space cost is

$$
\begin{aligned}
&O(n^{2-2t_2})+\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)})\cdot n^{(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)}+n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)})\\
&=O(n^{2-2t_2})+\tilde{O}(n^{2\theta(1-t_2)}\cdot n^{(t_1-t_2)}\cdot n^{(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)}+n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)})\\
&=\tilde{O}(n^{2-2t_2}+n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)\cdot(\rho+2+\frac{t_1}{1-t_2}-\sigma)}+n^{(1-t_2)\cdot(\rho+1+\frac{2t_1}{1-t_2}-\sigma)}).
\end{aligned}
$$

And the overall preprocessing time is

$$
\begin{aligned}
&O((n^{1-t_2+t_1}+n^{2-2t_2}+n^{(2-t_2)}/W)\lg n)\\
&+\tilde{O}(2(W-1)(3+Wn^{(t_1-t_2)})\cdot n^{(1-t_2)\cdot(\rho+\omega(1,t_1/(1-t_2),1)+\frac{t_1}{1-t_2}-\sigma)})\\
&+O((n^{1-t_2+t_1}+n^{(2-t_2)}/W)\lg n+n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)}+n^{2-2t_2})\\
&=\tilde{O}(n^{1-t_2+t_1}+n^{2-2t_2}+n^{(2-t_2)-\theta(1-t_2)})\\
&+\tilde{O}(n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)})\\
&+\tilde{O}(n^{1-t_2+t_1}+n^{2-t_2-\theta(1-t_2)}+n^{(1-t_2)\cdot(2+\frac{t_1}{1-t_2}-\rho)}+n^{2-2t_2})\\
&=\tilde{O}(n^{1-t_2+t_1}+n^{2-2t_2}+n^{1+(1-t_2)(1-\theta)}+n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)}\\
&+n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)}).
\end{aligned}
$$

The query time analysis has been given before. ∎

We can apply Lemma 31 to find the most frequent element and its frequency in the multiset of frequent colors assigned to the nodes in $\mathrm{P}'_{x',\perp}\cup\mathrm{P}'_{y',\perp}$ provided that $\mathrm{P}_{x',y'}$ is a predefined path.

Derandomization. In the preprocessing step 2, we construct $(c+2)\mu^\rho\ln n$ pairs of matrices $A^{(r)}$ and $B^{(r)}$, where $\mu=O(n^{1-t_2})$. Recall that we call a pair (i,k) covered if there is some $\hat{r}\in[(c+2)\mu^\rho\ln n]$ such that $A^{(\hat{r})}_{i,k}\neq\infty$. In Lemma 29, we have proved that after constructing $(c+2)\mu^\rho\ln n$ pairs of matrices $A^{(r)}$ and $B^{(r)}$, the number of

almost relevant and uncovered triples (i,k,j) is bounded by $O(n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)})$ with high probability.

In the derandomized algorithm, we will construct at most $n^{(1-t_2)\rho}$ pairs (less than the number of pairs required by the randomized algorithm) of matrices $A^{(r)}$ and $B^{(r)}$, where $A^{(r)}$ and $B^{(r)}$ are defined in the same way as Equation 4.2, while $j^{(r)}$ will be chosen deterministically instead. Recall that a pair (i,k) is covered if there exists a $\hat{r}$ such that $A^{(\hat{r})}_{i,k} \neq \infty$. Each matrix $A^{(r)}$ to be constructed covers at least $n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)}$ pairs of (i,k), so that there are at least $n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)}$ non-infinity entries in $A^{(r)}$. Furthermore, different matrices $A^{(r)}$ cover different pairs of (i,k).

The algorithm contains μ iterations for each j from 0 to $\mu-1$. In each iteration j, we maintain a red-black tree, T_i, for each $i \in [\mu]$. Fix $i \in [\mu]$. If (i,k) has not been covered yet, where $k \in [\kappa]$, then we add entry-pair $(\tilde{A}_{i,k}+\tilde{B}_{k,j},k)$ into T_i. Then, for each $i \in [\mu]$, we use T_i to compute the number of triples (i,k,j) such that $\tilde{C'}_{i,j} \leq \tilde{A}_{i,k}+\tilde{B}_{k,j} \leq \tilde{C'}_{i,j}+\frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$. That is the number of almost relevant and uncovered triples (i,k,j). If the sum of these numbers is no more than $n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)}$, we increment j by 1 and continue the next iteration, and in the next iteration, we only update entries of T_i when (i,k) has not been covered yet and $\tilde{B}_{k,j} \neq \tilde{B}_{k,j-1}$. Otherwise, we remove all entry-pairs $(\tilde{A}_{i,k}+\tilde{B}_{k,j},k)$ such that $\tilde{C'}_{i,j} \leq \tilde{A}_{i,k}+\tilde{B}_{k,j} \leq \tilde{C'}_{i,j}+\frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$ from T_i for each $i \in [\mu]$ and set their corresponding (i,k) to be covered. Then we set $j^{(r)}$ to be j, construct matrices $A^{(r)}$ and $B^{(r)}$ following from Equation 4.2, and continue the next iteration.

For each $j \in [\mu]$, each iteration j contributes at most $n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)}$ triples (i,k,j) that are uncovered and almost relevant. After performing the algorithm, the total number of uncovered and almost relevant triples is at most $\mu \cdot n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)} = n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)}$. Each time we construct matrices $A^{(r)}$ and $B^{(r)}$, we made at least $n^{(1-t_2)(1+\frac{t_1}{1-t_2}-\rho)}$ different pairs of (i,k) covered. Since there are only $n^{1-t_2+t_1}$ pairs of (i,k) in total, at most $n^{(1-t_2)\rho}$ pairs of matrices $A^{(r)}$ and $B^{(r)}$ are constructed. Especially, if triple $(i,k,j^{(r)})$ is set to be covered and $(i,k,j^{(r)})$ happens to be almost relevant, then $A^{(r)}_{i,k} \neq \infty$, following from Equation 4.2 and Lemma 24.

Finally, we give the running time analysis. Recall that the statement that $\tilde{C'}_{i,j} \leq \tilde{A}_{i,k}+\tilde{B}_{k,j}$ is always true. For each $i \in [\mu]$ and each $j \in [u]$, we simply perform one predecessor query over T_i to find the number of triples (i,k,j) such that $\tilde{A}_{i,k}+\tilde{B}_{k,j} \leq$

$\tilde{C'}_{i,j} + \frac{(W-1)\cdot(3+W\cdot n^{(t_1-t_2)})}{W}$, for a total of n^{2-2t_2} queries. Given that the total difference of matrix $\tilde{B}$ is bounded by $O(n/W)$, the total number of updates on all T_i's is bounded by $O(n^{1-t_2}\cdot(n/W+n^{t_1})) = O(n^{1-t_2+t_1}+n^{1+(1-t_2)(1-\theta)})$. Therefore, the overall running time of this algorithm is bounded by $\tilde{O}(n^{1-t_2+t_1} + n^{1+(1-t_2)(1-\theta)} + n^{2-2t_2})$.

4.4.4 Mode Queries on an Arbitrary Path

In the previous sections, we have seen a data structure that finds an infrequent color that appears at least as frequently as other infrequent colors that are assigned to nodes in an arbitrary path. We also described a data structure that finds a frequent color that appears at least as frequently as other frequent colors assigned to the nodes in $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ for any predefined path $\mathrm{P}_{x',y'}$. In this section, we present a solution that finds a mode for an arbitrary path by combining both data structures mentioned earlier.

We first represent tree T using the data structure of Lemma 2. As discussed in Section 4.2, this structure supports finding the number of appearances of a color on a path in $O(\lg\lg C)$ time. Then we mark $O(n^{1-t_2})$ nodes of T as discussed in Section 4.4.2. We compute the number of appearances of each color on T to determine whether it is frequent or infrequent. Then we construct the data structures of Lemmas 20 and 31 for queries over infrequent and frequent colors, respectively.

Let $\mathrm{P}_{x,y}$ be a query path that contains the root $\perp$. We consider two different cases: Either $\mathrm{P}_{x,y}$ contains only one marked node, i.e., $\perp$, or $\mathrm{P}_{x,y}$ contains more than one marked nodes, including $\perp$.

In the first case, it follows that $|\,\mathrm{P}_{x,y}\,| = O(n^{t_2})$, and a mode in $\mathrm{P}_{x,y}$ can be found in $O(n^{t_2})$ time. The query algorithm requires two variables: a potential mode, and its frequency, as well as an array $V[0..C-1]$. Initially, entries in V, as well as the frequency of the potential mode, are all set to be 0. Note that array V can be constructed once but shared by all queries. A query scans $\mathrm{P}_{x,y}$ twice. In the first pass, each time a node v is visited, we increment entry $V[c(v)]$ by 1, where $c(v)$ denotes the color of node v. If $V[c(v)]$ after being incremented happens to be more than the frequency of the potential mode, we set the potential mode to be $c(v)$ and update its frequency by $V[c(v)]$. Obviously, after the first pass, the potential mode becomes a mode in $\mathrm{P}_{x,y}$. Finally, we scan the nodes in $\mathrm{P}_{x,y}$ the second time to reset all the

non-zero entries in V back to 0 for future queries. Given that the number of nodes in $\mathrm{P}_{x,y}$ is bounded by $O(n^{t_2})$, so is the query time.

It remains to handle the second case. Let x' and y' denote the lowest marked ancestors of x and y, respectively. Given that the root node is always marked in the preprocessing stage, neither x' nor y' is a null. We divide $\mathrm{P}_{x,y}$ into four disjoint parts: $\mathrm{P}'_{x,x'}$, $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$, $\perp$ and $\mathrm{P}'_{y,y'}$. Note that any of $\mathrm{P}'_{x,x'}$, $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ and $\mathrm{P}'_{y,y'}$ might be empty. Especially, there are $O(n^{t_2})$ nodes in $\mathrm{P}'_{x,x'} \cup \mathrm{P}'_{y,y'} \cup \perp$; for each distinct color that appears in $\mathrm{P}'_{x,x'} \cup \mathrm{P}'_{y,y'} \cup \perp$, counting its occurrences in $\mathrm{P}_{x,y}$ takes $O(\lg \lg C)$ time by the data structure of Lemma 3, for a total of $O(n^{t_2} \lg \lg C)$ time. Let c_1 be the color with maximum number of occurrences found this way. Then we apply Lemma 20 to find the infrequent color, c_2, with maximum frequency in $\mathrm{P}_{x',y'}$ in $\tilde{O}(n^{1-t_1})$ time, and we query over the data structure of Lemma 31 to find the frequent color, c_3, with maximum frequency in $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ in $\tilde{O}(n^{(1-t_2)(\sigma+\rho)})$ time. We also obtain the frequency of c_2 in $P_{x',y'}$ and the frequency of c_3 in $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ when finding c_2 and c_3. Note that, if the mode is not c_1, then the mode does not appear in $\mathrm{P}'_{x,x'} \cup \mathrm{P}'_{y,y'} \cup \perp$, so it must be either c_2 or c_3. Hence it suffices to compare the frequency of c_1 in $\mathrm{P}_{x,y}$, the frequency of c_2 in $P_{x',y'}$, and the frequency of c_3 in $\mathrm{P}'_{x',\perp} \cup \mathrm{P}'_{y',\perp}$ to find the answer to the query.

Finally, we apply the technique in Section 4.3.3 to compute the mode in an arbitrary path:

Lemma 32 *Let T be a colored ordinal tree on n nodes with each node assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$. A data structure of $\tilde{O}(n^{2-t_1} + n^{2-2t_2} + n^{(1-t_2)(\rho+1+\frac{2t_1}{1-t_2}-\sigma)} + n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+2+\frac{t_1}{1-t_2}-\sigma)})$ words can be constructed in $\tilde{O}(n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)} + n^{1+(1-t_2)(1-\theta)} + n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)} + n^{2-t_1} + n^{2-2t_2} + n^{1-t_2+t_1})$ time such that a path mode query can be answered in $\tilde{O}(n^{t_2} + n^{1-t_1} + n^{(1-t_2)(\sigma+\rho)})$ time, where ρ, σ and θ denote any constants such that $\rho \geq 0$, $0 \leq \sigma \leq \frac{t_1}{1-t_2}$ and $0 \leq \theta \leq 1$.*

Proof. It remains to present the analysis for the space cost and the construction time. As shown in Lemma 20, the data structure for infrequent colors occupies $O(n^{2-t_1} \lg^4 n)$ words of space and can be built in $O(n^{2-t_1} \lg^5 n)$ time. Lemma 31 shows that the data structure for frequent colors requires $\tilde{O}(n^{(1-t_2)(\rho+1+\frac{2t_1}{1-t_2}-\sigma)} +$

$n^{2-2t_2} + n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+2+\frac{t_1}{1-t_2}-\sigma)})$ words of space and $\tilde{O}(n^{1-t_2+t_1} + n^{2-2t_2} + n^{1+(1-t_2)(1-\theta)} + n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)} + n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)})$ time to construct. By combining data structures for infrequent colors and frequent colors, we achieve the space cost and construction time as shown in Lemma 32. Note that to support queries over arbitrary path, we apply the centroid decomposition shown in Section 4.3.3. As a result, both the space cost and the preprocessing time will be increased by an $O(\lg n)$ factor, while this extra $\lg n$ factor is hidden in $\tilde{O}$. ■

Lemma 32 shows a data structure solution, which means that being aware of queries in the preprocessing stage is unnecessary. On the other hand, given the fast preprocessing and query time of this data structure, it can be naturally transformed into an efficient offline algorithm for batched path mode queries: We simply build the data structure following Lemma 32 and then use it to answer n queries. Setting the proper values for parameters in Lemma 32 yields Theorem 5.

Theorem 5 *A batch of n path mode queries over a colored tree T on n nodes can be answered in $\tilde{O}(n^{\frac{24+2\omega}{17+\omega}}) = O(n^{1.483814})$ time.*

Proof. The running time of batched path mode queries includes two parts: the overall query time of n queries and its preprocessing time. The prior one is bounded by $\tilde{O}(n \cdot (n^{t_2} + n^{1-t_1} + n^{(1-t_2)(\sigma+\rho)})) = \tilde{O}(n^{1+t_2} + n^{2-t_1} + n^{1+(1-t_2)(\sigma+\rho)})$, while the latter one is given in Lemma 32. By setting the overall query time to be equal to the preprocessing time, we obtain proper values for parameters, including t_1, t_2, ρ, σ, and θ, thereby achieving the optimum running time, i.e., $O(n^{1.483814})$, as promised. To do that, we first express t_2, ρ, σ and θ as functions of t_1 and decide the value of t_1 in the end.

By setting the term n^{1+t_2} from the total query time to be equal to the term n^{2-t_1} from the preprocessing time, we have $t_2 = 1 - t_1$. By comparing the term n^{2-t_1} from the total query time and the term $n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)}$ from the preprocessing time, we

find

$$\begin{aligned} n^{2-t_1} &= n^{(1-t_2)(2+\frac{t_1}{1-t_2}-\rho)} \\ 2-t_1 &= t_1(3-\rho) \\ \rho &= \frac{4t_1-2}{t_1}. \end{aligned}$$

By comparing the term $n^{1+(1-t_2)(\sigma+\rho)}$ from the total query time and the term n^{2-t_1} from the preprocessing time, we find

$$\begin{aligned} n^{1+(1-t_2)(\sigma+\rho)} &= n^{2-t_1} \\ (1-t_2)(\sigma+\rho) &= 1-t_1 \\ \sigma &= \frac{1-t_1}{1-t_2} - \rho \\ &= \frac{1-t_1}{t_1} - \frac{4t_1-2}{t_1} \\ &= \frac{3-5t_1}{t_1}. \end{aligned}$$

By comparing the term $n^{1+(1-t_2)(\sigma+\rho)}$ from the total query time and the term $n^{1+(1-t_2)(1-\theta)}$ from the preprocessing time, we find

$$\begin{aligned} n^{1+(1-t_2)(\sigma+\rho)} &= n^{1+(1-t_2)(1-\theta)} \\ (1-t_2)(\sigma+\rho) &= (1-t_2)(1-\theta) \\ (\sigma+\rho) &= (1-\theta) \\ \theta &= 1-(\sigma+\rho) \\ &= 1-(\frac{3-5t_1}{t_1} + \frac{4t_1-2}{t_1}) \\ &= 1-\frac{1-t_1}{t_1} \\ &= \frac{2t_1-1}{t_1}. \end{aligned}$$

Finally, by comparing the term $n^{1+(1-t_1)}$ from the total query time and the term $n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)}$ from the preprocessing time, we find the

proper value of t_1:

$$
\begin{aligned}
n^{1+(1-t_1)} &= n^{2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma)} \\
2-t_1 &= 2\theta(1-t_2)+(t_1-t_2)+(1-t_2)(\rho+\omega(1,\frac{t_1}{1-t_2},1)+\frac{t_1}{1-t_2}-\sigma) \\
2-t_1 &= \frac{4t_1-2}{t_1}\cdot t_1+(2t_1-1)+t_1(\frac{4t_1-2}{t_1}+\omega(1,1,1)+1-\frac{3-5t_1}{t_1}) \\
2-t_1 &= 6t_1-3+4t_1-2+t_1\omega(1,1,1)+t_1-(3-5t_1) \\
t_1 &= \frac{10}{17+\omega(1,1,1)}.
\end{aligned}
$$

The values, chosen as shown above, for parameters t_1, t_2, ρ, σ and θ ensure that the preprocessing time is proportional to the time of answering n path mode queries, ignoring polylog(n) factors. To compute the time complexity of our solution to batched path mode queries, it suffices to give the bound of the total running time for n queries, which is

$$
\begin{aligned}
&\tilde{O}(n^{1+t_2}+n^{2-t_1}+n^{1+(1-t_2)(\sigma+\rho)}) \\
=&\tilde{O}(n^{1+1-t_1}+n^{2-t_1}+n^{1+t_1(\frac{3-5t_1}{t_1}+\frac{4t_1-2}{t_1})}) \\
=&\tilde{O}(n^{2-t_1})=\tilde{O}(n^{2-\frac{10}{17+\omega(1,1,1)}})=\tilde{O}(n^{\frac{24+2\omega}{17+\omega}}).
\end{aligned}
$$

Observe that the function, $\frac{24+2x}{17+x}$, is monotonically increasing for positive x. As $\omega(1,1,1) < 2.37286$, n path mode queries can be answered in overall $\tilde{O}(n^{1.48381395}) = O(n^{1.483814})$ time. ■

4.5 Conclusion

In this chapter, we saw efficient solutions to batched colored path counting queries and batched path mode queries. The strategy used in both our solutions can be summarized as follows: Consider a restricted case, in which all query paths contain a pre-selected node; then apply centroid decomposition, relaxing the restriction. This strategy induces a reduction from queries over arbitrary paths to queries over paths that contain a pre-selected node. As we have seen, the latter problem is much easier to cope with. We expect that this versatile strategy to find its application to other path queries as well.

In our solutions to batched path queries, we choose a set of nodes in the tree as the marked nodes. Combining with the restriction mentioned before, we precompute queries for the predefined paths, which contain the pre-selected node and have a pair of pre-selected nodes as their endpoints. In this preprocessing phase, we manage to reduce colored path counting to matrix multiplication and reduce path mode queries to the min-plus product. For the former reduction, we show a non-trivial transformation that turns bit matrices created directly using the pre-selected nodes of the input tree into sparse matrices, in which the number of non-zero bits are bounded. Then we apply the efficient algorithm for sparse matrix multiplication, improving the overall preprocessing time. As we have seen, the latter reduction is different from the reduction [69, 68, 39] from range mode queries to the min-plus product, in which matrices generated from arrays are monotone. To capture the special structure of matrices generated from trees, we define a new notion, total difference, and we prove that the total differences of the matrices for path mode queries are always bounded by the number of nodes in the tree. Then, to further reduce the total difference of the matrices from $O(n)$ to $O(n/W)$, we design a more involved technique, two-level marking scheme, due to which, we further improve the computation time of the min-plus product. Our algorithm can be seen as a generalization of the method by Gu et al. [39], given that a monotone matrix always has bounded total difference, while a matrix with bounded total difference does not have to be monotone.

After all, we manage to design solutions that support n queries for colored path counting and path mode in $O(n^{1.4071})$ and $O(n^{1.483814})$ time, respectively.

The problem, batched path mode queries, is a generalized version of the batched mode query problem on 1D arrays, i.e., batched range mode queries. Recently, a new improvement on the latter problem has been achieved: The algorithm that supports n range mode queries is as fast as $\tilde{O}(n^{\frac{3+2\omega}{3+\omega}}) = O(n^{1.4416})$ time [24]. There is an obvious gap in the running time between batched path mode queries and batched range mode queries. We expect to see a faster algorithm for batched path mode queries than ours.

Chapter 5

Approximate Colored Path Counting

5.1 Introduction

In Chapter 4, we studied colored path counting queries, in which the precise number of distinct colors in a query path is expected as an answer. We have seen a data structure that occupies $\Omega(n^{1.1860})$ words of space and supports each query in $O(n^{0.4071})$ time. Given the conditional lower bound of this problem, this solution is efficient. As a result of the rapid growth of large data sets, linear space data structures are favored by applications that deal with massive amounts of data. The super-linear space used by that data structure is not ideal for handling large data sets. The only known linear space solution to this problem requires $\Omega(\sqrt{n})$ time in the worst case [44]. For other path queries, such as path minimum [18, 2, 56, 22, 8, 12], path medium [58, 64, 48, 49], path counting [18, 58, 64, 48, 49] and path majority [23, 30], linear-space solutions with sublogarithmic or even constant query times exist, which are much faster.

To achieve faster queries, approximate colored path counting problems have been studied. Similar to Chapter 4, these problems are defined over an ordinal tree T on n nodes, each assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$. Two different ways of bounding approximate ratios have been considered [44]: a 2-*approximate colored path counting query* computes a number in $[\mathrm{occ}, 2 \cdot \mathrm{occ}]$, while a $(1 \pm \epsilon)$-*approximate* query returns a number in $[(1-\epsilon)\,\mathrm{occ}, (1+\epsilon)\,\mathrm{occ}]$ for any $\epsilon \in (0, 1)$. In this chapter, we study approximate colored path counting and aim at improving previous results under both approximate measures.

We also note that 1D colored range counting is sometimes called *1D colored type-1 range counting* in the literature [40, 13], while *1D colored type-2 range counting* reports the number of occurrences of each distinct color in a query range. We can also generalize the latter to consider tree topology by defining *colored type-2 path counting* over a colored tree, which reports the number of occurrences of each distinct color in a query path. See Figure 5.1 for an example. The $O(n)$-word data structures

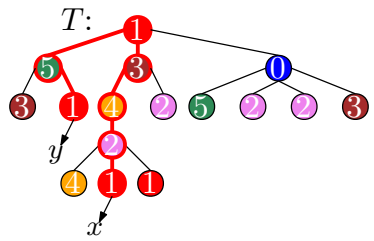


Figure 5.1: Examples of colored type-1 and type-2 path counting. The input is a tree T on 16 nodes, each assigned a color drawn from $\{0, 1, 2, 3, 4, 5\}$. Let $\mathrm{P}_{x,y}$ be a query path. The answer to the colored type-1 path counting query over $\mathrm{P}_{x,y}$ is 5, while the answer to the colored type-2 path counting query over $P_{x,y}$ is a set of entries, $\{(5, 1), (2, 1), (4, 1), (3, 1), (1, 3)\}$, where each entry (e, f) indicates that color e appears f times in $P_{x,y}$.

of Durocher et al. [23] can be used to answer a colored type-2 path counting query in $O(\mathrm{occ} \lg \lg n)$ time. This is slower than the $O(\mathrm{occ}+1)$-time support for 1D colored type-2 range counting over points in rank space [40, 33]. Thus we also investigate the possibility of closing this gap.

5.1.1 Previous Work

By reducing colored path counting to path counting over weighted trees [48] using the chaining approach [62], He and Kazi [44] designed a linear space data structure that supports 2-approximate colored path counting in $O(\lg n / \lg \lg n)$ time. For $(1 \pm \epsilon)$-approximate colored path counting queries, they described a sketching data structure that occupies $O(n + \frac{n}{\epsilon^2 t} \lg n)$ words and answers a query in $O(\epsilon^{-2} t \lg n)$ time with success probability no less than $1 - \delta$, where t is an arbitrary integer in $[1, n]$ and δ is an arbitrary constant in $(0, 1)$. [1] Setting $t = \lceil \epsilon^{-2} \lg n \rceil$ makes the space cost linear and the query time $O(\epsilon^{-4} \lg^2 n)$.

Similar approximate problems can also be defined for colored 1D range counting and colored 2D orthogonal range counting which generalizes the former by preprocessing colored points in 2D to efficiently compute the number of distinct colors assigned to points in an axis-aligned query rectangle. The same conditional lower bound for colored path counting also applies to the latter [55]. In the 1D case, given n colored points in the rank space, El-Zein et al. [27] designed an index data structure that

[1] He and Kazi [44] originally stated their result for constant ϵ, but it is easy to generalize their bounds when $\epsilon = o(1)$.

uses $O(n)$-bits of space[2] and supports c-approximate colored counting in constant time for any constant $c > 1$. In higher dimensions, Rahul [66] showed that $(1 \pm \epsilon)$-approximate colored range counting can be answered by combining a colored range reporting structure and a c-approximate colored range counting structure. With it, he designed an $O(n \lg n)$-word structure to support $(1 \pm \epsilon)$-approximate colored 2D orthogonal range counting in $O(\epsilon^{-2} \lg n)$ time.

Regarding 1D colored type-2 range counting, Gupta et al. [40] designed an $O(n)$-word structure with $O(\lg n + \text{occ})$ query time, over a set of n colored points on a real line. Ganguly et al. [33] stated that, by combining the approach of Gupta et al. and some other results [62, 67], the query time can be further improved to $O(1 + \text{occ})$ if all points are in rank space. This query problem can also be generalized to point sets on the plane, and we refer to [15, 13] for recent work on 2D colored type-2 range counting. For colored trees, the linear word structures designed by Durocher et al. [23] can answer a colored type-2 path counting query in $O(\text{occ} \lg \lg n)$ time. They did not state this result explicitly, but it is implied by the algorithmic steps stated in the proof of Theorem 6 in their paper.

5.1.2 Our Results

Under the word RAM model, we first design 2-approximate colored path counting structures with i) $O(n)$ words of space and $O(\lg^{\lambda} n)$ query time for any constant $0 < \lambda < 1$, ii) $O(n \lg \lg n)$ words of space and $O(\lg \lg n)$ query time and iii) $O(n \lg^{\lambda} n)$ words of space and $O(1)$ query time. In all three cases, the preprocessing time is $O(n \lg n)$. Hence the first trade-off beats the $O(\lg n / \lg \lg n)$ query time of the linear-word 2-approximate structure of He and Kazi [44].

We then design an $O(n)$-word $(1 \pm \epsilon)$-approximate colored path counting structure with $O(\frac{1}{\epsilon^2} \lg n)$ deterministic query time and $O(n^2 \lg C \lg \lg C)$ expected preprocessing time. Compared to the sketching structure by He and Kazi [44] with $O(n)$-word space and $O(\epsilon^{-4} \lg^2 n)$ query time, we not only achieve improvement for query time but also guarantee that the query algorithm always returns a $(1 \pm \epsilon)$-approximation, though the cost of preprocessing is higher.

[2]Note that storing the original data set would occupy $\Omega(n \lg C)$ bits of space. The index data structure given by El-Zein et al. [27] does not store the original data set.

Table 5.1: A summary of our results on approximate colored path counting and colored type-2 path counting, in which space costs are measured in words, ϵ is an arbitrary parameter in $(0, 1)$, λ is an arbitrary constant in $(0, 1)$, † marks an expected bound, and ‡ marks a solution that returns a correct answer with probability no less than $1 - \delta$ for any constant $\delta \in (0, 1)$.

	Space	Query	Preprocess	Ref
2-appr.	$O(n)$	$O(\frac{\lg n}{\lg \lg n})$	$O(n\frac{\lg n}{\lg \lg n})$	[44]
	$O(n)$	$O(\lg^\lambda n)$	$O(n \lg n)$	Thm 6a)
	$O(n \lg \lg n)$	$O(\lg \lg n)$		Thm 6b)
	$O(n \lg^\lambda n)$	$O(1)$		Thm 6c)
$(1 \pm \epsilon)$-appr.	$O(n + \frac{n \lg n}{\epsilon^2 t})$	$O(\epsilon^{-2} t \lg n)$	$O(\epsilon^{-2} n \lg n)$	$[44]^\ddagger$
	$O(n)$	$O(\epsilon^{-2} \lg n)$	$O(n^2 \lg C \lg \lg C)^\dagger$	Thm 7
Type-2	$O(n)$	$O(\text{occ} \lg \lg n)$	$O(n \lg n)$	[23]
		$O(\text{occ})$		Lemma 41

When designing our $(1 \pm \epsilon)$-approximate solutions, our techniques also lead to a linear-word data structure supporting colored type-2 path counting in $O(\text{occ})$ time. This result improves the solution of Durocher et al. [23] which has $O(\text{occ} \lg \lg n)$ query time. See Table 5.1 for a comparison of our results to all previous results.

To achieve these results, we develop new techniques. For 2-approximate colored path counting, note that no further improvement can be made using the strategy of He and Kazi for this problem, due to the lower bound on (uncolored) 2D orthogonal range counting [65] (which is a special case of path counting over weighted trees). Instead, we adopt the strategy, presented in the previous chapter, for batched colored path counting which applies centroid decomposition to decompose the tree into a hierarchy of components. A query is answered by locating and querying the component that satisfies these two conditions: This component contains the entire query path, and its centroid is in the path. This means, within each component, we need only support queries for the paths that pass through a fixed node, e.g, the selected centroid, given during the construction. This strategy however incurs $O(n \lg n)$ words of space cost in Chapter 4. To address this, we design a data structure of $O(n)$ bits that answers 2-approximate queries in constant time, provided that a query path must pass through a certain node. To speed up the mapping of the endpoints of a query path to nodes in a specific component in a space-efficient manner, we borrow ideas from the solutions to the ball inheritance problem, introduced in Section 2.3, and design data structures

with different time-space trade-offs.

With regard to $(1 \pm \epsilon)$-approximate colored path counting queries, we provide a completely different solution from the previous one by He and Kazi [44]. Previously, He and Kazi selected $O(n/t)$ nodes from the input tree as the marked nodes in the preprocessing stage, where t is a parameter defined in the preprocessing stage. Then they constructed and stored an $O(\epsilon^{-2} \lg n)$-word sketch at each marked node. Due to the sketches used in their solution, their query algorithm might not always return a $(1 \pm \epsilon)$-approximation. Unfortunately, we are unaware of any method that uses these sketches and guarantees a $(1 \pm \epsilon)$-approximation is always returned. In the new solution, we divide all query paths into $O(\lg n)$ tiers, depending on the number of distinct colors in each path. For the paths of each tier, we design an efficient solution to $(1 \pm \epsilon)$-approximate queries by combining the tree extraction and the random sampling techniques. We also describe a linear space data structure for colored type-2 path counting, which allows us to compute efficiently the number of distinct colors in a query path by simply reporting all the distinct colors in it, whenever a query path contains no more than $O(\epsilon^{-2} \lg n)$ distinct colors. For the query paths that contains $\Omega(\epsilon^{-2} \lg n)$ distinct colors, we use our solution to 2-approximate colored path counting to decide which tier the query path belongs to and then use the corresponding data structure for $(1 \pm \epsilon)$-approximate queries to find the answer. The idea that reduces $(1 \pm \epsilon)$- to 2-approximate colored path counting queries is inspired by the work of Rahul [66].

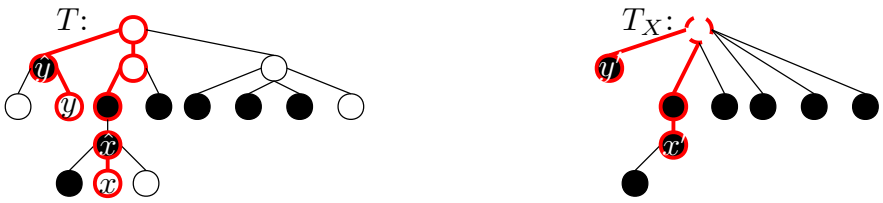
5.2 Preliminaries

This section introduces the previous results used in this chapter.

5.2.1 Partial Rank

Let $A[0..n-1]$ be a sequence of symbols drawn from alphabet $[\sigma] = \{1, 2, \ldots, \sigma\}$, where $\sigma \leq n$. The *partial rank* operation [4], $\text{rank}'(A, i)$, counts the number of elements equal to $A[i]$ in $A[0..i]$.

Lemma 33 *[4, Lemma 3.5] Given a sequence $A[0..n-1]$ drawn from alphabet $[\sigma]$, where $\sigma \leq n$, a data structure of $O(n \lg \sigma)$ bits can be constructed in $O(n)$ time to*



(a) Tree T with $\mathrm{P}_{x,y}$ (b) Tree T_X with $\mathrm{P}_{x',y'}$

Figure 5.2: An example of tree extraction. The first figure shows path $\mathrm{P}_{x,y}$ in the original tree T and its corresponding path $\mathrm{P}_{x',y'}$ in the extracted tree T_X, where the set, X, of extracted nodes are colored black in T. Since the root of T is not in X, T_X has a dummy root. The decompose(x, y) operation returns x' and y' in T_X, which correspond to nodes $\hat{x}$ and $\hat{y}$ in T, respectively.

support $\mathrm{rank}'(A, i)$ *in constant time.*

5.2.2 Tree extraction

Given a subset, X, of nodes of an ordinal tree T, the extracted tree, T_X, can be constructed by deleting each node $v \notin X$ using the following approach: If v is not the root, let $u = \mathrm{parent}(v)$. We remove v and its incident edges from T and insert its children into the list of children of u, replacing v in this list while preserving these children's original left-to-right order. This means that v's left and right siblings before the deletion will respectively become the left and right siblings of its children after the deletion. If v is the root, then, before we apply the same procedure to delete v, we add a dummy root to T and make it the parent of v, so that T_X will remain a tree. This technique first appears in [48].

To map a path $\mathrm{P}_{x,y}$ in T to a path in T_X, we use the operation, decompose(x, y), defined in [44]: If $\mathrm{P}_{x,y} \cap X = \emptyset$, it returns null. Otherwise, let $\hat{x}$ and $\hat{y}$ denote the nodes in $\mathrm{P}_{x,y} \cap X$ that are closest to x and y, respectively (this can be the node x or y itself if it is in X). Then decompose(x, y) returns the nodes x' and y' in T_X that correspond to (i.e., whose original copies are) nodes $\hat{x}$ and $\hat{y}$ in T, respectively. See Figure 5.2 for examples.

Lemma 34 *[44, Proposition 9] Given a tree T on n nodes and a tree extraction T_X, an $O(n)$-bit structure on top of T and T_X can be constructed in $O(n)$ time to support* decompose *in $O(1)$ time.*

5.2.3 The k-nearest distinct ancestor queries.

Let T be a colored tree. In a *k-nearest distinct ancestor query* [23], we are given a node v of T, and the goal is to find a sequence $a_1, a_2, \cdots$ of ancestors of v, in which a_1 is node v itself and a_i is the lowest ancestor of i such that $c(a_i)$ does not appear in $\mathrm{P}_{v,a_i} \setminus \{a_i\}$ for any $1 < i \leq k$. We have the following lemma:

Lemma 35 *[23, Lemma 13] Given a colored tree T on n nodes, an $O(n)$-word data structure can be constructed in $O(n \lg n)$ time to support k-nearest distinct ancestor queries over T in $O(k)$ time. The ancestors are reported in the lowest-to-highest order, and thus k need not be specified in a query.*

5.2.4 Colored Path Counting over All Paths

A tree of n nodes can have $\Theta(n^2)$ different query paths on it. He and Kazi [44] described an efficient algorithm that answers colored path counting queries over all different paths, summarized as Lemma 36.

Lemma 36 *[44, Theorem 10] Let T denote a labeled ordinal tree on n nodes, each of which is assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$. The colored path counting queries over all possible query paths can be answered in $O(n^2 \lg \lg C)$ time in total.*

5.2.5 The Chernoff Bound

The Chernoff Bound summarized as Lemma 37 will be later used in our solution to $(1 \pm \epsilon)$-approximate colored path counting.

Lemma 37 *(Chernoff Bound [60, Corollary 4.6]) Let $X_1 \cdots X_{n'}$ be independent Poisson trials such that $\Pr[X_i = 1] = p_i$ for each $1 \leq i \leq n'$. Let $X = \sum_{i=1}^{n'} X_i$ and $\mu = E[X]$. For any $0 < \epsilon < 1$, $\Pr[|X - \mu| \geq \epsilon\mu] \leq 2e^{-\mu\epsilon^2/3}$.*

5.3 2-Approximate Colored Path Counting

For 2-approximate colored path counting, we first consider a special case in which query paths must contain a fixed tree node specified in the preprocessing step (Section 5.3.1). Then we generalize it for arbitrary paths (Sections 5.3.2-5.3.3).

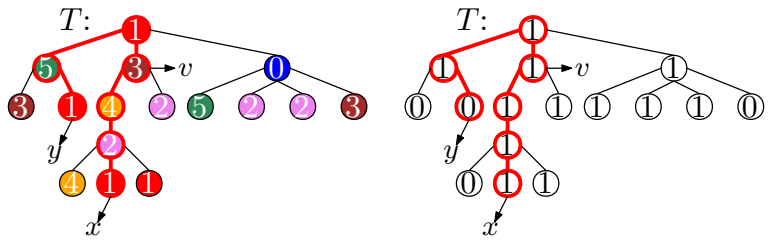


Figure 5.3: The data structure for 2-approximate queries. Given the input tree T and the pre-selected node v, the binary label $B(u)$ constructed for each node u of T is shown on the right-hand side. The query path $\mathrm{P}_{x,y}$ contains node v and has 5 distinct colors on it. Accordingly, we find 6 1-bits in $\mathrm{P}_{x,y}$ in the 0/1-labeled tree, so we return 6 as a 2-approximate of the exact answer.

5.3.1 Counting over a Path that Contains a Fixed Node

Fix a node v of T, and we design an $O(n)$-bit encoding data structure that supports 2-approximate colored path counting over any query path containing v. To answer a query, our encoding data structure does not need to access T after preprocessing, provided that the preorder ranks of the endpoints of the query path are known.

Lemma 38 *Let T be a colored tree on n nodes and fix any node v in T. A data structure of $O(n)$ bits can be constructed in $O(n)$ time to support 2-approximate colored path counting over any query path containing v in $O(1)$ time.*

Proof. We associate a Boolean label $B(u)$ to each node u of T as follows: If $u = v$, then $B(u) = 1$. Otherwise, locate the node, t, in $\mathrm{P}_{u,v}$ that is adjacent to u. If color $c(u)$ appears in $\mathrm{P}_{t,v}$, then set $B(u) = 0$. If not, set $B(u) = 1$. See Figure 5.3 for an example. We discard the original colors of T, treat these labels as node colors and represent T with these labels using the data structure of Lemma 2. Since there are only two possible labels, this uses $3n + o(n)$ bits.

Let $\mathrm{P}_{x,y}$ be a query path containing v. Consider the nodes in the subpath $\mathrm{P}_{x,v}$ one by one in the direction from v to x. Observe that, each time we see a node labeled by 1, we encounter a color that has not been seen previously. Therefore, the number of 1-bits assigned to nodes in $\mathrm{P}_{x,v}$ is equal to $|C(\mathrm{P}_{x,v})|$. Similarly, the number of 1-bits assigned to nodes in $\mathrm{P}_{y,v}$ is equal to $|C(\mathrm{P}_{y,v})|$. Therefore, the number of 1-bits assigned to nodes in $\mathrm{P}_{x,y}$ is a 2-approximate of the precise answer. Following Lemma 3, this number can be computed in $O(\lg \lg C) = O(1)$ time using operations

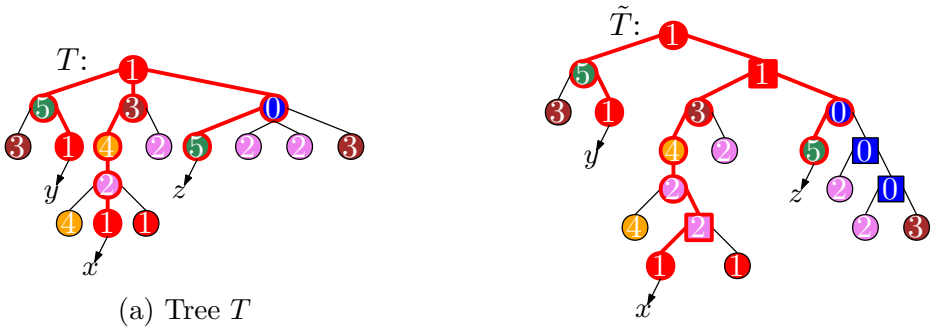
depth_1 and LCA, as $C \leq 2$.

To prove the bound on construction time, it suffices to show that these binary labels can be assigned in $O(n)$ time. This can be done by performing a depth-first traversal of T using v as the starting node. During this traversal, we also update an array $A[0..C-1]$, and the invariant that we maintain is that, each time we visit a node u, $A[i]$ stores the number of nodes in $\mathrm{P}_{v,u}$ that are assigned color i in the original tree T. The following are the steps: We start the traversal from vertex v, set $A[c(v)] = 1$ and initialize all other entries of A to 0. During the traversal, each time we follow an edge (x, y) with $x \in \mathrm{P}_{v,y}$, there are two cases. In the first case, we follow this edge to visit node y. Then this is the first time we visit y. We check if $A[c[y]] = 0$. If it is, then the color of y does not appeared in $\mathrm{P}_{v,x}$, so we set $B(y) = 1$. Afterwards, we increment $A[c[y]]$ to maintain the invariant. Otherwise, we set $B(y) = 0$ and also increment $A[c[y]]$. In the second case, we follow this edge to visit x. Since x is closer to v than y is, we have visited x before, and we will not traverse any paths containing y in the future. In this case, we decrement $A[c[y]]$ to maintain the invariant. ■

5.3.2 Counting over Arbitrary Paths

To support 2-approximate colored path counting queries over arbitrary paths, we transform the given ordinal tree T on n colored nodes into a binary tree $\tilde{T}$. Our transformation is similar to that used by Chan et al. [12], but some modifications are necessary. Ultimately, this is because, unlike the queries considered by Chan et al., to compute a 2-approximate answer for a query path $\mathrm{P}_{x,y}$, we cannot add up 2-approximate answers for query paths $\mathrm{P}_{x,z}$ and $\mathrm{P}_{y,z}$ where $z = \mathrm{LCA}(x, y)$.

Our transformation works as follows: For each node v with degree d, where $d > 2$, we remove the edges between v and its children, $v_1, v_2, \ldots, v_d$, to detach the subtrees rooted at these children from T. For the convenience of the description, let v_0 denote the node v. We then create $d - 2$ dummy nodes, $\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_{d-2}$, each assigned the color of v_0. Next we add edges to reconnect v_0 and its d children with the newly created dummy nodes as follows: For $t = 1, 2, \ldots, d - 2$, make v_t and $\tilde{v}_t$ the left and right children of v_{t-1}, respectively. Afterwards, make v_{d-1} and v_d the left and right children of $\tilde{v}_{d-2}$, respectively. The resulting tree is $\tilde{T}$ which has at most $2n$ nodes. See Figure 5.4 for an example. This transformation preserves preorder among the original



(a) Tree T (b) Tree $\tilde{T}$

Figure 5.4: The input tree T and its transformed binary tree $\tilde{T}$. In $\tilde{T}$, dummy nodes are represented by squares. Query paths can be conceptually divided into two categories, depending on whether or not the LCA of their endpoints in $\tilde{T}$ is a dummy node. For example, $\mathrm{P}_{x,y}$ and $\mathrm{P}_{x,z}$ are in different categories, as $\mathrm{LCA}(x, y)$ of $\tilde{T}$ and $\mathrm{LCA}(x, z)$ of $\tilde{T}$ are non-dummy and dummy nodes, respectively. In either category, a query path in T and its corresponding path in $\tilde{T}$ always share the same set of colors.

nodes of T. More importantly, any path $\mathrm{P}_{x,y}$ in T and its corresponding path, $\tilde{\mathrm{P}}$, in $\tilde{T}$ share the same set of colors. To see this, observe that the lowest common ancestor, z, of nodes x and y in T is the only original node that appears in $\mathrm{P}_{x,y}$ but may not necessarily appear in $\tilde{\mathrm{P}}$. If z does not appear in $\tilde{\mathrm{P}}$, then $\tilde{\mathrm{P}}$ must contain a dummy node created for z which is also colored in $c(z)$. On the other hand, any original node in $\tilde{\mathrm{P}}$ must also be in $\mathrm{P}_{x,y}$ in T, while any dummy node that appears in $\tilde{\mathrm{P}}$ must satisfy the condition that the original node it is created for must be in $\mathrm{P}_{x,y}$ in T.

After this transformation, for each node in T, we store the preorder rank of its corresponding node in $\tilde{T}$. Then we follow the strategy, presented in the previous chapter, to decompose $\tilde{T}$ recursively using centroid decomposition, but different data structures will be constructed this time.

We now give the details of the recursion. At level 0 of the recursion, we call tree $\tilde{T}$ the level-0 component. We find a centroid, u, of $\tilde{T}$ and construct the data structure in Lemma 38 supporting 2-approximate queries over paths containing u. Since $\tilde{T}$ is a binary tree, after removing u, we are left with at most three connected components, and we add u back into the smallest component. In this way, tree $\tilde{T}$ is partitioned into at most three pairwise-disjoint connected components in the level-0 recursion, each of which is a tree on no more than $|\tilde{T}|/2$ nodes. We call each of these three components a *level*-1 *component* and build the data structure recursively upon each

of them. In general, at level i of the recursion, we compute a centroid, v, of each level-i component γ. We then use Lemma 38 to construct a data structure $D(\gamma)$ supporting 2-approximate colored path counting over paths that are entirely contained within γ and also contain v. This component can be partitioned into up to three level-$(i+1)$ components using the approach described above. We call component γ the *parent* of these up to three level-$(i+1)$ components, and each of these up to three level-$(i+1)$ components is a *child* of γ. A component that has a single node is called a *base component* and is not partitioned further. Hence, the recursion contains $O(\lg n)$ levels in total.

Suppose that query path $\mathrm{P}_{x,y}$ is contained entirely within a level-t component γ but not in any level-$(t+1)$ components. This means that $\mathrm{P}_{x,y}$ contains the centroid of γ. Therefore, we can find a 2-approximate of $|C(\mathrm{P}_{x,y})|$ using the data structure $D(\gamma)$, provided that the preorder ranks of x and y in γ are known. To locate component γ and then to compute the preorder rank of x and y in it, we define a *component tree* $\mathcal{CT}$. A component tree is a 3-ary tree in which each node represents a component and the edges represent the parent-child relationship between components. More specifically, a node v at level l of $\mathcal{CT}$ represents a level-l component $\mathcal{C}_v$, where l starts from 0. A node v of $\mathcal{CT}$ is the parent of another node u iff component $\mathcal{C}_v$ is the parent of component $\mathcal{C}_u$. Among the nodes that share the same parent in $\mathcal{CT}$, the relative order between them does not matter, so we order them arbitrarily. The height of $\mathcal{CT}$ is bounded by $O(\lg n)$, and each leaf in it represents a base component. Since each internal node has at least two children, $\mathcal{CT}$ has $O(n)$ nodes in total.

At each internal node v of $\mathcal{CT}$, we build an array $\mathrm{SP}(v)$ of length $|\mathcal{C}_v|$, in which $\mathrm{SP}(v)[i]$ is set to be d if the i-th node (in preorder) in $\mathcal{C}_v$ is stored in the d-th child component of v in the next level. Then we represent $\mathrm{SP}(v)$ using Lemma 33 to support rank$'$. Since v can have at most 3 children, the alphabet size of $\mathrm{SP}(v)$ is constant. Therefore, $\mathrm{SP}(v)$ is represented in $O(|\mathrm{SP}(v)|)$ bits. See Figures 5.5 and 5.6 for an example. With these data structures, we can support queries over arbitrary paths and achieve Lemma 39.

Lemma 39 *Let T be an ordinal tree on n nodes in which each node is assigned a color. A data structure of $O(n)$ words can be constructed in $O(n \lg n)$ time to support 2-approximate colored path counting over T in $O(\lg n)$ time.*

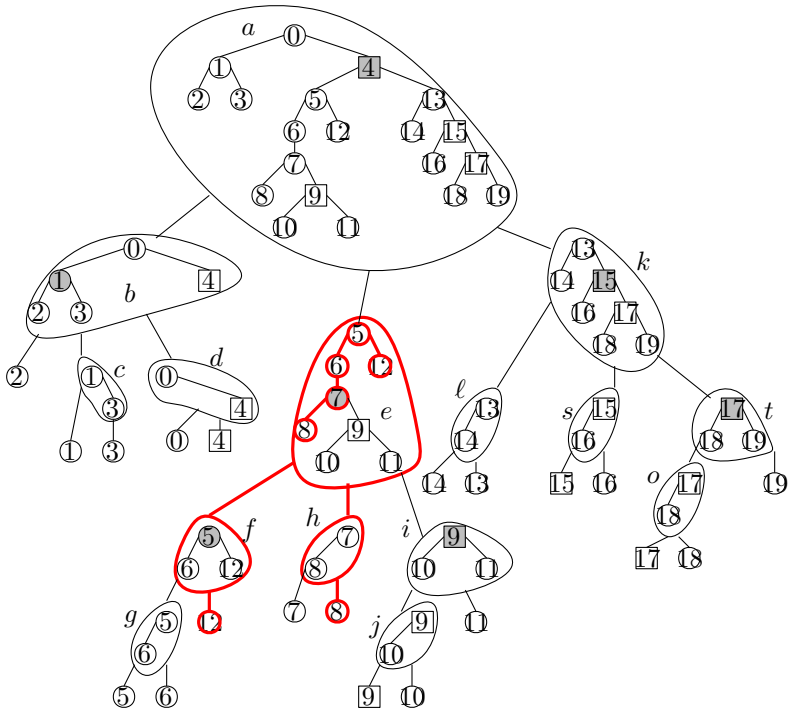


Figure 5.5: An example of a component tree. The component tree in the figure represents a centroid decomposition of the binary tree $\tilde{T}$ presented in Figure 5.4(b). Each node of the component tree represents a connected component of $\tilde{T}$. For illustrative purposes, we explicitly draw these components as well. Within each component, the label on each node is its original preorder rank in $\tilde{T}$, and the gray node represents the selected centroid. Consider the query path between nodes labeled 8 and 12 in $\tilde{T}$. The component labeled e, as the LCA of the base components that represent nodes labeled 8 and 12, contains the query path, and the pre-selected centroid of e is on the query path.

Proof. Given a node $i \in \tilde{T}$, let π denote the path from the root of $\mathcal{CT}$ to the leaf of $\mathcal{CT}$ representing the base component that contains i, and let π_l denote the node in π whose depth is l. First, we show how to locate π_l and to compute the preorder rank of i in component $\mathcal{C}_{\pi_l}$ for $l = 0, 1, 2, \ldots$. The procedure proceeds as follows: We start at the root π_0 of $\mathcal{CT}$. The preorder rank of i in $\mathcal{C}_{\pi_0}$ is i, and π_1 is the $\mathrm{SP}(\pi_0)[i]$-th child of π_0, following the definition of array $\mathrm{SP}(\pi_0)$. In general, given that the preorder rank of i in $\mathcal{C}_{\pi_l}$ is j, one can find node π_{l+1}, which is the $\mathrm{SP}(\pi_l)[j]$-th child of π_l. Since tree extraction preserves preorder, the preorder rank of i in $\mathcal{C}_{\pi_{l+1}}$ is $\mathrm{rank}'(\mathrm{SP}(\pi_l), j) - 1$. Each rank' query takes constant time, so this procedure uses constant time per level of $\mathcal{CT}$.

Since each node of T stores the preorder rank of its corresponding node in $\tilde{T}$, to

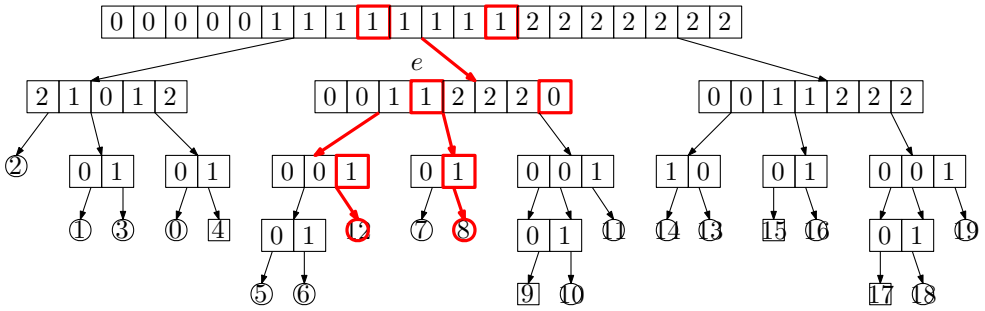


Figure 5.6: The SP's constructed for the component tree presented in Figure 5.5. Consider the query path between the endpoints labeled 8 and 12 in $\tilde{T}$. The entries of the SP arrays at each level that we examine in a top-down traversal to reach the base clusters for nodes 8 and 12 are highlighted in red.

answer a query, it is sufficient to compute a 2-approximate of $|C(\mathrm{P}_{x,y})|$ for a query path $\mathrm{P}_{x,y}$ in $\tilde{T}$. This can be done by performing the top-down traversals of $\mathcal{CT}$ described in the previous paragraph for x and for y simultaneously until we reach the lowest level, l, of $\mathcal{CT}$ such that x and y are contained in the same level-l component γ. This process also gives us the preorder ranks of x and y in γ, which allows us to query $D(\gamma)$ to find a 2-approximate answer. Since $\mathcal{CT}$ has $O(\lg n)$ levels, the query algorithm uses $O(\lg n)$ time.

To analyze the space cost, observe that the total number of nodes in the components at the same level is at most the number of nodes of $\tilde{T}$, which is $2n$. The component tree contains $O(\lg n)$ levels, and all $D(\gamma)$'s and $\mathrm{SP}(v)$'s at the same level use $O(n)$ bits, for a total of $O(n \lg n)$ bits, or $O(n)$ words. The $O(n)$-node component tree $\mathcal{CT}$ itself occupies another $O(n)$ words of space. Therefore, the total space cost is $O(n)$ words. The data structure at each level can be constructed in linear time, so the overall construction time is $O(n \lg n)$. ■

5.3.3 Speeding up the Query

To further improve the query efficiency in Lemma 39, observe that two procedures introduced before require $O(\lg n)$ time for a query $\mathrm{P}_{x,y}$ in $\tilde{T}$: The first locates the lowest component γ in $\mathcal{CT}$ that contains both nodes x and y, and the second computes the preorder ranks of x and y in γ. Previously, both procedures proceed in the same top-down traversal of $\mathcal{CT}$. Now, we perform them separately. For the first procedure,

observe that the node representing component γ in $\mathcal{CT}$ must be the lowest common ancestor of the two leaves of $\mathcal{CT}$ representing the base components that contain nodes x and y, respectively. To locate γ in constant time, we can represent $\mathcal{CT}$ using the data structure of Lemma 2 to support LCA in $O(1)$ time and store with each node x of $\tilde{T}$ a pointer to the base component that contains x. This incurs $O(n)$ words of space and $O(n)$ preprocessing time. To improve the second procedure, we model it by defining an operation, preorder(v, x), as follows: Given a node v of $\mathcal{CT}$ and a tree node x of $\tilde{T}$ that appears in component $\mathcal{C}_v$, preorder(v, x) returns the preorder rank of node x in $\mathcal{C}_v$.

To support preorder(v, x), we borrow ideas from the solutions to the ball inheritance problem, introduced in Section 2.3 and achieve various trade-offs. Let π denote the path between the root of $\mathcal{CT}$ and the leaf representing the base component that contains x, and let π_l denote the node in π whose depth is l. Then each component $\mathcal{C}_{\pi_l}$ contains a copy of node x. Array SP's in Section 5.3.2 work as pointers between these copies in components at consecutive levels. The algorithm in the proof of Lemma 39 follows these pointers one by one till we locate x in $\mathcal{C}_v$, which requires $O(\lg n)$ time. To speed it up, we construct *skipping pointers* which allow us to jump over many levels at one time: Suppose that we have computed the preorder rank, i, of node x in component $\mathcal{C}_{\pi_l}$, and we need to locate x in $\mathcal{C}_{\pi_{l+\Delta}}$ for some positive integer Δ. What we can do is to build an array $\mathrm{SP}_\Delta(\pi_l)$ with length $|\mathcal{C}_{\pi_l}|$, in which $\mathrm{SP}_\Delta(\pi_l)[k]$ is set to d if the k-th node in preorder in $\mathcal{C}_{\pi_l}$ appears in the component represented by the d-th descendant of π_l at depth $l + \Delta$ of $\mathcal{CT}$. If this node is not stored in any level-$(l + \Delta)$ descendant component of π_l (this may happen when $\mathcal{CT}$ is not a complete tree), then $\mathrm{SP}_\Delta(\pi_l)[k] = -1$. Since π_l has up to 3^Δ descendants at level $l + \Delta$, we can represent $\mathrm{SP}_\Delta(\pi_l)$ in $O(|\mathcal{C}_{\pi_l}|\Delta)$ bits by the data structure of Lemma 33 to support rank$'$. Then rank$'(\mathrm{SP}_\Delta(\pi_l), i) - 1$ is the preorder rank of node x in $\mathcal{C}_{\pi_{l+\Delta}}$ and can be computed in $O(1)$ time. We regard $\mathrm{SP}_\Delta(\pi_l)$ as an array of skipping pointers that connect the nodes in component $\mathcal{C}_{\pi_l}$ to the nodes in level-$(l + \Delta)$ descendant components of π_l.

If an array of skipping pointers map nodes in a level-l component to nodes in level-$(l + \Delta)$ descendants of this component, then we say that the length of each of these skipping pointers is Δ. Furthermore, based on previous discussions, storing a skipping pointer of length Δ incurs a space cost of $O(\Delta)$ bits. To achieve good time/space

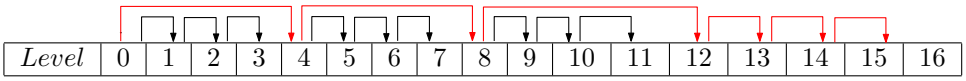


Figure 5.7: An example of the first traversal strategy. In this example, the component tree $\mathcal{CT}$ has 17 levels, and parameter B is set to be 4. The arrows represent the skipping pointer. By following 6 skipping pointers, one can reach level-15 from the root level.

trade-offs, we design two strategies to decide what skipping pointers to construct for each level. Henceforth, let $h = O(\lg n)$ denote the height of $\mathcal{CT}$. Let $B \in [2, h]$ be an integer parameter to be chosen later, and let $\tau = \log_B h$; for simplicity, assume that τ is an integer.

In the first strategy, consider level l of the component tree $\mathcal{CT}$. For each integer $i \in [0, \tau-1]$ such that l is a multiple of B^i but $l+B^i$ is not a multiple of B^{i+1}, we build an array of length B^i skipping pointers for each level-l component. See Figure 5.7 for an example. Since at most $\frac{h}{B^i}$ levels of $\mathcal{CT}$ have skipping pointers of length B^i, the total space cost of all the skipping pointers constructed using this strategy is $\sum_{i=0}^{\tau-1}(\frac{h}{B^i}) \cdot O(nB^i) = O(n \lg n \log_B \lg n)$ bits, which is $O(n \log_B \lg n)$ words of space.

To use these skipping pointers to compute preorder(v, x), let $b_{\tau-1}b_{\tau-2}\cdots b_0$ denote the base-B expression of the depth[3], l_v, of node v in $\mathcal{CT}$. That is, each b_i is in $[0, B-1]$ and $l_v = \sum_{i=0}^{\tau-1} b_i B^i$. We then compute preorder(v, x) in τ phases. In phase-1, we start from the root of $\mathcal{CT}$ and follow length $B^{\tau-1}$ skipping pointers $b_{\tau-1}$ times. Each time after we follow a skipping pointer to reach a level of $\mathcal{CT}$, we use level_anc to locate the ancestor, u, of v at that level. We then follow the skipping pointers in $\mathrm{SP}_{B^{\tau-1}}(u)$ to continue this phase. At the end of phase-1, we have located the ancestor of v at level $b_{\tau-1}B^{\tau-1}$ of $\mathcal{CT}$ and computed the preorder rank of node x in the component that this ancestor represents. In phase-2, we start from this ancestor and follow length $B^{\tau-2}$ skipping pointers $b_{\tau-2}$ times, and so on. In general, in phase-p, we follow length $B^{\tau-p}$ skipping pointers $b_{\tau-p}$ times, reach the ancestor of v at level $\sum_{j=1}^{p} b_{\tau-p}B^{\tau-p}$ of $\mathcal{CT}$ and compute the preorder rank of node x in the component represented by this ancestor. Thus, we reach v and compute the answer after τ phases. Since we follow at most $B-1$ skipping pointers in each phase, the total running time is

[3]Note that a component tree has $O(\lg n)$ depths and the base-B expression of any depth can be encoded in $O(\log B \times \log_B \lg n) = O(\lg n)$ bits. Storing the base-B expression of each of the $O(\lg n)$ depths uses $O(\lg n)$ words of space overall.

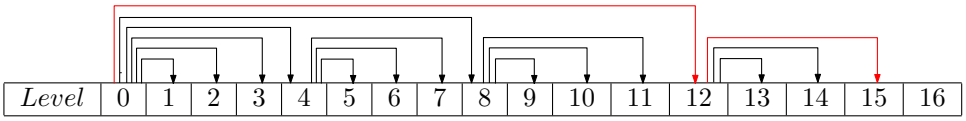


Figure 5.8: An example of the second traversal strategy. In this example, the component tree $\mathcal{CT}$ has 17 levels, and parameter B is set to be 4. The arrows represent the skipping pointer. By following 2 skipping pointers, one can reach level-15 from the root level.

$O(B\tau) = O(B \log_B \lg n)$. Setting $B = \lg^\lambda n$ for an arbitrary constant $\lambda \in (0, 1)$ yields a solution with $O(n)$ space and $O(\lg^\lambda n)$-time support for preorder(v, x).

The second strategy improves the running time of the above process by constructing a different set of skipping pointers so that each phase can be completed by following exactly one skipping pointer. Let l be an arbitrary level of $\mathcal{CT}$. For each integer $i \in [0, \tau - 1]$ such that l is a multiple of B^i, we construct for level-l clusters skipping pointers of length $B^{i-1}, 2B^{i-1}, \ldots, (B-1)B^{i-1}$. See Figure 5.8 for an example. In the first strategy, in phrase-p, for each $p \in [\tau]$, we proceed $b_{\tau-p}$ hops. Now, with these new skipping pointers, we need only one hop, following a skipping pointer of length $b_{\tau-p}B^{\tau-p}$. As a result, the total query time is improved from $O(B\tau)$ to $O(\tau) = O(\log_B \lg n)$. The total space cost of these skipping pointers is then at most $\sum_{i=0}^{\tau-1} \frac{h}{B^i} \cdot (B-1)O(nB^i) = O(nB \lg n \log_B \lg n)$ bits, which is $O(nB \log_B \lg n)$ words. Setting $B = 2$ bounds the space cost by $O(n \lg \lg n)$ and query time by $O(\lg \lg n)$, while setting $B = \lg^\lambda n$ bounds the space cost by $O(n \lg^\lambda n)$ and query time by $O(1)$.

With these three trade-offs for preorder, we have the following theorem:

Theorem 6 *Let T be an ordinal tree on n nodes in which each node is assigned a color. A data structure of $s(n)$ words can be constructed in $O(n \lg n)$ time to support 2-approximate colored path counting over T in $q(n)$ time, where a) $s(n) = O(n)$ and $q(n) = O(\lg^\lambda n)$; b) $s(n) = O(n \lg \lg n)$ and $q(n) = O(\lg \lg n)$; or c) $s(n) = O(n \lg^\lambda n)$ and $q(n) = O(1)$ for any constant $0 < \lambda < 1$.*

5.4 $(1 \pm \epsilon)$-Approximate Colored Path Counting

Our solution to $(1 \pm \epsilon)$-approximate colored path counting consists of three parts: a structure for *colored path reporting*, which lists the set of distinct colors assigned to

the nodes in a query path, a structure for 2-approximate colored path counting and a refinement structure that reduces $(1 \pm \epsilon)$- to 2-approximate colored path counting.

We first present in Section 5.4.1 a linear space data structure that solves colored type-2 path counting in optimal time, thereby solving the colored path reporting problem. In the remaining sections, we show the new data structure solution to $(1 \pm \epsilon)$-approximate colored path counting.

In Section 5.4.2, we describe a random sampling technique that applies to node colors. Then we construct a tree extraction of the input tree based on the sampled colors, which uses less space cost in the expected case, comparing to the original input tree. More importantly, the tree extraction can be used for finding an $(1 \pm \epsilon)$-approximate answer under certain conditions. The conditions will be specified in the same section.

In Section 5.4.3, we will see a solution that is only applicable to *canonical paths*. A canonical path is a path in which the number of distinct colors assigned to the nodes in the path is $[\kappa/2, 2\kappa]$, where κ is an integer parameter specified in the preprocessing stage. The tree extraction constructed on the sampled color set is adopted. Recall that a query path $\mathrm{P}_{x,y}$ in the original input tree has a counterpart, $\mathrm{P}_{x',y'}$, in a tree extraction. We will see how to use $|C(\mathrm{P}_{x',y'})|$ to compute an $(1 \pm \epsilon)$-approximate value of $|C(\mathrm{P}_{x,y})|$ with high probability. The probability is guaranteed by the Chernoff bound stated as Lemma 37. To find $|C(\mathrm{P}_{x',y'})|$, we apply the solution to colored type-2 path counting described in Section 5.4.1. We also provide a verifier, the algorithm that computes the number of distinct colors for all paths, stated as Lemma 36, and turn this *Monte Carlo* data structure to a *Las Vegas* data structure, so that the results returned by our data structure are always correct. Unfortunately, the preprocessing time bound of our data structure is still non-deterministic.

In the last subsection, we generalize the data structure solution, described in Section 5.4.3, for arbitrary query paths, and we will see a reduction from $(1 \pm \epsilon)$- to 2-approximate colored path counting.

5.4.1 Colored Type-2 Path Counting in Optimal Time

We describe a data structure for colored type-2 path counting, which implies the support for colored path reporting. It turns out the colored type-2 path counting

problem is decomposable as shown as Observation 1.

Observation 1 *Let T denote a labeled ordinal tree on n nodes, each of which is assigned a color from $\{0, 1, \ldots, C-1\}$. Consider a query path, P, of T. Suppose that P is partitioned into a constant number of subpaths, and that the answer to the colored type-2 path counting query over each subpath is known. Then the answer to the colored type-2 path counting query over P can be computed in $O(\mathrm{occ})$ time, where* occ *denotes the number of distinct colors in P, with the help of an $O(C)$-word array with all entries initially set to* 0 *during preprocessing.*

Proof. During preprocessing, we construct an array, V, of C integers with each entry set to 0. This is the array mentioned in the lemma. Note that array V is constructed once and will be used by all the queries in the future.

Suppose that the answer to the colored type-2 path counting query over each subpath is given as a list of pairs, in which each entry (c, f_c) stores a color c that occurs in this subpath as well as its frequency f_c. For each entry, (c, f_c), in this list, we increase $V[c]$ by f_c. After scanning these pairs for all the subpaths, each non-zero entry $V[c]$ of V corresponds to a color c that appears in P and stores its frequency in P. To find these non-zero entries of V, we iterate through the answers for each subpath a second time. For each pair (c, f_c) in the answer, if $V[c]$ is non-zero, we return $(c, V[c])$ and set $V[c]$ to zero. All the pairs returned in this procedure form the answer to the colored type-2 path counting query over P. Furthermore, it maintains the invariant that all the entries of V are reset to 0 when the procedure ends.

For each color $c \in C(P)$, entry $V[c]$ is updated a constant number of times, as there are only a constant number of subpaths. Therefore, the overall running time is $O(\mathrm{occ})$. ∎

Let $\mathrm{P}_{x,x'}$ denote any query path such that x' is an ancestor of x. We first consider colored type-2 path counting for $\mathrm{P}_{x,x'}$. The following is our main strategy: For each color $c \in C(\mathrm{P}_{x,x'})$, we locate the lowest node, ℓ_c, in $\mathrm{P}_{x,x'}$ whose color is c, as well as the highest node, h_c, in $\mathrm{P}_{x,x'}$ colored in c. Then the frequency of color c in $\mathrm{P}_{x,x'}$ is $\mathrm{depth}_c(\ell_c) - \mathrm{depth}_c(h_c) + 1$. If we precompute the value of $\mathrm{depth}_{c(v)}(v)$ for each node v, then, after locating ℓ_c and h_c, we can compute the frequency of color c in $\mathrm{P}_{x,x'}$

immediately. Lemma 35 can locate ℓ_c's for all $c \in C(\mathrm{P}_{x,x'})$ in $O(|C(\mathrm{P}_{x,x'})|)$ time. Next, we discuss how to find all h_c's efficiently.

Lemma 40 *Let T denote a labeled ordinal tree on n nodes, each of which is assigned a color from $[C]$. A data structure of $O(n)$ words of space can be constructed in $O(n \lg \lg n)$ time to support finding h_c's of any path $\mathrm{P}_{x,x'}$ in $O(|C(\mathrm{P}_{x,x'})|)$ time, where x' is an ancestor of x.*

Proof. It turns out that the problem of finding all h_c's can be reduced to *path minimum queries*, which ask for a node with the minimum weight in a query path. The idea is borrowed from the chaining approach [62], which was also used for colored path counting [44]. For this reduction, we assign a weight $\mathtt{w}(u)$ to each node $u \in T$ as follows: If $c(u)$ does not appear in $\mathrm{P}_{\perp,\mathrm{parent}(u)}$, then $\mathtt{w}(u)$ is set to -1; otherwise, $\mathtt{w}(u)$ is set to be $\mathrm{depth}(\mathrm{parent}_{c(u)}(u))$, i.e., the depth of the lowest proper ancestor of u that is colored in $c(u)$. In addition, $\mathtt{w}(\perp)$ is set to be -1. Observe that for each node v in $\mathrm{P}_{x,x'}$, if its weight is less than $\mathrm{depth}(x')$, then color $c(v)$ does not appear in $\mathrm{P}_{\mathrm{parent}(v),x'}$, so v is $h_{c(v)}$. Therefore, finding all h_c's is equivalent to finding in $\mathrm{P}_{x,x'}$ all nodes whose weight are less than $\mathrm{depth}(x')$, for which we can use a path minimum query structure. The algorithm proceeds as follows: We find the node, y, with the smallest weight in $\mathrm{P}_{x,x'}$. Due to the way that weights are assigned, the inequality $\mathtt{w}(y) < \mathrm{depth}(x')$ always holds, and we report y. Next, consider the two disjoint subpaths that $\mathrm{P}_{x,x'} \setminus \{y\}$ consists of, i.e., $\mathrm{P}_{\mathrm{parent}(y),x'}$ and $\mathrm{P}_{x,y} \setminus \{y\}$, and for either of them, perform a path minimum query. If the minimum weight of a subpath, P, is no less than $\mathrm{depth}(x')$, then the weights of the nodes in this subpath are all greater than or equal to $\mathrm{depth}(x')$. In this case, we do not report any node in this subpath and return. Otherwise, report the minimum-weight node z. Then perform this process recursively over the two disjoint subpaths whose union is $\mathrm{P} \setminus \{z\}$. The total number of path minimum queries we perform is $O(|C(\mathrm{P}(x, x'))|)$.

In the data structure part, we represent T in $O(n \lg C)$ bits using the data structure of Lemma 2 for fast navigation. As a result, $\mathrm{depth}_{c(v)}(v)$ can be computed in $O(\lg \lg C)$ time for any node v, so precomputing these values for all nodes uses $O(n \lg \lg C)$ time in total and storing these values requires $O(n)$ words of space. It remains to assign weights to the nodes of T and to build the data structure over the node weights for

path minimum queries. To assign node weights, we perform a preorder traversal of T. Each time a node v is visited, we locate $v' = \text{parent}(v)$ and perform a colored path emptiness query to find out whether $c(v)$ appears in $\text{P}_{v',\perp}$. If it does not, set $\texttt{w}(v) = -1$; otherwise, set $\texttt{w}(v) = \text{parent}_{c(v)}(v)$. Each colored path emptiness query and each $\text{parent}_\alpha(v)$ query takes $O(\lg\lg C)$ time by the data structure of Lemma 2, so assigning weights to all n nodes uses $O(n \lg\lg C)$ time. To answer each path minimum query in $O(1)$ time, the linear data structure given in [12, Theorem 1.1(a)] is applied, which can be constructed in $o(n \lg\lg n)$ time. In the end, the overall preprocessing time is bounded by $O(n \lg\lg C) + o(n \lg\lg n) = O(n \lg\lg n)$, and the overall space cost is $O(n)$ words. ■

Note that the order in which we report the h_c's by the data structure of Lemma 40, differs from the order in which ℓ_c's are reported by the data structure of Lemma 35. In the proof of Lemma 41, we show how to match h_c to ℓ_c for the same color c efficiently and also extend this method to general query paths.

Lemma 41 *Let T be an ordinal tree on n nodes with each node assigned a color from $\{0, 1, \ldots, C-1\}$, where $C \leq n$. A data structure occupying $O(n)$ words of space can be constructed over T in $O(n \lg n)$ time to support colored type-2 path counting in $O(\text{occ})$ time, where* occ *denotes the number of distinct colors in a query path.*

Proof. The data structure part is straightforward. We apply the linear space data structures shown in Lemmas 35 and 40, respectively. In addition, we construct an array, V, of C integers with each entry set to 0. Overall, the space cost of our data structures is $O(n)$ words, and the construction requires $O(n \lg n)$ time, bounded by building the data structure of Lemma 35.

Let $\text{P}_{x,y}$ denote an arbitrary query path. To answer the query, we first locate $u = \text{LCA}(x, y)$ in constant time. Nodes x, u and y together partition $\text{P}_{x,y}$ into three pairwise disjoint subpaths, $\text{P}_{x,x'}$, $\text{P}_{u,u}$ and $\text{P}_{y,y'}$, where x' and y' denote the children of u in $\text{P}_{x,u}$ and $\text{P}_{y,u}$, respectively. Next, we show how to answer the colored type-2 path counting query over $\text{P}_{x,x'}$. As defined before, ℓ_c and h_c denote the lowest and highest nodes colored in c in $\text{P}_{x,x'}$ for any color $c \in C(\text{P}_{x,x'})$, and all ℓ_c's and h_c's can be computed in $O(|C(\text{P}_{x,x'})|)$ time, but the order in which all ℓ_c's are reported differs from the order in which all h_c's are reported. To compute the answer in optimal time,

each time we compute a ℓ_c, we retrieve the precomputed $\text{depth}_c(\ell_c)$ and store it in entry $V[c]$. Then, for each h_c, we return $(c, V[c] - \text{depth}_c(h_c) + 1)$. The pairs returned form the answer to the colored type-2 path counting query over $\text{P}_{x,x'}$, and this process uses $O(|C(\text{P}_{x,x'})|)$ time. The colored type-2 path counting query for subpath $\text{P}_{y,y'}$ can be answered in a similar way, and it is trivial to answer the query over $\text{P}_{u,u}$.

Finally, to find the answer for $\text{P}_{x,y}$, we merge the answers for all subpaths, which uses $O(\text{occ})$ time by Observation 1. Therefore, the overall query time is bounded by $O(\text{occ})$. ■

Remark. Our $(1 \pm \epsilon)$-approximate colored path counting structure only needs a component that supports colored path reporting in $O(\text{occ})$ time. Thus, Lemma 35 is already sufficient. Here we prove Lemma 41 not only because it supports the more powerful colored type-2 path counting queries, but also because our techniques imply a simple optimal solution to path colored reporting: Given $\text{P}_{x,x'}$, we compute all h_c's (without computing ℓ_c's) and report their colors, and this solution can be generalized to arbitrary query paths using the same approach we just described. Our data structure is simpler than the structures of Lemma 35 which use hive graphs and point location (to report ℓ_c's in the order needed for other results in [23]). Henceforth, when we apply Lemma 41 to support colored path reporting, we mean to construct this simple solution only.

5.4.2 Random Sampling

Our solution, to be presented later, to $(1 \pm \epsilon)$-approximate queries, combines the techniques, random sampling and tree extraction. The details can be found as follows:

Set $\theta = \frac{6(c_1+3)\lg n}{\epsilon^2 \lg e}$, where e denotes Euler's number and $c_1 \geq 1$ is an arbitrary positive constant. Let $\kappa \in (\theta, n]$ be an integer parameter to be chosen later, and define $\text{M} = \theta/\kappa$. We create a random color set C' by choosing each color that appears in T independently at random with probability M. Then we construct a tree extraction T' from T by removing nodes whose colors are not in C' using the approach described in Section 5.2.2. All the nodes in T' are assigned their original color in T except for the dummy root; if a dummy root is added, it is uncolored. For each color $c \in \{0, \cdots, C-1\}$, let X_c denote a random variable indicating whether

color c is sampled: X_c is set to 1 if c has been sampled and 0 otherwise. Thus, $\Pr[X_c = 1] = \mathrm{M}$. Furthermore, for an arbitrary path $\mathrm{P}_{x,y}$ in T, we define a random variable $X_{x,y} = \sum_{c \in C(\mathrm{P}_{x,y})} X_c$. Lemma 42 states the conditions under which $X_{x,y}/\mathrm{M}$ is a $(1 \pm \epsilon)$-approximate of $|C(\mathrm{P}_{x,y})|$ with high probability.

Lemma 42 *Consider an arbitrary path* $\mathrm{P}_{x,y}$ *in* T*. If* $\mathrm{occ} \geq \kappa/2$*, where* occ *denotes* $|C(\mathrm{P}_{x,y})|$*, then* $\Pr[(1-\epsilon)\,\mathrm{occ} \leq \frac{X_{x,y}}{\mathrm{M}} \leq (1+\epsilon)\,\mathrm{occ}] > 1 - \frac{2}{n^{c_1+3}}$.

Proof. Let μ be $\mathbb{E}[X_{x,y}]$. Following the Chernoff bound in Lemma 37, we observe that $\Pr(|X_{x,y} - \mu| > \epsilon\mu) \leq 2e^{-\mu\epsilon^2/3}$ for any $0 < \epsilon < 1$. Since $\mu = \mathrm{occ} \cdot \mathrm{M}$ and $\kappa/2 \leq \mathrm{occ}$, we have

$$\begin{aligned}
\Pr[|X_{x,y} - \mathrm{occ} \cdot \mathrm{M}\,| > \epsilon \cdot \mathrm{occ} \cdot \mathrm{M}] &\leq 2e^{-\,\mathrm{occ} \cdot \mathrm{M}\,\epsilon^2/3} \\
&\leq 2e^{-\frac{\kappa}{2} \cdot \frac{6(c_1+3)\lg n}{\kappa\,\epsilon^2 \lg e} \cdot \frac{\epsilon^2}{3}} \\
&= 2e^{\ln n^{-(c_1+3)}} \\
&= 2 \cdot n^{\frac{1}{c_1+3}}.
\end{aligned}$$

Therefore, $\Pr[(1-\epsilon)\,\mathrm{occ} \leq \frac{X_{x,y}}{\mathrm{M}} \leq (1+\epsilon)\,\mathrm{occ}] > 1 - \frac{2}{n^{c_1+3}}$. ∎

5.4.3 Approximate Colored Path Counting over Canonical Paths

Now, we are ready to present our data structure for $(1 \pm \epsilon)$-approximate colored path counting queries. To use Lemma 42 and also due to other considerations, we call a path in T *canonical* if the number of colors that appear in the path is in $[\kappa/2, 2\,\kappa]$, for an integer $\lceil\theta\rceil \leq \kappa \leq C/2$ to be decided later. In this section, we solve the problem for canonical paths, and then generalize our solution to be workable for arbitrary paths in the next section.

Lemma 43 *Let* T *be an ordinal tree on* n *nodes represented by the data structure of Lemma 2. With success probability more than* $1 - \frac{1}{n^{c_1+1}}$*, one can construct a data structure in* $O(n \lg n)$ *worst-case time to answer* $(1 \pm \epsilon)$*-approximate colored path counting queries over canonical paths in* $O(\epsilon^{-2} \lg n)$ *worst-case time. The space cost is* $O(n \cdot \mathrm{M} + n/\lg n)$ *words in the expected case (and* $O(n)$ *words in the worst case).*

Proof. First, we present the data structures. As described in Section 5.4.2, we choose a random color set and construct a tree extraction T' consisting of nodes with the sampled colors. Tree T' has $O(n \cdot \mathrm{M})$ expected number of nodes but $O(n)$ nodes in the worst case. We represent T' by the data structure of Lemma 2 in $O(|T'| \lg C)$ bits to support fast navigation. We also construct the data structure of Lemma 34 over T and T' to support decompose(x, y) in constant time; the data structure uses $O(n)$ bits which is $O(n/\lg n)$ words. Finally, we construct in $O(n \lg n)$ time in the worst case the linear space data structure for colored path reporting queries over T' by applying Lemma 41. The overall space cost is $O(n \cdot \mathrm{M} + n/\lg n)$ words in the expected case and $O(n)$ words in the worst case. The construction time is bounded by $O(n \lg n)$.

To describe the query algorithm, let $\mathrm{P}_{x,y}$ be a canonical query path. Since $|C(\mathrm{P}_{x,y})| \geq \kappa/2$, $X_{x,y}/\mathrm{M}$ is a $(1 \pm \epsilon)$-approximate of $|C(\mathrm{P}_{x,y})|$ with probability greater than $1 - \frac{2}{n^{c_1+3}}$ following Lemma 42. Since M is given in preprocessing, we need only compute $X_{x,y}$. Let x' and y' be the nodes of T' returned by decompose(x, y) in $O(1)$ time. If x' and y' are null, no color in $\mathrm{P}_{x,y}$ has been sampled, so we set $X_{x,y}$ to be 0. Otherwise, $X_{x,y}$ is either $|C(\mathrm{P}_{x',y'})|$ or $|C(\mathrm{P}_{x',y'})| - 1$, and we can determine which case it is by performing these steps: If the color of $z = \mathrm{LCA}(x, y)$ in T is sampled, then its corresponding node, z', in T' belongs to $\mathrm{P}_{x',y'}$. In this case, there is a one-to-one correspondence between the nodes in $\mathrm{P}_{x',y'}$ and the nodes in $\mathrm{P}_{x,y}$ whose colors are sampled, so $X_{x,y} = |C(\mathrm{P}_{x',y'})|$. If the color of z is not sampled, then the node $z'' = \mathrm{LCA}(x', y')$ in T' does not correspond to z, but all other nodes in $\mathrm{P}_{x',y'}$ correspond to nodes with sampled colors in the query path. Then there are two sub-cases to be considered, depending on whether the color of z'' also happens to appear in $\mathrm{P}_{x',y'} \setminus \{z''\}$. If it does, then we have $X_{x,y} = |C(\mathrm{P}_{x',y'})|$, and otherwise, $X_{x,y} = |C(\mathrm{P}_{x',y'})| - 1$. Navigational operations such as LCA can be performed over T and T' in constant time, and whether $c(z'')$ appears in $\mathrm{P}_{x',y'} \setminus \{z''\}$ can be tested by two path emptiness queries in $O(\lg \lg C)$ time. Therefore, if we know the value of $|C(\mathrm{P}_{x',y'})|$, we can compute $X_{x,y}$ in $O(\lg \lg C)$ extra time.

It remains to show how to compute $|C(\mathrm{P}_{x',y'})|$. To do it, observe that if $X_{x,y}/\mathrm{M}$ is a $(1 \pm \epsilon)$-approximate, then $X_{x,y} \leq (1+\epsilon) \cdot \mathrm{M} \cdot |C(\mathrm{P}_{x,y})| \leq (1+\epsilon) \cdot \mathrm{M} \cdot 2\kappa$. Since $X_{x,y}$ is at least $|C(\mathrm{P}_{x',y'})| - 1$, we have $|C(\mathrm{P}_{x',y'})| \leq (1+\epsilon) \cdot \mathrm{M} \cdot 2\kappa + 1$. With this, we can

apply Lemma 41 to report the distinct colors in $C(\mathrm{P}_{x',y'})$, and instead of reporting all these colors, we stop when the number of reported colors reaches $(1+\epsilon)\cdot \mathrm{M}\cdot 2\,\kappa+2$. If this happens, we terminate our query algorithm with failure. Otherwise, the number of colors reported is $|C(\mathrm{P}_{x',y'})|$. Since $(1+\epsilon)\cdot \mathrm{M}\cdot 2\,\kappa+2 = O(\frac{1}{\epsilon^2}\lg n)$, this process uses $O(\frac{1}{\epsilon^2}\lg n)$ time.

By Lemma 42, for an arbitrary query path, our data structure fails to returns a correct answer with probability $\Pr[|\frac{X}{\mathrm{M}} - \mathrm{occ}\,| > \epsilon\cdot \mathrm{occ}] < \frac{2}{n^{c_1+3}}$. Since there are $\binom{n}{2}$ different query paths, the probability of constructing a data structure that answers all queries correctly is more than $1-\binom{n}{2}\cdot\frac{2}{n^{c_1+3}} > 1-\frac{1}{n^{c_1+1}}$. ∎

Next, we consider finding a data structure that occupies $O(n\cdot \mathrm{M}+\frac{n}{\lg n})$ words in the worst case and can always answer correctly for canonical paths. Our approach, presented in the proof of Lemma 44, is to keep resampling colors and building the data structure of Lemma 43 for the sample, until such a data structure is found.

Lemma 44 *Let T be an ordinal tree on n nodes represented by the data structure of Lemma 2. A data structure occupying $O(n\cdot \mathrm{M}+n/\lg n)$ extra words in the worst case can be constructed in $O(n^2\lg\lg C)$ expected time to support $(1\pm\epsilon)$-approximate colored path counting over canonical paths in $O(\epsilon^{-2}\lg n)$ worst-case time.*

Proof. We first analyze how many tries we need until such a data structure is built. Each try could fail in two ways: Either the data structure constructed returns an incorrect answer for at least one canonical query path, or it uses more than $O(n\cdot \mathrm{M}+n/\lg n)$ words. By Lemma 43, the former happens with probability less than $\frac{1}{n^{c_1+1}}$. For the latter, we bound the probability that the extracted tree T' has more than $3\cdot n\cdot \mathrm{M}$ nodes. Due to the possible addition of a dummy root, the expected number of nodes in T' is at most $n\cdot \mathrm{M}+1$. Then, by Markov's inequality, $\Pr[|T'|\geq 3\cdot n\cdot \mathrm{M}] \leq \frac{n\cdot \mathrm{M}+1}{3\cdot n\cdot \mathrm{M}} \leq \frac{2}{3}$. Therefore, the overall probability of failure for a single try is less than $\frac{1}{n^{c_1+1}}+\frac{2}{3}$, where $c_1\geq 1$. Hence, the expected number of tries is at most $1/(1-(\frac{1}{n^{c_1+1}}+\frac{2}{3})) = O(1)$ for sufficiently large n.

The time spent on each try is dominated by the time of verifying whether the structure built in this try can answer queries over all possible canonical paths correctly. This can be done in $O((n^2+|T'|^2)\cdot\lg\lg C) = O(n^2\lg\lg C)$ time with the help of Lemma

36. As discussed in the proof of Lemma 43, up to two colored path emptiness queries may be performed for each path in T', and this cost is absorbed in the above bound. Since the expected number of tries is $O(1)$ and each tries uses $O(n^2 \lg\lg C)$ time, this lemma follows. ■

5.4.4 Approximate Colored Path Counting over Arbitrary Paths

To solve queries over arbitrary paths, we first represent T by the data structure of Lemma 2 to support navigational operations. We also construct the data structures of part a) of Theorem 6 to support 2-approximate colored path counting over T. In addition, we build the data structures of Lemma 41 to support colored path reporting. These data structures use $O(n)$ words and can be built in $O(n \lg n)$ time.

Then, for each $i \in [\lceil \lg\theta \rceil, \lceil \lg C \rceil)$, let κ_i be 2^i. We refer to a query path as a *tier-i canonical path* if the number of distinct colors that appear in it is in $[\kappa_i/2, 2\kappa_i]$. For each possible value of i, we apply Lemma 44 to construct a data structure $\mathbb{DS}_i$ to support $(1 \pm \epsilon)$-approximate colored path counting over tier-i canonical paths. Data structure $\mathbb{DS}_i$ uses $O(n\theta/\kappa_i + n/\lg n) = O(n\theta/2^i + n/\lg n)$ words in the worst case and can be constructed in $O(n^2 \lg\lg C)$ expected time. Summing up over all $i \in [\lceil \lg\theta \rceil, \lceil \lg C \rceil)$, the overall space cost of these data structures is $O(n)$ words in the worst case, and they can be constructed in $O(n^2 \lg C \lg\lg C)$ expected time. Theorem 7 summarizes our final result.

Theorem 7 *Let T be an ordinal tree on n nodes with each node assigned a color from $[C]$, where $C \leq n$. A data structure of $O(n)$ words of space in the worst case can be constructed in $O(n^2 \lg C \lg\lg C)$ expected time to support $(1 \pm \epsilon)$-approximate colored path counting in $O(\epsilon^{-2} \lg n)$ worst-case time.*

Proof. It remains to present the query algorithm. Let $\mathrm{P}_{x,y}$ denote the query path. We first use the colored path reporting structure to report up to θ distinct colors that appear in $\mathrm{P}_{x,y}$. If this step reports less than θ colors, then we return the number of colors reported, taking $O(\epsilon^{-2} \lg n)$ time. Otherwise, $\mathrm{occ} > \theta$. In this case, we compute a 2-approximate result, occ_a in $O(\lg^{\lambda} n)$ time. Then, $\mathrm{occ} \leq \mathrm{occ}_a \leq 2\,\mathrm{occ}$. Observe that, for any $i \in [\lceil \lg\theta \rceil, \lceil \lg C \rceil)$, if $\kappa_i \leq \mathrm{occ}_a \leq 2\kappa_i$, then $\kappa_i/2 \leq \mathrm{occ} \leq 2\kappa_i$.

This allows us to perform a binary search in $O(\lg \lg n)$ time to find the value of i such that $\mathrm{P}_{x,y}$ is a tier-i canonical path. Finally, by querying $\mathbb{DS}_i$, we can find a $(1 \pm \epsilon)$-approximate of occ in $O(\epsilon^{-2} \lg n)$ worst-case time. ■

5.5 Conclusion

In this chapter, we saw data structures that solve approximate colored path counting under two different approximate measures, i.e., 2- and $(1\pm\epsilon)$-approximate. In terms of 2-approximate colored path counting, we achieve three different time-space tradeoffs, which have $O(n)$ words of space usage and $O(\lg^{\lambda} n)$ query time, $O(n \lg \lg n)$ words of space usage and $O(\lg \lg n)$ time and $O(n \lg^{\lambda} n)$ words of space usage and $O(1)$ time, respectively. For $(1 \pm \epsilon)$-approximate colored path counting, we design a linear space data structure that supports each counting query in $O(\epsilon^{-2} \lg n)$ time.

For 2-approximate queries, our work has shown that the strategy of using centroid decomposition for colored path counting not only works for exact queries as in Chapter 4 but also works for approximate queries. We have also seen how to use succinct data structures carefully to achieve better time-space tradeoffs. For $(1 \pm \epsilon)$-approximate queries, we apply the technique, tree extraction, which has been shown to be a powerful tool to answer various types of path queries exactly such as path counting [48], path reporting [46], path medium [48] and colored path counting [44]. Our work has shown that it can be further combined with random sampling to design approximate solutions.

However, there are several problems left to be solved in the future. When a path contains $\epsilon^{-2} \lg n$ distinct colors, answering the $(1 \pm \epsilon)$-approximate query for this path requires $\Omega(\epsilon^{-2} \lg n)$ time in our solution. In this case, using colored type-2 path counting queries, we find the exact number of distinct colors, instead of an approximate number. Due to this, the overall query time of $(1 \pm \epsilon)$-approximate colored path counting is bounded by $\Theta(\epsilon^{-2} \lg n)$ in the worst case, although each 2-approximate query used in our solution takes as little as $O(\lg^{\lambda} n)$ time. It is worth looking into a faster solution to $(1 \pm \epsilon)$-approximate colored path counting over paths that contain no more than $O(\epsilon^{-2} \lg n)$ distinct colors.

To guarantee that the query algorithm always returns a $(1 \pm \epsilon)$-approximation, we

need spend $O(n^2 \lg C \lg \lg C)$ time on the preprocessing. An open problem is whether the overall preprocessing time can be further improved.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude this thesis by discussing all the results achieved in the previous three chapters and proposing open problems.

6.1 Results

In Chapter 3, we design three different data structures for solving the colored 2D orthogonal range counting problem, resulting in four time-space tradeoffs, which have $O(n \lg^3 n)$ words of space usage and $O(\sqrt{n} \lg^{5/2} n \lg \lg n)$ query time, $O(n \lg^2 n)$ words of space usage and $O(\sqrt{n} \lg^{4+\lambda} n)$ query time, $O(n \frac{\lg^2 n}{\lg \lg n})$ words of space usage and $O(\sqrt{n} \lg^{5+\lambda} n)$ query time, and $O(n \lg n)$ words of space usage and $O(n^{1/2+\lambda})$ query time. In Chapter 4, we present the data structures that solve the colored path counting and path mode query problems. Our data structures are efficient in terms of query time and the preprocessing time so that we are able to answer n colored path counting queries in overall $O(n^{1.4071})$ time and n path mode queries in overall $O(n^{1.483814})$ time, both including the preprocessing time. In Chapter 5, we consider the approximate colored path counting problem. For 2-approximate colored path counting, we design data structures with $O(n)$ words of space and $O(\lg^\lambda n)$ query time, $O(n \lg \lg n)$ words of space and $O(\lg \lg n)$ time and $O(n \lg^\lambda n)$ words of space and $O(1)$ time, respectively. We also present a linear space data structure that supports $(1 \pm \epsilon)$-approximate colored path counting in $O(\epsilon^{-2} \lg n)$ time.

As mentioned in the first chapter, the common difficulty in solving colored counting is that the query is not decomposable. We manage to overcome this difficulty by fulfilling the following principle: After dividing a query range or a query path into subranges or subpaths, make sure that each color that appears in multiple subranges or subpaths is counted only once. We design new techniques to guarantee this, and by doing so, we add to the tool-set that can be used to answer colored queries.

To solve the colored 2D orthogonal range counting problem, we partition the 4-sided query range $[x_1, x_2] \times [y_1, y_2]$ into two subranges, each bounded by 3 sides, and then we reduce colored 2D 3-sided range counting to stabbing queries over 3D canonical boxes, so that the number of boxes that contain the query point, transformed by a 2D 3-sided query range, equals to the number of distinct colors in this 2D 3-sided query range. Using different combinations of the geometric data structures, including segment trees, interval trees and wavelet trees, we design three different data structures for stabbing queries. In each solution, we preprocess the input boxes and assign them into bottom lists, and in the query procedure, we decompose the boxes containing the query point into a bounded number of sub-lists. Unlike the solution by Kaplan et al. [55], in which one has to make sure that each sub-list has to be exactly the same as some bottom list predefined, ours only requires that each sub-list is a prefix of some bottom list. This extra flexibility allows us to design a new scheme that computes the number of distinct colors that appear in both 2D 3-sided query ranges generated from a 2D 4-sided query range. Furthermore, this new scheme is applicable to all our three different stabbing query data structures, resulting in three different data structures for the colored 2D orthogonal range counting in the end.

As we have seen in Chapters 4 and 5, the technique, centroid decomposition, is a cornerstone in our solutions to all path queries, apart from colored type-2 path counting. Previously, He and Kazi [44] solved the colored path counting problem by performing one-level partition of trees: One of their solutions uses an approach from Durocher et al. [23] and the other one based on the pigeon hole principle. What's new in our work is that we put together two techniques, i.e., centroid decomposition and node-marking with the pigeon hole principle. This combination appears in our solutions to the colored path counting and the path mode query problems. By applying it, we manage to reduce the prior to the matrix multiplication and the latter to the min-plus product. Unlike the previous solution by Kaplan et al. [56] to batched colored 2D orthogonal range counting and the previous solution by Gu et al. [39] to batched range mode queries, the matrices in our reductions are neither sparse nor monotone. To resolve the difficulty in colored path counting, we use the properties of our one-level node marking scheme to carefully reduce the problem of multiplying these non-sparse matrices to the multiplication of two different but related matrices

that are sparse, and to overcome the difficulty in path mode queries, we propose a two-level marking scheme that reduces our non-monotone matrix to a related matrix that has smaller number of different entries between each consecutive columns. As a result, we are able to apply the sparse matrix multiplication and the three-phase algorithm presented by Gu et al. [39] for the min-plus product and achieve the improved the preprocessing time bounds in building both data structures.

In Chapter 5, we saw centroid decomposition adopted to solve approximate colored path counting. As a recursive technique, it divides the input tree into connected components of smaller sizes; as a result, a path $\mathrm{P}_{x,y}$ is always contained within some component and at the same time contains the pre-selected centroid of this component. To find the preorder ranks of nodes x and y in the component, unlike the strategy adopted in Chapter 4, in which we simply store at nodes x and y $O(\lg n)$ pointers that point to their copies at $O(\lg n)$ recursive levels, we manage to design space-efficient data structures that give three different time-space bounds. Namely, within $O(n)$, $O(n \lg\lg n)$ and $O(n \lg^{\lambda} n)$ words of space, we can find the preorder rank of x in any component in $O(\lg^{\lambda} n)$, $O(\lg\lg n)$ and $O(1)$ time, respectively. We believe these data structures could find their applications in other path queries, as it can be seen as a generalization of rank reduction on binary range trees, presented in Section 2.3. As we know, the latter has numerous applications in range searching and string processing. Furthermore, with the restriction that all query paths contain the same pre-selected node and are contained within a component of smaller size, path queries can be always significantly simplified. For example, as we have seen in Lemma 38, after adding the restriction, an index data structure that supports 2-approximate colored path counting in optimal time only requires $O(n)$ bits of space.

6.2 Future Work

Some open problems which aim at improving our results have been already proposed at the end of each chapter. In this section, we discuss possible future work on closely related problems.

In Chapter 4, we saw efficient algorithms that answer n colored path counting queries in $O(n^{1.4071})$ time and n path mode queries in $O(n^{1.483814})$ time. However, the working space of our algorithm for batched colored path counting is $\Omega(n^{1.1860})$

words. Currently, the only known solution in linear space supports each colored path counting in $O(\sqrt{n}\lg\lg C)$ time [44]. On the other hand, to path mode queries, a linear space solution with $O(\sqrt{n/w}\lg\lg n) = o(\sqrt{n})$ query time has been achieved by Durocher et al. [23]. An interesting open problem is whether we can achieve a similar result for colored path counting, i.e., a linear space data structure that supports each colored path counting query in $O(n^{1/2-o(1)})$ time.

In Chapter 5, we described space-efficient data structures built for the components generated after applying the centroid decomposition to a binary tree, which support computing efficiently the preorder rank of any tree node in any component. Although the data structure is only workable for a binary tree, it is sufficient for colored path counting as we have seen after turning a labeled tree with unbounded degree into a binary tree, the numbers of distinct colors in query paths remain unchanged. However, after the transformation, we added extra colored nodes. As a result, the frequency of a color in a path might change after the transformation. In terms of color-frequency related problems, such as path mode queries, our solution only applies to binary trees. We would like to see similar data structures that are workable for arbitrary-ary labeled trees so that they could have a bigger scope of applications in path queries.

In Chapter 5, we saw a linear space data structure that supports colored type-2 path counting queries in optimal time, which can also be used for finding a *α-minority* in a query path. Given a query path $\mathrm{P}_{x,y}$, a *α-minority*, for any $\alpha \in [0,1]$, of $\mathrm{P}_{x,y}$ is a color that appears at least once and at most $\alpha|\,\mathrm{P}_{x,y}\,|$ times in $\mathrm{P}_{x,y}$, and a path α-minority query returns one α-minority, if there are multiple of them. As observed by Chan et al. [11], any set of α^{-1} distinct colors that appear in $\mathrm{P}_{x,y}$ must has at least one α-minority. Durocher et al. [23] described an $O(n)$-word space data structure that supports path α-minority queries in $O(\alpha^{-1}\lg\lg n)$ time. In their solution, they report α^{-1} distinct colors in $\mathrm{P}_{x,y}$. Each time a color is reported, they spend $O(\lg\lg n)$ on computing its frequency in the query path and check whether or not it is a α-minority. Naturally, their solution supports colored type-2 path counting in $O(\mathrm{occ}\lg\lg n)$ time by simply reporting all occ distinct colors and their respective frequencies. Although we improve the query time for type-2 queries from $O(\mathrm{occ}\lg\lg n)$ to $O(\mathrm{occ})$, our data structure would take $O(\mathrm{occ})$ time to find a α-minority, since our solution cannot guarantee to find all the frequencies of α^{-1} distinct colors until it reports all occ

distinct colors. So we would like to ask whether there is a linear space data structure that supports path α-minority queries in $O(\alpha^{-1})$ time.

Bibliography

[1] Josh Alman and Virginia Vassilevska Williams. A Refined Laser Method and Faster Matrix Multiplication. In *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*. Society for Industrial and Applied Mathematics, 2021.

[2] Stephen Alstrup and Jacob Holm. Improved Algorithms for Finding Level Ancestors in Dynamic Trees. In *Proceedings of the 27th International Colloquium on Automata, Languages, and Programming (ICALP 2000)*. Springer, 2000.

[3] Nikhil Bansal and Ryan Williams. Regularity Lemmas and Combinatorial Algorithms. *Theory of Computing*, 2012.

[4] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time String Indexing and Analysis in Small Space. *ACM Transactions on Algorithms*, 2020.

[5] Djamal Belazzougui and Gonzalo Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Transactions on Algorithms*, 2015.

[6] Jon Louis Bentley. Solutions to Klee's Rectangle Problems. *Unpublished manuscript*, 1977.

[7] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 2008.

[8] Gerth Stølting Brodal, Pooya Davoodi, and S Srinivasa Rao. Path Minima Queries in Dynamic Weighted Trees. In *Proceedings of the 12th Algorithms and Data Structures Symposium (WADS 2011)*. Springer, 2011.

[9] Timothy M. Chan. Speeding up the Four Russians Algorithm by about One More Logarithmic Factor. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*. Society for Industrial and Applied Mathematics, 2015.

[10] Timothy M Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T Wilkinson. Linear-Space Data Structures for Range Mode Query in Arrays. *Theory of Computing Systems*, 2014.

[11] Timothy M Chan, Stephane Durocher, Matthew Skala, and Bryan T Wilkinson. Linear-Space Data Structures for Range Minority Query in Arrays. *Algorithmica*, 2015.

[12] Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct Indices for Path Minimum, with Applications. *Algorithmica*, 2016.

[13] Timothy M. Chan, Qizheng He, and Yakov Nekrich. Further Results on Colored Range Searching. In *Proceedings of the 36th Annual Symposium on Computational Geometry (SoCG 2020)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[14] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal Range Searching on the RAM, Revisited. In *Proceedings of the 27th Annual Symposium on Computational Geometry (SoCG 2011)*. ACM, 2011.

[15] Timothy M. Chan and Yakov Nekrich. Better Data Structures for Colored Orthogonal Range Reporting. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*. Society for Industrial and Applied Mathematics, 2020.

[16] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS 2000)*. ACM, 2000.

[17] Bernard Chazelle. Filtering Search: A New Approach to Query-Answering. *SIAM Journal on Computing*, 1986.

[18] Bernard Chazelle. Computing on a Free Tree via Complexity-Preserving Mappings. *Algorithmica*, 1987.

[19] Bernard Chazelle and Leonidas J Guibas. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica*, 1986.

[20] David R Clark and J Ian Munro. Efficient Suffix Trees on Secondary Storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete algorithms (SODA 1996)*. Society for Industrial and Applied Mathematics, 1996.

[21] Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A New Linear-Time Algorithm for Centroid Decomposition. In *Proceedings of the 26th Annual Symposium on String Processing and Information Retrieval (SPIRE 2019)*. Springer, 2019.

[22] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On Cartesian Trees and Range Minimum Queries. *Algorithmica*, 2014.

[23] Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-Space Data Structures for Range Frequency Queries on Arrays and Trees. *Algorithmica*, 2016.

[24] Anita Dürr. Improved Bounds for Rectangular Monotone Min-Plus Product and Applications. *Information Processing Letters*, 2023.

[25] Hicham El-Zein, Meng He, J. Ian Munro, Yakov Nekrich, and Bryce Sandlund. On Approximate Range Mode and Range Selection. In *Proceedings of the 30th International Symposium on Algorithms and Computation (ISAAC 2019)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[26] Hicham El-Zein, Meng He, J Ian Munro, and Bryce Sandlund. Improved Time and Space Bounds for Dynamic Range Mode. In *Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[27] Hicham El-Zein, J. Ian Munro, and Yakov Nekrich. Succinct Color Searching in One Dimension. In *Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC 2017)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[28] Michael L Fredman and Dan E Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *Journal of computer and system sciences*, 1993.

[29] Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the 16th Annual ACM SIGACT Symposium on Theory of Computing (STOC 1984)*. ACM, 1984.

[30] Travis Gagie, Meng He, Gonzalo Navarro, and Carlos Ochoa. Tree Path Majority Data Structures. *Theoretical Computer Science*, 2020.

[31] Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J Puglisi. Colored Range Queries and Document Retrieval. *Theoretical Computer Science*, 2013.

[32] François Le Gall and Florent Urrutia. Improved Rectangular Matrix Multiplication Using Powers of the Coppersmith-Winograd Tensor. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. Society for Industrial and Applied Mathematics, 2018.

[33] Arnab Ganguly, J. Ian Munro, Yakov Nekrich, Rahul Shah, and Sharma V. Thankachan. Categorical Range Reporting with Frequencies. In *Proceedings of the 22nd International Conference on Database Theory (ICDT 2019)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[34] Younan Gao and Meng He. Space Efficient Two-Dimensional Orthogonal Colored Range Counting. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[35] Younan Gao and Meng He. Faster Path Queries in Colored Trees via Sparse Matrix Multiplication and Min-Plus Product. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[36] Younan Gao and Meng He. On Approximate Colored Path Counting. submitted, under review, 2023.

[37] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*. Society for Industrial and Applied Mathematics, 2003.

[38] Roberto Grossi and Søren Vind. Colored Range Searching in Linear Space. In *Proceedings of the 14th Scandinavian Symposium and Workshops (SWAT 2014)*. Springer, 2014.

[39] Yuzhou Gu, Adam Polak, Virginia Vassilevska Williams, and Yinzhan Xu. Faster Monotone Min-Plus Product, Range Mode, and Single Source Replacement Paths. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, (ICALP 2021)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[40] P. Gupta, R. Janardan, and M. Smid. Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *Journal of Algorithms*, 1995.

[41] Prosenjit Gupta, Ravi Janardan, Saladi Rahul, and Michiel HM Smid. Computational Geometry: Generalized (or Colored) Intersection Searching. *Handbook of Data Structures and Applications*, 2018.

[42] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Algorithms for Generalized Halfspace Range Searching and Other Intersection Searching Problems. *Computational Geometry*, 1996.

[43] Yijie Han. Deterministic Sorting in O(n log log n) Time and Linear Space. In *Proceedings of the 34th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2002)*, 2002.

[44] Meng He and Serikzhan Kazi. Data Structures for Categorical Path Counting Queries. *Theoretical Computer Science*, 2022.

[45] Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct Ordinal Trees Based on Tree Covering. *ACM Transactions on Algorithms*, 2007.

[46] Meng He, J. Ian Munro, and Gelin Zhou. Dynamic Path Counting and Reporting in Linear Space. In *Proceedings of the 25th International Symposium on Algorithms and Computation (ISAAC 2014)*. Springer, 2014.

[47] Meng He, J. Ian Munro, and Gelin Zhou. A Framework for Succinct Labeled Ordinal Trees over Large Alphabets. *Algorithmica*, 2014.

[48] Meng He, J. Ian Munro, and Gelin Zhou. Data Structures for Path Queries. *ACM Transactions on Algorithms*, 2016.

[49] Meng He, J Ian Munro, and Gelin Zhou. Dynamic Path Queries in Linear Space. *Algorithmica*, 2018.

[50] David Hutchinson, Anil Maheshwari, and Norbert Zeh. An External Memory Data Structure for Shortest Path Queries. *Discrete Applied Mathematics*, 2003.

[51] Joseph JáJá, Christian W Mortensen, and Qingmin Shi. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC 2004)*. Springer, 2004.

[52] Ravi Janardan and Mario Lopez. Generalized Intersection Searching Problems. *International Journal of Computational Geometry & Applications*, 1993.

[53] Ce Jin and Yinzhan Xu. Tight Dynamic Problem Lower Bounds from Generalized BMM and OMv. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2022)*. ACM, 2022.

[54] Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1869.

[55] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient Colored Orthogonal Range Counting. *SIAM Journal on Computing*, 2008.

[56] Haim Kaplan and Nira Shafrir. Path Minima in Incremental Unrooted Trees. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008)*. Springer, 2008.

[57] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored Intersection Searching via Sparse Rectangular Matrix Multiplication. In *Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG 2006)*, 2006.

[58] Danny Krizanc, Pat Morin, and Michiel Smid. Range Mode and Range Median Queries on Lists and Trees. *Nordic Journal of Computing*, 2005.

[59] Ying Kit Lai, Chung Keung Poon, and Benyun Shi. Approximate Colored Range and Point Enclosure Queries. *Journal of Discrete Algorithms*, 2008.

[60] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017.

[61] J. Ian Munro, Yakov Nekrich, and Sharma V. Thankachan. Range Counting with Distinct Constraints. In *Proceedings of the 27th Canadian Conference on Computational Geometry (CCCG 2015)*. Queen's University, Ontario, Canada, 2015.

[62] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*. Society for Industrial and Applied Mathematics, 2002.

[63] Yakov Nekrich. Efficient Range Searching for Categorical and Plain Data. *ACM Transactions on Database Systems*, 2014.

[64] Manish Patil, Rahul Shah, and Sharma V Thankachan. Succinct Representations of Weighted Trees Supporting Path Queries. *Journal of Discrete Algorithms*, 2012.

[65] Mihai Pătraşcu. Lower Bounds for 2-Dimensional Range Counting. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC 2007)*. ACM, 2007.

[66] Saladi Rahul. Approximate Range Counting Revisited. *Journal of Computational Geometry*, 2021.

[67] Kunihiko Sadakane. Succinct Data Structures for Flexible Text Retrieval Systems. *Journal of Discrete Algorithms*, 2007.

[68] Bryce Sandlund and Yinzhan Xu. Faster Dynamic Range Mode. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[69] Virginia Vassilevska Williams and Yinzhan Xu. Truly Subcubic Min-Plus Product for Less Structured Matrices, with Applications. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*. Society for Industrial and Applied Mathematics, 2020.

[70] Huacheng Yu. An Improved Combinatorial Algorithm for Boolean Matrix Multiplication. *Information and Computation*, 2018.