DATA STRUCTURES FOR GEOMETRIC RETRIEVAL IN TREE-LIKE TOPOLOGIES

by

Serikzhan Kazi

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

at

Dalhousie University
Halifax, Nova Scotia
December 2020

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
1.2 Our Techniques 5

Chapter 2 Preliminaries **8**

2.1 Notation 8
2.1.1 Notation and Conventions 8
2.1.2 Graph-Theoretic Notation 9
2.1.3 Word-RAM Model and Information-Theoretic Lower Bound 9
2.2 Bitvectors and Sequences 10
2.2.1 Bitvectors 10
2.2.2 Sequences 11
2.3 Trees 12
2.3.1 Tree Extraction 12
2.3.2 Succinct Representations of Ordinal Trees 13

Chapter 3 Path and Ancestor Queries over Trees with d-Dimensional Weight Vectors **16**

3.1 Introduction 16
3.1.1 Previous Work 17
3.1.2 Our Results 19
3.2 Preliminaries 20

3.2.1 Notation . . . 20
3.2.2 Representation of a Range Tree on Node Weights by Hierarchical Tree Extraction . . . 20
3.2.3 Path Minimum in 1D-Weighted Trees . . . 22
3.3 Reducing to Lower Dimensions . . . 22
3.3.1 Space Reduction Lemma Using Range Trees with Branching Factor 2 23
3.3.2 Space Reduction Lemma Using Range Tree with a Non-Constant Branching Factor . . . 25
3.4 Ancestor Dominance Reporting . . . 26
3.4.1 Path Dominance in $(1, d, \epsilon)$ . . . 27
3.4.2 2D Ancestor Dominance Reporting . . . 29
3.4.3 Ancestor Dominance Reporting in $(2, d, \epsilon)$ and Generalization to Higher Dimensions . . . 36
3.5 Path Successor . . . 37
3.6 Path Counting . . . 42
3.7 Path Reporting . . . 45
3.8 Conclusion . . . 46

Chapter 4 Data Structures for Categorical Path Counting Queries . . . 48

4.1 Introduction . . . 48
4.1.1 Previous Work . . . 50
4.1.2 Our Results . . . 51
4.2 Categorical Path Counting . . . 52
4.2.1 Hardness of Categorical Path Counting . . . 52
4.2.2 Uniform Partitioning of the Tree . . . 54
4.2.3 Categorical Path Counting . . . 56
4.2.4 2-Approximate Categorical Path Counting . . . 57
4.3 Categorical Path Range Counting . . . 58
4.3.1 Path Range Emptiness Queries . . . 59
4.3.2 Categorical Path Range Counting in d Dimensions . . . 59
4.4 Sketching Data Structures for Approximate Categorical Path Counting . . . 64
4.4.1 Sketches . . . 64
4.4.2 $(1 \pm \epsilon)$-Approximate Categorical Path Range Counting . . . 65
4.5 Conclusion . . . 67

Chapter 5 Path Query Data Structures in Practice . . . 69

5.1 Introduction . . . 69
5.1.1 Previous Work . . . 70
5.1.2 Our Results . . . 71
5.2 Preliminaries . . . 72
5.2.1 Balanced Parentheses Representations of Ordinal Trees . . . 72
5.2.2 Heavy-Path Decomposition . . . 74

5.3 Data Structures for Path Queries 75
5.3.1 Data Structures Based on Heavy-Path Decomposition 75
5.3.2 Data Structures Based on Tree Extraction 77
5.4 Experimental Results 83
5.4.1 Implementation 83
5.4.2 Experimental Setup 84
5.4.3 Space Performance and Construction Costs 86
5.4.4 Path Median Queries 88
5.4.5 Path Counting Queries 90
5.4.6 Path Reporting Queries 90
5.5 Conclusion 93

Chapter 6 Conclusion **94**

6.1 Results and Discussion 94
6.2 Future Directions 96

Bibliography **99**

List of Tables

2.1 Primitive operations in trees. 14

3.1 Summary table for the some of the data structures in Lemma 3.8. . . . 33

5.1 Datasets description. 84

5.2 The implemented data structures. 85

5.3 Storage space and construction characteristics. 87

5.4 Average time of a path median or a path counting query, in μs. 88

5.5 Average time of a path reporting query, in μs. 91

List of Figures

1.1 An example of a weighted tree. The weights are shown inside the circles. 3

2.1 A wavelet tree for $\mathtt{S} = \mathtt{ecgbcahd}$ and alphabet $[\mathtt{a}, \mathtt{h}]$. 11

2.2 An edit operation during tree extraction: removing the node y and connecting all its children, in order, to y's parent, node p. 12

2.3 Tree extraction. 13

2.4 Tree covering of Farzan and Munro [39], with parameter $L = 5$. . . . 15

3.1 A fragment of hierarchical tree extraction. 21

3.2 An illustration to the proof of Lemma 3.4. 24

3.3 The 2-maximal set in a tree T weighted over $\sigma = 10$. 29

3.4 Hierarchical tree extraction with branching factor 2. 31

3.5 An illustration to the proof of Lemma 3.8. 35

3.6 An illustration to the proof of Lemma 3.10. 39

4.1 Boolean matrix multiplication and categorical path counting. 53

4.2 An illustration to the proof of Lemma 4.2. 55

4.3 Disjoint decomposition of the rim in 2D. 62

5.1 An ordinal tree with its BP sequence. 73

5.2 Heavy paths in a tree. 74

5.3 Original tree T and transformed tree T' encoding HPD. 76

5.4 An illustration to the procedure of finding the t-parent of v. 79
5.5 Average time to answer a query *vs* number of segments in HPD. . . . 89
5.6 Visualization of some of the entries in Table 5.4. 92

Abstract

This thesis studies the generalizations of orthogonal range searching to the case in which one of the dimensions is replaced with a tree topology. We design data structures for efficient support of *path queries*, which generalize the corresponding range queries. We work in the word-RAM model with word size $w = \Omega(\lg n)$.

Let T be an ordinal tree on n nodes, each of which is assigned a d-dimensional *weight vector* from $[n]^d = \{1, 2, \ldots, n\}^d$, where d is a constant integer. We solve: (i) *ancestor dominance reporting*, in $\mathcal{O}(\lg^{d-1} n + k)$ time and $\mathcal{O}(n \lg^{d-2} n)$ space (where k is the size of the output, and $d \geq 2$). Also achieved is a trade-off of $\mathcal{O}(\lg^{d-1} n/(\lg\lg n)^{d-2} + k)$ time and $\mathcal{O}(n \lg^{d-2+\epsilon} n)$ space ($d \geq 3$), for an arbitrary constant $\epsilon > 0$; (ii) *path successor*, in $\mathcal{O}(\lg^{d-1+\epsilon} n)$ time and $\mathcal{O}(n \lg^{d-1} n)$ space ($d \geq 1$); (iii) *path counting*, in $\mathcal{O}((\lg n/\lg\lg n)^d)$ time and $\mathcal{O}(n(\lg n/\lg\lg n)^{d-1})$ space ($d \geq 1$); (iv) *path reporting*, in $\mathcal{O}(\lg^{d-1} n/(\lg\lg n)^{d-2} + k)$ time and $\mathcal{O}(n \lg^{d-1+\epsilon} n)$ space ($d \geq 2$). All these bounds match or nearly match the best bounds in corresponding range queries.

Next, we study trees in which each node is also assigned a *category*. For unweighted trees, we solve *categorical path counting*, in $\mathcal{O}(t \lg\lg_w \sigma)$ time and $\mathcal{O}(n + (n/t)^2)$ space, for fixed $1 \leq t \leq n$, implying linear space for $\mathcal{O}(\sqrt{n} \lg\lg_w \sigma)$ time. By a reduction from Boolean matrix multiplication, we show that the time is optimal within polylogarithmic factors, with current knowledge and only combinatorial methods. For weighted trees, we solve *categorical path range counting* in $\mathcal{O}(t \lg\lg n)$ time and $\mathcal{O}(n \lg\lg n + (n/t)^4)$ space, or $\mathcal{O}(t \lg^\epsilon n)$ time and $\mathcal{O}(n + (n/t)^4)$ space, which implies linear space for $\mathcal{O}(n^{3/4} \lg^\epsilon n)$ time. The solution is extended to the trees weighted with vectors from $[n]^d$, $d \geq 2$, with time- and space-bounds which are within polylogarithmic factors of the best bounds for point-sets. We also solve approximate variations of these categorical queries.

Finally, we experimentally study the data structures for path median, path counting and path reporting queries. We propose practical realizations of the best theoretical results, and provide both succinct and pointer-based implementations. Our experiments show the viability of the former in practical scenarios.

Acknowledgements

I would like to thank my supervisor, Dr. Meng He, for honest feedback, for being generous with his time and expertise, and for having highest expectations of me. It is to his guidance that I owe my so-far proudest professional accomplishments. But first of all, I thank him for having the faith to embark on this four-year-long journey.

I thank my Thesis Committee members, Dr. Norbert Zeh and Dr. Travis Gagie, for the thoughtful comments on the thesis proposal, and for making me take heart again (and succeed) in a project that I deemed dead-end.

I thank my Aptitude Defense committee members, Dr. Nauzer Kalyaniwalla and Dr. Qigang Gao for the sincere interest they took in my research, and for the enlightening discussions we had in the course of these years.

Dr. Vlado Keselj engaged my former-teacher self by entrusting me with leading TA roles in his courses and involving me with the Dal competitive programming team. For all the conversations about computer programming, algorithmic puzzles, and for the unwavering confidence in my pedagogical abilities – *thank you*.

I thank the administrative staff of the Faculty of Computer Science for taking immaculate care of the practical side of things. Big thanks are due to CS Help Desk staff, who went beyond the call of duty in resolving technical issues and accommodating my requests (which I tried to keep moderate).

Thanks also goes to my colleagues and Halifax-based friends – you know who you are!

Finally, I thank my family, for the time they generously granted me to realize my professional goals – many an early morning and holiday season that I spent working were rightfully yours.

Chapter 1

Introduction

In orthogonal range searching, one preprocesses a given finite set $S \subset \mathbb{R}^d$ into a data structure so that the points inside an axis-aligned query (hyper-)rectangle can be efficiently searched. Theoretical computer science owes remarkable new methodologies and elegant data structures to the field (see [74, 21, 3] and references therein). Orthogonal range searching is among those theoretical disciplines whose results enjoy deployment at industrial scale. Any textbook on relational database management systems (RDBMS) is sure to include a question similar to "*Give me the number of the employees with age between* `x` *and* `y`*, earning between* `$A` *and* `$B` *per year*". The latter is an instance of a 2D *range counting query* in the field's parlance; the domain of each of the two characteristics (*age* and *income*) is a linear order, and therefore the query is fully defined by the lower and upper bounds on the relevant values.

At the same time, natural extensions of these types of queries to the case in which one of the dimensions is a tree topology, are also conceivable [24, 6, 77]. There being a unique path between any two given nodes in the tree, if a pair of nodes is specified instead of the lower and upper limits of a linear order, the query is still well-defined. On the other hand, when tree degenerates into a single path, the corresponding problems become identical.

Whereas grids may be our primary heuristic for imposing structure, a tree-shaped hierarchy is also a versatile data organization tool. Trees are ubiquitous across disciplines. For example, there are phylogenies in bioinformatics, abstract syntactic trees in translation and compilers, inheritance hierarchies in object-oriented programming (OOP), and mark-up formats such as XML. Linguists study semantic similarities between terms by querying an *is-a* relationship tree [99] such as WordNet. In parallel computing, it is known that for any number A, message passing in any *area A* network can be simulated with a certain tree on-line with slowdown of only $\mathcal{O}(\lg A)$[1] with high probability [80]. Finally, there are minimum spanning trees, which non-redundantly capture connectivity information in graphs.

It is common that trees, apart from conveying a hierarchical or connectivity information, also carry meaningful labels (weights) on their nodes. For example, in data mining, the nodes in decision trees are naturally associated with the information gain, gain ratio, or Gini index value that arises in the splitting process [63]. In the previous example of inheritance hierarchies in OOP, a label can mark whether the corresponding derived class overrides a given abstract method. Phylogenetic trees may contain *molecular clock*-related information at split junctures. Mark-up formats are typically composed of various *tags*.

To capture even more aspects of the underlying phenomenon, the nodes of the tree can further be weighted not with a single weight or label, but rather with a d-dimensional ($d \in \mathbb{N}$) *vector* of weights. Endowing the tree nodes with *categories* (or *colours*) further adds to the expressive power of the models. All these scenarios generalize (multidimensional) orthogonal point-sets and point-sets with categories, in $\mathbb{R}^{d+1}$. Preprocessing of such input trees for efficient geometric retrieval is the topic of this thesis.

The problem of preprocessing a weighted tree to support various *path queries* has been studied extensively [6, 24, 62, 77, 32, 70, 19]. Given a query path $P_{x,y}$ and a query rectangle Q, a path query evaluates a certain function on those nodes of $P_{x,y}$ whose weights belong to Q. For example, in *path counting* (resp. *path reporting*), the nodes of the given path with weights lying in the given query interval are counted (resp. reported).

In Figure 1.1, two nodes, x and y, are marked and used in the running example that follows. A path counting query with arguments $P_{x,y}$ and $Q = [7, 9]$ would return a number 1, as the only node on $P_{x,y}$ with the weight inside $[7, 9]$ is the first child of the root. A path reporting query with arguments $P_{x,y}$ and $Q = [1, 4]$ would return the root and its second child as only the weights of these nodes (resp. 2 and 3) fall within Q.

Research on path queries has spawned widely-used metrics such as the median [70], minimum/maximum [32, 19], mode/minority [36], and (α-)majority/minority [45].

This thesis, too, studies data structures for path queries. It differs from previous work on a

[1]We set $\lg x = \log_2 x$, unless the base is explicitly specified.

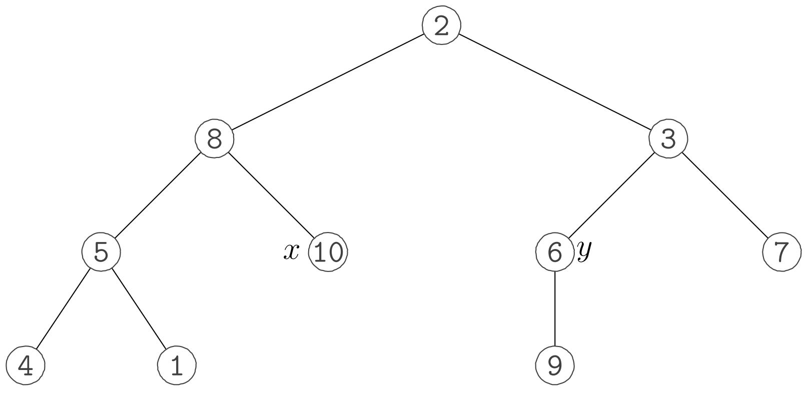

Figure 1.1: An example of a weighted tree. The weights are shown inside the circles.

few significant counts. First, the trees themselves are "multidimensional" in the sense that a node in a tree is associated with not just a single weight, but rather with an entire *vector* (tuple) of weights. Second, we are the first to study path query data structures on trees that are not only "multidimensional" in the preceding sense, but can also be "coloured" in the sense that each node is assigned a category. Third, we are the first to study some of the path queries even for regular weighted trees (i.e. when $d = 1$), such as the path successor and the categorical path range counting queries. Finally, we are also the first to experimentally evaluate the currently best theoretical proposals in data structures for path queries in weighted trees.

1.1 Organization of the Thesis

This thesis is organized as follows.

Chapter 2 offers some background on the discipline. We first agree on notation and conventions used throughout the work. We proceed by reviewing the fundamental building blocks for our solutions. The topics reviewed range from well-known techniques such as efficient storage and navigation in sequences and trees, to relatively recent ones such as succinct tree representations and tree extractions.

Chapter 3 studies path queries over trees with multidimensional weight vectors on nodes, and presents a set of results. We solve the path counting problem, where the number of nodes lying on a given query path, with weight vectors belonging to a given query range, is to be returned; we also solve the path reporting problem, where such nodes are to be explicitly enumerated. We also describe a solution to the path successor problem; it asks for a node (on the query path) with the weight vector lying in the query range, and such that the first component of the weight vector is the smallest. Finally, we propose a solution to the ancestor dominance reporting problem, where ancestors of a query node, with weight vectors dominating that of the query node, are to be reported. This chapter is based on part of the joint work with Meng He [64].

Chapter 4 then studies the generalization of categorical counting to tree topologies. We solve the categorical path counting problem; it asks to count the number of distinct categories occurring on a query path. We demonstrate the hardness of the categorical path counting problem in unweighted trees by a reduction from the Boolean matrix multiplication problem. We mitigate the limitations posed by the thus established lower bound by solving an approximate variant of the problem. Although Durocher et al. [36] introduced and solved for trees the *reporting* version of the problem, we believe that neither the counting type thereof, nor the approximate scenario has been studied before.

We further study the categorical path range counting problem, which now presents us not only with a query path, but also a weight range in the form of an axis-aligned (hyper-)rectangle. The query asks for the number of distinct categories on a query path, if only the nodes with weights in the query rectangle are considered. In this weighted setting, we also describe an approach based on *sketches*, whose properties make them desirable for use as (approximate) summaries for categorical queries, at the cost of a certain failure probability. This chapter, too, is based on part of the joint work with Meng He [66].

Furthermore, Chapter 5 considers the details of the practical implementation of several data structures supporting path counting, path reporting, and path selection queries, and evaluates the performance thereof on some interesting datasets. Here, the path selection problem asks for the node with the k^{th} (specified at query time) largest weight, on the given path. Described are our experimental framework, the tools used, and key design choices made. We measure both the space occupancy and the query times of the data structures we implement. This chapter is also based on part of the joint work with Meng He [65].

Chapter 6 concludes the thesis by stating the obtained results, highlighting the techniques we have developed. In a few concrete examples, we weigh in on the difficulties of porting the existing approaches from point-sets to trees. We also discuss some open problems and possible future research directions.

1.2 Our Techniques

Central to our techniques is the notion of *tree extraction* [70]. It allows for "subsetting" the tree while retaining the partial order of the nodes. (Indeed, tree is a partial order in the sense that two nodes x and y are in relation *iff* x is an ancestor of y.) The correspondence between the nodes in the "host" tree and those in its extraction is the key to our approaches.

Going further still, we use *hierarchical* tree extraction of He et al. [70] to build, essentially, the wavelet tree of a weighted tree. It is the key to our solutions in Chapter 3, and we also use it in the actual succinct implementations in Chapter 5. Apart from many advantages (e.g. the mapping infrastructure discussed in Section 3.2.2), one major plus of hierarchical tree extraction is simplified space analysis. Indeed, the sum of the sizes of disparate data structures at each level of extraction may not be as clear-cut as we would like, because of the lower-order terms that are added together non-constant number of times.

In most use-cases of tree extraction, one stores the mapping structures explicitly. In solving the path successor problem (Section 3.5) for a tree T weighted over $\{1, 2, \ldots, n\}$, however, we needed to search for the path maximum/path minimum on the query path, with an additional requirement that only the nodes whose weights are in a certain given range are considered. (Indeed, the path maximum/path minimum problem proper, i.e. without the latter non-trivial requirement, has already been solved [32, 19].) We solve this problem for the special case when the ranges Q considered are the ranges associated with the nodes of the range tree over $\{1, 2, \ldots, n\}$. The crux is that for such a range r (associated with a node) of the range tree, we do not require an explicit mapping structure between the extraction of the r-weighted nodes of T, and T itself. Solving this sub-problem was sufficient to find the path successor via binary search.

For path counting in d dimensions, we use an another technique – *tree covering* (Section 2.3.2). We recursively decompose the tree into *mini-* and *micro-*trees (collectively referred to as *cover elements*). The answers for micro-trees are tabulated, and the roots of the cover elements store the answers up-to the roots of the respective encompassing cover elements (the "global" input tree for mini-trees, and a mini-tree for micro-trees). Here, a careful choice of the parameters of tree covering and succinct storage of the pre-computed answers was the key to space-efficiency.

Furthermore, for the categorical version of path counting, we apply an interesting tree mark-up technique (Lemma 4.1; see [36] and references therein). This technique helps to balance pre-computation versus explicit on-the-fly computation, via a trade-off parameter t.. Namely, the marked nodes are akin to block borders in the sense that any path $P_{x,y}$ that is longer than t can be decomposed into sub-paths $P_{x,x'}$, $P_{x',y'}$, and $P_{y,y'}$, where x' and y' are marked, $\max\{|P_{x,x'}|, |P_{y,y'}|\} \leq t$. The answer to the query between any two marked nodes is pre-computed and stored in a table, and the flanking ends of the path are explicitly traversed, refining the overall answer. Not to be deceived by its simplicity, the technique is applicable only if the query answer for $P_{x',y'}$ stays the same regardless of the "context" – the nodes x and y.

Our approach to categorical path (range) counting is similar to that used in the best categorical results for $\mathbb{R}^d$, for $d \geq 2$ [76]. Again, after pair-to-pair pre-computations, we walk the flanking ends of the query path and find out whether a certain category has already been accounted for by a range emptiness query in a monochromatic region. Here, we use either the best results in path range emptiness or labeled ancestor queries.

Recall that for categorical counting, in contrast to the "plain" variant, simple-mindedly adding the counts of the parts does not yield the count of the whole. (Indeed, therein lies the intuitive "hardness" of categorical counting, which we formalize in Theorem 4.1 for paths.) In approximate categorical path counting, we use a probabilistic technique called *sketches*, which allows one to do exactly that (Lemma 4.8). At the logarithmic cost per operation, logarithmic storage, a certain failure probability δ, and a certain tolerance of ϵ to deviations around the true answer, one can maintain a *summary*. A summary is a probabilistic variant of a "count" in the sense that the summaries for two sets can be added to obtain the summary for their set union, and can be subtracted to obtain summaries for their set difference. Importantly, the parameter ϵ remains intact in such additions and subtractions.

Further, in approximate categorical counting, we use the fact that summaries can be updated in logarithmic time to obtain a space/time trade-off. Namely, we store summaries for only a fraction of the nodes of the tree, and recover other summaries relevant to the query at query time proper.

We extend most of our data structures to higher dimensions (i.e. to trees weighted with multidimensional weight vectors) using the framework we develop in Section 3.3.1 of Chapter 3. Once the base data structure is described, our job reduces to an appropriate choice of the semigroup with the corresponding sum operator.

For our experimental studies, we design a practical variant of the time- and space-optimal data structure of He et al. [70]. Instead of tree covering and hierarchical tree extraction with non-constant branching factors, which are likely to have large constant factors in practice, we use the balanced parentheses (BP) representation of trees, which has been extensively studied [7] and is readily available in well-known libraries such as `sdsl-lite`. This results in a three-fold increase in space and a sub-logarithmic increase in query time (with respect to the optimal solution of [70]), but offers attractive trade-offs nevertheless, compared to the other data structures studied.

Chapter 2

Preliminaries

This chapter lays the groundwork for subsequent chapters. Section 2.1 introduces notation and basic terminology; its contents are rather conventional and intuitive, and therefore can be consulted as needed. Sections 2.2 and 2.3 then offer background material of a more specialized nature. Therein, we review previous results at the core of our toolset: storage and navigation in sequences (Section 2.2) and trees (Section 2.3).

2.1 Notation

This section gives the notation we operate with, conventions we adopt, as well as the background knowledge on the general concepts we rely on.

2.1.1 Notation and Conventions

Whenever possible, we try to follow standard notation. We assume base-2 logarithm, denoted by lg, whenever no explicit base is given, and denoted as $[n]$ the set $\{1, 2, \ldots, n\}$, for any $n \in \mathbb{N}$. Also, $[a, b]$ denotes the range of integers $\{a, a+1, \ldots, b\}$. The terms *succinct* and *compact* are used interchangeably. A *sequence* or *string* of length n drawn from the alphabet $[\sigma]$ is a tuple from Cartesian product $[\sigma]^n$, with its i^{th} coordinate denoted by $S[i]$. Landau symbols $\mathcal{O}/o/\Omega/\omega/\Theta$ are defined as elsewhere in the literature [27], except for $\widetilde{\mathcal{O}}$, which is used when leaving out polylogarithmic factors. The primitive data structures we use as building blocks are regarded as black-box interfaces with certain time/space guarantees; for the detailed exposition of their inner workings, an interested reader is encouraged to refer to more comprehensive sources on compact data structures, such e.g. as a recent book by Navarro [91].

Given a d-dimensional weight vector $\mathbf{w} = (w_1, w_2, \ldots, w_d)$, we define the vector $\mathbf{w}_{i,j}$ to be $(w_i, w_{i+1}, \ldots, w_j)$. We extend the definition to a range $Q = \prod_{i=1}^{d}[q_i, q'_i]$ by setting $Q_{i,j} = \prod_{k=i}^{j}[q_k, q'_k]$. Furthermore, a 3D orthogonal range is denoted as 3-*sided iff* it is bounded one one side only, at each dimension. In general, a range is $(3+t)$-*sided* if it is bounded on both sides for t dimensions, and unbounded on one side for the remaining $3-t$ dimensions; the definitions are analogous for any other dimensionality.

For brevity, we shall also use *Iverson notation* [55]: For a Boolean predicate P, the symbol $[\![P]\!] \in \{0,1\}$ equals 1 *iff* $P = \texttt{true}$. A sequence of objects $I_1, I_2, \ldots, I_k$ is denoted as $\{I_j\}_{j=1}^{k}$.

2.1.2 Graph-Theoretic Notation

We denote by $|T|$ the size (i.e. the number of nodes) of the tree T, whose set of nodes is denoted as $V(T)$. For brevity, we write $x \in T$ to denote $x \in V(T)$, if no confusion ensues. The path between the nodes $x, y \in T$ is denoted as $P_{x,y}$, both ends inclusive. We write $P_{x,y} \subseteq T$ to indicate that a path belongs to a tree. During a preorder traversal of a given tree T, the i^{th} node visited is said to have *preorder rank* i. Preorder ranks are commonly used to identify tree nodes in various succinct data structures which we use as building blocks. Thus, we also identify a node by its preorder rank, i.e. node i in T is the node with preorder rank i in T. For a node $x \in T$, its set of ancestors, denoted as $\mathcal{A}(x)$, includes x itself; $\mathcal{A}(x) \setminus \{x\}$ is then the set of the *proper ancestors* of x. Given two nodes $x, y \in T$, where $y \in \mathcal{A}(x)$, we set $A_{x,y} \triangleq P_{x,y} \setminus \{y\}$. Thus $P_{x,y} = A_{x,z} \sqcup \{z\} \sqcup A_{y,z}$, for any $x, y \in T$ and z being the lowest common ancestor [13] of x and y. Further, we denote by $\perp$ the root of the relevant tree; therefore, $P_{x,\perp}$ stands for the path from the root to the given node x. Finally, when the nodes of the tree T are associated with scalar weights, we say that T is 1D-*weighted*, or simply *weighted*; both terms are used interchangeably throughout the work.

2.1.3 Word-RAM Model and Information-Theoretic Lower Bound

Word-RAM model of computation emulates the operations available on modern CPUs. A memory unit is a *word* of $w = \Omega(\lg n)$ bits; basic arithmetic operations plus bitwise logical operations (e.g. negation, `AND`, `OR`, left/right shifts) are assumed to be executed in $\mathcal{O}(1)$ time on machine word-size operands. In addition, any location addressed by a machine word is fetched in constant time. In practical terms, this model's convenience lies in its ability to tabulate pre-computed answers to certain queries, [1] when the relevant entity fits in a machine word. For example, in [12] the answers to all possible ± 1 range minima queries (i.e. when the input array's successive entries differ by ± 1) inside blocks of size $\frac{\lg n}{2}$ are stored in a table of size $\mathcal{O}(2^{\lg n/2}\lg^2 n \lg\lg n) = \mathcal{O}(\sqrt{n}\lg^2 n\lg\lg n) = o(n)$ words, to be answered in $\mathcal{O}(1)$ time.

Information theory provides a framework for analyzing space occupancy of data structures. The information-theoretic lower bound is the number of bits needed to represent an object, with no *a priori* knowledge. For example, a sequence $S \in [\sigma]^n$ can be represented in $n\lg\sigma$ bits by simply encoding each symbol from $[\sigma]$ in binary. As trivially follows from the Pigeonhole Principle, no shorter uniform-length codes would guarantee unique interpretation of an arbitrary sequence. On the other hand, this barrier is crossed through the notion of *entropy*, to achieve shorter *expected* lengths via variable-length codes [29]. Entropy of a sequence $S \in [\sigma]^n$ is defined as

$$H_0(S) = -\sum_{c\in[\sigma]} n_c/n \cdot \lg\,(n_c/n),$$

and can be shown to be at most $\lg\sigma$:

$$H_0(S) = \sum_{c\in[\sigma]} n_c/n \cdot \lg\,(n/n_c) \le \lg\Big(\sum_{c\in[\sigma]}(n_c/n\cdot n/n_c)\Big) = \lg\sigma,$$

by Jensen's inequality. In practical terms, compression of S to $nH_0(S)$ (plus negligible terms) generally improves space requirements as compared to straightforward binary encoding.

2.2 Bitvectors and Sequences

This section reviews compact storage and support of core primitives for bitvectors and sequences.

2.2.1 Bitvectors

In the context of this thesis, bitvector B is a data structure occupying $n + o(n)$ bits of space to encode a (static) subset of $[n]$, and supporting the following primitives in $\mathcal{O}(1)$ time:

- `access`(B, i) return $B[i]$;

[1]i.e. the so-called "Four Russians Speedup" [60]

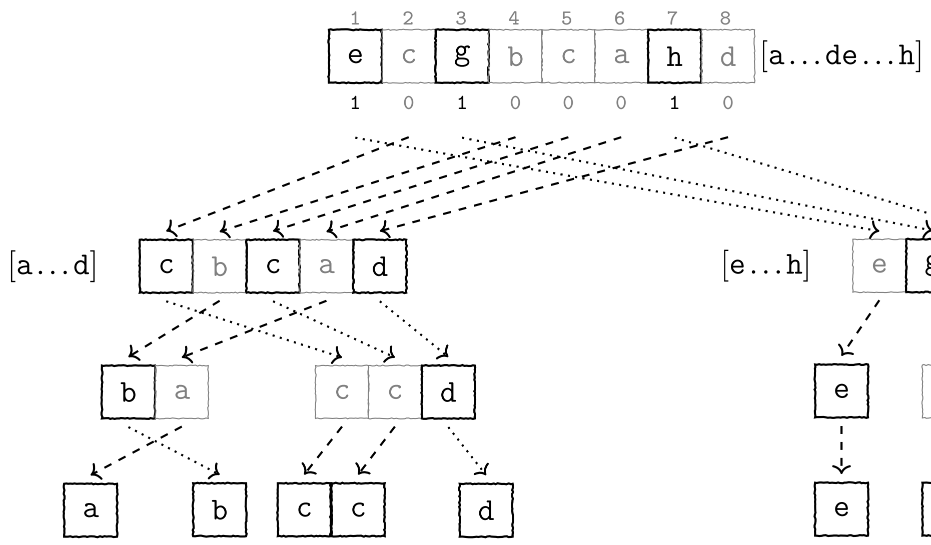

FIGURE 2.1: A wavelet tree for $\mathtt{S} = \mathtt{ecgbcahd}$ and alphabet $[\mathtt{a}, \mathtt{h}]$; left subtree is for range $[\mathtt{a}, \mathtt{d}]$. The symbols c, b, c, a, and d follow the left path (shown with dashed arrows), while e, g, and h follows the right path (dotted arrows). The 0/1-bitvector stored in each node marks which symbol follows which path. The symbols retain their original order.

- $\mathtt{rank}_t(B, i)$ return the number of $t \in \{0, 1\}$ in $B[1, i-1]$, i.e. $\mathtt{rank}_t(B, i) = \sum_{j=1}^{i-1} [\![B[j] = t]\!]$;
- $\mathtt{select}_t(B, i)$ return the position of the i^{th} $(1 \leq i \leq n)$ occurrence of $t \in \{0, 1\}$.

The central result concerning bitvectors is the following

LEMMA 2.1 ([101]). *Let $B[1..n]$ be a bitvector with m 1-bits. Then B can be represented in $\lg \binom{n}{m} + \mathcal{O}(n \lg \lg n / \lg n)$ bits to support **rank**, **select** and **access** in $\mathcal{O}(1)$ time.*

To simplify notation, we assume a bitvector is a data structure occupying $n + o(n)$ bits of space, although generally $\lg \binom{n}{m} \leq n$.

2.2.2 SEQUENCES

An arbitrary sequence $S \in [\sigma]^n$ can be stored in space $nH_0(S) + o(n \lg \sigma)$ bits, supporting the above primitives, now for arbitrary $c \in [\sigma]$, in $\mathcal{O}(\lg \sigma)$ time, in a structure known as *wavelet tree* [90]. Intuitively, given an invariant that a sequence S consists of numbers in the range $[a, b]$, wavelet tree proceeds by splitting S into two complementary sub-sequences according to whether the value falls into $[a, m]$ or $[m+1, b]$, where $m = \lfloor \frac{a+b}{2} \rfloor$, and recursively building wavelet trees for the sub-sequences. Figure 2.1 shows an example of a wavelet tree storing

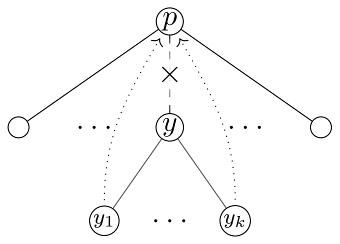

Figure 2.2: An edit operation during tree extraction: removing the node y and connecting all its children, in order, to y's parent, node p.

the sequence $S =$ `ecgbcahd`. At the topmost level, the alphabet $[a, h]$ is partitioned into $[a, d] \bigcup [e, h]$, and a bitvector encodes which sub-tree inherits the corresponding element of the sequence (0 for left, 1 for right). The actual sequences are given for illustrative purposes only; stored is only a bitvector. Each level is thus nothing but a collection of bitvectors; a `rank/access` is resolved by a top to bottom, and `select` – by a bottom-up pass on the tree.

2.3 Trees

This section gives the background on more specialized concepts and data-structural components in our solutions: compact storage of trees, fast navigation therein, and various decompositions.

2.3.1 Tree Extraction

Tree extraction [70] selects a subset of nodes while preserving the relative preorder ranks, as well as the hierarchical relations among the nodes. Precisely, given a subset $X \subseteq V$ of nodes (X is called the *extracted nodes*), the *extracted tree* T_X is constructed from T as follows. Fix an arbitrary node $y \notin X$, and let $p \in T$ be the parent of y (see Figure 2.2 for intuition on the next few sentences). Let y be the i^{th} child of p, in preorder. Let us erase, from T, the node y together with its incident edges. This frees the i^{th} slot in the list of children of p, as well as the children $y_1, y_2, \ldots, y_k$ of the node y. Then y_1 becomes the i^{th} child of p, y_2 becomes its $(i+1)^{st}$ one, and so on, until y_k becomes p's $(i+k-1)^{st}$ child. The node that was the $(i+1)^{st}$ child of p prior to deletion becomes the $(i+k)^{th}$ child of p, i.e. all the initial children occurring after the i^{th} are shifted to k positions to the right. After erasing all the nodes $y \notin X$ in the described way, the resulting forest F_X is either a tree (in which case we do nothing), or a forest, in which case we create a dummy root r (with preorder rank and depth set to 0) that becomes the parent of all the roots of the trees in F_X, again preserving the relative preorder ranks of the roots. See Figure 2.3 for an example of tree extraction.

An original node $x \in X$ of T and its copy, x', in T_X are said to *correspond* to each other; x' is also said to be the T_X*-view* of x, and x is the T*-source* of x'. The T_X-view of a node $y \in T$ (y is not required to be in X) is more generally defined to be the node $y' \in T_X$ corresponding to the lowest ancestor of y that has been extracted, i.e. to the lowest node in $\mathcal{A}(y) \cap X$. Figure 2.3 also gives examples on the sources, views, and the correspondence of nodes.

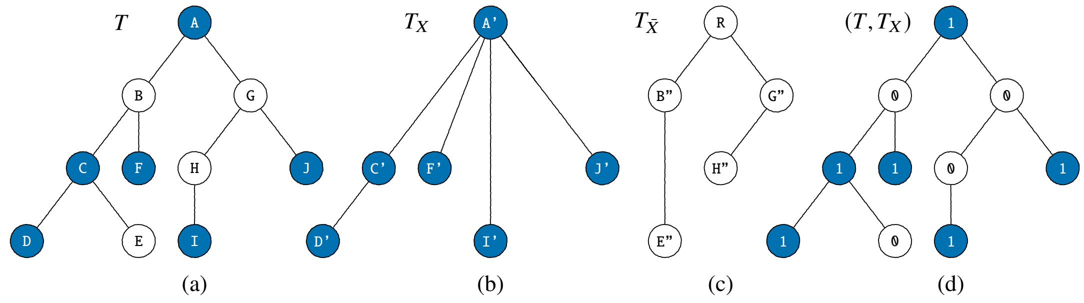

FIGURE 2.3: Tree extraction. Original tree (a), extracted tree T_X (b), extraction of the complement of X, tree $T_{\bar{X}}$ (c) and the indicator tree (T, T_X) (d). The blue shaded nodes in T form the set X. In the tree T_X, node `C'` corresponds to node `C` in the original tree T, and node `C'` in the extracted tree T_X is the T_X-view of nodes `C` and `E` in the original tree T. Finally, node `C` in T is the T-source of the node `C'` in T_X. Extraction of the complement, $T_{\bar{X}}$, demonstrates the case of adding a dummy root `R`.

A common scenario of using tree extraction in our solutions is captured in the following

DEFINITION 2.1. *For a given tree T and an extraction T_X therefrom, let T' be a tree with the topology of T and in which a node is labeled with 1 if it has been extracted into T_X, or with 0 otherwise. Then T' is referred to as the* indicator tree *of (T, T_X).*

Figure 2.3 also gives an example of an indicator tree.

2.3.2 Succinct Representations of Ordinal Trees

Succinct representations of unlabeled and labeled ordinal trees is a widely researched area. The following lemma presents a previous result on unlabeled trees that will be used in our solutions.

LEMMA 2.2 ([67]). *An ordinal tree T on n nodes can be represented in $2n + o(n)$ bits of space to support the operations in Table 2.1(a) in $\mathcal{O}(1)$ time.*

In a labeled tree, each node is associated with a label over an alphabet. Such a label can serve as a scalar weight; in our solutions, however, they typically categorize tree nodes into

	Operation	Description
(a)	$\texttt{depth}(x)$	the number of nodes on $P_{x,\perp}$
	$\texttt{level\_anc}(x, i)$	the i^{th} nearest ancestor of x ($\texttt{level\_anc}(x, 1) = x$)
	$\texttt{pre\_rank}(x)$	the number of nodes preceding x in preorder
	$\texttt{pre\_select}(j)$	the j^{th} node in preorder
	$\texttt{LCA}(x, y)$	the lowest common ancestor of x and y
(b)	$\texttt{depth}_\alpha(x)$	the number of α-nodes on $P_{x,\perp}$
	$\texttt{level\_anc}_\alpha(x, i)$	the i^{th} nearest α-ancestor of x ($\texttt{level\_anc}_\alpha(x, 1) = x$ if x is an α-node)
	$\texttt{pre\_rank}_\alpha(x)$	the number of α-nodes preceding x in preorder
	$\texttt{pre\_select}_\alpha(j)$	the j^{th} α-node in preorder

TABLE 2.1: Primitive operations in trees.

different classes. Hence we call these assigned values *labels* instead of *weights*. We summarize the previous result used in our solutions, in which a node (resp. ancestor) with label α is called an *α-node* (resp. *α-ancestor*):

LEMMA 2.3 ([106]). *Let T be an ordinal tree on n nodes, each of which is assigned a label over $[\sigma]$, $\sigma \leq n$. Then, under the word-RAM with word size $w = \Omega(\lg n)$, T can be represented using $n(\lg \sigma + 2) + o(n \lg \sigma)$ bits of space to support the operations in Table 2.1(b) in $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time.*

Of particular interest is the following corollary to Lemma 2.3:

LEMMA 2.4. *Let T be an ordinal tree on n nodes, each of which is assigned a label over $[\sigma]$. If $\sigma = \mathcal{O}(\lg^c n)$ for a positive constant c, then, under the word-RAM model, T can be represented using $n(\lg \sigma + 2) + o(n)$ bits of space to support the operations in Table 2.1(b) in $\mathcal{O}(1)$ time.*

An important special case of Lemma 2.4 is when $\sigma = 2$; here, T is referred to as a $0/1$-*labeled*, with the storage space correspondingly being $3n + o(n)$ bits. From Lemma 2.4 it also follows that an indicator tree (Definition 2.1) of (T, T_X) occupies $3n + o(n)$ bits of space, for any tree extraction T_X from T.

When translating node identifiers between T and T_X, the following fact is immediate:

PROPOSITION 2.1. *Let T' be the indicator tree of (T, T_X). Then, (i) the corresponding node $x \in T$ of a node $x^* \in T_X$ can be recovered as*

$$x = \texttt{pre\_select}_1(T', x^*);$$

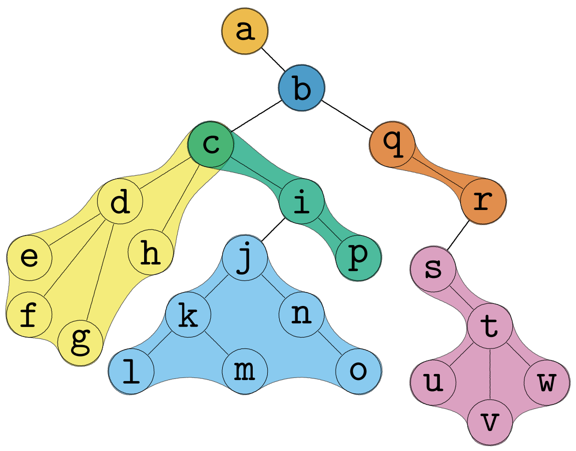

FIGURE 2.4: Tree covering of Farzan and Munro [39], with parameter $L = 5$. Mini-trees are represented by splinegons. The mini-trees can share roots only, and there is at most one arc leading from a non-root node of a mini-tree to the root of another mini-tree.

and (ii) the T_X-view x^* of a node $x \in T$ can be computed as

$$x^* = 1 + \texttt{pre\_rank}_1(T', \texttt{level\_anc}_1(T', x, 1)).$$

TREE COVERING. Tree covering first appeared in [51, 67, 39] as a method of succinct representation of ordinal trees. The tree T is split into *mini*-trees, given a certain parameter L:

LEMMA 2.5 ([39]). *A tree with n nodes can be decomposed into $\Theta(n/L)$ subtrees of size at most $2L$. These are pairwise disjoint aside from the subtree roots. Furthermore, aside from the edges incident to the subtree roots, there is at most one edge per subtree leaving a node of a subtree to its child in another subtree.*

Figure 2.4 gives an example of tree covering with parameter $L = 5$. Each of the mini-trees in turn can be recursively decomposed into *micro*-trees, with another parameter $L' < L$. The idea is to choose the parameter L' such that *intra*-micro-tree queries are executed in constant-time by virtue of a pre-computed table $\mathcal{T}$ of size $o(n)$, indexed by micro-trees. For any given node $x \in T$, the solutions of [51, 67, 39] provide constant-time access to the mini-tree τ and micro-tree τ' containing the node x, as well as the address of the micro-tree τ' in the table $\mathcal{T}$, using $\mathcal{O}(n)$ bits of space, with suitably chosen parameters L and L'.

Chapter 3

Path and Ancestor Queries over Trees with d-Dimensional Weight Vectors

3.1 Introduction

The problem of preprocessing a weighted tree, i.e. a tree in which each node is associated with a weight value, to support various queries evaluating a certain function on the node weights of a given path, has been studied extensively [6, 24, 62, 77, 32, 70, 19]. For example, in *path counting* (resp. *path reporting*), the nodes of the given path with weights lying in the given query interval are counted (resp. reported). *Path minimum* queries, where a node with the smallest weight on the given query path is to be returned, arises in the context of multi-terminal network flows [54]. Precisely, the well-known Gomory-Hu theorem states that with a given graph G with edge capacities, a certain tree T on the same set of nodes can be associated, so that the minimum weight on the path from x to y in T equals the value of the maximum flow from x to y in the original graph G. Path queries, therefore, address the general need of fast information retrieval from tree-structured data.

For many applications, meanwhile, a node in a tree is associated not with just a single weight, but rather with a vector of weights. Consider a simple scenario of an online forum thread, where users rate responses and respond to posts. Induced is a tree-shaped structure with posts representing nodes, and replies to a post being its children. One can imagine enumerating all the ancestor posts of a given post that are not too short and have sufficiently high average ratings. Ancestor dominance query, which is among the problems we consider, provides an appropriate model in this case.

For a d-dimensional weight vector $\mathbf{w} = (w_1, w_2, \ldots, w_d)$, w_i is referred to as the i^{th} *weight* of $\mathbf{w}$. We then consider an ordinal tree T on n nodes, each node x of which is assigned a d-dimensional weight vector $\mathbf{w}(x)$, each weight w_i of which is in rank space $[n]$. For a d-dimensional orthogonal range $Q = \prod_{i=1}^{d}[q_i, q_i']$, a weight vector $\mathbf{w}$ is in Q *iff* $\forall i \in [1, d]$ it holds that $q_i \le w_i \le q_i'$. In our queries we are given a pair of vertices $x, y \in T$, and an arbitrary orthogonal range Q. With $P_{x,y}$ being the path from x to y in the tree T, the goal is to preprocess the tree T for the following types of queries:

Path Counting: return $|\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}|$;

Path Reporting: enumerate $\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}$;

Path Successor: return $\arg\min\{w_1(z) \mid z \in P_{x,y} \text{ and } \mathbf{w}(z) \in Q\}$;

Ancestor Dominance Reporting: a special case of path reporting, in which y is the root of the tree and $q_i' = +\infty$ for all $i \in [d]$. That is, the query reports the ancestors of x whose weight vectors dominate the vector $\mathbf{q} = (q_1, q_2, \ldots, q_d)$.

This is indeed a natural generalization of the traditional weighted tree, which we refer to as 1D-*weighted* (or simply as *weighted*, when context is clear), to the case when the weights are multidimensional vectors. At the same time, when the tree degenerates into a single path, these queries become respectively $(d+1)$-dimensional orthogonal range counting, reporting and successor, as well as $(d+1)$-dimensional dominance reporting, queries. Thus, the queries we study are generalizations of these fundamental geometric queries in high dimensions. We also go along with the state-of-the-art in orthogonal range search by considering weights in rank space, since the case in which weights are from a larger universe can be reduced to it [44].

3.1.1 Previous Work

Path queries in weighted trees. For weighted trees, Chazelle [24] gave an $\mathcal{O}(n)$-word *emulation dag*-based data structure that answers path counting queries in $\mathcal{O}(\lg n)$ time; it works primarily with the topology of the tree and is thus oblivious to the distribution of weights. Later, He et al. [70] proposed a solution with $nH(W_T) + o(n \lg \sigma)$ bits of space and $\mathcal{O}(\frac{\lg \sigma}{\lg \lg n} + 1)$ query time, when the weights are from $[\sigma]$; here, $H(W_T)$ is the entropy of the multiset of the weights in T.

He et al. [70] introduced and solved the path reporting problem using (i) linear space and $\mathcal{O}((1+k)\lg\sigma)$ query time, or (ii) $\mathcal{O}(n\lg\lg\sigma)$ words of space but $\mathcal{O}(\lg\sigma + k\lg\lg\sigma)$ query time, in the word-RAM model; henceforth for this chapter, we reserve k for the size of the output. Patil et al. [98] presented a succinct data structure for path reporting with $n\lg\sigma + 6n + o(n\lg\sigma)$ bits of space and $\mathcal{O}((\lg n + k)\lg\sigma)$ query time. Although the latter solution uses less space than the version (i) of the former when $\sigma \ll n$, it suffers a logarithmic slowdown in the additive term. An optimal-space solution with $nH(W_T) + o(n\lg\sigma)$ bits of space and $\mathcal{O}((1+k)(\frac{\lg\sigma}{\lg\lg n}+1))$ reporting time is due to He et al. [70]. One of the trade-offs proposed by Chan et al. [19] requires $\mathcal{O}(n\lg^{\epsilon} n)$ words of space for the query time of $\mathcal{O}(\lg\lg n + k)$.

Orthogonal range queries. Dominance reporting in 3D was solved by Chazelle and Edelsbrunner [25] in linear space with either $\mathcal{O}((1+k)\lg n)$ or $\mathcal{O}(\lg^2 n + k)$ time, in pointer-machine (PM) model, with the latter being improved to $\mathcal{O}(\lg n\lg\lg n + k)$ by Makris and Tsakalidis [82]. The same authors [82] developed, in the word-RAM, a linear-size, $\mathcal{O}(\lg n + k)$ and $\mathcal{O}((\lg\lg n\lg\lg\lg n + k)\lg\lg n)$ query-time data structures for the unrestricted case and for points in rank space, respectively. Nekrich [93] presented a word-RAM data structure for points in rank space, supporting queries in $\mathcal{O}((\lg\lg n)^2 + k)$ time, and occupying $\mathcal{O}(n\lg n)$ words; this space was later reduced to linear by Afshani [2], retaining the same query time. Finally, in the same model, a linear-space solution with $\mathcal{O}(\lg\lg n + k)$ query time was designed for 3D dominance reporting in rank space [2, 17]. In the PM model, Afshani [2] also presented an $\mathcal{O}(\lg n + k)$ query time, linear-space data structure for points in $\mathbb{R}^3$.

For the word-RAM, JáJá et al. [74] generalized the range counting problem for $d \geq 2$ dimensions and proposed a data structure of $\mathcal{O}(n(\frac{\lg n}{\lg\lg n})^{d-2})$ words of space and $\mathcal{O}(\left(\frac{\lg n}{\lg\lg n}\right)^{d-1})$ query time. Chan et al. [21] solved orthogonal range reporting in 3D rank space in $\mathcal{O}(n\lg^{1+\epsilon} n)$ words of space and $\mathcal{O}(\lg\lg n + k)$ query time.

Nekrich and Navarro [95] proposed two trade-offs for the range successor problem, with either $\mathcal{O}(n)$ or $\mathcal{O}(n\lg\lg n)$ words of space, and respectively with $\mathcal{O}(\lg^{\epsilon} n)$ or $\mathcal{O}((\lg\lg n)^2)$ query time. Zhou [110] later improved upon the query time of the second trade-off by a factor of $\lg\lg n$, within the same space. Both results are for points in rank space.

3.1.2 Our Results

As d-dimensional path queries generalize the corresponding $(d+1)$-dimensional orthogonal range queries, we compare results on them to show that our bounds match or nearly match the best results or some of the best trade-offs on geometric queries in Euclidean space. We present solutions for the (we assume d is a positive integer constant):

- ancestor dominance reporting problem, in $\mathcal{O}(n \lg^{d-2} n)$ words of space and $\mathcal{O}(\lg^{d-1} n + k)$ query time for $d \geq 2$. When $d = 2$, this matches the space bound for 3D dominance reporting of [2, 17], while still providing efficient query support. When $d \geq 3$, we also achieve a trade-off of $\mathcal{O}(n \lg^{d-2+\epsilon} n)$ words of space, with query time of $\mathcal{O}(\lg^{d-1} n/(\lg\lg n)^{d-2} + k)$;

- path successor problem, in $\mathcal{O}(n \lg^{d-1} n)$ words and $\mathcal{O}(\lg^{d-1+\epsilon} n)$ query time, for an arbitrarily small positive constant ϵ, and $d \geq 1$. These bounds match the first trade-off for range successor of Nekrich and Navarro [95]. [1] Previously this problem has not been studied even on 1D-weighted trees;

- path counting problem, in $\mathcal{O}(n(\frac{\lg n}{\lg\lg n})^{d-1})$ words and $\mathcal{O}((\frac{\lg n}{\lg\lg n})^{d})$ query time for $d \geq 1$. This matches the best bound for range counting in $d+1$ dimensions [74];

- path reporting problem, in $\mathcal{O}(n \lg^{d-1+\epsilon} n)$ words and $\mathcal{O}(\frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}} + k)$ query time, for $d \geq 2$. When $d = 2$, the space matches that of the corresponding result of Chan et al. [21] on 3D range reporting, while the first term in the query complexity is slowed down by a sub-logarithmic factor.

To achieve our results, we introduce a framework for solving range sum queries in arbitrary semigroups and extend base-case data structures to higher dimensions using universe reduction. A careful design with results hailing from succinct data structures and tree representations has been necessary, both for building space- and time-efficient base data structures, and for porting, using tree extractions (Section 2.3.1), the framework of range tree decompositions from general point-sets to tree topologies (Lemma 3.4). The notion of *maximality* in Euclidean sense is central to the solutions of orthogonal dominance problems. We employ a few novel techniques and extend this notion to tree topologies and provide the means of efficient computation thereof

[1] Range successor can be generalized to higher dimensions via standard techniques based on range trees.

(Section 3.4). Furthermore, given a weighted tree T, we propose efficient means of zooming into the nodes of T with weights in the given range in the range tree (Lemma 3.10). Given the ubiquity of the concepts, these technical contributions are likely to be of independent interest.

3.2 Preliminaries

3.2.1 Notation

We use the symbol $\succeq$ for domination: $\mathbf{p} \succeq \mathbf{q}$ *iff* $\mathbf{p}$ dominates $\mathbf{q}$, i.e. each of the weights of $\mathbf{p}$ is greater than or equal to the corresponding weight of $\mathbf{q}$; a vector $\mathbf{p}$ *k-dominates* $\mathbf{q}$ *iff* $\mathbf{p}_{1,k} \succeq \mathbf{q}_{1,k}$. With $d' \leq d$ and $0 < \epsilon < 1$ being constants, a weight vector $\mathbf{w}$ is said to be (d', d, ϵ)*-dimensional iff* $\mathbf{w} \in [n]^{d'} \times [\lceil \lg^{\epsilon} n \rceil]^{d-d'}$; i.e. each of its first d' weights is drawn from $\{1, 2, \ldots, n\}$, while each of its last $d - d'$ weights is in $\{1, 2, \ldots, \lceil \lg^{\epsilon} n \rceil\}$. When stating theorems, we set $i/0 \triangleq \infty$ for $i > 0$.

3.2.2 Representation of a Range Tree on Node Weights by Hierarchical Tree Extraction

Range trees are widely used in solutions to query problems in Euclidean space. He et al. [70] further applied the idea of range trees to 1D-weighted trees. They defined a conceptual range tree on node weights and represented it by a hierarchy of tree extractions. Let us summarize how the hierarchy is built.

We first define a conceptual range tree on $[n]$ with branching factor f, where $f = \mathcal{O}(\lg^{\epsilon} n)$ for some constant $0 < \epsilon < 1$. Its root represents the entire range $[n]$. Starting from the root level, we keep partitioning each range, $[a, b]$, at the current lowest level into f child ranges $[a_1, b_1], \ldots, [a_f, b_f]$, where $a_i = \lceil (i-1)(b-a+1)/f \rceil + a$ and $b_i = \lceil i(b-a+1)/f \rceil + a - 1$. This ensures that, if a given weight j is in $[a, b]$, then j is contained in the child range with subscript $\lceil f(j - a + 1)/(b - a + 1) \rceil$, which can be determined in $\mathcal{O}(1)$ time. We stop partitioning a range when its size is 1, i.e. when $b - a + 1 = 1$. This range tree has $h = \lceil \log_f n \rceil + 1$ levels. The root is at level 1 and the bottom level is level h.

For $1 \leq l < h$, we construct an auxiliary tree T_l for level l of this range tree as follows. Let $[a_1, b_1], \ldots, [a_m, b_m]$ be the ranges at level l. For a range $[a, b]$, let $F_{a,b}$ stand for the extracted forest of the nodes of T with weights in $[a, b]$. Then, for each range $[a_i, b_i]$, we extract F_{a_i,b_i} and

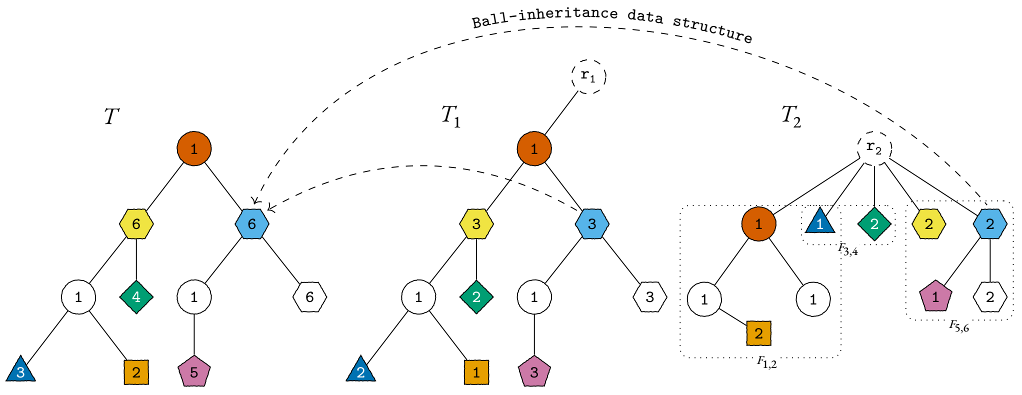

FIGURE 3.1: A fragment of hierarchical tree extraction with branching factor $f = 3$, for a tree T weighted over an alphabet of size 6. The labels are given inside the circles. For some nodes, the correspondence across the levels of the hierarchy is shown using same non-plain colours and node shapes. Rounded rectangular areas enclose the nodes of the corresponding forests. Dashed arrow maps a node in T_l to the original node, and illustrate the *ball-inheritance* data structure of Lemma 3.2.

plug its roots as children of a dummy root r_l, retaining the original left-to-right order of the roots within the forest. Between forests, the roots in $F_{a_{i+1},b_{i+1}}$ are the right siblings of the roots in F_{a_i,b_i}, for any $i \in [m-1]$. We then label the nodes of T_l using the reduced alphabet $[f]$, as follows. Note that barring the dummy root r_l, there is a bijection between the nodes of T and those of T_l. Let $x_l \in T_l$ be the node corresponding to $x \in T$. In the range tree, let $[a, b]$ be the level-l range containing the weight of x. Then, at level $l+1$, if the weight of x is contained in the j^{th} child range of $[a, b]$, then $x_l \in T_l$ is labeled j. Each T_l is represented by Lemma 2.4 in $n(\lg f + 2) + o(n)$ bits, so that the space cost of all the T_ls is $n \lg n + (2n + o(n)) \log_f n$ bits. When $f = \omega(1)$, this space cost is $n + o(n)$ words ([70]). This completes the outline of hierarchical tree extraction, with Figure 3.1 showing an example thereof.

Furthermore, the following lemma maps x_l to x_{l+1}:

LEMMA 3.1 ([70]). *Given a node $x_l \in T_l$ and the range $[a, b]$ of level l containing the weight of x, node $x_{l+1} \in T_{l+1}$ can be located in $\mathcal{O}(1)$ time, for any $l \in [h-2]$.*

Later, Chan et al. [19] augmented this representation with the *ball-inheritance* data structure to map an arbitrary x_l back to x:

LEMMA 3.2 ([19]). *Given a node $x_l \in T_l$, where $1 \le l < h$, the node $x \in T$ that corresponds to x_l can be found using $\mathcal{O}(n \lg n \cdot \mathtt{s}(n))$ bits of additional space and $\mathcal{O}(\mathtt{t}(n))$ time, where*

(a) $\mathtt{s}(n) = \mathcal{O}(1)$ *and* $\mathtt{t}(n) = \mathcal{O}(\lg^\epsilon n)$;

(b) $\mathtt{s}(n) = \mathcal{O}(\lg\lg n)$ *and* $\mathtt{t}(n) = \mathcal{O}(\lg\lg n)$; *or*

(c) $\mathtt{s}(n) = \mathcal{O}(\lg^{\epsilon} n)$ *and* $\mathtt{t}(n) = \mathcal{O}(1)$.

In what follows, we denote as $\mathscr{T}_v$ the extraction from T of the nodes with weights in v's range, for a node v of the range tree, denoted as $\mathscr{R}$.

3.2.3 Path Minimum in 1D-Weighted Trees

In a weighted tree, a *path minimum query* asks for the node with the smallest weight in the given path. We summarize the best result on path minimum; in it, $\alpha(m, n)$ and $\alpha(n)$ are the inverse-Ackermann functions:

Lemma 3.3 ([19]). *An ordinal tree T on n weighted nodes can be indexed (a) using $\mathcal{O}(m)$ bits of space to support path minimum queries in $\mathcal{O}(\alpha(m,n))$ time and $\mathcal{O}(\alpha(m,n))$ accesses to the weights of nodes, for any integer $m \geq n$; or (b) using $2n + o(n)$ bits of space to support path minimum queries in $\mathcal{O}(\alpha(n))$ time and $\mathcal{O}(\alpha(n))$ accesses to the weights of nodes. In particular, when $m = \Theta(n \lg^{**} n)$,[2] one has $\alpha(m,n) = \mathcal{O}(1)$, and therefore (a) includes the result that T can be indexed in $\mathcal{O}(n \lg^{**} n)$ bits of space to support path minimum queries in $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ accesses to the weights of nodes.*

3.3 Reducing to Lower Dimensions

This section presents a general framework for reducing the problem of answering a d-dimensional query to the same query problem in $(d-1)$ dimensions, by generalizing the standard technique of range tree decomposition for the case of tree topologies weighted with multidimensional vectors. To describe this framework, we introduce a *d-dimensional semigroup path sum query* problem, which is a generalization of all the query problems we consider in this chapter.

Definition 3.1. *Let $(G, \oplus)$ be a semigroup with support G and semigroup sum operator $\oplus$. Let T be a tree on n nodes, in which each node x is assigned a d-dimensional weight vector $\mathbf{w}(x)$ and a semigroup element $g(x) \in G$. Then, in a d-dimensional semigroup path sum query, we are given a path $P_{x,y}$ in T, an orthogonal query range Q in d-dimensional space, and we are asked to compute*

$$\bigoplus_{z \in P_{x,y} \text{ and } \mathbf{w}(z) \in Q} g(z).$$

[2] $\lg^{**} n$ stands for the number of times an iterated logarithm function $\lg^{*}$ needs to be applied to n in order for the result to become at most 1.

When the weight vectors of the nodes and the query range are both from a (d', d, ϵ)-dimensional space, the *(d', d, ϵ)-dimensional semigroup path sum* problem is defined analogously.

3.3.1 Space Reduction Lemma Using Range Trees with Branching Factor 2

The following lemma presents our framework for solving a *d-dimensional semigroup path sum query* problem; its counterpart in (d', d, ϵ)-dimensional space is given in Section 3.3.2.

Lemma 3.4. *Let d be a positive integer constant. Let $G^{(d-1)}$ be an $\mathtt{s}(n)$-word data structure for a $(d-1)$-dimensional semigroup path sum problem of size n. Then, there is an $\mathcal{O}(\mathtt{s}(n) \lg n + n)$-word data structure $G^{(d)}$ for a d-dimensional semigroup path sum problem of size n, whose components include $\mathcal{O}(\lg n)$ structures of type $G^{(d-1)}$, each of which is constructed over a tree on $n+1$ nodes. Furthermore, $G^{(d)}$ can answer a d-dimensional semigroup path sum query by performing $\mathcal{O}(\lg n)$ $(d-1)$-dimensional queries using these components and returning the semigroup sum of the answers. Determining which queries to perform on structures of type $G^{(d-1)}$ requires $\mathcal{O}(1)$ time per query.* [3]

Proof. We define a conceptual range tree R with branching factor 2 over the d^{th} weights of the nodes of T and represent it using hierarchical tree extraction as in Section 3.2.2. For each level l of the range tree, we define a tree T_l^* with the same topology as T_l. We assign $(d-1)$-dimensional weight vectors and semigroup elements to each node, x', in T_l^* as follows. If x' is not the dummy root, then $\mathbf{w}(x')$ is set to be $\mathbf{w}_{1,d-1}(x)$, where x is the node of T corresponding to x'. We also set $g(x') = g(x)$. If x' is the dummy root, then its first $(d-1)$ weights are $-\infty$, while $g(x')$ is set to an arbitrary element of the semigroup. We then construct a data structure, G_l, of type $G^{(d-1)}$, over T_l^*. The data structure $G^{(d)}$ comprises the structures T_l and G_l, over all l. The range tree has $\mathcal{O}(\lg n)$ levels, each T_l^* has $n+1$ nodes, and the G_ls are the $\mathcal{O}(\lg n)$ structures of type $G^{(d-1)}$ referred to in the statement. By the exposition of hierarchical tree extraction in Section 3.2.2, all the structures T_l in total occupy $n + o(n)$ words, and $G^{(d)}$ therefore occupies $\mathcal{O}(\mathtt{s}(n) \lg n + n)$ words.

Next we show how to use $G^{(d)}$ to answer queries. Let $P_{x,y}$ be the query path and $Q = \prod_{j=1}^{d} [q_j, q_j']$ be the query range. To answer the query, we first decompose $P_{x,y}$ (see Section 2.1.2) into $A_{x,z}$, $\{z\}$, and $A_{y,z}$, where z is the lowest common ancestor of x and y, found in $\mathcal{O}(1)$

[3]It may be tempting to simplify the statement of the lemma by defining $\mathtt{t}(n)$ as the query time of $G^{(d-1)}$ and claiming that $G^{(d)}$ can answer a query in $\mathcal{O}(\mathtt{t}(n) \cdot \lg n)$ time. However, this bound is too loose when applying this lemma to reporting queries.

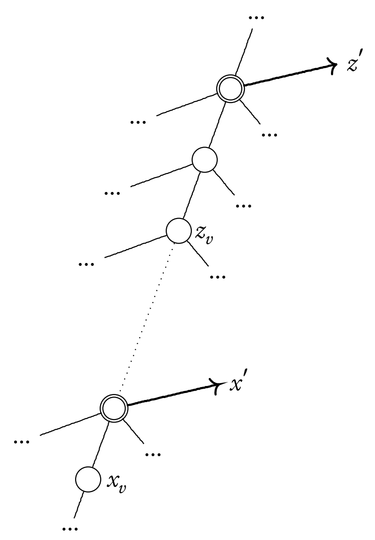

FIGURE 3.2: An illustration to the proof of Lemma 3.4. Shown is a root-to-leaf path that contains both x_v and z_v. *Double* circles represent the nodes with weights falling into the range of v's j^{th} child. They are retrieved via a call to $\texttt{level\_anc}_j(T_l, \cdot, 1)$. The nodes x' and z' are the nodes corresponding to (resp.) $\texttt{level\_anc}_j(T_l, x_v, 1)$ and $\texttt{level\_anc}_j(T_l, z_v, 1)$ in the next level of the hierarchy of extractions.

time via LCA in T_1. It suffices to answer three path semigroup sum queries using each subpath and Q as query parameters, as the semigroup sum of the answers to these queries is the answer to the original query. Since the query on subpath $\{z\}$ reduces to checking whether $\mathbf{w}(z) \in Q$, we show how to answer the query on $A_{x,z}$; the query on $A_{y,z}$ is then handled analogously. To answer the query on $A_{x,z}$, we perform a standard top-down traversal in the range tree to identify up to two nodes at each level representing ranges that contain at least one of q_d and q'_d. Let v be the node that we are visiting, in the range tree R. We maintain *current nodes*, x_v and z_v (initialized as respectively x and z) local to the current level l; they are the nodes in T_l that correspond to the $\mathscr{T}_v$-views of the original query nodes x and z. Nodes x_v and z_v are kept up-to-date in $\mathcal{O}(1)$ time as we descend the levels of the range tree. Namely, when descending to the j^{th} ($j \in \{0, 1\}$) child of the node v, we identify, via Lemma 3.1, the corresponding nodes in T_{l+1}, for the nodes $\texttt{level\_anc}_j(T_l, x_v, 1)$ and $\texttt{level\_anc}_j(T_l, z_v, 1)$, as illustrated in Figure 3.2.

For each node v identified at each level l, such that v's range contains q_d but not q'_d, we check if it is its left child-range that contains q_d. If so, we perform a $(d-1)$-dimensional semigroup range sum query with the following parameters: (i) the query range $[q_1, q'_1] \times [q_2, q'_2] \times \ldots \times [q_{d-1}, q'_{d-1}]$ (i.e. we drop the last range); and (ii) the query path is A_{x_u,z_u}, where x_u and z_u are the nodes in T_{l+1} corresponding to the $\mathscr{T}_u$-views of x and z, with u being the right child of v ; this is analogous to updating x_v and z_v, i.e. applying Lemma 3.1 to the nodes $\texttt{level\_anc}_1(T_l, x_v, 1)$ and $\texttt{level\_anc}_1(T_l, z_v, 1)$. For each node whose range contains q'_d but not q_d, a symmetrical procedure is performed by considering its left child.

The semigroup sum of the answers to these $\mathcal{O}(\lg n)$ queries is the answer to the original query. □

3.3.2 Space Reduction Lemma Using Range Tree with a Non-Constant Branching Factor

This section presents a general framework for reducing the problem of answering a (d', d, ϵ)-dimensional query to the same query problem in $(d' - 1, d, \epsilon)$ dimensions, by generalizing the approach of JáJá et al. [74] for the case of trees weighted with multidimensional vectors.

Lemma 3.5. *Let d and d' be positive integer constants such that $d' \leq d$, and ϵ be a constant in $(0, 1)$. Let $G^{(d'-1)}$ be an $\mathtt{s}(n)$-word data structure for a $(d' - 1, d, \epsilon)$-dimensional semigroup path sum problem of size n. Then, there is an $\mathcal{O}(\mathtt{s}(n) \lg n/ \lg\lg n + n)$-word data structure $G^{(d')}$ for a (d', d, ϵ)-dimensional semigroup path sum problem of size n, whose components include $\mathcal{O}(\lg n/ \lg\lg n)$ structures of type $G^{(d'-1)}$, each of which is constructed over a tree on $n + 1$ nodes. Furthermore, $G^{(d')}$ can answer a (d', d, ϵ)-dimensional semigroup path sum query by performing $\mathcal{O}(\lg n/ \lg\lg n)$ queries in $(d' - 1, d, \epsilon)$ dimensions using these components, and returning the semigroup sum of the answers. Determining which queries to perform on structures of type $G^{(d'-1)}$ requires $\mathcal{O}(1)$ time per query.*

Proof. We define a conceptual range tree with branching factor $f = \mathcal{O}(\lg^{\epsilon} n)$ over the d'^{th} weights of the nodes of T and represent it using hierarchical tree extraction as in Section 3.2.2. For each level l of the range tree, we define a tree T^*_l with the same topology as T_l. We assign $(d' - 1, d, \epsilon)$-dimensional weight vectors and semigroup elements to each node, x', in T^*_l, as follows. If x' is not the dummy root, then $\mathbf{w}(x')$ is set to be

$$(w_1(x), \dots, w_{d'-1}(x), \lambda(T_l, x'), w_{d'+1}(x), \dots, w_d(x)),$$

where x is the corresponding node of x' in T, and $\lambda(T_l, x')$ is the label assigned to x' in T_l. We also set $g(x') = g(x)$. If x' is the dummy root, then its first $d' - 1$ weights are $-\infty$ and last $d - d' + 1$ weights are $-\lceil \lg^{\epsilon} n \rceil$, while $g(x')$ is set to an arbitrary element of the semigroup. We further construct a data structure, G_l, of type $G^{(d'-1)}$, over T^*_l. The data structure $G^{(d')}$ then comprises the structures T_l and G_l, over all l. The range tree has $\mathcal{O}(\lg n/\lg\lg n)$ levels and each T_l^* has $n+1$ nodes, and the structures G_l are the $\mathcal{O}(\lg n/\lg\lg n)$ structures of type $G^{(d'-1)}$ referred to in the statement. By the exposition of hierarchical tree extraction in Section 3.2.2, all the T_ls in total occupy $n + o(n)$ words. Therefore, $G^{(d')}$ occupies $\mathcal{O}(\mathtt{s}(n)\lg n/\lg\lg n + n)$ words.

Next we show how to use $G^{(d')}$ to answer queries. Let $P_{x,y}$ be the query path and $Q = \prod_{j=1}^{d}[q_j, q'_j]$ be the query range. As discussed in the proof of Lemma 3.4, it suffices to describe the handling of the path $A_{x,z}$, where z is the lowest common ancestor of x and y.

To answer the query on $A_{x,z}$, we perform a top-down traversal in the range tree to identify the up to two nodes at each level representing ranges that contain at least one of $q_{d'}$ and $q'_{d'}$. For each node v identified at each level l, we perform a $(d'-1, d, \epsilon)$-dimensional semigroup range sum query with parameters computed as follows: (i) the query path is P_{x_v,z_v}, where x_v and z_v are the nodes in T_l corresponding to the $\mathscr{T}_v$-views of x and z; and (ii) the query range is

$$Q_v = [q_1, q'_1] \times [q_2, q'_2] \times \ldots \times [q_{d'-1}, q'_{d'-1}] \times [i_v, j_v] \times [q_{d'+1}, q'_{d'+1}] \times \ldots \times [q_d, q'_d],$$

such that the children of v representing ranges that are entirely within $[q_{d'}, q'_{d'}]$ are children $i_v, i_v+1, \ldots, j_v$ ("child i" refers to the i^{th} child); no queries are performed if such children do not exist. The semigroup sum of these $\mathcal{O}(\lg n/\lg\lg n)$ queries is the answer to the original query. It remains to show that the parameters of each query are computed in $\mathcal{O}(1)$ time per query. By Section 3.2.2, i_v and j_v are computed in $\mathcal{O}(1)$ time via simple arithmetic, which is sufficient to construct Q_v. Nodes x_v and z_v are computed in $\mathcal{O}(1)$ time each time we descend down a level in the range tree: Initially, when v is the root of the range tree, x_v and z_v are nodes x and z in T_1. When we visit a child, v_j, of v whose range contains at least one of $q_{d'}$ and $q'_{d'}$, we compute (via Lemma 3.1) x_{v_j} as the node in T_{l+1} corresponding to the node $\mathtt{level\_anc}_j(T_l, x_v, 1)$ in T_l, which uses constant time. Node z_{v_j} is located similarly. $\square$

3.4 Ancestor Dominance Reporting

This section proposes a solution to the ancestor dominance reporting problem. Our solution is designed around the *layers of maxima* paradigm [25]. We consider a partial ordering of the nodes of a weighted tree, in which two nodes are in relation *iff* one is the ancestor of the other, and its weight is greater than that of the other. We then design data structures that allow iteration through the set of the maximal elements – often referred to as the *layer of maxima* [25] – of this partial ordering, as well as switching from one layer to the next.

This section comprises Sections 3.4.1 to 3.4.3. In Section 3.4.1, we solve the $(1, d, \epsilon)$-dimensional *path dominance reporting problem*, which asks one to enumerate the nodes in the query path whose weight vectors dominate the query vector. Then, in Section 3.4.2 we solve the 2-dimensional ancestor dominance reporting problem. Finally, Section 3.4.3 uses the results of the latter two sections to solve the $(2, d, \epsilon)$-dimensional ancestor dominance reporting problem and extends the result to the general d-dimensional case.

3.4.1 Path Dominance in $(1, d, \epsilon)$

The strategy employed in Lemma 3.6 is that of zooming into the extraction dominating the query point in the last $(d-1)$ weights, and therein reporting the relevant nodes based on the 1st weight and tree topology only.

Lemma 3.6. *Let $d \geq 1$ be a constant integer and $0 < \epsilon < \frac{1}{d-1}$ be a constant number. A tree T on $m \leq n$ nodes, in which each node is assigned a $(1, d, \epsilon)$-dimensional weight vector, can be represented in $m + o(m)$ words, so that a path dominance reporting query can be answered in $\mathcal{O}(1 + k)$ time, where k is the number of the nodes reported.*

Proof. We represent the topology of T using Lemma 2.2. For any $(0, d-1, \epsilon)$-dimensional vector $\mathbf{g}$, we consider a conceptual 1D-weighted tree $E_\mathbf{g}$ by first extracting the node set

$$\{x \mid x \in T \text{ and } \mathbf{w}_{2,d}(x) \succeq \mathbf{g}\}$$

from T. The weight of a non-dummy node in $E_\mathbf{g}$ is the 1st weight of its T-source. If $E_\mathbf{g}$ has a dummy root, then its weight is $-\infty$.

Instead of storing $E_\mathbf{g}$ explicitly, we create the following structures, the first two of which are built for any possible $(0, d-1, \epsilon)$-dimensional vector $\mathbf{g}$:

- The indicator tree (Definition 2.1) $T_\mathbf{g}$ of $(T, E_\mathbf{g})$;
- A succinct index $I_\mathbf{g}$ for path maximum queries in $E_\mathbf{g}$ (using Lemma 3.3(a));
- An array W_1 where $W_1[x]$ stores the 1st weight of the node x in T;
- A table C which stores pointers to $T_\mathbf{g}$ and $I_\mathbf{g}$ for each possible $\mathbf{g}$.

For any node $x' \in E_{\mathbf{g}}$, its T-source x equals $\texttt{pre\_select}_1(T_{\mathbf{g}}, x')$ (Proposition 2.1). Then, the weight of x' is $W_1[x]$. With this $\mathcal{O}(1)$-time access to node weights in $E_{\mathbf{g}}$, by Lemma 3.3 we can use $I_{\mathbf{g}}$ to answer path maximum queries in $E_{\mathbf{g}}$ in $\mathcal{O}(1)$ time.

We now show how to answer a path dominance reporting query in T. Let $P_{x,y}$ and $\mathbf{q} = (q_1, q_2, \ldots, q_d)$ be respectively the path and the weight vector given as query parameters. First, we use C to locate $T_{\mathbf{q}'}$ and $I_{\mathbf{q}'}$, where $\mathbf{q}' = \mathbf{q}_{2,d}$. As discussed in the proof of Lemma 3.4, it suffices to show how to answer the query with $A_{x,z}$ as the query path, where $z = \texttt{LCA}(T, x, y)$. To that end, we fetch the $E_{\mathbf{q}'}$-view, x', of x, using Proposition 2.1 and analogously the view, z', of z. Next, $I_{\mathbf{q}'}$ locates a node $t' \in A_{x',z'}$ with the maximum weight. If the weight of t' is less than q_1, then no node in $A_{x,y}$ can possibly have a weight vector dominating $\mathbf{q}$, and our algorithm is terminated without reporting any nodes. Otherwise, the T-source t of t' is located as in Proposition 2.1. The node $t \in T$ then claims the following two properties: (i) as $T_{\mathbf{q}'}$ contains a node corresponding to t, one has $\mathbf{w}_{2,d}(t) \succeq \mathbf{q}'$; and (ii) as $w_1(t)$ equals the weight of t', it is at least q_1. We therefore have that $\mathbf{w}(t) \succeq \mathbf{q}$ and duly report t. Afterwards, we perform the same procedure recursively on paths $A_{x',t'}$ and $A_{s',z'}$ in $E_{\mathbf{q}'}$, where s' is the parent of t' in $E_{\mathbf{q}'}$ and can be computed as the $E_{\mathbf{q}'}$-view of the parent of t, using Proposition 2.1.

To analyze the running time, the key observation is that we perform path maximum queries using $I_{\mathbf{q}'}$ at most $2k+1$ times. Since both each query itself and the operations performed to identify the query path use $\mathcal{O}(1)$ time, our algorithm runs in $\mathcal{O}(1+k)$ time.

To analyze the space cost, we observe that W_1 occupies m words. The total number of possible $(0, d-1, \epsilon)$-dimensional vectors is $\mathcal{O}(\lg^{(d-1)\epsilon} n)$. Since each $T_{\mathbf{g}}$ uses $\mathcal{O}(m)$ bits and each $I_{\mathbf{g}}$ uses $\mathcal{O}(m \lg^{**} m)$ bits, the total space space cost of storing $T_{\mathbf{g}}$s and $I_{\mathbf{g}}$s for all possible vectors $\mathbf{g}$ is $\mathcal{O}((m + m\lg^{**} m)\lg^{(d-1)\epsilon} n) = \mathcal{O}(m \lg^{**} m \lg^{(d-1)\epsilon} n) \le \mathcal{O}(m \lg^{**} n \lg^{(d-1)\epsilon} n) = o(m \lg n)$ bits for any constant $0 < \epsilon < 1/(d-1)$, or $o(m)$ words. Furthermore, C stores $\mathcal{O}(\lg^{(d-1)\epsilon} n)$ pointers. To reduce the space cost for each pointer, we concatenate the encodings of all the $T_{\mathbf{g}}$s and $I_{\mathbf{g}}$s and store them in a memory block of $o(m \lg n)$ bits. Thus, each pointer stored in C can be encoded in $\mathcal{O}(\lg(m \lg n))$ bits, and the table C thus uses $\mathcal{O}((\lg m + \lg\lg n)\lg^{(d-1)\epsilon} n) = \mathcal{O}(\lg m \lg^{(d-1)\epsilon} n) + \mathcal{O}(\lg\lg n \lg^{(d-1)\epsilon} n) = o(\lg m \lg n) + o(\lg n) = o(\lg m \lg n)$ bits for any constant $0 < \epsilon < 1/(d-1)$, which is $o(\lg m)$ words. Finally, the encoding of T using Lemma 2.4 is $2m + o(m)$ bits. Therefore, the total space cost is $m + o(m)$ words. $\square$

3.4.2 2D Ancestor Dominance Reporting

We next design a solution to the 2-dimensional ancestor dominance reporting problem, by first generalizing the notion of 2-dominance in Euclidean space (Section 3.2.1) to weighted trees. More precisely, in a tree T in which each node is assigned a d-dimensional weight vector, we say that a node x 2-*dominates* another node y *iff* $x \in \mathcal{A}(y)$ and $w_1(x) > w_1(y)$. (The semantics of "2" in 2-domination is that only two dimensions are taken into account – the tree topology and the first weight.) Then a node x is defined to be 2-*maximal iff* no other node in T 2-dominates x. An example of a 2-maximal set of nodes is given in Figure 3.3.

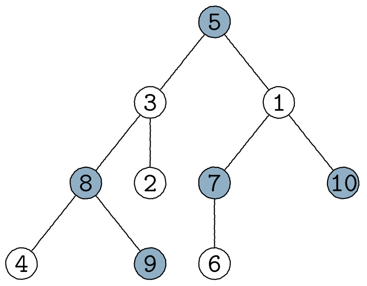

Figure 3.3: The 2-maximal set in a tree T weighted over $\sigma = 10$. The numbers inside the circles represent the assigned weights. The 2-maximal set is shown in shaded circles. The shaded nodes in any upward path form a decreasing sequence.

The following property is then immediate: Given a set, X, of 2-maximal nodes, let T_X be the corresponding extraction from T. Let the weight of a node $x' \in T_X$ be the 1^{st} weight of its T-source x. Then, in any upward path of T_X, the node weights are strictly decreasing.

Now, T_X is a valid input to the *weighted ancestor problem* of Farach and Muthukrishnan [38]. In this problem, one is given a weighted tree with monotonically decreasing node weights along any upward path. We preprocess such a tree to answer *weighted ancestor queries*, which, for any given node x and value κ, ask for the highest ancestor of x whose weight is at least κ. Farach and Muthukrishnan [38] presented an $\mathcal{O}(n)$-word solution that answers this query in $\mathcal{O}(\lg\lg n)$ time, for an n-node tree weighted over $[n]$. With an easy reduction we can further achieve the following result:

Lemma 3.7. *Let T be a tree on $m \le n$ nodes, in which each node is assigned a weight from $[n]$. If the node weights along any upward path are strictly decreasing, then T can be represented using $\mathcal{O}(m)$ words to support weighted ancestor queries in $\mathcal{O}(\lg\lg n)$ time.*

Proof. Let W be the set of weights actually assigned to the nodes of T. We replace the weight,

h, of any node x in T by the rank of h in W, which is in $[m]$. We then represent the resulting tree T' in $\mathcal{O}(m)$ words to support a weighted ancestor query in T' in $\mathcal{O}(\lg\lg m)$ time [38]. We also construct a y-fast trie [107], Y, on the elements of W ; the rank of each element is also stored with this element in Y. Y uses $\mathcal{O}(m)$ space. Given a weighted ancestor query over T, we first find the rank, κ, of the query weight in W in $\mathcal{O}(\lg\lg n)$ time by performing a predecessor query in Y, and κ is further used to perform a query in T' to compute the answer. $\square$

We tackle the 2-dimensional ancestor dominance problem with the following data structures. We define a conceptual range tree R with branching factor $f = \lceil \lg^{\epsilon} n \rceil$ over the 2^{nd} weights of the nodes in T and represent T using hierarchical tree extraction as in Section 3.2.2. Let v be a node in this range tree R, and let R_l denote all the nodes on level l of R. In $\mathscr{T}_v$[4], we assign to each node the weight vector of the T-source and call the resulting weighted tree $T(v)$. We then define $M(v)$ as follows: If v is the root of the range tree, then $M(v)$ is the set of all the 2-maximal nodes in T. Otherwise, let u be the parent of v. Then a node, t, of $T(v)$ is in $M(v)$ *iff* t is 2-maximal in $T(v)$ and its corresponding node in $T(u)$ is not 2-maximal in $T(u)$. Thus, for any node x in T, there exists a unique node v in the range tree such that there is a node in $M(v)$ corresponding to x. Furthermore, for a non-leaf node v we define the set $N(v)$ as the set $\{t \in T \mid \exists \text{ a child } v' \text{ of } v \text{ such that } t \in M(v')\}$.

We further conceptually extract two trees from T_l : (i) M_l is an extraction from T_l of the node set

$$\{x \mid x \in T_l \text{ and } \exists \text{ a node } u \in R_l \text{ s.t. } x \text{ has a corresponding node in } M(u)\};$$

while (ii) N_l is an extraction from T_l of the node set

$$\{x \mid x \in T_l \text{ and } \exists \text{ a node } v \in R_l \text{ s.t. } x \text{ has a corresponding node in } N(v)\}.$$

Figure 3.4 provides an example of the trees M_l and N_l.

Furthermore, for each level l, we also create the following sets of data structures (when defining these structures, we assume that the root, r_l, of T_l corresponds to a dummy node s' in T with weight vector $(-\infty, -\infty)$; the node s' is omitted when determining the rank space, preorder ranks, and depths in T). Each set of the data structures can be conceptualized as a pair,

[4]The tree extraction of all the nodes with weights in v's range, where v is a node of the range tree.

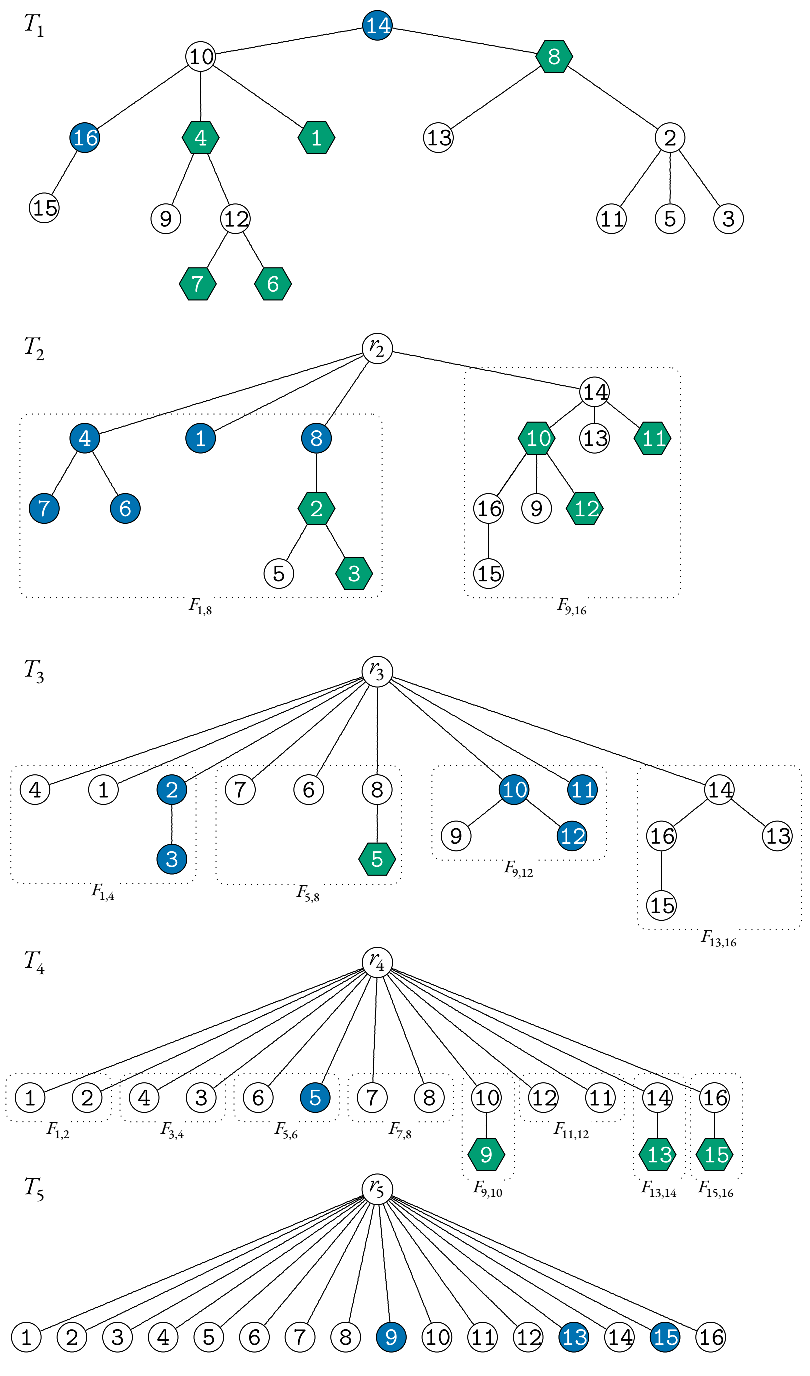

FIGURE 3.4: Hierarchical tree extraction with branching factor 2. For each $1 \leq l \leq 5$ blue, circle-shaped nodes in T_l are the nodes extracted to construct M_l, while green, hexagon-shaped nodes in T_l are the nodes extracted to construct N_l.

consisting of a reporting structure proper and a certain navigational, auxiliary data structure; below we introduce them in this particular order.

One set comprises the structures D_l and A_l, defined as follows:

- D_l is a 1-dimensional path dominance reporting structure (Lemma 3.6) over the tree obtained by assigning weight vectors to the nodes of M_l as follows: each node x' of M_l is assigned a scalar weight $w_2(x)$, where x is the node of T corresponding to x' ;
- A_l is a weighted ancestor query structure over M_l (Lemma 3.7), when its nodes are assigned the 1^{st} weights of the corresponding nodes in T.

Another set comprises the structures E_l and F_l, defined as follows:

- E_l is a 1-dimensional path dominance reporting structure (Lemma 3.6) over the tree obtained by assigning weight vectors to the nodes of M_l as follows: each node x' of M_l is assigned a scalar weight $w_1(x)$, where x is the node of T corresponding to x';
- F_l is a $(1, 2, \epsilon)$-dimensional path dominance reporting structure (Lemma 3.6) over the tree obtained by assigning weight vectors to the nodes of N_l as follows: each node x' of N_l is assigned $(w_1(x), \kappa)$, where x is the node of T corresponding to x', and κ is the label assigned to the node in T_l corresponding to x'.

Finally, the following data structures are also maintained:

- T_l', the indicator tree of (T_l, M_l);
- T_l'', the indicator tree of (T_l, N_l);
- P_l, an array where $P_l[x]$ stores the preorder number of the node in T corresponding to a node x in M_l.

The features of some of these data structures are summarized in Table 3.1.

Before delving into details of the search algorithm, it may be useful to see the "big picture", first. A synopsis of a somewhat more detailed exposition given at the beginning of Section 3.4.2 could then go as follows. The search in the 2D case proceeds by eliminating the second weight from consideration and heeding the first weight only. One employs two strategies to that end, each necessitated by the anatomy of range trees. Recall that when descending down a path in the range tree, the nodes of interest are those lying on the path together with their right

	Nodes	Assigned Weights	Query	Source
D_l	$\forall x' \in M_l$	$w_2(x)$	1D ancestor dominance reporting	Lemma 3.6
A_l	$\forall x' \in M_l$	$w_1(x)$	weighted ancestor	Lemma 3.7
E_l	$\forall x' \in M_l$	$w_1(x)$	1D ancestor dominance reporting	Lemma 3.6
F_l	$\forall x' \in N_l$	$(w_1(x), \texttt{label}(x''))$, $x'' \in T_l$	$(1, 2, \epsilon)$-dimensional ancestor dominance	Lemma 3.6

Table 3.1: Summary table for the data structures D_l, A_l, E_l, and F_l built in Lemma 3.8. Denoted by $\mathbf{w}(x)$ is the original weight of the node $x \in T$ that corresponds to x'. In the last row, $x'' \in T_l$ is the node corresponding to x.

siblings. If a (range tree) node is strictly to the right of the search path, we are left with only the first weight to worry about. Otherwise, the second weight is eliminated using the monotonicity property of the maximal nodes. Finding out the right siblings to explore is a "meta" query of its own, which is now $(1, 2, \epsilon)$-dimensional owing to "small" second weights.

Having thus dealt with a higher-level view, we now describe the algorithm for answering queries in detail, and analyze its time- and space complexity:

Lemma 3.8. *A tree T on n nodes, in which each node is assigned a 2-dimensional weight vector, can be represented in $\mathcal{O}(n)$ words, so that an ancestor dominance reporting query can be answered in $\mathcal{O}(\lg n + k)$ time, where k is the number of the nodes reported.*

Proof. Let x and $\mathbf{q} = (q_1, q_2)$ be the node and the weight vector given as query parameters, respectively. We define Π as the path in the range tree between and including the root and the leaf storing q_2. Let π_l denote the node at level l in Π; then the root of the range tree is π_1. To answer the query, we perform a traversal of a subset of the nodes of the range tree, starting from π_1. The invariant maintained during this traversal is that a node u of the range tree is visited *iff* one of the following two conditions holds: (i) $u = \pi_l$ for some l; or (ii) $M(u)$ contains at least one node whose corresponding node in T must be reported. We now describe how the algorithm works when visiting a node, v, at level l of this range tree, during which we shall show how the invariant is maintained. Let x_v denote the node in T_l that corresponds to the $\mathscr{T}_v$-view of x; x_v can be located in constant time each time we descend down one level in the range tree, as described in the proof of Lemma 3.4. Our first step is to report all the nodes in the answer to the query that have corresponding nodes in $M(v)$. There are two cases depending on whether $v = \pi_l$; this condition can be checked in constant time by determining whether q_2 belongs to the range represented by v. In either of these cases, we first locate the M_l-view, x'_v, of x_v, using Proposition 2.1.

If $v = \pi_l$, then the non-dummy ancestors of x'_v in M_l correspond to all the ancestors of x in T that have corresponding nodes in $M(v)$. We then perform a weighted ancestor query using A_l to locate the highest ancestor, y, of x'_v in M_l whose 1^{st} weight is at least q_1. Since the 1^{st} weights of the nodes along any upward path in M_l are decreasing, the 1^{st} weights of the nodes in path $P_{x'_v,y}$ are greater than or equal to q_1, while those of the proper ancestors of y are strictly less. Hence, by performing a 1-dimensional path dominance reporting query in D_l using $P_{x'_v,y}$ as the query path and $\mathbf{q}' = (q_2)$ as the query weight vector, we can find all the ancestors of x'_v whose corresponding nodes in T have weight vectors dominating $\mathbf{q}$. Then, for each of these nodes, we retrieve from P_l its corresponding node in T which is further reported.

If $v \neq \pi_l$, the maintained invariant guarantees that the 2^{nd} weights of the nodes in $M(v)$ are greater than q_2. Therefore, by performing a 1-dimensional path dominance reporting query in E_l using the path between (inclusive) x'_v and the root of M_l as the query path and $\mathbf{q}'' = (q_1)$ as the query weight vector, we can find all the ancestors of x'_v in M_l whose corresponding nodes in T have weight vectors dominating $\mathbf{q}$. By mapping these nodes to nodes in T via P_l, we have reported all the nodes in the answer to the query that have corresponding nodes in $M(v)$.

After we handle both cases, the next task is to decide which children of v we should visit. Let v_i denote the i^{th} child of v. We always visit π_{l+1} if it happens to be a child of v. To maintain the invariant, for any other child v_i, we visit it *iff* there exists at least one node in $M(v_i)$ whose corresponding node in T should be reported. To find the children that we will visit, we locate the N_l-view, x''_v, of x_v, using Proposition 2.1. Then the non-dummy ancestors of x''_v correspond to all the ancestors of x in T that have corresponding nodes in $\cup_{i=1,2,\ldots} M(v_i)$. We then perform a $(1, 2, \epsilon)$-dimensional path dominance reporting query in F_l using the path between (inclusive) x''_v and the root of N_l as the query path and $(q_1, \kappa + 1)$ as the query weight vector if π_{l+1} is the κ^{th} child of v, and we set $\kappa = 0$ if π_{i+1} is not a child of v. For each node, t, returned when answering this query, if its 2^{nd} weight in F_l is j, then t corresponds to a node in $M(v_j)$. Since the node corresponding to t in T should be included in the answer to the original query, we iteratively visit v_j if we have not visited it before (checked using an f-bit word to flag the children of v). Figure 3.5 illustrates the considerations in this paragraph.

The total query time is dominated by the time used to perform queries using A_l, D_l, E_l and F_l. We only perform one weighted ancestor query when visiting each π_l, and this query is not performed when visiting other nodes of the range tree. Given the $\mathcal{O}(\lg n/\lg\lg n)$ levels of the range tree, all the weighted ancestor queries collectively use $\mathcal{O}(\lg\lg n \times (\lg n/\lg\lg n)) = \mathcal{O}(\lg n)$

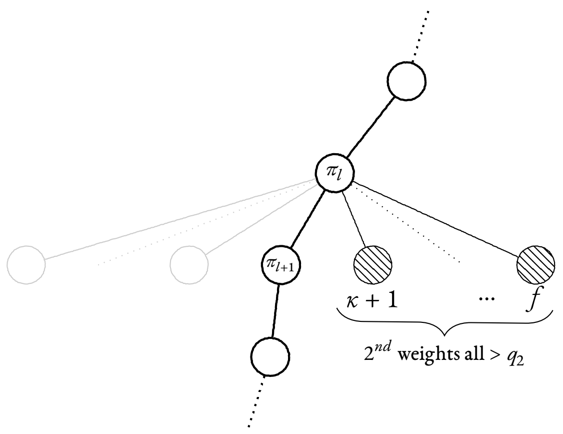

FIGURE 3.5: An illustration to the proof of Lemma 3.8. When visiting π_{l+1}, we use the A_l structure to adjust query parameters. When deciding which children, among $\kappa + 1, \ldots, f$, to visit, we use the F_l structure.

time. Similarly, we perform one query using D_l at each level of the range tree, and the query times summed over all levels is $\mathcal{O}(\lg n/\lg\lg n + k)$. Our algorithm guarantees that, each time we perform a query using E_l, we report a not-reported hitherto, non-empty subset of the nodes in the answer to the original query. Therefore, the queries performed over all E_ls use $\mathcal{O}(k)$ time in total.

Querying the F_l-structures is $\mathcal{O}(k)$ time when visiting nodes not in Π, and $\mathcal{O}(\frac{\lg n}{\lg\lg n} + k)$ time when visiting nodes in Π. Indeed, F_l is built using Lemma 3.6 and therefore has $\mathcal{O}(1 + k)$ query time. Per a range-tree node in Π we pay $\mathcal{O}(1)$ to initiate the search, hence the $\mathcal{O}(\frac{\lg n}{\lg\lg n})$ additive factor. On the other hand, the $\mathcal{O}(1)$-time cost of initiating the search is charged to a tree node that has been reported during the visit of the current range-tree node outside of Π.

We thus conclude that the query times spent on all these structures throughout the execution of the algorithm sum up to $\mathcal{O}(\lg n + k)$.

Next, we analyze the space cost of our data structures. As mentioned in Section 3.2.2, all the T_ls occupy $n + o(n)$ words. By Lemma 2.4, each T'_l or T''_l uses $3n + o(n)$ bits, so over all $\lg n/\lg\lg n$ levels, they occupy $\mathcal{O}(n\frac{\lg n}{\lg\lg n})$ bits, which is $\mathcal{O}(n/\lg\lg n)$ words. As discussed earlier, we know that, for any node x in T, there exists one and only one node v in the range tree such that there is a node in $M(v)$ corresponding to x. Furthermore, $M(v)$s only contain nodes that have corresponding nodes in T. Therefore, the sum of the sizes of all the $M(v)$s is precisely n. Hence all the P_ls have n entries in total and thus use n words. By Lemma 3.6, the size of each D_l in words is linear in the number of nodes in M_l. The sum of the numbers of nodes in the M_ls over all levels of the range tree is equal to the sum of the sizes of all the $M(v)$s plus the number of dummy roots, which is $n + \mathcal{O}(\lg n/\lg\lg n)$. Therefore, all the D_ls occupy $\mathcal{O}(n)$ words. By similar reasoning, all the E_ls and A_ls occupy $\mathcal{O}(n)$ words in total. Finally, it is also true that, for any node x in T, there exists a unique node v in the range tree such that there is a node in $N(v)$ corresponding to x. Thus, we can upper-bound the total space cost of all the F_ls by $\mathcal{O}(n)$ words in a similar way. All our data structures, therefore, use $\mathcal{O}(n)$ words. □

3.4.3 Ancestor Dominance Reporting in $(2, d, \epsilon)$ and Generalization to Higher Dimensions

Further, we describe the data structure for $(2, d, \epsilon)$-dimensional ancestor dominance reporting, and analyze its time- and space-bounds:

Lemma 3.9. *Let $d \geq 2$ be a constant integer and $0 < \epsilon < \frac{1}{d-2}$ be a constant number. A tree T on n nodes, in which each node is assigned a $(2, d, \epsilon)$-dimensional weight vector, can be represented in $\mathcal{O}(n \lg^{(d-2)\epsilon} n)$ words, so that an ancestor dominance reporting query can be answered in $\mathcal{O}(\lg n + k)$ time, where k is the number of the nodes reported.*

Proof. In our design, for any $(0, d-2, \epsilon)$-dimensional vector $\mathbf{g}$, we consider a conceptual 1D-weighted tree $E_{\mathbf{g}}$ as the tree extraction from T of the node set $\{x \mid x \in T \text{ and } \mathbf{w}_{3,d}(x) \succeq \mathbf{g}\}$. The weight of a node x' in $E_{\mathbf{g}}$ is the (2-dimensional) weight vector $\mathbf{w}_{1,2}(x)$, where x the T-source of x'. If $E_{\mathbf{g}}$ has a dummy root, then its weight is $(-\infty, -\infty)$. Rather than storing $E_{\mathbf{g}}$ explicitly, we follow the strategy in the proof of Lemma 3.6 and store indicator tree $T_{\mathbf{g}}$ for each possible $\mathbf{g}$. We also maintain arrays W_1 and W_2 storing respectively the 1^{st} and 2^{nd} weights of all nodes of T, in preorder, which enables accessing the weight of an arbitrary node of $E_{\mathbf{g}}$ in $\mathcal{O}(1)$ time. Let $n_{\mathbf{g}}$ be the number of nodes in $E_{\mathbf{g}}$. We convert the node weights of each $E_{\mathbf{g}}$ to rank space $[n_{\mathbf{g}}]$. For each such $E_{\mathbf{g}}$, we build the 2-dimensional ancestor dominance reporting data structure, $V_{\mathbf{g}}$, from Lemma 3.8. Thus, the space usage of the resulting data structure is upper-bounded by $\mathcal{O}(n \lg^{(d-2)\epsilon} n)$ words.

Let x and $\mathbf{q} = (q_1, q_2, \ldots, q_d)$ be the node and the weight vector given as query parameters, respectively. In $\mathcal{O}(1)$ time, we fetch the data structures pertaining to the weight vector $\mathbf{q}' = \mathbf{q}_{3,d}$; this way, all the weights 3 through d of the query vector have been taken care of, and all we need to consider is the tree topology and the first two weights, q_1 and q_2, of the original query vector. We localize the query node x to $E_{\mathbf{q}'}$ by finding the $E_{\mathbf{q}'}$-view x' of x via Proposition 2.1 and launch the query in $V_{\mathbf{q}'}$ with x' as a query node, having reduced the components of the query vector (q_1, q_2) to the rank space of $E_{\mathbf{q}'}$. The time- and space-bounds for the reductions are absorbed in the final bounds. Indeed, adjusting the path in order to query $E_{\mathbf{q}'}$ is constant-time, whereas rank-space reduction is $\mathcal{O}(\lg\lg n)$ time using y-fast tries [107]. $\square$

Instantiating Section 3.3 with $g(x) = \{x\}$ and the semigroup sum operator $\oplus$ as the set-theoretic union operator $\cup$, Lemma 3.4 iteratively applied to Lemma 3.8 yields

Theorem 3.1. *Let $d \geq 2$ be a constant integer. A tree T on n nodes, in which each node is assigned a d-dimensional weight vector, can be represented in $\mathcal{O}(n \lg^{d-2} n)$ words, so that an ancestor dominance reporting query can be answered in $\mathcal{O}(\lg^{d-1} n + k)$ time, where k is the number of the nodes reported.*

Analogously, Lemma 3.5 which is the counterpart of Lemma 3.4 when the range tree has a non-constant branching factor $f = \mathcal{O}(\lg^{\epsilon} n)$, with Lemma 3.9 which concerns itself with the $(2, d, \epsilon)$-dimensional ancestor reporting, together yield a different trade-off:

Theorem 3.2. *Let $d \geq 3$ be a constant integer. A tree T on n nodes, in which each node is assigned a d-dimensional weight vector, can be represented in $\mathcal{O}(n \lg^{d-2+\epsilon} n)$ words of space, so that an ancestor dominance reporting query can be answered in $\mathcal{O}((\lg^{d-1} n)/(\lg\lg n)^{d-2} + k)$ time, where k is the number of the nodes reported. Here, ϵ is an arbitrary constant in $(0, 1)$.*

3.5 Path Successor

We first solve the path successor problem when $d = 1$, and extend the result to $d > 1$ via Lemma 3.4.

The topology of T is stored using Lemma 2.4. We define a binary range tree R over $[n]$, and build the associated hierarchical tree extraction as in Section 3.2.2; as in Section 3.4, T_l denotes the auxiliary tree built for each level l of R, and $\mathcal{T}_v$ denotes the tree extraction from T associated with the range of node $v \in R$. We represent R using Lemma 2.4, and augment it with the ball-inheritance data structure $\mathcal{B}$ from Lemma 3.2(a), as well as with the data structure from the following

Lemma 3.10. *Let R be a binary range tree with topology encoded using Lemma 2.4, and augmented with the ball-inheritance data structure $\mathcal{B}$ from Lemma 3.2(a). With an additional space of $\mathcal{O}(n)$ words, the node $x_{u,l}$ in T_l corresponding to the $\mathcal{T}_u$-view of x can be found in $\mathcal{O}(\lg^{\epsilon'} n)$ time, for an arbitrary node $x \in T$ and an arbitrary node $u \in R$ residing on a level l. Here, $0 < \epsilon' < 1$ is an arbitrarily small constant.*

Proof. For a node $u \in R$ at a level l, and a node $x \in T$, the query can be thought of as a chain of transformations $T \to \mathscr{T}_u \to T_l$. In the first transition, $T \to \mathscr{T}_u$, given an original node $x \in T$, we are looking for its $\mathscr{T}_u$-view, x_u. That is, although $\mathscr{T}_u$ is (conceptually) obtained from T through a *series* of extractions (i.e. as we construct the range tree), the wish is to "jump" many successive extractions at once, as if $\mathscr{T}_u$ were extracted from T *directly*. This would be trivial to achieve through storing an indicator tree per range u, i.e. for each pair $(T, \mathscr{T}_u)$, if it were not for a prohibitive space-cost – the number of bits at least quadratic in the number of nodes. One can avoid extra space cost altogether and directly use Lemma 3.1 to explicitly traverse the hierarchy of extractions. In this case, the time cost is proportional to the height of the range tree, and hence becomes the bottleneck.

In turn, in the $\mathscr{T}_u \to T_l$-transition, we are looking for the identity of x_u in T_l. For this second transformation, we recall (from Section 3.2.2) that $\mathscr{T}_u$ is embedded within T_l, and the nodes of $\mathscr{T}_u$ must lie contiguously in the preorder sequence of T_l.

With these considerations in mind, we build the following data structures.

For R, we maintain an annotation array I, such that $I[u]$ stores a quadruple $\langle a_u, b_u, s_u, t_u \rangle$ for an arbitrary node $u \in R$ at level l, such that (i) the weight range associated with u is $[a_u, b_u]$; and (ii) all the nodes of T with weights in $[a_u, b_u]$ occupy precisely the preorder ranks s_u through t_u in T_l. The space occupied by the annotation array I is clearly $\mathcal{O}(n)$ words (the number of nodes in R is $\mathcal{O}(n)$).

For each level L that is a multiple of $\lceil \lg \lg n \rceil$, which we call a *marked* level, we maintain a data structure enabling the direct $T \to \mathscr{T}_u \to T_L$-conversion. Namely, for each individual node u on marked level L of R, we define a conceptual array A_u, which stores, in increasing order, the (original) preorder ranks of all the nodes of T whose weights are in the range represented by u. Rather than maintaining A_u explicitly, we store a succinct index, S_u, for predecessor/successor search [56] in A_u. Assuming the availability of a $o(n^\delta)$-bit universal table, where $0 < \delta < 1$ is a constant, given an arbitrary value in $[n]$, this index can return the position of its predecessor/successor in A_u in $\mathcal{O}(\lg \lg n)$ time plus accesses to $\mathcal{O}(1)$ entries of A_u. The size of the index in bits is $\mathcal{O}(\lg \lg n)$ times the number of entries in A_u. For a marked (in fact, for any) level L all the S_u-structures thus sum up to $\mathcal{O}(n \lg \lg n)$ bits. There being $\mathcal{O}(\lg n / \lg \lg n)$ marked levels, the total space cost for the S_u-structures over the entire tree R is $\mathcal{O}(n)$ words.

Let us show how to answer queries using the data structures built. Resolving a query falls into two distinct cases. The first is when the level l, at which the query node u resides, is marked; the second is when it is not.

When the level l is marked, we use the structures S_u stored therein, directly. We adopt the strategy of He et al. [69] to find $x_{u,l}$. First, for an arbitrary index i to A_u, we observe that node $A_u[i] \in T$ corresponds to the node $(s_u + i - 1)$ in T_l. We thus fetch $\langle a_u, b_u, s_u, t_u \rangle$ from $I[u]$. Then the predecessor $p \in A_u$ of x is obtained through S_u via an $\mathcal{O}(\lg \lg n)$-time query and $\mathcal{O}(1)$ calls to the $\mathcal{B}$-structure, which results in $\mathcal{O}(\lg^{\epsilon'} n)$ time in total. We then determine the lowest common ancestor $s \in T$ of x and p, in $\mathcal{O}(1)$ time.

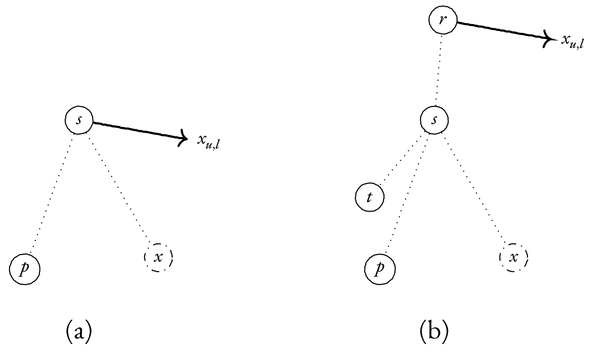

FIGURE 3.6: An illustration to the proof of Lemma 3.10. Node x, which could have weight in u's range or not, is represented by a *dash-dotted* circle. The subfigures (a) and (b) respectively represent the cases in which the weight of s is in and not in u's range.

If the weight of s is in $[a_u, b_u]$, then it must be present in A_u by the latter's very definition. By another predecessor query, therefore, we can find the position, j, of s in A_u, and $(s_u + j - 1)$ is the sought $x_{u,l}$. This case subsumes all the corner cases, too: If the node x had the weight from range $[a_u, b_u]$ itself, the predecessor query would duly return $p = x$; if $p \neq x$ *and* $p \in \mathcal{A}(x)$, then $p = s$.

If the weight of s is not in $[a_u, b_u]$, a final query to S_u returns the successor t in A_u of s; the node t must exist because of p. Let κ be the position of t in A_u. Then the parent of the node $(s_u + \kappa - 1)$ in T_l is the sought node $x_{u,l}$. Indeed, the node $t \notin \mathcal{A}(x)$, because otherwise the very first predecessor query would have returned t and we would have already found $x_{u,l}$. Figure 3.6 illustrates the discussions of this and the previous paragraphs.

We perform a constant number of predecessor/successor queries, and correspondingly a constant number of calls to the ball-inheritance problem. The time complexity is thus $\mathcal{O}(\lg^{\epsilon'} n)$.

When the level l is *not* marked, we ascend to the lowest ancestor u' of u residing on a marked level l', and reduce the problem to the previous case. More precisely, via the navigation operations (`level_anc` to move to a parent, and `depth` to determine whether the level is marked) available through R's encoding, we climb up at most $\lceil \lg\lg n \rceil$ levels to the closest marked level l'. Let u' be therefore the ancestor of u found on that marked level l'. We then find the node $x'_{u',l'}$ in $T_{l'}$ that corresponds to the $\mathscr{T}_{u'}$-view of the node x in T. Afterwards, we initialize a variable s to be $x_{u',l'}$. At each level $l' \le l'' \le l$ this variable s represents the node $x''_{u'',l''} \in T_{l''}$ that corresponds to the $\mathscr{T}_{u''}$-view of the node $x \in T$; here u'' is the node of the range tree such that it is an ancestor of u and a descendant of u'. We descend down to the original level l, back to the original query node u, all the while adjusting the node s as we move down a level, analogously to the proof of Lemma 3.4. As we arrive, in time $\mathcal{O}(\lg\lg n)$, at node u, the variable s stores the answer, $x_{u,l}$.

In the second case, too, the term $\mathcal{O}(\lg^{\epsilon'} n)$ dominates the time complexity, as climbing from the current level up to the closest marked level[5] and back is an additive term of $\mathcal{O}(\lg\lg n)$. Therefore, the query is answered in $\mathcal{O}(\lg^{\epsilon'} n)$ time. □

Finally, each T_l is augmented with succinct indices m_l (resp. M_l) from Lemma 3.3(b), for path minimum (resp. path maximum) queries; here, the weights of the nodes of T_l are the weights of their corresponding nodes in T. We now describe the algorithm for answering queries and analyze its running time, as well as give the space cost of the data structures built:

Lemma 3.11. *A 1D-weighted tree T on n nodes can be represented in $\mathcal{O}(n)$ words, so that a path successor query is answered in $\mathcal{O}(\lg^{\epsilon} n)$ time, where $0 < \epsilon < 1$ is an arbitrary constant.*

Proof. Let x, y and $Q = [q_1, q'_1]$ be respectively the nodes and the orthogonal range given as query parameters. As in the proof of Lemma 3.4, we focus only on the path $A_{x,z}$, where z is $\texttt{LCA}(T, x, y)$. We locate in $\mathcal{O}(1)$ time the leaf L_{q_1} of R that corresponds to the singleton range $[q_1, q_1]$. Let Π be the root-to-leaf path to L_{q_1} in R; let π_l be the node at level l of Π. We binary search in Π for the deepest node $\pi_f \in \Pi$ whose associated extraction $\mathscr{T}_{\pi_f}$ contains the node corresponding to the answer to the given query, as follows.

We initialize two variables: $high$ as 1 so that π_{high} is the root of R, and low as the height of R so that π_{low} is the leaf L_{q_1}. We first check whether $\mathscr{T}_{\pi_{low}}$ already contains the answer, by fetching the node x' in T_{low} corresponding to the $\mathscr{T}_{\pi_{low}}$-view of x, using Lemma 3.10. If x' exists, we examine its corresponding node x'' in T (fetched via $\mathcal{B}$). We check whether x'' is

[5]One could reach the marked level in one leap, in $\mathcal{O}(1)$ time, using `level_anc`. This however has no bearing on the asymptotical time bound, because we later descend to the original level one level at a time, anyway.

on $A_{x,z}$, using `depth` and `level_anc` operations. If $x'' \in A_{x,z}$, then x'' is the final answer. If not, this establishes the invariant of the ensuing search: $\mathscr{T}_{\pi_{high}}$ contains a node corresponding to the answer, whereas $\mathscr{T}_{\pi_{low}}$ does not.

At each iteration, therefore, we set (via `level_anc` in R) π_{mid} to be the node mid-way from π_{low} to π_{high}. We then fetch the nodes x', z' in T_{mid} corresponding to the $\mathscr{T}_{\pi_{mid}}$-views of respectively x and z, using Lemma 3.10. The non-existence of x' or the emptiness of $A_{x',z'}$ sets low to mid, and the next iteration of the search ensues. (If z' does not exist, z' is set to the root of T_{mid}.) A query to the M_{mid}-structure then locates a node in $A_{x',z'}$ for which the 1st weight, μ, of its corresponding node in T is maximized. Accounting for the mapping of a node in T_{mid} to its corresponding node in T via $\mathcal{B}$, this query uses $\mathcal{O}((\lg^{\epsilon'} n)\alpha(n))$ time. The variables are then updated as $high \leftarrow mid$ if $\mu \geq q_1$, and $low \leftarrow mid$, otherwise.

Once π_f is located, it must hold for π_f that (i) it is its left child that is on Π; and (ii) its right child, v, contains the query result, even though v represents a range of values all larger than q_1 [95]. When locating π_f, we also found the nodes in T_f corresponding to the $\mathscr{T}_{\pi_f}$-views of x and z; they can be further used to find the nodes in T_{f+1}, x^* and z^*, corresponding to the $\mathscr{T}_v$-views of x and z. We then use m_{f+1} to find the node in A_{x^*,z^*} with the minimum 1st weight, whose corresponding node in T is the answer.

The total query time is determined by that needed for binary search. An iteration of the binary search is in turn dominated by the path maximum query in T_{mid}, which is $\mathcal{O}((\lg^{\epsilon'} n)\alpha(n))$. Given the $\mathcal{O}(\lg n)$ levels of R, the binary search has $\mathcal{O}(\lg\lg n)$ iterations. Therefore, the total running time is $\mathcal{O}(\lg\lg n \cdot \lg^{\epsilon'} n \cdot \alpha(n)) = \mathcal{O}(\lg^{\epsilon} n)$ if we choose $\epsilon' < \epsilon$.

To analyze the space cost, we observe that the topology of T, represented using Lemma 2.4, uses only $2n + o(n)$ bits. As stated in Section 3.2.2, all the structures T_l occupy $\mathcal{O}(n)$ words in total. The space cost of the structure from Lemma 3.10 built for R is $\mathcal{O}(n)$ words. The $\mathcal{B}$-structure occupies another $\mathcal{O}(n)$ words. The m_l- and M_l-structures occupy $\mathcal{O}(n)$ bits each, or $\mathcal{O}(n)$ words in total over all levels of R. Thus, the final space cost is $\mathcal{O}(n)$ words. □

Lemmas 3.4 and 3.11 yield the following

THEOREM 3.3. *Let $d \geq 1$ be a constant integer. A tree T on n nodes, in which each node is assigned a d-dimensional weight vector can be represented in $\mathcal{O}(n\lg^{d-1} n)$ words, so that a path successor query can be answered in $\mathcal{O}(\lg^{d-1+\epsilon} n)$ time, for an arbitrarily small positive constant ϵ.*

Proof. We instantiate Section 3.3 with $g(x) = x$ and the semigroup sum operator $\oplus$ as $x \oplus y =$

$\arg\min_{t\in\{x,y\}}\{w_1(t)\}$. Lemma 3.4 applied to Lemma 3.11 yields the space bound of $\mathcal{O}(n \lg^{d-1} n)$ words and the query time complexity of $\mathcal{O}(\lg^{d-1+\epsilon} n)$. □

3.6 Path Counting

He et al. [70] have solved the path counting problem for 1D-weighted trees. Using the standard technique of range trees, one accommodates the remaining $(d-1)$ weights at the cost of polylogarithmic factors both in space and query time [93]. To do better than the straightforward approach, one can use the tree covering technique, described in Section 2.3.2.

In this section we design a data structure to solve the path counting problem in the special case of the nodes being assigned $(0, d, \epsilon)$-dimensional weight vectors. It turns out that when the weight vector of a node can be packed into $o(\lg n)$ bits, counting queries can be executed in constant time. The key machinery used is tree covering.

Let T be a given ordinal tree on n nodes, each node of which is assigned a weight vector in $(0, d, \epsilon)$ dimensions. Let r be the root of T. In our solution to the $(0, d, \epsilon)$-dimensional path counting problem for T, we set $c = \lceil \lg^{\epsilon} n \rceil$, and use Lemma 2.5 to decompose T into mini-trees with parameter $L = c^{2d} \lg n$. Each of the mini-trees is further subject to decomposition into micro-trees with parameter $L' = c^{2d}$. We denote by r_b the root of a mini- or micro-tree b, for any b that shall be clear from the context. Each mini- or micro-tree b stores an array $b.cnt$, indexed by a tuple from $([c] \times [c])^d$, with the following contents:

- for a mini-tree b, an $\mathcal{O}(\lg n)$-bit number $b.cnt[Q]$ stores the number of the nodes with weight vectors falling within the range Q on the path $A_{r_b,r}$ in T, where r is the root of T;
- for a micro-tree b' inside a mini-tree b, an $\mathcal{O}(\lg\lg n)$-bit number $b'.cnt[Q]$ stores the number of the nodes with weight vectors falling within the range Q on the path $A_{r_{b'},r_b}$.

We also pre compute a look-up table D that is indexed by a quadruple from the following Cartesian product:

- all possible micro-tree topologies τ, *times*
- all possible assignments λ of weight vectors to the nodes of τ, *times*
- all nodes in τ, *times*

- all possible query orthogonal ranges Q.

Let λ be a labeling of a micro-tree τ with $(0, d, \epsilon)$-dimensional weight vectors. Then the entry $D[\tau, \lambda, x, Q]$ stores the number of nodes on the path A_{x,r_τ} in τ, such that their weight vectors belong to the range Q.

We now show how to use these data structure to answer queries.

LEMMA 3.12. *The data structures in this section can answer a $(0, d, \epsilon)$-dimensional path counting query in $\mathcal{O}(1)$ time, for any constant integer $d \geq 1$.*

Proof. Let $P_{x,y}$ and Q be, respectively, the path and the orthogonal range given as the parameters to the query. For the reasons given in Lemma 3.4, we describe only how to answer the query over $A_{x,z}$, where $z = \texttt{LCA}(T, x, y)$. We assume the encoding of T as in Lemma 2.4, so the $\texttt{LCA}$ operator is available; the space overhead is only $\mathcal{O}(n)$ bits, i.e. negligible with respect to the space bound we are ultimately aiming at.

We further notice that answering the path counting query over $A_{x,z}$ is equivalent to answering two path counting queries, one over $A_{x,r}$ and another over $A_{z,r}$, and taking their arithmetic difference. It is thus sufficient to describe the procedure of answering the query over $A_{x,r}$, and analyze its running time, as the query over $A_{z,r}$ can be handled similarly.

The key observation is that overall we perform a constant number of constant-time operations. Indeed, using the data structures for navigating the mini- and micro-trees [39], we first identify, in $\mathcal{O}(1)$ time, the mini-tree b and the micro-tree b' containing the node x, as well as the encoding of b'. From the (disjoint) decomposition of the path $A_{x,r} = A_{x,r_{b'}} \cup A_{r_{b'},r_b} \cup A_{r_b,r}$, it is now immediate that the answer to the query over $A_{x,r}$ is $\texttt{res}_x = D[b', \texttt{lab}(b'), x', Q] + b.cnt[Q] + b'.cnt[Q]$, where $\texttt{lab}(b')$ is the labeling of the micro-tree b', and x' is the preorder number of the node $x \in T$ in b', provided by the standard encodings [51, 67, 39]. One finds the answer $\texttt{res}_z$ to the path counting query over $A_{z,r}$ analogously. Then the final answer is $\texttt{res}_x - \texttt{res}_z$. Therefore, our algorithm runs in $\mathcal{O}(1)$ time. □

We now analyze the space cost of our data structures.

LEMMA 3.13. *The data structures in this section occupy $\mathcal{O}(n \lg\lg n)$ bits when $\epsilon \in (0, \frac{1}{4d})$.*

Proof. To analyze the space cost, we tally up the costs of the main constituents of our data structure: the *cnt*-arrays stored at the roots of each mini- and micro-tree, and the D-table.

There being $\Theta(\frac{n}{c^{2d}\lg n})$ mini-trees, each of which contains an array of c^{2d} elements, $\mathcal{O}(\lg n)$ bits each, the associated cnt-arrays contribute $\mathcal{O}(\frac{n}{c^{2d}\lg n} \times c^{2d} \times \lg n) = \mathcal{O}(n)$ bits.

Analogously, for the $\Theta(n/c^{2d})$ micro-trees, the net contribution of the associated cnt-arrays is $\mathcal{O}(n/c^{2d} \times c^{2d} \times \lg(c^{2d}\lg n)) = \mathcal{O}(n\lg\lg n)$ bits, which is the space claimed in the statement.

It suffices to show, therefore, that the space occupied by the D-structure can not exceed $\mathcal{O}(n\lg\lg n)$ bits; as demonstrated below, it is much less. Indeed, as mentioned in Lemma 2.5, each micro-tree can have up to $2L' = 2c^{2d}$ nodes, which gives us less than $2^{2\cdot 2L'} = 2^{2\cdot 2c^{2d}}$ possible topologies τ. In turn, each of the $2c^{2d}$ nodes can independently be assigned c^d possible weight vectors; hence the number of possible configurations λ is at most $(c^d)^{2c^{2d}}$ (the number of strings of length $2c^{2d}$ over alphabet $[c^d]$). Furthermore, the c^{2d} query orthogonal ranges Q make for $2c^{2d} \times c^{2d} = 2c^{4d}$ distinct queries, i.e. the number of nodes times the number of ranges. The number of entries in the table D is thus $\mathcal{O}(2^{4c^{2d}} \cdot (c^d)^{2c^{2d}} \cdot 2c^{4d}) = \mathcal{O}((4c^d)^{2c^{2d}} c^{4d})$. The term c^{4d} is $o(\lg n)$. To upper-bound the term $(4c^d)^{2c^{2d}}$, we notice that $c^d = \lceil \lg^{\epsilon} n \rceil^d < \sqrt[4]{\lg n}/4$ for sufficiently large n. Therefore, we have the following chain of inequalities for sufficiently large n:

$$\begin{aligned} (4c^d)^{2c^{2d}} &< \left(\sqrt[4]{\lg n}\right)^{2c^{2d}} < \left(\sqrt[4]{\lg n}\right)^{2\sqrt{\lg n}} = \left(\sqrt{\lg n}\right)^{\sqrt{\lg n}} \\ &= \left(2^{\lg\sqrt{\lg n}}\right)^{\sqrt{\lg n}} = 2^{\sqrt{\lg n}\cdot\lg\sqrt{\lg n}} < 2^{\sqrt{\lg n}\cdot\frac{\sqrt{\lg n}}{2}} = \sqrt{n} \end{aligned}$$

Thus, the number of entries in D is at most $\mathcal{O}(\sqrt{n}\lg^{4d\epsilon} n)$. Each entry holding a value of $\mathcal{O}(\lg c^{2d}) = \mathcal{O}(\lg\lg n)$ bits, the table D occupies $\mathcal{O}(\sqrt{n}\lg\lg n\lg^{4d\epsilon} n) = o(n)$ bits, in total.

Finally, as shown above, the number of ways to assign weight vectors to nodes of a micro-tree is a $\mathcal{O}(L'\lg\lg n)$-bit number. Thus, the storage space for the labelings of each of the $\Theta(n/L')$ micro-trees amounts to $\mathcal{O}(n\lg\lg n)$ bits. $\square$

With Lemmas 3.12 and 3.13, we have the following

LEMMA 3.14. *Let $d \geq 0$, $\epsilon \in (0, \frac{1}{4d})$ be constants. A tree T on n nodes, in which each node is assigned a $(0, d, \epsilon)$-dimensional weight vector, can be represented in $\mathcal{O}(n\lg\lg n)$ bits of space such that a path counting query is answered in $\mathcal{O}(1)$ time.*

Finally, instantiating Section 3.3.2 with $g(x) \equiv 1$ and $\oplus$ as the regular arithmetic addition operation $+$ in $\mathbb{R}$, we can apply Lemma 3.5 to Lemma 3.14 iteratively and obtain the following result:

Theorem 3.4. *Let $d \geq 1$ be a constant integer. A tree T on n nodes, in which each node is assigned a d-dimensional weight vector, can be represented in $\mathcal{O}(n(\lg n/\lg\lg n)^{d-1})$ words such that a path counting query can be answered in $\mathcal{O}((\lg n/\lg\lg n)^d)$ time.*

3.7 Path Reporting

In this section, we solve the path reporting problem. We use the following result of Chan et al. [19]:

Lemma 3.15 ([19]). *An ordinal tree on n nodes whose weights are drawn from $[n]$ can be represented using $\mathcal{O}(n\lg^{\epsilon} n)$ words of space, such that path reporting queries can be supported in $\mathcal{O}(\lg\lg n + k)$ time, where k is the number of reported nodes and ϵ is an arbitrary positive constant.*

Lemma 3.15 implies the following

Lemma 3.16. *Let $d \geq 1$ and $0 < \epsilon < \frac{1}{2(d-1)}$ be constants, and let T be an ordinal tree on n nodes, in which each node is assigned a $(1, d, \epsilon)$-dimensional weight vector. Then, T can be represented in $\mathcal{O}(n\lg^{\epsilon'} n)$ words of space, for any constant $\epsilon' \in (2(d-1)\epsilon, 1)$, so that a path reporting query can be answered in $\mathcal{O}(\lg\lg n + k)$ time, where k is the number of reported nodes.*

Proof. In brief, we build a path reporting data structure from Lemma 3.15 for each possible orthogonal range over the last $(d-1)$ dimensions. When presented with a query, we directly proceed to the appropriately-tagged (by the last $(d-1)$ weights) reporting structure, and launch the query therein. A detailed exposition follows.

We assume the encoding of T as in Lemma 2.4; the space incurred is only $\mathcal{O}(n)$ bits, i.e. negligible with respect to the terms derived below.

For any $(0, d-1, \epsilon)$-dimensional orthogonal range G, we build an explicit 1D-weighted tree E_G as the extraction of the node set $\{v \mid v \in T \text{ and } \mathbf{w}_{2,d}(v) \in G\}$ from T. The weight of a node in E_G is the 1st weight of its T-source. If E_G has a dummy root, then its weight is $-\infty$.

E_G is represented using Lemma 3.15, in space that is at most $\mathcal{O}(n\lg^{\delta} n)$ words, for an arbitrarily small positive δ. In order to adjust the nodes between T and E_G, indicator tree T_G of (T, E_G) is maintained.

Accounting for all possible ranges G, we therefore have $\mathcal{O}(n\lg^{\delta+2(d-1)\epsilon} n)$ words of space, in total. Setting δ to be sufficiently small and assigning $(\delta + 2(d-1)\epsilon)$ to ϵ' justifies the space claimed. All the T_G-structures collectively occupy $\mathcal{O}(n \lg^{2(d-1)\epsilon} n)$ bits of space, which is $o(n)$ words.

Let $P_{x,y}$ and Q be, respectively, the path and the orthogonal range given as the query parameters. Using the notation of Lemma 3.4, and for the same reasons as given therein, we concern ourselves only with answering the query over the path $A_{x,z}$, where $z = \texttt{LCA}(T, x, y)$. To answer the query, we first locate the relevant tree $E_{Q'}$, where $Q' = Q_{2,d}$, and launch a path reporting query in $E_{Q'}$, having adjusted the nodes x and z to their $T_{Q'}$-views as described in Proposition 2.1. Finally, the one-dimensional query in $E_{Q'}$ executes in $\mathcal{O}(\lg\lg n + k)$ time, by Lemma 3.15, thereby establishing the claimed time bound. Each returned node x's original identifier is recovered as in Proposition 2.1. □

Instantiating Section 3.3.2 with $g(x) = \{x\}$ and the semigroup sum operator $\oplus$ as the set-theoretic union operator $\cup$, Lemma 3.5 and Lemma 3.16 combined imply the following

THEOREM 3.5. *Let $d \geq 2$ be a constant integer. A tree T on n nodes, in which each node is assigned a d-dimensional weight vector, can be represented in $\mathcal{O}(n \lg^{d-1+\epsilon} n)$ words such that a path reporting query can be answered in $\mathcal{O}((\lg^{d-1} n)/(\lg\lg n)^{d-2} + k)$ time where k is the number of the nodes reported, for an arbitrarily small positive constant ϵ.*

3.8 Conclusion

This chapter proposes solutions to the ancestor dominance reporting, path successor, path counting and path reporting problems, over ordinal trees weighted with multidimensional weight vectors. These problems generalize the classical orthogonal range searching problems, to the case of one dimension being replaced by a tree topology.

In solving these problems, we combine diverse data-structuring ideas and propose a few novel techniques that could be of interest in their own right.

We propose an $\mathcal{O}(n \lg^{d-2} n)$ words-of-space data structure to answer an *ancestor dominance reporting* query in $\mathcal{O}(\lg^{d-1} n + k)$ time. For $d = 2$, this data structure provides a poly-logarithmic query time while matching the corresponding space bound for 3D dominance reporting of [2, 17]. For $d \geq 3$, it is possible to achieve a trade-off of $\mathcal{O}(n \lg^{d-2+\epsilon} n)$ words of space and $\mathcal{O}(\lg^{d-1} n/(\lg\lg n)^{d-2} + k)$ query time. Here, k is the size of the output.

We solve the *path successor problem* in $\mathcal{O}(n \lg^{d-1} n)$ words of space and time $\mathcal{O}(\lg^{d-1+\epsilon} n)$ for $d \geq 1$ and an arbitrary constant $\epsilon > 0$. We propose a solution to the *path counting problem*, with

$\mathcal{O}(n(\lg n/\lg\lg n)^{d-1})$ words of space and $\mathcal{O}((\lg n/\lg\lg n)^{d})$ query time, for $d \geq 1$. Finally, we solve the *path reporting problem* in $\mathcal{O}(n\lg^{d-1+\epsilon} n)$ words of space and $\mathcal{O}(\frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}} + k)$ query time, for $d \geq 2$. These results match or nearly match the best trade-offs of the respective range queries. We are also the first to solve path successor even for $d = 1$.

By the dint of any two nodes uniquely determining a path, tree topologies generalize one-dimensional arrays. And when tree degenerates into a single path, the path query problems and their Euclidean counterparts become identical. The big question is, how far this similarity stretches? And for which classes of problems can one hope to close the gap between the best results on the either side?

Chapter 4

Data Structures for Categorical Path Counting Queries

4.1 Introduction

Chapter 3 studied data structures for trees weighted with multidimensional weight vectors. The path queries studied therein generalized the classical orthogonal range searching problems in Euclidean space.

Going further, in some applications, of the actual interest (enshrined in the SQL clause `GROUP BY`) is not so much each individual point itself as the distinct *categories* occurring in the query region. For example, the number of distinct soil types in the query region is a a result of a *categorical range counting* query, if the points on a map are marked by soil types. These distinct values also find uses in SQL query optimization [23]. Categorical (or *coloured*) range searching is thus an area of active research in computer science [59, 4, 76, 78, 79, 57, 94, 48, 22, 20].

To motivate the study of categorical path queries, let us annotate a phylogenetic tree for a set of genomes by marking each speciation event with a *type* of mutation that caused it. The number of distinct mutation event types between two given species then could in some contexts serve as a proxy for evolutionary "distance" between them. A *categorical path counting* query, which asks for the number distinct categories on a query path, provides an adequate model in this case. These are the types of queries we consider in this chapter.

A few aspects make the categorical variants of range searching generally harder than their "plain" counterparts. First, there can be far fewer categories than points. Second, such problems are not easily decomposable — for two disjoint regions S_1 and S_2, knowing just the number of distinct categories in each of them is insufficient to infer the count for the union $S_1 \cup S_2$.

For all the progress in categorical *reporting* queries (see Section 3.1.1, and also [22, 20]), with results almost matching the state of the art in regular reporting [21], efficient *counting* (Section 3.6) in categorical setting remains elusive, with the currently best results standing at $\mathcal{O}(n^2 \lg^2 n)$ words and $\mathcal{O}(\lg^2 n)$ query time [59, 76], or $\mathcal{O}(n \lg^6 n)$ words and $\mathcal{O}(\sqrt{n} \lg^7 n)$ query time [76], versus the optimal linear-space and $\mathcal{O}(\frac{\lg n}{\lg \lg n})$-time data structure for 2D orthogonal range counting [74]. In contrast to the "plain" case, the categorical version of range counting is deemed to be harder than its reporting counterpart [76], when $d \geq 2$.

Generalizing the 1D categorical reporting problem, Durocher et al. [36] solved the *top-k colour reporting problem* on trees. We believe that in trees, neither the counting problem in the categorical setting, nor the scenario of weighted nodes has been studied before. In this chapter, we formalize these problems and propose solutions to them.

We consider an ordinal tree T on n nodes, such that each node z of T is associated with a category $\mathsf{c}(z) \in [\sigma]$. Specified at query time is a query path $P_{x,y}$ between two nodes x, y in T. We are to preprocess T into a data structure to compute in an efficient manner the number $n_{real} = |\{\mathsf{c}(z) \mid z \in P_{x,y}\}|$. This is the **categorical path counting problem** studied in this chapter.

Next, for $d \geq 1$, we consider a tree T in which each node, in addition to a category, is also associated with a weight vector $\mathbf{w}(z) \in [n]^d$. In addition to a query path $P_{x,y}$, specified at query time is then also an axis-aligned (hyper-)rectangle Q from $[n]^d$. We are to preprocess T into a data structure to compute in an efficient manner the number $n_{real} = |\{\mathsf{c}(z) \mid z \in P_{x,y} \wedge \mathbf{w}(z) \in Q\}|$. This is the **categorical path range counting problem** studied in this chapter.

For both problems, a *c-approximate* (for $c > 1$) answer is a number n_{appr} such that $n_{real} \leq n_{appr} \leq c \cdot n_{real}$. A $(1 \pm \epsilon)$-approximate (for $0 < \epsilon < 1$) answer is a number n_{appr} such that $\frac{|n_{appr} - n_{real}|}{n_{real}} \leq \epsilon$.

All these problems generalize the corresponding categorical range counting problems in Euclidean space $\mathbb{R}^{d+1}$, for respective d, by replacing one of the dimensions with a tree topology.

4.1.1 Previous Work

For points on a line, Gagie and Kärkkäinen [46] have proposed an $\mathcal{O}(n)$-word solution to the 1D categorical counting, with query time $\mathcal{O}(\lg^{1+\epsilon} n)$, for any $\epsilon > 0$. Nekrich [93] proposed another $\mathcal{O}(n)$-space solution with query time $\mathcal{O}(\lg \sigma / \lg\lg n)$, where σ is the number of categories.

Grossi and Vind [57] solve the 2D categorical range counting problem in linear space and $o(n)$ time, and higher-dimensional variants in almost-linear space and $o(n)$ time. The core idea is to divide the universe of categories into chunks of size $\lg n$, and use bitwise-`OR` when querying the restriction of the input set to each such chunk. The best result with polylogarithmic time in the two-dimensional categorical counting problem remains at $\mathcal{O}(n^2 \lg^2 n)$ words and $\mathcal{O}(\lg^2 n)$ query time [59, 76], with [76] also proposing an $\mathcal{O}(X \lg^7 n)$ query-time data structure with $\mathcal{O}((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$ storage space, for a trade-off parameter $1 \leq X \leq n$; for $X = \sqrt{n}$, the space is $\mathcal{O}(n \lg^6 n)$ and query time is $\mathcal{O}(\sqrt{n} \lg^7 n)$. Whereas Gupta et al. [59] use *persistence*, Kaplan et al. [76] proceed by a disjoint decomposition of the region covering an individual category, with the subsequent reduction to the *rectangle-stabbing problem* (the counting variant of which asks to count all the rectangles containing the query point). In higher dimensions ($d > 2$), [76] proposed an $\mathcal{O}(n^d \lg^{2d-2} n)$-word data structure with $\mathcal{O}(\lg^{2d-2} n)$ query time. They also show that an algorithm for categorical range counting in $\mathbb{R}^2$ that answers m queries over the set of $\mathcal{O}(n)$ points in $o(\min\{n, m\}^{\omega/2})$ time would yield an algorithm for obtaining the matrix product $M \cdot M^{\intercal}$ in $o(k^{\omega/2})$ time, for any $k \times k$ matrix M over $\{0, 1\}$, where ω is the best current exponent for Boolean matrix multiplication. Further, Kaplan et al. [76] proposed an $\mathcal{O}((\frac{n}{X})^{2d} + n \lg^{d-1} n)$-word data structure with $\mathcal{O}(X \lg^{d-1} n)$ query time, for a trade-off parameter $1 \leq X \leq n$. This implies an $\widetilde{\mathcal{O}}(n)$-space data structure with $\widetilde{\mathcal{O}}(n^{\frac{2d-1}{2d}})$ query time.

Nekrich [94] proposed an $\mathcal{O}(n(\lg\lg n)^2)$-word data structure to support $(4+\epsilon)$-approximate 2D categorical counting in $\mathcal{O}((\lg\lg n)^2)$ time, which also translates to a linear-size data structure for points on an $n \times n$ grid, that returns in $\mathcal{O}(1)$ time a $(1+\epsilon)$-approximation for the number of points in a 3-sided 2D query range, for a constant $0 < \epsilon < 1$. El-Zein et al. [37] solved the approximate categorical range counting problem in 1D in succinct $\mathcal{O}(n)$ bits of space and $\mathcal{O}(1)$ time. The core technique is to sample the prefixes of the array with exponentially increasing number of distinct categories covered, and "sandwich" the query point between two sampled values using transdichotomous data structures [42].

Lai et al. [78] used *sketching* data structures [28] to solve the approximate categorical range counting problem in probabilistic setting. In d dimensions, they propose a data structure of

$\mathcal{O}(dn \lg^{d-1} n)$ words of space, to support queries in $\mathcal{O}(d \lg^{d+1} n)$ time, with probability $1 - \delta$, where $0 < \delta < 1$ is a given constant. Sketches approximate the number of distinct categories occurring in a collection; they are small and additive, and in the solution of Lai et al. serve as summary structures.

To the best of our knowledge, the only categorical range searching problem considered so far for tree topologies is the *top-k colour reporting* problem, by Durocher et al. [36]. Therein, the categories have priorities, and the k highest-priority categories occurring on the given path are to be reported. Durocher et al. [36] introduce and solve this problem in (optimal) $\mathcal{O}(n)$ space and $\mathcal{O}(1 + k)$ time. They use heavy-path decomposition (Sleator and Tarjan [104], see also a review in Section 5.2) and *chaining* [88] to reduce the problem to two-dimensional reporting in a narrow grid.

4.1.2 Our Results

For the categorical path counting problem, we propose a linear-space data structure with query time $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$, where w is the word size on the word-RAM model. We show, by reduction from Boolean matrix multiplication, that the query time is optimal within polylogarithmic factors, with current knowledge and when only combinatorial methods are allowed. This conditional lower bound is surprising, because the 1D counterpart in the Euclidean case admits a linear-size solution with a sub-logarithmic query time, and a similar conditional lower bound can only be proven in 2D. In other words, having a tree structure in the presence of categories is about as hard as having a second dimension, making the query time go up from polylogarithmic to polynomial, when the space usage is linear. This is however is not the case in the previous work on path queries [36, 70, 64]. For a trade-off parameter $1 \le t \le n$, we propose an $\mathcal{O}(n + \frac{n^2}{t^2})$-word, $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$ query time data structure. We also describe a linear-space data structure that supports 2-approximate categorical path counting queries in $\mathcal{O}(\lg n / \lg \lg n)$ time. These problems have not been considered in trees before.

We also generalize the categorical path counting queries to weighted trees. For $d = 1$, we propose an $\mathcal{O}(n \lg \lg n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg \lg n)$ query time, or an $\mathcal{O}(n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg^{\epsilon} n)$ query time. This implies a linear-space data structure with $\mathcal{O}(n^{3/4} \lg^{\epsilon} n)$ query time. The corresponding $\mathcal{O}(n \lg^6 n)$-word solution to categorical range counting in $\mathbb{R}^2$ by [76] achieves $\mathcal{O}(\sqrt{n} \lg^7 n)$ query time. Compared to the best result in the Euclidean counterpart, we thus sacrifice an $\widetilde{\mathcal{O}}(\sqrt[4]{n})$-factor in query time, to accommodate the tree structure.

We further extend the approach to the trees weighted with multidimensional vectors from $[n]^d,\ d \geq 2$. We describe an $\mathcal{O}(n \lg^{d-1+\epsilon} n + \left(\frac{n}{t}\right)^{2d+2})$-word data structure with $\mathcal{O}(t\frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}})$ query time. For an $\widetilde{\mathcal{O}}(n)$-space solution, this yields $\widetilde{\mathcal{O}}(n^{\frac{2d+1}{2d+2}})$ query time. When $d \geq 2$, this result matches the best corresponding result in $\mathbb{R}^{d+1}$ by [76], within polylogarithmic factors.

Our sketching data structure for unweighted trees solves the approximate categorical path counting problem, which asks for a $(1 \pm \epsilon)$-approximation for the number of distinct categories on the given path, with probability $1 - \delta$. The data structure occupies $\mathcal{O}(n + \frac{n}{t}\lg n)$ words of space, for the query time of $\mathcal{O}(t \lg n)$. For trees weighted with d-dimensional weight vectors ($d \geq 1$), we propose an $\mathcal{O}((n + \frac{n}{t}\lg n)\lg^d n)$-word data structure with $\mathcal{O}(t \lg^{d+1} n)$ query time. Here, $0 < \epsilon, \delta < 1$ are arbitrarily small constants.

4.2 Categorical Path Counting

In this section, we consider the categorical path counting problem in the exact and approximate formulations. First, in Section 4.2.1 we prove a conditional lower bound on the categorical path counting problem in unweighted trees. Then, Section 4.2.2 offers some background on the techniques used in our data structures. In Section 4.2.3, we design a data structure that matches the lower bound within polylogarithmic factors when only combinatorial approaches are allowed. Finally, in Section 4.2.4 we design a 2-approximate solution with much faster query time.

4.2.1 Hardness of Categorical Path Counting

In this section we show a reduction from the Boolean matrix multiplication (i.e. matrix multiplication over $\{0, 1\}$ such that the multiplication and addition are replaced by respectively `AND` and `OR`) problem to the categorical path counting problem over unweighted trees.

Theorem 4.1. *Let $p(n)$ (for $n \in \mathbb{N}$) be the preprocessing time of a categorical path counting data structure and $q(n)$ its query time, over an ordinal tree T on n nodes, each of which is assigned a category over a finite alphabet. Then Boolean matrix multiplication on two $\sqrt{n} \times \sqrt{n}$ matrices can be solved in $\mathcal{O}(p(n) + nq(n) + n)$ time.*

Proof. Let A and B be two $\sqrt{n} \times \sqrt{n}$ Boolean matrices, and we are to compute the product $C = AB$. Let $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ be the elements in row i and column j of the matrices A, B and

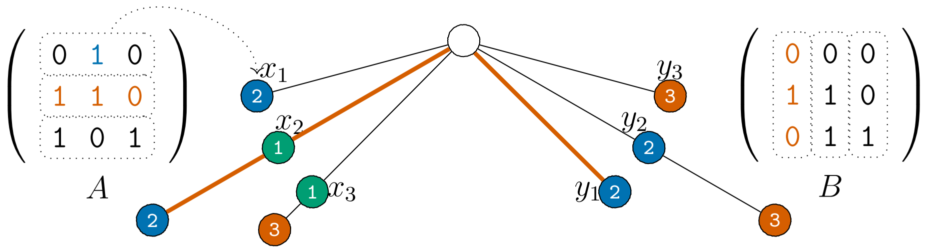

FIGURE 4.1: Two matrices A and B each of size $\sqrt{n} \times \sqrt{n}$ with $n = 9$ give rise to a tree over $\sqrt{n} + 1$ categories. The dummy root is the node marked r, and the numbers inside circles, as well as the distinct colours, denote the category of the corresponding node. The path shown in thick coloured line corresponds to a path queried when computing the cell $(2, 1)$ of the product $A \times B$. This entry corresponds to the product of the second row and the first column, respectively of the matrices A and B (which are also colored).

C, respectively. For the i^{th} row of A we construct the set $A_i = \{j \mid a_{i,j} = 1\}$, and for the j^{th} column of B we construct the set $B_j = \{i \mid b_{i,j} = 1\}$. We notice that $c_{i,j} = [\![A_i \cap B_j \neq \emptyset]\!]$.

As $|A_i \cup B_j| = |A_i| + |B_j| - |A_i \cap B_j|$, it is sufficient to focus on computing $|A_i \cup B_j|$, which in turn motivates the following construction of a tree T of size $\mathcal{O}(n)$:

1. We create a dummy root r with dummy category $\sqrt{n} + 1$;

2. The root r has $2\sqrt{n}$ children $x_1, x_2, \ldots, x_{\sqrt{n}}$ and $y_1, y_2, \ldots, y_{\sqrt{n}}$;

3. The subtree rooted at each x_i, $1 \leq i \leq \sqrt{n}$, is a single path of length $m_i = |A_i|$, consisting of nodes $x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}$, listed in preorder, i.e. with $x_i = x_{i,1}$ and x_{i,m_i} being the leaf;

4. The subtree rooted at each y_j, $1 \leq j \leq \sqrt{n}$, is a single path of length $n_j = |B_j|$, consisting of nodes $y_{j,1}, y_{j,2}, \ldots, y_{j,n_j}$, listed in preorder, i.e. with $y_j = y_{i,1}$ and y_{j,n_j} being the leaf;

5. $\forall\, 1 \leq i \leq \sqrt{n}$ and $\forall\, 1 \leq j \leq m_i$, the node $x_{i,j}$ is assigned a category – the j^{th} largest entry of A_i;

6. $\forall\, 1 \leq j \leq \sqrt{n}$ and $\forall\, 1 \leq i \leq n_j$, the node $y_{j,i}$ is assigned a category – the i^{th} largest entry of B_j.

Thus T is a tree of size $\mathcal{O}(n)$, in which each node is assigned a category from $[\sqrt{n} + 1]$. See Figure 4.1 for an example of the tree constructed for two matrices A and B. Now, it is clear that computing $|A_i \cup B_j|$ is nothing but a categorical path query with query parameters x_{i,m_i} and y_{j,n_j} (subtracting 1 from the result, to correct for the root r).

Assuming the preprocessing time of $p(n)$, and when processing n (=$\sqrt{n} \times \sqrt{n}$) queries each of time complexity $q(n)$, we arrive at the time bound claimed. $\square$

Thus, Theorem 4.1 states that the Boolean matrix multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices can be computed using n categorical path counting queries in a tree of size $\mathcal{O}(n)$. This implies that for any data structure for categorical path counting that uses $p(n)$ preprocessing time for a tree of size n, with $q(n)$ time for a query, it follows that $n^{\omega/2} = \mathcal{O}(p(n) + n \cdot q(n) + n)$, if the best algorithm for Boolean matrix multiplication uses $\Theta(n^{\omega})$ time to multiply two $n \times n$ matrices. Therefore, $p(n) = \Omega(n^{\omega/2})$ and $q(n) = \Omega(n^{\omega/2-1})$, in the worst case. Williams [109] show that in the currently best algorithm for matrix multiplication, $\omega = 2.3727$. Therefore, with current knowledge, one cannot have preprocessing time faster than $n^{1.8635}$ and query time faster than $n^{0.18635}$, simultaneously. At the same time, the best current *combinatorial* algorithm for Boolean matrix multiplication has running time only a polylogarithmically better than cubic [10, 18]. Thus, preprocessing time better than $n^{3/2}$ and query time better than $\sqrt{n}$ simultaneously by purely combinatorial techniques can not be achieved with current knowledge, save for polylogarithmic speedups.

4.2.2 Uniform Partitioning of the Tree

Next, we review a tree mark-up technique that we use in our solutions in Sections 4.2.3 and 4.3.2.

Lemma 4.1 ([36]). *Given an ordinal tree T on n nodes and an integer $t \leq n$ which is called the* blocking factor, *a subset $V' \subseteq V$ of the nodes can be chosen, called the* marked *nodes, such that: (i) $|V'| = \mathcal{O}(n/t)$; (ii) for any $x, y \in V'$ it follows that $\texttt{LCA}(x, y) \in V'$; and (iii) a path containing unmarked nodes only consists of less than t nodes and the edges between them.*

Path decomposition using the marked nodes in particular and nodes with certain labels in general is encapsulated in a `decompose`-operator introduced in Definition 4.1. Lemma 4.2 implements `decompose`, as a simple corollary to Lemma 2.3. In Definition 4.1 and Lemma 4.2, T is an ordinal tree on n nodes, each of which is assigned a label over the alphabet $[\sigma]$, where $\sigma \leq n$.

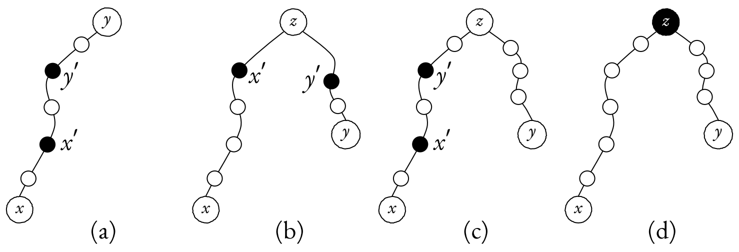

FIGURE 4.2: An illustration to the proof of Lemma 4.2. Shaded nodes are marked. Subfig. (a) illustrates the case of y being an ancestor of x. Subfigures (b)-(d) illustrate the case when x and y are not ancestors of each other. In subfig. (b), there are marked nodes on both of the paths $P_{x,z}$ and $P_{y,z}$, whereas in subfig. (c) there are marked nodes on $P_{x,z}$ only. Finally, subfig. (d) shows the case when $x' = y' = z$. In all subfigures, $z = \texttt{LCA}(x, y)$.

DEFINITION 4.1. *For any pair of nodes x, y of T, for any $\alpha \in [\sigma]$, consider the α-nodes x' and y' on $P_{x,y}$ that are closest to respectively x and y. Then, the operation $\texttt{decompose}(x, y, \alpha)$ returns the corresponding pair of nodes x' and y', or a special symbol* ***undefined*** *when no such x' and y' exist.*

LEMMA 4.2. *The tree T represented via Lemma 2.3 supports $\texttt{decompose}(x, y, \alpha)$ in $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time.*

Proof. Let $\mathcal{K} = \mathcal{K}(T)$ be the data structure of Lemma 2.3. One then has the following cases and the corresponding courses of action:

Case 1 If $\texttt{LCA}(x, y) = y$, we have $x' = \texttt{level\_anc}_\alpha(\mathcal{K}, x, 1)$. Node y' is set to be y itself if y is an α-node; if it is not, then $y' = \texttt{level\_anc}_\alpha(\mathcal{K}, x, a)$, where $a = \texttt{depth}_\alpha(\mathcal{K}, x) - \texttt{depth}_\alpha(\mathcal{K}, y)$. The result is $undefined$ if y is not an α-node and $a = 0$. The case when x is the ancestor of y is symmetrical. See Figure 4.2 (a) for an illustration.

Case 2 If x and y are not ancestors of each other, we set $z = \texttt{LCA}(T, x, y)$. Depending on the values $a = \texttt{depth}_\alpha(\mathcal{K}, x) - \texttt{depth}_\alpha(\mathcal{K}, z)$ and $b = \texttt{depth}_\alpha(\mathcal{K}, y) - \texttt{depth}_\alpha(\mathcal{K}, z)$, there are four sub-cases:

$a > 0,\ b > 0$: One has $x' = \texttt{level\_anc}_\alpha(\mathcal{K}, x, 1), y' = \texttt{level\_anc}_\alpha(\mathcal{K}, y, 1)$; see Figure 4.2 (b).

$a > 0,\ b = 0$: This case is reduced to **Case 1** by setting $y := z$; see Figure 4.2 (c).

$a = 0,\ b > 0$: This case is reduced to **Case 1** by setting $x := z$;

$a = 0,\ b = 0$: The result is $undefined$ if z is not an α-node, and $x' = y' = z$, otherwise; see Figure 4.2 (d) for an illustration.

Figure 4.2 illustrates the essential configurations. We perform an constant number of operations, each of which costs $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time; the claimed running time follows. □

From the properties of tree extraction and Lemmas 2.3 and 4.2 it follows that

Proposition 4.1. *In the tree T represented via Lemma 2.3, let $P_{x,y} \subseteq T$ be an arbitrary path and $\alpha \in [\sigma]$ an arbitrary label. Let T_α be a tree extraction from T of the node-set $X = \{z \in V(T) \mid \texttt{label}(z) = \alpha\}$. Let, furthermore, x' and y' be the nodes returned by $\texttt{decompose}(x, y, \alpha)$. Then, all the nodes in $P_{x,y} \cap X$ form a contiguous path π in T_α, with the end-points $(x_\alpha, y_\alpha) = (\texttt{pre\_rank}_\alpha(T, x'), \texttt{pre\_rank}_\alpha(T, y'))$. Furthermore, the result of $\texttt{decompose}(x, y, \alpha)$ is* ***`undefined`*** *iff $P_{x,y} \cap X = \emptyset$.*

4.2.3 Categorical Path Counting

In this section, we solve the exact categorical path counting problem. We do so by pre-computing certain information, with additional work on top of it at query time. Hence the storage space and explicit work at query time are balanced by a trade-off parameter.

Namely, the tree T is subject to the following preprocessing:

Nodes marking For the parameter $t \leq n$ to be chosen later, we mark $\mathcal{O}(n/t)$ nodes in T using Lemma 4.1. Let K be the indicator tree (Definition 2.1) for the set of marked nodes in T. We preprocess K using Lemma 4.2;

Path emptiness Let G be a copy of T labeled over $[\sigma]$ in such a way that the node $z \in V(G)$ has label α *iff* its copy in T has category α. We preprocess G using Lemma 4.2;

Tabulation We store a table M such that, for the x^{th} and y^{th} (in preorder) marked node of T, one has $M[x, y] \triangleq |\{\texttt{c}(z) \mid z \in P_{x',y'}\}|$ (i.e. $M[x, y]$ is the number of distinct categories occurring on the path $P_{x',y'} \subseteq T$). Here, $x' = \texttt{pre\_select}_1(K, x)$ and $y' = \texttt{pre\_select}_1(K, y)$ (by Proposition 2.1).

The data structures built in Section 4.2.3 result in the following

Theorem 4.2. *Let T be an ordinal tree on n nodes, each of which is assigned a category over an alphabet $[\sigma]$, where $\sigma \leq n$. Then, T can be preprocessed into a data structure of size $\mathcal{O}(n + \frac{n^2}{t^2})$, for some parameter $1 \leq t \leq n$, so that a categorical path counting query is solved in $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$ time. In particular, setting $t = \sqrt{n}$ yields a linear-space data structure with $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$ query time.*

Proof. We preprocess the input tree T as described in Section 4.2.3. The structures K, G and M contribute respectively $o(n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n^2/t^2)$ words, and hence the claimed space.

We thus turn to answering queries and then analyzing the query time. Answering the query when $|P_{x,y}| \le t$ is subsumed in our analysis. Hence let the query path $P_{x,y}$ be of length greater than t.

First, a call to $\texttt{decompose}(x, y, 1)$ on K returns two nodes x' and y' such that x', y' are marked and $\max\{|P_{x,x'}|, |P_{y,y'}|\} \le t$..

Let x_M and y_M respectively be the relative preorder ranks of x' and y' among the marked nodes of T; one computes x_M and y_M using Proposition 4.1. We use x_M and y_M to address the table M.

The answer to our query is contained in the following sets of nodes: **Group** $0 : P_{x',y'}$ (the *span*); **Group** $1 : P_{x,x'} \setminus \{x'\}$; and **Group** $2 : P_{y,y'} \setminus \{y'\}$. We note that Groups 1-2 are each of size at most t.

The strategy is to process each group sequentially, so that a category contributes to the answer as long as it appears neither in the groups one has so far traversed, nor in the portion of the path preceding the current node, in the current group.

Namely, the processing of Group 0 reduces to initializing the result counter `res` with $M[x_M, y_M]$. Next, one traverses Group 1 in the direction towards x. Let z be the current node, and p_z be the node immediately preceding z, on the current path $P_{x,x'}$ oriented from x' towards x. We check whether $\texttt{c}(z)$ occurs in $P_{p_z,y'}$ using G and Proposition 4.1; if not, we increment `res`. Finally, we traverse Group 2 in the direction towards y. Let z be the current node, and p_z be the node immediately preceding z, on the current path $P_{y,y'}$ and in the direction of traversal. We check whether $\texttt{c}(z)$ occurs in P_{x,p_z} using G and Proposition 4.1; if it does not, we increment `res`.

We call operations provided by Lemmas 2.3 and 4.2 $\mathcal{O}(t)$ times; the claimed time bound follows. $\square$

4.2.4 2-Approximate Categorical Path Counting

We provide a 2-approximation for the number of distinct categories on $P_{x,y}$ by decomposing the path $P_{x,y}$ as $P_{x,z}$ followed by $P_{y,z}$, with $z = \texttt{LCA}(T, x, y)$, and counting the number of distinct categories in $P_{x,z}$ and $P_{y,z}$ separately. It turns out that in contrast to general paths a query path in which one end is an ancestor of the other lends itself to efficient categorical counting.

We apply the *chaining* approach of Muthukrishnan [88], by assigning weights to the nodes of T as follows: If for $q \in T$ one has $\mathtt{c}(q) = \gamma$, then we identify q's lowest proper ancestor p such that $\mathtt{c}(p) = \gamma$ and set $\mathbf{w}(q) = \mathtt{depth}(T, p)$. We set $\mathbf{w}(q) = -1$, if there is no such p. We use the result of He et al. [70] to encode, in a structure C, the weighted tree and support path counting queries (see Section 3.1 for the definition) over its paths:

Lemma 4.3 ([70]). *Let T be an ordinal tree on n nodes, each having a weight drawn from $[m]$. Under the word-RAM model, T can be encoded in $\mathcal{O}(n)$ words to support path counting queries in $\mathcal{O}(\frac{\lg m}{\lg\lg n} + 1)$ time.*

For the data structures built in Section 4.2.4, we claim the following

Theorem 4.3. *An ordinal tree T on n nodes, each of which assigned a category, can be preprocessed into an $\mathcal{O}(n)$-word data structure to solves the 2-approximate categorical path counting problem in $\mathcal{O}(\frac{\lg n}{\lg\lg n})$ time. When one query node is an ancestor of the other, the answer is exact.*

Proof. We preprocess the input tree T as described in Section 4.2.4.

The dominant-size data structure C is linear-size (Lemma 4.3); hence the claimed space.

We focus on how to answer the query on the path $P_{x,z}$, where $z = \mathtt{LCA}(x, y)$, for the query nodes x and y. Given the query $P_{x,z}$, we execute a path counting query in C with parameters $P_{x,z}$ (as the query path) and $(-\infty, \mathtt{depth}(z))$ (as query weight range). By repeating the procedure verbatim for $P_{y,z}$, we return the sum of the two queries as the sought 2-approximation. We note that when y is an ancestor of x, the answer is exact.

The total running time therefore is the sum of the running times of at most two path counting queries, and the claimed time bound follows. $\square$

4.3 Categorical Path Range Counting

In this section, we solve the categorical path counting problem in the case of weighted trees, including those weighted with multidimensional weight vectors.

In solving the categorical path range counting problem, we apply the marking technique of Lemma 4.1, too. The core idea remains, but we guard against over-counting using somewhat more complex data structures. Namely, in Section 4.3.1 we extend the repertoire of useful tree operations (Table 2.1) by the *path range emptiness* query, which, in the case of unweighted trees (Section 4.2.3), was simulated using labeled ancestors and labeled depth (Lemma 2.3).

4.3.1 Path Range Emptiness Queries

First, let us formally introduce path range emptiness queries:

Definition 4.2. *For a constant $d \in \mathbb{N}$, let T be an ordinal tree on n nodes, each node z of which is assigned a weight vector $\mathtt{w}(z) \in [n]^d$. For any two nodes $x, y \in T$ and any axis-aligned hyper-rectangle Q from $[n]^d$, a* path range emptiness *query is a path query that returns* `false` *if the set $\{z \in T \mid z \in P_{x,y} \wedge \mathtt{w}(z) \in Q\}$ is empty, and* `true`, *otherwise.*

From Theorem 3.5 in Section 3.7 and the solution to the path reporting problem (see Section 3.1 for the definition) by [19], it follows that [1]

Lemma 4.4 ([19, 64]). *Let T be an ordinal tree on n nodes, each of which is assigned a weight vector from $[n]^d$, where $d \in \mathbb{N}$ is a constant. Then, T can be preprocessed into a data structure so that a path emptiness query is answered in*

$d = 1$: *either (a) $\mathcal{O}(\lg\lg n)$; or (b) in $\mathcal{O}(\lg^{\epsilon} n)$ time. The data structures occupy respectively (a) $\mathcal{O}(n \lg\lg n)$; and (b) $\mathcal{O}(n)$ words of space.*

$d \geq 2$: *$\mathcal{O}(\frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}})$ time, for an $\mathcal{O}(n \lg^{d-1+\epsilon} n)$-word data structure.*

Since Lemma 4.4 presents different trade-offs to be used in different cases in our solutions for different values of d, we shall refer to the query time as $\tau_d(n)$ and and to the space cost as $\mathtt{s}_d(n)$.

4.3.2 Categorical Path Range Counting in d Dimensions

As in Section 4.2, here we also trade explicit traversals for the storage for pre-computed information, with a few notable differences to accommodate weights. Precisely, the preprocessing of the tree T encompasses the following procedures and data structures:

Nodes marking We mark the nodes of T using Lemma 4.1, with blocking factor t;

Weights partitioning Along each of the d dimensions, we partition the space $[n]^d$ into $\lceil n/t \rceil$ slabs, using axis-aligned hyper-planes, in such a way that each slab (possibly, except for the last) contains exactly t nodes of the tree T (this is always possible, as the weights are in rank space). More precisely, we maintain a list λ_i of slabs per weight component:

[1]The original sources state the results for path reporting but these results imply the results on path emptiness as stated in Lemma 4.4.

$\lambda_{i,j} \triangleq \{z \in T \mid (j-1)t < w_i(z) \leq \min\{jt, n\}\}$, for $1 \leq i \leq d$ and $1 \leq j \leq \lceil n/t \rceil$. Slightly abusing notation, here we use "slab $\lambda_{i,j}$" to denote both the orthogonal range and the corresponding set of nodes defined above;

Path emptiness For each category $\gamma \in [\sigma]$, we build the tree extraction T_γ of all the nodes with category γ. The nodes of T_γ inherit the weights of the original nodes in T. Each T_γ, in turn, is associated with the following data structures:

- The path emptiness data structure C_γ of Lemma 4.4;
- y-fast tries [108] $\{Y_{\gamma,i}\}_{i=1}^{d}$ s.t. $Y_{\gamma,j}$ maps the j^{th} weights of T_γ into rank space $[|T_\gamma|]$;

Mapping structures Maintained using Lemma 2.3 are also trees K and G:

- Let T_X be the (conceptual) extraction of the set of marked nodes in T. Then K is the indicator tree (Definition 2.1) of (T, T_X);
- G has the topology of T and is labeled over $[\sigma]$ such that a node $z \in G$ is given a label γ *iff* its copy in T has category γ;

Tabulation For each of the $\Theta((n/t)^{2d+2})$ *spans* we pre-compute and store, in a table M, the number of distinct categories occurring in the span. Precisely, let the indices $i_1, i_2, \ldots, i_d$ and $j_1, j_2, \ldots, j_d$ be such that $\forall k\ 1 \leq i_k \leq j_k \leq \lceil n/t \rceil$ and two nodes x' and y' be marked. Then, the *span* corresponding to this tuple of indices and the pair of marked nodes is the set $\{z \in P_{x',y'} \mid z \in \cap_{k=1}^{d}(\cup_{l=i_k}^{j_k} \lambda_{k,l})\}$ (i.e. the set of nodes on the path $P_{x',y'}$ such that their weights fall into the relevant rectangle in $[n]^d$). One final detail: To save space, the nodes x' and y' should be referred to by their relative preorder ranks, say x_M and y_M, among the marked nodes. Now, M is a table whose element $M[x_M, y_M, i_1, j_1, i_2, j_2, \ldots, i_d, j_d]$ stores the number of distinct categories in the relevant span.

LEMMA 4.5. *The data structures built in Section 4.3.2 occupy $\mathcal{O}(\mathtt{s}_d(n) + (n/t)^{2d+2})$ words of space.*

Proof. Indeed, weights partitioning in total stores d copies of the nodes of T, as each node is contained in exactly d distinct $\lambda_{i,j}$; hence the contribution is linear. The sizes of the tree extractions T_γ sum up to $\mathcal{O}(n)$, too. The structures C_γ occupy $\mathcal{O}(\mathtt{s}_d(n))$ words of space, in total. Each y-fast trie occupying linear space [108] and by design storing disjoint sets, the total contribution of all the y-fast tries is linear. The mapping structures K and G in total occupy another $\mathcal{O}(n)$ words. Finally, the table M is of size $\mathcal{O}((n/t)^{2d+2})$ words. □

In Lemma 4.7, we describe how to resolve queries and analyze the query time. The main idea is to divide the set of all the candidate nodes (i.e. the nodes that can potentially lie on the query path with weight vectors in the query rectangle) into groups, one of which is "large", and several are "small". We use a pre-computed result for a "large" group. Then, one walks the other, "small", groups while checking whether a new category is already contained in the preceding groups. The latter is accomplished via restricting a previous group to a *monochromatic* region – the tree extraction whose nodes have the same category.

To facilitate presentation, we abstract the procedure of zooming into a monochromatic region in a lemma of its own:

LEMMA 4.6. *With respect to the data structures built in Section 4.3.2, let $P_{x,y} \subseteq T$ and Q be respectively an arbitrary path and an axis-aligned rectangle in $[n]^d$, where $d \in \mathbb{N}$ is a constant. Let, furthermore, $\gamma \in [\sigma]$ be an arbitrary category. Then, whether the set $\{z \in P_{x,y} \mid \mathtt{w}(z) \in Q \wedge \mathtt{c}(z) = \gamma\}$ is empty can be answered in $\mathcal{O}(\tau_d(n))$ time.*

Proof. First, we adjust the query parameters $P_{x,y}$ and Q to the tree T_γ. We map the query path to a path in T_γ in $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time (Lemma 4.2). We then use the y-fast tries $\{Y_{\gamma,k}\}_{k=1}^{d}$ to restrict the range Q to the rank space of the weights of T_γ. If $Q = \prod_{k=1}^{d}[a_k, b_k]$, then each a_k is searched for a successor in $Y_{\gamma,k}$, and b_k is searched for a predecessor in $Y_{\gamma,k}$, which results in an additive $\mathcal{O}(\lg \lg n)$ time, over all $1 \leq k \leq d$.

Next, we launch a path emptiness query using C_γ for the query time of $\tau_d(n)$ (Lemma 4.4). Both of the time bounds in the previous paragraph do not exceed $\tau_d(n)$, and hence the claim follows. □

To proceed with the formal statement and its proof:

LEMMA 4.7. *The data structures built in Section 4.3.2 solve a categorical path range counting query in $\mathcal{O}(t \cdot \tau_d(n))$ time.*

Proof. Let $P_{x,y}$ and $Q = \prod_{k=1}^{d}[a_k, b_k]$ be the query arguments.

If $|P_{x,y}| \leq t$, we explicitly traverse the path $P_{x,y}$ and count the number of unique categories encountered; the exact procedure is subsumed in the discussion that follows. We therefore assume $|P_{x,y}| > t$, and split the path $P_{x,y}$ into $P_{x,x'}, P_{x',y'}$, and $P_{y,y'}$, where x' and y' are

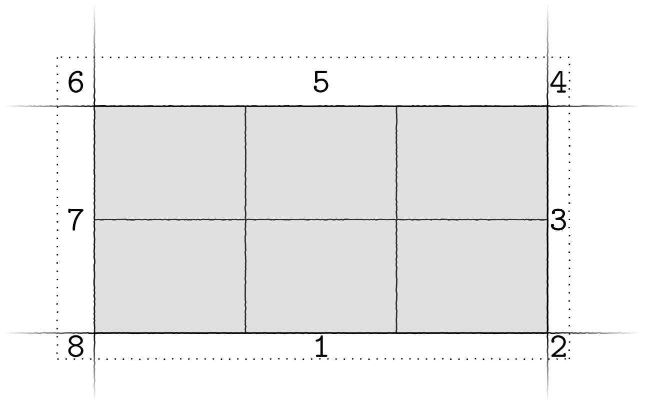

FIGURE 4.3: Disjoint decomposition of the rim in 2D (the tree dimension is omitted). The first weight corresponds to the x-axis, the second weight – to the y-axis. The shaded region marks the span. The dotted rectangle is the query rectangle, while solid lines denote the grid lines induced by the slabs λ. The rectangular subdivisions marked 1 and 5 belong to a slab $\lambda_{1,j}$ for some j, whereas the subdivisions 7 and 3 belong respectively to the slabs $\lambda_{1,j-1}$ and $\lambda_{1,j+1}$. Furthermore, the subdivisions marked 8, 6 lie respectively on slabs $\lambda_{2,k-1}$ and $\lambda_{2,k+1}$, for some k. At the same time, 6 and 8 lie also on $\lambda_{1,j-1}$. The situation is symmetrical to the subdivisions 2, 4.

marked nodes that are on the path $P_{x,y}$ and are closest to respectively x' and y'. To find x' and y', we use `decompose`$(x, y, 1)$ of Lemma 4.2. One has that $|P_{x,x'}|, |P_{y,y'}| \le t$.

The grid of the marked nodes and the slabs induce a decomposition of the query region into the span and the "rim" – the parts of the query region abutting it. Of these, the rim is meant to be explicitly traversed. The details follow.

THE SPAN. First, we initialize the indices $i_1, i_2, \ldots, i_d$ and $j_1, j_2, \ldots, j_d$ as $i_k := \lceil a_k/t \rceil$ and $j_k := \lceil b_k/t \rceil$, for all $1 \le k \le d$. That is, i_k^{th} range contains a_k, and j_k^{th} range contains b_k. Furthermore, let x' and y' be respectively the x_M^{th} and y_M^{th} marked node, in preorder; one computes x_M and y_M using Proposition 4.1. Now the tuple $(x_M, y_M, i_1+1, j_1-1, i_2+1, j_2-1, \ldots, i_d+1, j_d-1)$ determines a span, for which the answer – the number of distinct categories occurring therein – is already pre-computed. We initialize the counter variable `res` holding the answer to the query with the table entry $M[x_M, y_M, i_1+1, j_1-1, i_2+1, j_2-1, \ldots, i_d+1, j_d-1]$. With this span, we also associate the pair of query arguments $P^{(span)} = P_{x',y'}$ and $Q^{(span)} = \prod_{k=1}^{d}[i_k t + 1, \min\{(j_k-1)t, n\}]$.

THE RIM. Our goal is to traverse the rim systematically, scanning for categories, while being careful to neither double-count nor miss them. One thus takes note of a category only if it has not been seen in "the past". With this consideration in mind, we present a way of walking the rim.

Consider all the nodes $z \in P_{x',y'}$ whose weight vector $\mathbf{w}(z)$ pushes z outside of the span. The loci of such vectors in $[n]^d$ clearly can be covered with $\mathcal{O}(d) = \mathcal{O}(1)$ disjoint axis-aligned rectangles – henceforth *canonical rectangles* – in such a way that each canonical rectangle r lies entirely within some $\lambda_{i,j}$. For each dimension k, there are up to two canonical rectangles within slabs λ_{k,i_k} and λ_{k,j_k}. We assume the availability of such a cover $\mathcal{D}$. (An example of such a disjoint decomposition is shown in Figure 4.3.)

We then "separate" the prefix/suffix from $\mathcal{D}$ by creating, out of each canonical rectangle r, a *canonical set* $s(r)$ as the set $\{z \in P_{x',y'} \,|\, \mathbf{w}(z) \in r\}$. As each r lies inside a slab, one has $|s(r)| \leq t$.

We enumerate the nodes in the rim in the (say) following order: the nodes of the prefix, the nodes of the suffix, and the nodes in each canonical set. Within each set, the nodes are conceptually ordered in the direction from x to y (the traversal order is ascertained via tree operators of Lemma 2.2). As we walk through these sets, we refer to the processed sets as *previously seen*. Importantly, the previously seen set contains the span as its first item.

Let us maintain a conceptual set E of query parameters: it consists of the pairs (*path*, *weight range*) that specify the previously seen sets. For the span, this set of parameters is $(P^{(span)}, Q^{(span)})$ defined above. For the prefix and the suffix, the query path is respectively the prefix and the suffix, and the weight range is Q. For a canonical set, the path is $P_{x',y'}$ and the weight range is given by the associated slab.

Let z be a current node in our traversal of the rim, and let $\gamma = \mathtt{c}(z)$. The category γ contributes towards `res` *iff* each of the path range emptiness queries executed using the query parameters stored in E comes back as `false`. Each item of E is used to execute the query described in Lemma 4.6. We however also need to perform the final test for γ: it involves the part of the *current* set (be it the prefix, the suffix, or a canonical set) that precedes z in our conventional x-to-y ordering. Regardless of whether the current set is the prefix, the suffix, or a canonical set, we launch a path range emptiness query for the path $P_{x,z} \setminus \{z\}$ and the range Q via Lemma 4.6.

The decomposition of the rim thus consists of $\mathcal{O}(1)$ disjoint sets. Each set can be enumerated in $\mathcal{O}(t)$ time, and therefore the entire rim is traversed in $\mathcal{O}(t)$ time. For each node traversed, we consider all the previously seen sets, which is still $\mathcal{O}(t)$ time overall. The claim for query time follows from the time bounds in Lemma 4.6. $\square$

Combining Lemmas 4.5 and 4.7 one has

THEOREM 4.4. *Let $d \in \mathbb{N}$ be a constant. Let T be an ordinal tree on n nodes, each node $z \in T$ of which is assigned a category $\mathsf{c}(z) \in [\sigma]$, as well as a d-dimensional weight vector $\mathbf{w}(z)$ in rank space, where $d \in \mathbb{N}$ is a constant. Then, for the categorical path range counting problem there exists a data structure such that it uses:*

$d = 1$: *either*

- $\mathcal{O}(n \lg\lg n + (n/t)^4)$ *words of space for the query time of* $\mathcal{O}(t \lg\lg n)$*; or*
- $\mathcal{O}(n + (n/t)^4)$ *words of space for the query time of* $\mathcal{O}(t \lg^{\epsilon} n)$*;*

In particular, one can have a linear-space data structure with query time $\mathcal{O}(n^{3/4} \lg^{\epsilon} n)$*;*

$d \geq 2$: $\mathcal{O}(n \lg^{d-1+\epsilon} n + (n/t)^{2d+2})$ *words of space for the query time of* $\mathcal{O}(t \frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}})$*; For an* $\widetilde{\mathcal{O}}(n)$*-space solution, one thus has* $\widetilde{\mathcal{O}}(n^{\frac{2d+1}{2d+2}})$ *query time.*

Here, $1 \leq t \leq n$ *is a trade-off parameter set prior to construction.*

4.4 Sketching Data Structures for Approximate Categorical Path Counting

In this section we solve the $(1 \pm \epsilon)$-approximate categorical counting problem, with probability $1 - \delta$, for arbitrarily small constants $0 < \epsilon, \delta < 1$. Section 4.4.1 reviews the concept of *sketches* [28, 78] that we use in our solutions. Then, Section 4.4.2 describes our data structures.

4.4.1 Sketches

For an arbitrary vector $\mathbf{a} = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$, Cormode et al. [28] introduce the *Hamming norm* $|\mathbf{a}|_H$ of $\mathbf{a}$, defined as $|\mathbf{a}|_H \triangleq \sum_{i=1}^{n} |a_i|^0$, with $|0|^0 \triangleq 0$. It is clear that $|\mathbf{a}|_H = |\{a_i \mid a_i \neq 0\}|$, i.e. the Hamming norm equals the number of non-zero components in $\mathbf{a}$. Therefore, if $\mathbf{a} \in (\mathbb{N} \cup \{0\})^n$ is a *frequency array*, i.e. a_i records the multiplicity of the number i in a certain collection, then the number of distinct elements in the collection is given by the Hamming norm of $\mathbf{a}$.

Furthermore, with the vector $\mathbf{a}$ a certain length-m vector $h(\mathbf{a})$, called a *sketch*, can be associated, that possesses several properties turning $h(\mathbf{a})$ into a useful summary structure. While referring the reader to [28] and references therein for a discussion in more detail, we state the main previous result used in our work:

LEMMA 4.8 ([28, 78]). *Let $0 < \epsilon, \delta < 1$ be constants. Given a vector* **a**, *there exists a sketch, $h(\mathbf{a})$, that requires $m = \mathcal{O}(\frac{1}{\epsilon^2} \cdot \lg \frac{1}{\delta})$ words and allows approximation of $|\mathbf{a}|_H$ within a factor of $1 \pm \epsilon$ of the true answer with probability $1 - \delta$. Updating the sketch and computing $|\mathbf{a}|_H$ both take $\mathcal{O}(m)$ time. Furthermore, if* **a** *and* **b** *are two vectors, then $h(\mathbf{a} \pm \mathbf{b}) = h(\mathbf{a}) \pm h(\mathbf{b})$.*

A clarification is in order regarding the *update* operation mentioned in Lemma 4.8. The scenario of Cormode et al. [28] is that of observing a stream and at each instant maintaining the sketch for the frequency array associated with the stream. *Update* refers to updating the sketch upon observing the next value in the stream.

4.4.2 $(1 \pm \epsilon)$-APPROXIMATE CATEGORICAL PATH RANGE COUNTING

We now use sketches for preprocessing the tree T for $(1 \pm \epsilon)$-approximate (with probability $1 - \delta$) categorical path range counting queries. We first solve the problem for unweighted trees; Lemma 3.4 from Section 3.3 then extends the data structures to trees weighted with d-dimensional weight vectors, where $d \in \mathbb{N}$ is a constant.

Recall that in Section 4.4.1, the sketch $h(\mathbf{a})$ was used to approximate the number of non-zero entries in a vector **a**. In the solution of Cormode et al. [28], this original vector **a** was the frequency array of a stream. The gist of our solution is in treating certain paths $P_{x,\perp} \subseteq T$ each as a stream of its own and hence maintaining *several* sketch-summaries thereof. Namely, our adaptation comprises [28, 78] (i) using the same *transformation matrix* for all computations; and (ii) building the sketches using $\delta' = \frac{\delta}{n^{2d+2}}$. Ensured by (i) is the "compatibility" (i.e. additivity in the sense of Lemma 4.8) of any two arbitrarily chosen summaries — sketches are obtained by a linear transformation [28, 78] of (in our case) the frequency array, with linearity implying additivity. Guaranteed by (ii) is that even if each of the $\Theta(n^{2d+2})$ queries (i.e. $\Theta(n^2)$ distinct paths *times* $\Theta(n^{2d})$ distinct d-dimensional rectangles) fails independently with probability δ/n^{2d+2}, then the overall measure of the failure events is still δ. With (i) and (ii) in mind, the value of m in Lemma 4.8 works out to be $m = \mathcal{O}(\frac{1}{\epsilon^2} \lg \frac{1}{\delta/n^{2d+2}}) = \mathcal{O}(\lg n)$.

We next describe how to preprocess T.

First, we apply Lemma 4.9 with parameter t to mark $\mathcal{O}(n/t)$ nodes in the tree (the proof of Lemma 4.9 easily follows from the Pigeonhole Principle):

LEMMA 4.9 ([71]). *Let $1 \leq t \leq n$ be an integer parameter. There exists a level l' no deeper than t such that, when one* marks *the nodes on every t^{th} level of the tree T, starting from l', then there are $\mathcal{O}(n/t)$ marked nodes in total.*

Next, at each marked node $z \in T$, one stores the sketch $h(z)$ as a summary of the categories occurring on the path $P_{z,\perp}$. Indeed, let $\mathbf{a}(z)$ be the (conceptual) frequency vector for the categories on the path $P_{z,\perp}$. Then we associate with z a length-m vector $h(z)$ – the sketch of $\mathbf{a}(z)$. One thus obtains

LEMMA 4.10. *The data structures built in Section 4.4.2 occupy $\mathcal{O}(n + (n/t)\lg n)$ words and answer a $(1 \pm \epsilon)$-approximate categorical path counting query in $\mathcal{O}(t \lg n)$ time, with probability $1 - \delta$.*

Proof. There are $\mathcal{O}(n/t)$ marked nodes, each storing $m = \mathcal{O}(\lg n)$ words, hence the claimed space.

The path $P_{x,\perp}$ can be represented as $P_{x,x'} \cup P_{x',\perp}$, where x' is the closest marked ancestor of x'. If there is no such x', then the depth of x is no greater than t and this case is solved by an explicit traversal, as shown below. Let us therefore assume the existence of such x'. The case $x = x'$ is trivial, as we use $h(x')$. If $x \neq x'$, then by construction $|P_{x,x'}| \leq t$. We initialize a zero-vector s of length m and the *current node* to x. We then ascend the path $P_{x,x'}$ in the direction of x' until the current node equals x'. (Informally, the path $P_{x,x'}$ in the direction towards x' is our "stream", and the "next value" is the category of the next node encountered on this path.) Let the category of the current node be $j \in [\sigma]$. Then the current sketch s is updated using Lemma 4.8. This increment thus being an $\mathcal{O}(m) = \mathcal{O}(\lg n)$-time operation, the traversal's time cost is $\mathcal{O}(tm) = \mathcal{O}(t \lg n)$. At the node x', we return the sum of s and the sketch $h(x')$ (which is pre-computed), as the sketch for the entire path $P_{x,\perp}$.

Now, by virtue of the additivity property of sketches (Lemma 4.8), the answer to a query with arbitrary query nodes x and y is simply

$$h(x) + h(y) - 2 \cdot h(\texttt{LCA}(x, y)),$$

corrected for $\texttt{c}(\texttt{LCA}(x, y))$ using Lemma 4.8. □

Furthermore, Lemma 4.8 implies that the sketches that are thus associated with a fixed tree form a semigroup. Therefore, the combination of Lemma 4.10 and Lemma 3.4 from Section 3.3.1 yields the following

THEOREM 4.5. *Let $0 < \epsilon, \delta < 1$ be arbitrarily small constants, and $d \in \mathbb{N}$ be a constant. Let, furthermore, T be an ordinal tree on n nodes, each node z of which is assigned a weight $\mathtt{w}(z) \in [n]^d$, as well as a category $\mathtt{c}(z) \in [\sigma]$. Then, there exists a data structure of $\mathcal{O}((n + \frac{n}{t}\lg n)\lg^d n)$ words* *that solve a* $(1 \pm \epsilon)$*-approximate categorical path range counting query in* $\mathcal{O}(t \lg^{d+1} n)$ *time, with success probability no less than* $1 - \delta$.

Proof. The set $\mathbb{R}^m$ forms a semigroup with respect to the regular component-wise addition operation in vectors. Thus, when the marked nodes in Lemma 4.10 are assigned the sketches, they are assigned semigroup elements in the sense of Definition 3.1 from Section 3.3. Unmarked nodes are assigned conceptual zero-vectors in view of formal compliance with Definition 3.1; since the sketches associated with unmarked nodes are never used for computations, this has no effect on our algorithm.

Now, Lemma 4.10 provides the base data structure – of type G^0, in Lemma 3.4's terminology – for the application of Lemma 3.4. We iteratively apply Lemma 3.4 to Lemma 4.10. Since we start with G^0 – supplied by Lemma 4.10 – the Lemma 3.4 is applied exactly d times; hence the space cost of $\mathcal{O}((n + \frac{n}{t} \lg n) \lg^d n)$ words and the query time of $\mathcal{O}(t \lg^{d+1} n)$.

Furthermore, our data structure fails *iff* at least one of the $\Theta(n^{2d+2})$ queries fails. The total probability of failure therefore is at most the sum of the failure probabilities of each of these $\Theta(n^{2d+2})$ queries. When building the data structure, therefore, we use a stronger guarantee of $\delta' = \frac{\delta}{n^{2d+2}}$, which also means $m = \mathcal{O}(\lg \frac{1}{\delta'}) = \mathcal{O}(\lg n)$. □

4.5 Conclusion

This chapter describes data structures that solve the categorical path counting problem, in both weighted and unweighted trees. For unweighted trees, it shows a conditional lower bound via a reduction from the Boolean matrix multiplication problem, and then presents a data structure matching this lower bound up to polylogarithmic factors. The categorical path range counting data structures devised for trees weighted with d-dimensional weight vectors also achieve the best previous bounds on categorical range counting, save for polylogarithmic factors, when $d \geq 2$.

Namely, for a given $1 \leq t \leq n$, we solve the categorical path counting in unweighted trees in $\mathcal{O}(t \lg \lg_w \sigma)$ time and $\mathcal{O}(n + (n/t)^2)$ space, which implies linear space for $\mathcal{O}(\sqrt{n} \lg \lg_w \sigma)$ time, where $w = \Omega(\lg n)$ is the word size in RAM. For weighted trees, the categorical path range counting is solved in $\mathcal{O}(t \lg \lg n)$ time and $\mathcal{O}(n \lg \lg n + (n/t)^4)$ space, or $\mathcal{O}(t \lg^\epsilon n)$ time and $\mathcal{O}(n + (n/t)^4)$ space, and then extend to the trees weighted with vectors from $[n]^d$, $d \geq 2$, We also present solutions to the categorical path counting problem in certain approximate settings. One of our solutions is probabilistic and uses *sketches*, while the other one achieves a 2-approximation in sub-logarithmic time, for linear space, but applicable for unweighted trees only. Namely, the data structure based on sketches uses $\mathcal{O}((n+\frac{n}{t}\lg n)\lg^{d} n)$ words to solve $(1\pm\epsilon)$-approximate categorical path range counting query in trees weighted with d-dimensional weight vectors in $\mathcal{O}(t\lg^{d+1} n)$ time, with failure probability at most δ, for arbitrarily small positive constants ϵ, δ and a constant $d \in \mathbb{N}$.

Our results in Section 4.2 and Section 4.3 use the same underlying idea of balancing precomputation versus explicit traversals. Durocher et al. [36] slash the running time by a further $\sqrt{w}$ when using the exact same approach for their linear-space path mode data structure for unweighted trees. We however could not replicate this step in our solution (presented in Section 4.2) to categorical path counting. It is therefore an open problem whether better time than $\mathcal{O}(\sqrt{n}\lg\frac{\lg\sigma}{\lg w})$ can be achieved for linear-space. For example, when $\sigma=\mathcal{O}(n)$, the query time becomes $\mathcal{O}(\sqrt{n}\lg\lg n)$. Can one achieve $\mathcal{O}(n)$ space and $\mathcal{O}(\sqrt{n})$ time, even if a speedup by $\sqrt{w}$ proves too ambitious?

Secondly, a lower bound for the categorical path *range* counting is an interesting open problem, too. In the proof of their hardness results, Kaplan et al. [76] use an ingenious geometric design based on perturbing the points of the original point-set and projecting the points to a subset of the coordinate axes, when applying the inclusion-exclusion principle (akin to what we did, albeit in a rudimentary form, in the proof of Theorem 4.1). One however has no obvious way of "perturbing" the nodes of a tree, nor is there a way of ensuring "tree-aware" projections.

Chapter 5

Path Query Data Structures in Practi-ce

5.1 Introduction

Chapter 3 studied the generalizations of the classical orthogonal range search problems to trees weighted with multidimensional weight vectors. Therein, we developed a framework for extending any data structure for a semigroup path sum query problem to higher dimensions, and applied it to ancestor dominance reporting, path successor, path counting and path reporting. Chapter 4 further considered categorical versions of path counting, both in exact and approximate settings.

At the same time, the sheer volume of raw data, the ever-growing need for low-latency query responses, resource-limited environments such as in embedded and edge computing, all drive the practical applications of the advanced data structures to the forefront. Apart from emerging fields, queries on tree topologies are gaining ground even in established domains such as RDBMS [103]. At the same time, the expected height of a tree T on n nodes (i.e. the height over a fixed root of a tree selected uniformly at random from the set of the n^{n-2} labeled trees) is $\sqrt{2\pi n}$ [102], which calls for the development of methods beyond naïve.

All of this motivated us to design and implement the data structures for path queries and evaluate their performance in practice. We perform experimental studies on data structures that answer path median, path counting, and path reporting queries in 1D-weighted trees. These query problems generalize (respectively) the well-known range median query problem in arrays, as well as the 2D orthogonal range counting and reporting problems (defined in Section 3.1) in planar point sets, to tree structured data.

Our practical realizations of the best theoretical results on the three queries include tree extraction, heavy-path decomposition, and wavelet trees among its major components. Our data structures are implemented in both pointer-based and succinct form, and hence we rely also on primitives such as bitvectors, balanced parentheses sequences, and `rank`/`select`. Our succinct data structures are even further specialized to be plain or entropy-compressed.

Through experiments on large sets, we show that succinct data structures for path queries present a viable alternative to standard pointer-based realizations, in practical scenarios; occupying space close to a compact storage of the input, yet retaining competitive time guarantees, they represent an inflexion point between slow naïve solutions and rather fast but space-prohibitive verbatim implementations of the advanced approaches.

To the best of our knowledge, the material of this chapter is the first foray of algorithm engineering into path queries over weighted trees.

For a formal introduction, let T be an ordinal tree on n nodes, with each node x associated with a *weight* $\mathbf{w}(x)$ over an alphabet $[\sigma]$. Query arguments consist of a pair of vertices $x, y \in T$ along with a 1D interval Q. The goal is to preprocess the tree T for the following types of queries:

Path Counting: return $|\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}|$;

Path Reporting: enumerate $\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}$;

Path Selection: return the k^{th} $(0 \leq k < |P_{x,y}|)$ weight in the sorted list of weights on $P_{x,y}$; k is given at query time (and Q is not applicable). In the special case of $k = \lfloor |P_{x,y}|/2 \rfloor$, a path selection is a *path median query*.

5.1.1 Previous Work

Previous work on the query problems considered in this chapter includes that of Krizanc et al. [77], who were the first to introduce the path median problem (referred to in this chapter as `PM`) in trees, and gave an $\mathcal{O}(\lg n)$ query-time data structure with the space cost of $\mathcal{O}(n \lg^2 n)$ words. They also gave an $\mathcal{O}(n \log_b n)$ words data structure to answer `PM` queries in time $\mathcal{O}(b \lg^3 n / \lg b)$, for any fixed $1 \leq b \leq n$. Chazelle [24] gave an *emulation dag*-based linear-space data structure for solving path counting (referred to as PC in this chapter) queries in weighted trees in time $\mathcal{O}(\lg n)$.

While [77, 24] design different data structures for PM and PC, He et al. [68, 70] use tree extraction to solve both PC and the path selection problem (henceforth PS), as well as the path reporting problem (PR, in this chapter), which they were the first to introduce. The running times for PS/PC were $\mathcal{O}(\lg \sigma)$, while a PR query is answered in $\mathcal{O}((1+\kappa)\lg \sigma)$ time, with κ henceforth in this chapter denoting output size. Also given is an $\mathcal{O}(n \lg\lg \sigma)$-word and $\mathcal{O}(\lg \sigma + \kappa \lg\lg \sigma)$ query time solution, for PR, in the RAM model.

Later, solutions based on *succinct* data structures emerged. (The convention throughout this chapter is that a data structure is *succinct* if its size in bits is close to the information-theoretic lower bound, reviewed in Section 2.1.3.) Patil et al. [98] presented an $\mathcal{O}(\lg n \cdot \lg \sigma)$ query time data structure for PS/PC, occupying $6n + n\lg \sigma + o(n \lg \sigma)$ bits of space. Therein, the tree structure and the weights distribution are decoupled and delegated to respectively heavy-path decomposition [104] and wavelet trees [91]. Their data structure also solves PR in $\mathcal{O}(\lg n \lg \sigma + (1+\kappa)\lg \sigma)$ query time.

Parallel to [98], He et al. [68, 70] devised a succinct data structure occupying $nH(W_T) + o(n\lg \sigma)$ bits of space to answer PS/PC in $\mathcal{O}(\frac{\lg \sigma}{\lg\lg n} + 1)$, and PR in $\mathcal{O}((1+\kappa)(\frac{\lg \sigma}{\lg\lg n} + 1))$ time. Here, W_T is the multiset of weights of the tree T, and $H(W_T)$ is the entropy thereof. Combining tree extraction and the ball-inheritance problem [21] (see also Lemma 3.2), Chan et al. [19] proposed further trade-offs, one of them being an $\mathcal{O}(n \lg^{\epsilon} n)$-word structure with $\mathcal{O}(\lg\lg n + \kappa)$ query time, for PR.

Despite the vast body of theoretical work, little is known on the practical performance of the data structures for path queries, with empirical studies on *weighted* trees definitely lacking, and existing related experiments being limited to navigation in unlabeled trees only [8], or to very specific domains [1, 92]. By contrast, traditional orthogonal range queries are attracting fair amount of attention from the experimental angle [9, 16, 72]. This chapter of the thesis therefore contributes to remedying this imbalance.

5.1.2 Our Results

In this chapter, we provide an experimental study of data structures for path queries. The types of queries we consider are PM, PC, and PR. The theoretical foundation of our work are the data structures and algorithms developed in [68, 98, 69, 70]. The succinct data structure by He et al. [70] is optimal both in space and time in the word-RAM model. However, it builds on components that are likely to be cumbersome in practice. We therefore present a practical compact implementation of this data structure that uses $3n \lg \sigma + o(n \lg \sigma)$ bits of space as opposed to the original $nH(W_T) + o(n \lg \sigma)$ bits of space in [70]. In the interests of brevity, we henceforth refer to the data structures based on tree extraction using the prefix `ext`. Our implementation of `ext` achieves the query time of $\mathcal{O}(\lg \sigma)$ for `PM` and `PC` queries, and $\mathcal{O}((1 + \kappa) \lg \sigma)$ time for `PR`. Further, we present an exact implementation of the data structure (henceforth referred to using the prefix `hpw`) by Patil et al. [98]. The theoretical guarantees of `hpw` are $6n + n \lg \sigma + o(n \lg \sigma)$ bits of space, with $\mathcal{O}(\lg n \lg \sigma)$ and $\mathcal{O}(\lg n \lg \sigma + (1 + \kappa) \lg \sigma)$ query times for respectively `PM`/`PC` and `PR`. Although `hpw` is optimal neither in space nor in time, on average it proves competitive with `ext` on the practical datasets we use. Further, we evaluate time- and space-impact of succinctness by realizing plain pointer-based versions of both `ext` and `hpw`. We show that succinct data structures based on `ext` and `hpw` offer an attractive alternative for their fast but space-consuming counterparts, with query-time slow-down of 30-40 times yet commensurate savings in space. We also implement a naïve approach of not preprocessing the tree at all but rather answering the query by explicit scanning, in pointer-based and succinct variations. The succinct solutions `ext` and `hpw` compare favourably to the naïve ones, the slowest former being 7-8 times faster than naïve in `PM`, while occupying up to 20 times less space. We also compare the performance of different succinct solutions relative to each other.

5.2 Preliminaries

This section introduces main algorithmic techniques at the core of our data structures.

5.2.1 Balanced Parentheses Representations of Ordinal Trees

Compact representation of ordinal trees is a well-researched area, mainstream methodologies including *balanced parentheses* (BP) [73, 86, 50, 81, 85], *depth-first unary degree sequence* (DFUDS) [14, 51, 75], *level-order unary degree sequence* (LOUDS) [73, 31], and *tree covering* (TC) [51, 67, 39]. Of these, BP-based representations "combine good time- and space-performance with rich functionality" in practice [8], and we use BP for succinct representation of trees, in our solutions. BP is a way of linearising the tree by emitting '`(`" upon first entering a node and "`)`" upon exiting, having explored all its descendants during the preorder traversal of the tree.

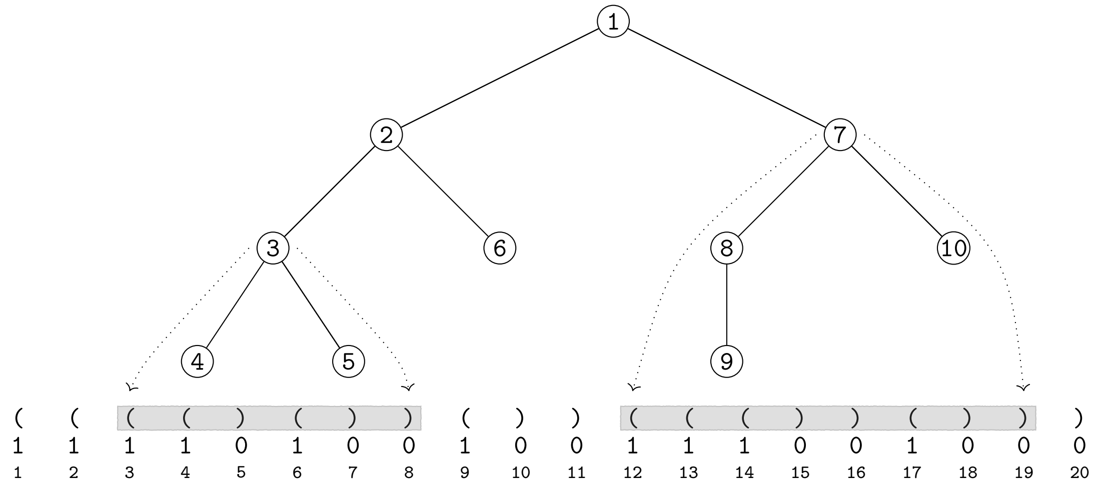

FIGURE 5.1: An ordinal tree, preorder labeled. Sequences: (top) Balanced parentheses sequence BP; (middle) bitvector, representing BP; (bottom) positions. In BP, "`(`" is `1`. Arrows point to the opening and closing parentheses, corresponding to the node. Shadings mark subtrees.

To develop intuition, let us refer to Figure 5.1. Technically, a BP sequence is nothing but an ordinary bitvector (Section 2.2.1), with an opening parenthesis encoded as *one*, and the closing parenthesis encoded as *zero*. Since we identify nodes with their preorder numbers, we move back and forth between a node and its opening parenthesis using `pos2node`$(\cdot)$ and `node2pos`$(\cdot)$ convenience wrappers around the corresponding `rank` and `select` calls, as in `node2pos`(x) = `select`$_1(BP, x)$, `pos2node`(i) = `rank`$_1(BP, i) + 1$, given that $BP[i]$ is an opening parenthesis. For further examples of BP primitives, one observes that a sub-tree rooted at a particular node x is encoded in a portion between the parentheses-pair (say, at positions i and j) corresponding to x. It follows that a node x is an ancestor of a node y if and only if the interval of the former encloses the interval of the latter; the corresponding primitive is `is_ancestor()`. For these i and j, a call to `is_opening`(i) returns `true`, whereas `is_opening`(j) is `false`. Furthermore, for a closing parenthesis at j a primitive `find_open`$(\cdot)$ is defined to return i; and `enclose`(i) returns the opening position of the tightest pair enclosing i and j. Now, finding the LCA of two nodes would be equivalent to enclosing both of the opening parentheses corresponding to these nodes by the tightest pair of matching parentheses.

To sum up, BP is one of the concrete tree representations that support the operations in Table 2.1(a) in optimal time and space.

5.2.2 Heavy-Path Decomposition

Heavy-path decomposition (HPD) [104] is a well-known approach of imposing a structure on trees. In a broader context, HPD could perhaps be conceptualized as another instantiation of the folklore "smaller half trick",[1] when judicious partitioning of the problem results in better time complexity. In HPD, for each non-leaf node, we define its *heavy child* as the child whose sub-tree has the maximum cardinality (ties broken arbitrarily). For example, if a node x has children y and z with sub-tree sizes 2 and 3 respectively, then z is x's heavy child. HPD decomposes the tree into a set of disjoint *chains*, in each of which a node is followed by its heavy child. The crucial property is that any root-to-leaf path in the tree encounters $\mathcal{O}(\lg n)$ distinct chains. Indeed, when we descend to the heavy child, we are still in the same chain; when we descend to a non-heavy child, hanging off the chain, the size of the sub-tree we found ourselves in has shrunk by at least a half. To illustrate the decomposition, in Figure 5.2, there are five chains overall: $1 \to 2 \to 3 \to 5$, three singleton chains 4, 6, and 10, and one more chain $7 \to 8 \to 9$. The heads of these chains are, respectively, $1, 4, 6, 10$, and 7. To characterize a chain in plainer words: its tail is a leaf, and its head is either the root of the tree, or a node that is not the heavy child of its parent. For example, in Figure 5.2, the head 7 of the chain $7 \to 8 \to 9$ is not the heavy child of its parent, node 1.

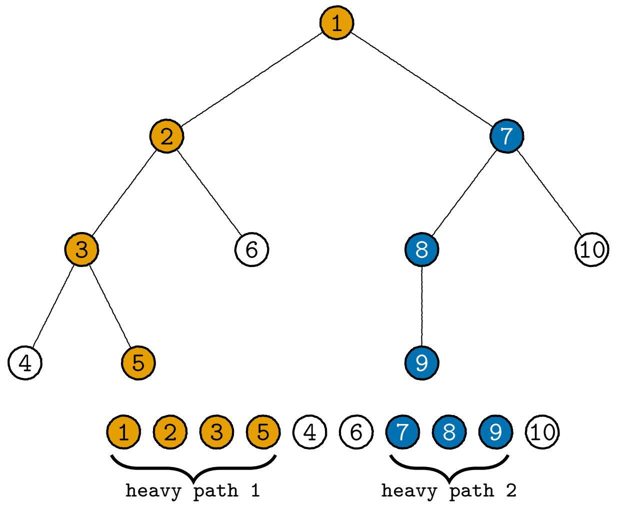

Figure 5.2: Heavy paths in a tree. Heavy paths (shown in blue and orange) lie contiguously in a segment, if one descends to a heavy child first during the depth-first search.

Assume each chain is preprocessed to answer some type of range query. Since a chain is a

[1] See also [30] and [61], where the smaller half trick is used for finding repetitions in a string.

linear structure, we avail ourselves of the plethora of data structures for simple 1D arrays. Then, since each path in a tree can be decomposed into $\mathcal{O}(\lg n)$ disjoint sub-paths – sub-chains of some chains – the entire query is spread across $\mathcal{O}(\lg n)$ disjoint sub-queries over the already-preprocessed chains. In practical terms, HPD enables traversing the path between two arbitrary nodes x and y by skipping over the portions that lie in one chain.

There are many applications of HPD. Interesting examples include random access to grammar-compressed strings [15] and compressed string dictionaries [40]. In these use-cases, by contracting each chain into a node, a tree is condensed to guarantee a height of $\mathcal{O}(\lg n)$. In the context of our work, the role of HPD is that of enabling efficient path queries. To sum up the section with a specific example, one could use HPD for path minimum queries (Section 3.2.3), as follows: (1) each chain is *individually* preprocessed to answer RMQ in constant time [41]; (2) the query path is decomposed into $\mathcal{O}(\lg n)$ disjoint segments each belonging to a distinct chain; (3) the answer is combined to be the minimum of the sub-query call-back values. In our running example in Figure 5.2, a minimum weight on the path from node $u = 5$ to node $v = 10$ would be found querying three chains: $1 \to \ \dots \ \to 5$ in its entirety, a sub-chain 7 of $7 \to \ \dots \ \to 9$, and a singleton chain 10.

5.3 Data Structures for Path Queries

This section gives the design details of the `hpw` and `ext` data structures.

5.3.1 Data Structures Based on Heavy-Path Decomposition

We describe the approach of [98], which is based on heavy-path decomposition (HPD) [104] described in Section 5.2.

Patil et al. [98] used HPD to decompose a path query into $\mathcal{O}(\lg n)$ queries in sequences. To save space, they designed the following data structure to represent the tree and its HPD. If x is the head of a chain ϕ, all the nodes in ϕ have a (conceptual) *reference* pointing to x, while x points to itself. A *reference count* of a node x (denoted as rc_x) stands for the number of times a node serves as a reference. Obviously, only heads feature non-zero reference counts – precisely the lengths of their respective chain. The reference counts of all the nodes are stored in unary in preorder in a bitvector $B = 10^{rc_1}10^{rc_2}\dots10^{rc_n}$ using $2n + o(n)$ bits. Then, one has that $rc_x = \texttt{rank}_0(B, \texttt{select}_1(B, x+1)) - \texttt{rank}_0(B, \texttt{select}_1(B, x))$. The topology

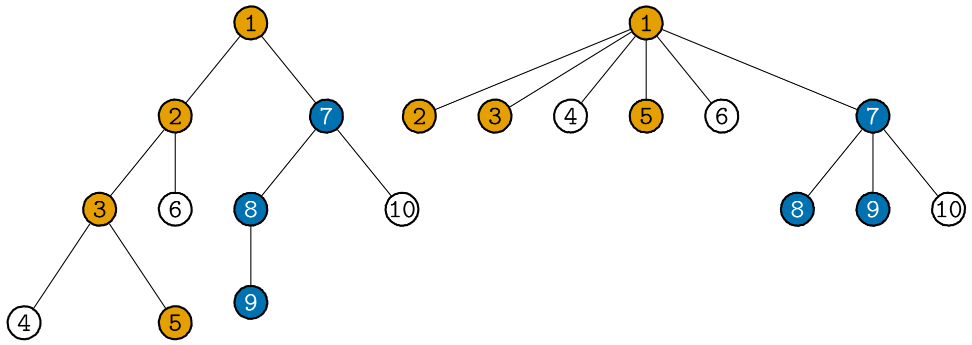

FIGURE 5.3: The original tree T (left) and its transformation T' (right). The tree T' encodes the structure of the heavy-path decomposition of T.

of the original tree T is represented succinctly in another $2n + o(n)$ bits. In addition, they encode the HPD structure of T using a new tree T' that is obtained from T via the following transformation. All the non-head nodes become leaves and are directly connected to their respective heads; the heads themselves (except for the root) become children of the references of their original parents. All these connections are established respecting the preorder ranks of the nodes in the original tree T. Namely, a node farther from the head attaches to it only after the higher-residing nodes of the chain have done so. This transformation preserves the original preorder ranks. Figure 5.3 presents an example of such transformation. On T', operation $\texttt{ref}(\mathrm{x})$ is supported, which returns the head of chain to which the node x in the original tree belongs.

To encode weights they call C_x the weight-list of x if it collects, in preorder, all the weights of the nodes for which x is a reference. Thus, a non-head node's list is empty; a head's list spells the weights in the relevant chain. Define $C = C_1 C_2 \dots C_n$. Then, in C, the weight of $x \in T$ resides at position

$$1 + \texttt{select}_1(B, \texttt{ref}(x)) - \texttt{ref}(x) + \texttt{depth}(x) - \texttt{depth}(\texttt{ref}(x)) \qquad (5.1)$$

(where $\texttt{depth}(x)$ and $\texttt{ref}(x)$ are provided by T and T', respectively). C is then encoded in a wavelet tree (WT). To answer a query, T, T', B, and Equation (5.1) are used to partition the query path into $\mathcal{O}(\lg n)$ sub-chains that it overlaps in HPD; and for each sub-chain, one computes the interval in C storing the weights of the nodes in the chain. I_m denotes the set of intervals computed. Precisely, for a node x, one uses B to find out whether x is the head of its chain; if not, the parent of x in T' returns one (say, y). Then Equation (5.1) maps the path $A_{x,y}$ to its corresponding interval in C. One proceeds to the next chain by fetching the (original) parent of y, using T. Then, the WT is queried with $\mathcal{O}(\lg n)$ simultaneous (i) range quantile[2](for `PM`); or (ii) 2D orthogonal range (for `PC` and `PR`) queries.

Range quantile query over a collection of ranges is accomplished via a straightforward extension of the algorithm of Gagie et al. [47]. One descends the levels of the wavelet tree W_C maintaining a set of current weights $[a, b]$ (initially $[\sigma]$), the current node v (initially the root of W_C), and I_m. When querying the current node v of W_C with an interval $[l_j, r_j] \in I_m$, one finds out, in $\mathcal{O}(1)$ time, how many weights in the interval are lighter than the mid-point c of $[a, b]$, and how many of them are heavier. The sum of these values then determines which subtree of W_C to descend to. There being $\mathcal{O}(\lg \sigma)$ levels in W_C, and spending $\mathcal{O}(1)$ time for each segment in I_m, the overall running time is $\mathcal{O}(m \lg \sigma) = \mathcal{O}(\lg n \lg \sigma)$. `PC`/`PR` proceed by querying each interval, independently of the others, with the standard 2D search over W_C.

5.3.2 Data Structures Based on Tree Extraction

The solution by He et al. [70] is based on performing a hierarchy of tree extractions, as described in Section 3.2.2. Namely, one starts with the original tree T weighted over $[\sigma]$, and extracts two trees $T_0 = T_{1,m}$ and $T_1 = T_{m+1,\sigma}$, respectively associated with the intervals $I_0 = [1, m]$ and $I_1 = [m+1, \sigma]$, where $m = \lfloor \frac{1+\sigma}{2} \rfloor$. Then both T_0 and T_1 are subject to the same procedure, stopping only when the current tree is weight-homogeneous. We refer to the tree we have started with as the *outermost* tree.

The key insight of tree extraction is that the number of nodes n' with weights from I_0 on the path from u to v equals $n' = \texttt{depth}_0(u_0) + \texttt{depth}_0(v_0) - 2 \cdot \texttt{depth}_0(z_0) + [\![w(z) \in I_0]\!]$, where $\texttt{depth}_0(\cdot)$ is the depth function in T_0, $z = \texttt{LCA}(u, v)$, u_0, v_0, z_0 are the T_0-views of u, v, and z. The key step is then, for a given node x, how to efficiently find its $0/1$-*parent*, whose purpose is analogous to a `rank`-query when descending down a wavelet tree. Consider a node $x \in T$ and its T_0-view x_0. The corresponding node $x' \in T$ of $x_0 \in T_0$ is then called 0-*parent* of x. The 1-parent is defined analogously. This is indeed a special case of the `level_anc` operator in Table 2.1, since $\texttt{level\_anc}_0(T, x', 1)$ is nothing but the 0-parent of x'. Since level ancestors are not used in this chapter (i.e. except for the special case mentioned), for clarity we use the terms $0/1$-parent, instead.

Supporting $0/1$-parents in compact space is one of the main implementation challenges of

[2]In the range quantile problem, one preprocesses a numerical array $a_1 a_2 \ldots a_n$ for queries that return the k^{th} smallest element among $a_i a_{i+1} \ldots a_j$, with k and i, j specified at query time.

the technique, as storing the views explicitly is expensive. In [70], the hierarchy of extractions is done as described in Section 3.2.2, by dividing the range not to 2 but $f = \mathcal{O}(\lg^{\epsilon} n)$ parts, with $0 < \epsilon < 1$ being a constant. They classify the nodes according to weights using these $f = \lceil \lg^{\epsilon} n \rceil$ labels and use tree covering (reviewed in Section 2.3.2) to represent the tree with small labels in order to find T_{α}-views for arbitrary $\alpha \in [\sigma]$, in $\mathcal{O}(1)$ time. They also use this representation to identify, in constant time, which extractions to explore. Therefore, at each of the $\mathcal{O}(\lg \sigma / \lg \lg n)$ levels of the hierarchy of extractions, constant time work is done, yielding an $\mathcal{O}(\lg \sigma / \lg \lg n)$-time algorithm for PC. Space-wise, it is shown that each of the $\mathcal{O}(\lg \sigma / \lg \lg n)$ levels can be stored in $2n + nH_0(W) + o(n \lg \sigma)$ bits of space in total (where W is the multiset of weights on the level) which, summed over all the levels, yields $nH_0(W_T) + \mathcal{O}(n \lg \sigma / \lg \lg n)$ bits of space. The components of this optimal result, however, use word-parallel techniques that are unlikely to be practical. In addition, one of the components, tree covering for trees labeled over $[\sigma]$, $\sigma = \mathcal{O}(\lg^{\epsilon} n)$ has not been implemented and experimentally evaluated even for unlabeled versions thereof. Finally, lookup tables for the word-RAM data structures may either be rendered too heavy by e.g. word alignment, or too slow by the concomitant arithmetic for accessing its entries. In practice, small blocks of data are usually explicitly scanned [8]. However, we can see no fast way to scan small labeled trees. At the same time, a generic multi-parentheses approach [91] would spare the effort altogether, immediately yielding a $4n \lg \sigma + 2n + o(n \lg \sigma)$-bit encoding of the tree, with $\mathcal{O}(1)$-time support for 0/1-parents. We achieve instead $3n \lg \sigma + o(n \lg \sigma)$ bits of space, as we proceed to describe next.

We store $2n + o(n)$ bits as a regular BP-structure S (Section 5.2) of the original tree, in which a 1-bit represents an opening parenthesis, and a 0-bit represents a closing one, and mark in a separate length-n bitvector B the *types* (i.e. whether it is a 0- or 1-node) of the n opening parentheses in S. The type of an opening parenthesis at position i in S is thus given by $\texttt{access}(B, \texttt{rank}_1(S, i))$. Given S and B, we find the $t \in \{0, 1\}$-parent of v with an approach described in [69]. For completeness, we outline in Algorithm 1 how to locate the T_t-view of a node v.

First, find the number of t-nodes preceding v (line 4). If none exists (line 5), we are done; otherwise, let u be the t-node immediately preceding v in preorder (line 7). If u is an ancestor of v, it is the answer (line 9); otherwise, set $z = \texttt{LCA}(u, v)$. If z is a t-node, or non-existent (because the tree is actually a forest), then return z or $\texttt{null}$, respectively. Otherwise (z exists and not a t-node), in line 14 we find the first t-descendant r of z (it exists because of u). This

Algorithm 1 Locate the view of $v \in T$ in T_t, where T_t is the extraction from T of the t-nodes

```
Require: $t \in \{0, 1\}$
1: function VIEW_OF($v, t$)
2:     if B[$v$] $==$ $t$ then                              ▷ $v$ is a $t$-node itself
3:         return rank$_t$($B, v$)
4:     $\lambda \leftarrow$ rank$_t$($B, v$)                           ▷ how many $t$-nodes precede $v$?
5:     if $\lambda == 0$ then
6:         return null
7:     $u \leftarrow$ select$_t$($B, \lambda$)                          ▷ find the $\lambda^{th}$ $t$-node
8:     if LCA($u, v$) $== u$ then
9:         return rank$_t$($B, u$)
10:    z $\leftarrow$ LCA($u, v$)               ▷ $z$ is $LCA$ of a $t$-node $u$ and a non-$t$-node $v$
11:    if $z ==$ null or B[$z$] $== t$ then        ▷ $z$ is a $t$-node $\Rightarrow \nexists$ $t$-parent closer to $v$
12:        return rank$_t$($B, z$)                                  ▷ or null
13:    $\lambda \leftarrow$ rank$_t$($B, z$)                           ▷ how many $t$-nodes precede $z$?
14:    $r \leftarrow$ select$_t$($B, \lambda + 1$)                     ▷ the first $t$-descendant of $z$
15:    $z_t \leftarrow$ rank$_t$($B, r$)                               ▷ $z_t$ is the $T_t$-view of $r$
16:    $p \leftarrow$ T$_t$.parent($z_t$)                          ▷ $p$ can be null if $z_t$ is 0
17:    return p
```

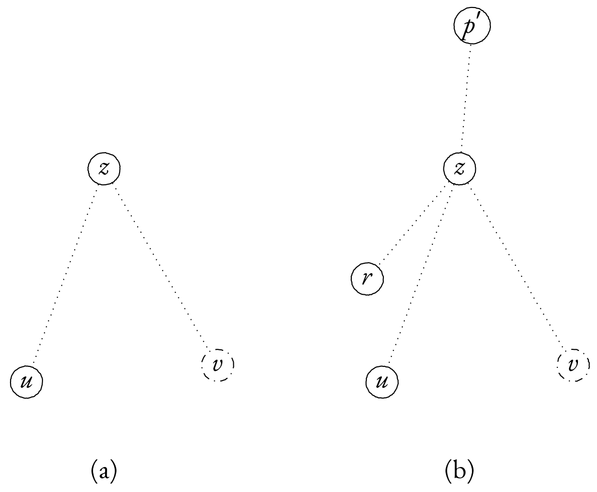

FIGURE 5.4: An illustration to the procedure of finding the t-parent of v and the T_t-view of v, given in Algorithm 1. Sub-fig. (a) illustrates the case when z is a t-node, whereas in (b), one first maps r to a node in T_t, finds the parent of the result, and then maps it back to the original node $p' \in T$.

descendant cannot be a parent of v, since otherwise we would have found it before. It must share though the same t-parent with v. We map this descendant to a node z_t in T_t (line 15). Finally, we find the parent p of z_t in T_t (line 16). Figure 5.4 summarizes the discussions of this paragraph.

The combined cost of S and B is $2n + n + o(n) = 3n + o(n)$ bits. At each of the $\lg \sigma$ levels of extraction, we encode 0/1-labeled trees in the same way, so the total space is $3n \lg \sigma + o(n \lg \sigma)$ bits.

Query algorithms in the `ext` data structure proceed within the generic framework of extracting T_0 and T_1. Let $n' = |P_{u_0,v_0}|$. We next describe the algorithms for `PM` and `PC`/`PR`.

Path median. In `PM`, we recurse on T_0 if $k < n'$, for a query that asks for a node with the k^{th} smallest weight on the path P_{u_0,v_0}; otherwise, we recurse on T_1 with $k \leftarrow k - n'$ and u_1, v_1. We stop upon encountering a tree with homogeneous weights. This logic is embodied in Algorithm 2. Theoretical running time is $\mathcal{O}(\lg \sigma)$, as all the primitives used are $\mathcal{O}(1)$-time. We enter Algorithm 2 with several parameters – the current tree T, the query nodes u, v, the LCA z of the two nodes, the quantile k we are looking for, the weight-range $[a, b]$, and a number w. These are initially set, respectively, to be the outermost tree, the original query nodes, the LCA of the original query nodes, the median's index (i.e. half the length of the corresponding path in the original tree), the weight range $[\sigma]$, and the weight of the LCA of the original nodes. We maintain the invariant that T is weighted over $[a, b]$, z is the LCA of u and v in T. Line 2 checks whether the current tree is weight-homogeneous. If it is, we immediately return the current weight a (line 3). Otherwise, the quantile value we are looking for is either on the left or on the right half of the weight-range $[a, b]$. In lines 5-11 we check, successively, the ranges $[a_0, b_0]$ and $[a_1, b_1]$ to determine how many nodes on the path from u to v in T have weights from the corresponding interval. The accumulator variable `acc` keeps track of these values and is certain to always be at most k. When the next value of `acc` is about to become larger than k (line 11), we are certain that the current weight-interval is the one we should descend to (line 12). The invariants are maintained in line 6: there, we calculate the views of the current nodes u, v, and z in the extracted tree we are looking at. It is clear that $\mathcal{O}(\lg \sigma)$ levels of recursion are explored. At each level of recursion, a constant number of `view_of` and `depth` operations are performed (lines 6-7). Hence, assuming the $\mathcal{O}(1)$-time for the latter operations themselves, we have a $\mathcal{O}(\lg \sigma)$ query-time algorithm, overall.

Algorithm 2 Selection: return the k^{th} smallest weight on the path from $u \in T$ to $v \in T$

```
Require: $z = LCA(u,v),\ a \le b,\ k \ge 0$
 1: function SELECT($T, u, v, z, k, w, [a..b]$)
 2:     if $a == b$ then
 3:         return $a$
 4:     $acc \leftarrow 0$
 5:     for $t \in 0..1$ do
 6:         $iu, iv, iz \leftarrow$ view_of($u, t$), view_of($v, t$), view_of($z, t$)
 7:         $du, dv, dz \leftarrow$ depth($B_t, ix$), depth($B_t, iy$), depth($B_t, iz$)
 8:         $dw \leftarrow du + dv - 2 \cdot dz$
 9:         if $a_t \le w \le b_t$ then          ▷ $[a_0..b_0] = [a..c],\ [a_1..b_1] = [c+1..b],\ c = (a+b)/2$
10:             $dw \leftarrow dw + 1$
11:         if $acc + dw > k$ then
12:             return SELECT($T_t, iu, iv, iz, k - acc, w, [a_t..b_t]$)
13:         $acc \leftarrow acc + dw$
14:     assert(false);          ▷ unreachable statement – line 12 must execute at some point
```

Path counting and path reporting. A procedure for the PC and PR is essentially similar to that for the PM problem. We maintain two nodes, u and v, as the query nodes with respect to the current extraction T, and a node z as the lowest common ancestor of u and v in the current tree T. Initially, $u, v \in T$ are the original query nodes, and T is the outermost tree. Correspondingly, z is the LCA of the nodes u and v in the original tree; we determine the weight of z and store it in w, which is passed down the recursion. Let $[a, b]$ be the query interval, and $[p, q]$ be the current range of weights of the tree. Initially, $[p, q] = [1, \sigma]$. First, we check whether the current interval $[p, q]$ is contained within $[a, b]$. If so, the entire set $A_{u,z} \cup A_{v,z}$ belongs to the answer. Here, we also check whether $w \in [a, b]$. Then we recurse on T_t ($t \in \{0, 1\}$) having computed the corresponding T_t-views of the nodes u, v, and z, and with the corresponding current range.

Algorithm 3 is adapted from [70], and reasoning similar to Algorithm 2 applies. Now we have a weight-range $[p, q]$, and maintain that $[p, q] \cap [a, b] \neq \emptyset$ (the appropriate action is in line 12). In line 2 we check if the query range $[p, q]$ is completely inside the current range. If so, we return all the nodes (if *report* argument is set to `true`) and the number thereof (for counting case). If not, we descend to T_0 and T_1 (line 14), as discussed previously. Algorithm 3 emulates traversal of a path in range tree, maintaining the current weight range $[a, b]$ and halving at at each step (line 14). As operations in lines 15 and 16 are constant-time, the algorithm runs in time $\mathcal{O}(\lg \sigma)$.

Algorithm 3 Counting and reporting.

```
Require: $z = LCA(u, v)$, $p \le q$
 1: function COUNTREPORT($T, u, v, z, w, [p, q], [a, b], vec = null, report = False$)
 2:     if $p \le a \le b \le q$ then
 3:         if $report$ then
 4:             for $pu \in \mathcal{A}(u)$ and $pu \ne z$ do
 5:                 $vec \leftarrow vec + \texttt{original\_node}(pu)$
 6:             for $pv \in \mathcal{A}(v)$ and $pv \ne z$ do
 7:                 $vec \leftarrow vec + \texttt{original\_node}(pv)$
 8:             if $a \le w \le b$ then
 9:                 $vec \leftarrow vec + \texttt{original\_node}(pv)$
10:         return $\texttt{depth}(u) + \texttt{depth}(v) - 2\texttt{depth}(z) + [\![w \in [p, q]]\!]$
11:     if $[p, q] \cap [a, b] = \emptyset$ then
12:         return $0$
13:     $res \leftarrow 0$
14:     for $t \in 0..1$ do                ▷ $[a_0, b_0] = [a, m],\ [a_1, b_1] = [m + 1, b],\ m = (a + b)/2$
15:         $iu, iv, iz \leftarrow \texttt{view\_of}(u, t),\ \texttt{view\_of}(v, t),\ \texttt{view\_of}(z, t)$
16:         $du, dv, dz \leftarrow \texttt{depth}(B_t, ix),\ \texttt{depth}(B_t, iy),\ \texttt{depth}(B_t, iz)$
17:         $res \leftarrow res + \texttt{countreport}(T_t, iu, iv, iz, w, [p, q], [a_t, b_t], vec, report)$
18:     return $res$
```

To summarize, the variant of `ext` that we design here uses $3n \lg \sigma + o(n \lg \sigma)$ bits to support `PM` and `PC` in $\mathcal{O}(\lg \sigma)$ time, and `PR` in $\mathcal{O}((1 + \kappa) \lg \sigma)$ time. Compared to the original succinct solution [70] based on tree extraction, our variant uses about 3 times the space with a minor slow-down in query time, but is easily implementable using bitvectors and BP, both of which have been studied experimentally (see e.g. [8, 91] for an extensive review).

5.4 Experimental Results

We now conduct experimental studies on the data structures for path queries that we have implemented.

5.4.1 Implementation

For ease of reference, we outline the implemented data structures in Table 5.2.

Naïve approaches (both plain pointer-based `nv`/`nv`^{L} and succinct `nv`^{c}) resolve a query on the path $P_{x,y}$ by explicitly traversing it from x to y. At each encountered node, we either (i) collect its weight into an array (for `PM`); (ii) check if its weight is in the query range (for `PC`/`PR`); (iii) if the check in (ii) succeeds, we collect the node into a container (for `PR`). In `PM`, we subsequently call the standard *introspective selection* algorithm [87] over the array of collected weights. Depths and parent pointers, explicitly stored at each node, guide in traversal from x and y upwards to their common ancestor. Plain pointer-based tree topologies are stored using the *forward-star* [5] representation. In `nv`^{L}, we equip `nv` with the linear-space LCA-support structure of [13] to find lowest common ancestors of the query nodes in $\mathcal{O}(1)$ time.

Succinct structures `ext`^{c}/`ext`^{P}/`hpw`^{c}/`hpw`^{P} are implemented with the help of the succinct data structures library `sdsl-lite` of Gog et al. [53]. To implement `hpw` and the practical variant of `ext` we designed in Section 5.3.2, two types of bitvectors are used: a compressed bitvector [100] (implemented in `sdsl::rrr_vector` of `sdsl-lite`) and plain bitvector (implemented in `sdsl::bit_vector` of `sdsl-lite`). For `nv`^{c}, the weights are stored using $\lceil \lg \sigma \rceil$ bits each in a sequence and the structure theoretically occupies $2n + n \lg \sigma + o(n \lg \sigma)$ bits. For uniformity, across our data structures, tree navigation is provided solely by a BP representation based on [51] (implemented in `sdsl::bp_support_gg`), chosen on the basis of our benchmarks.

Plain pointer-based implementation `ext`† is an implementation of the solution by He et al. [70] for the pointer-machine model, which uses tree extraction. In it, the views $x_0 \in T_0$, $x_1 \in T_1$ for each node that arises in the hierarchy of extractions, as well as the depths in T, are explicitly stored. Similarly, `hpw`† is a plain pointer-based implementation of the data structure by Patil et al. [98].

The source code of the library resides at `https://github.com/serkazi/tree_path_queries`.

	`num nodes`	`diameter`	σ	$\log\sigma$	H_0	Description
`eu.mst.osm`	27,024,535	109,251	121,270	16.89	9.52	An MST we constructed over map of Europe [97]
`eu.mst.dmcs`	18,010,173	115,920	843,781	19.69	8.93	An MST we constructured over European road network [58]
`eu.emst.dem`	50,000,000	175,518	5020	12.29	9.95	An Euclidean MST we constructed over DEM of Europe [96]
`mrs.emst.dem`	30,000,000	164,482	29,367	14.84	13.23	An Euclidean MST we constructed over DEM of Mars [89]

Table 5.1: Datasets description. DEM stands for Digital Elevation Model, and MST for minimum spanning tree. Weights are over $[\sigma]$, and H_0 is the entropy of the multiset of weights. In DEM, elevation (in meters) is used as weights. For `eu.mst.osm`, distance in meters between locations, and for `eu.mst.dmcs`, travel time between locations, for a proprietary "car" profile in tenths of a second, are used as weights.

5.4.2 Experimental Setup

The platform used is a 128GiB RAM, `Intel(R) Xeon(R) Gold` 6234 CPU 3.30GHz server running 4.15.0-54-generic 58-Ubuntu SMP x86_64 kernel. The build is due to `clang-8` with `-g,-O2, -std=c++17`,`mcmodel=large`,`-NDEBUG` flags.

Query generation. We generated query paths by choosing a pair uniformly at random (u.a.r.). To generate a range of weights, $[a, b]$, we follow the methodology of [26] and consider `large`, `medium`, and `small` configurations: given K, we generate the left bound $a \in [W]$ u.a.r., whereas b is generated u.a.r. from $[a, a + \lceil \frac{W-a}{K} \rceil]$. We set $K = 1, 10$, and 100 for respectively `large`, `medium`, and `small`. To counteract skew in weight distribution in some of the datasets, when generating the weight-range $[a, b]$, we in fact generate a pair from $[n]$ rather than $[\sigma]$ and map the positions to the sorted list of the input-weights, ensuring the number of nodes covered by the generated weight-range to be proportional to K^{-1}.

Experimental data. In the context of our experiments, a relevant dataset would both have a tree inherent to it *and* "naturally-occurring" weights. While tree topologies *per se* are abound (see e.g. University of Chile's collection of balanced parentheses representations of various tree-shaped entities [43]), large instances of "real-world" *weighted* trees are harder to come by. We found *Geographical Information Systems (GIS)* to be a potential source of such data; hence our choices of datasets outlined in Table 5.1.

	Symbol	Description
pointer-based	`nv`	Naïve data structure in Section 5.4.1
	$\texttt{nv}^{\texttt{L}}$	Naïve data structure in Section 5.4.1, augmented with LCA of [13]
	$\texttt{ext}^{\dagger}$	A solution based on tree extraction [70] in Section 5.2
	$\texttt{hpw}^{\dagger}$	A non-succinct version of the wavelet tree- and heavy-path decomposition-based solution of [98] in Section 5.3.
succinct	$\texttt{nv}^{\texttt{c}}$	Naïve data structure of Section 5.4.1, using succinct data structures to represent the tree structure and weights
	$\texttt{ext}^{\texttt{c}}$	$3n \lg \sigma + o(n \lg \sigma)$-bits-of-space scheme for tree extraction of Section 5.3.2, with compressed bitvectors
	$\texttt{ext}^{\texttt{P}}$	$3n \lg \sigma + o(n \lg \sigma)$-bits-of-space scheme for tree extraction of Section 5.3.2, with uncompressed bitvectors
	$\texttt{hpw}^{\texttt{c}}$	Succinct version of **`hpw`**, with compressed bitvectors
	$\texttt{hpw}^{\texttt{P}}$	Succinct version of **`hpw`**, with uncompressed bitvectors

Table 5.2: The implemented data structures and the abbreviations used to refer to them.

In Table 5.1, we endow our datasets with descriptive handles, and give the relevant statistics, including tree diameter (i.e. the longest distance in nodes between any two nodes of the tree), alphabet size, and entropy of the weight-set.

We next proceed to describe in greater detail each of the entries of Table 5.1.

The `eu.mst.osm` dataset originates from *Open Street Map* (OSM) project [97]. Specifically, we converted OSM data from [52] to `*.sql` insert-files using a well-known software tool [84]. Our instance of PostgreSQL RDBMS, extended with Postgis [34], was then populated with the `INSERT` statements obtained using `osm2po`, thereby making the OSM data *routable*. Finally, since the OSM data still contains information that is irrelevant to our purposes, we extract all possible links in `source target m` format into a regular `*.csv` file, using a simple SQL-query. [3] Thus, each line in our final file states that there is a road from `source` to `target` of length `m` meters, where the former two are some entities internal to OSM, and mere integer IDs for all practical purposes.

Next, we find a largest-cardinality minimum spanning tree (MST), for simplicity considering the graph to be bidirectional. We then orient the tree via a depth-first-search from a randomly selected but fixed source. The nodes are then assigned the weights of the incoming arcs (with the source being assigned zero).

[3]There are other ways of accomplishing this step, e.g. via **`pgRouting`** at **`http://pgrouting.org/`**.

The `eu.mst.dmcs` dataset is from the 9th DIMACS challenge [33], and was downloaded from [58]. The graph was converted from the raw DIMACS format, keeping only the largest strongly-connected component and removing multi-arcs. The metric is travel distance, in meters [11]. The tree was obtained in the way exactly similar to the `eu.mst.osm` dataset.

Digital elevation models (DEM) lend itself to the conversion into weighted trees rather naturally, as the (x, y) coordinates supply the basis for tree topology, and the elevation z serves as the weight. These datasets represent elevations above certain levels on a planet's surface (`eu.emst.dem`, `mrs.emst.dem`) and are each a result of the following two-phase procedure.

We outline the procedure for the first dataset, as the procedure for the second one is analogous.

The first, preprocessing phase, fetches original raw data from [96], as 30-by-30 degree `GeoTIFF` tiles. Then the tiles are projected into Cartesian plane, in *Transverse Mercator projection* using meters, via the `gdalwarp` tool of the *GDAL Geospatial Data Abstraction software Library* [49]. The projection is further dumped into the XYZ format via `gdal_translate` tool from GDAL, invoking the command `gdal_translate` with relevant input/output files and command-line arguments. Here, what is referred to as a file in XYZ format is essentially a collection of rows, containing x, y, and z values, being respectively the xy-coordinates of the projection along the (original) elevation of the projected point. Finally, the *nodata*-values concomitant to a projection were removed with UNIX's standard `sed` editor.

The second phase constructs the *Euclidean Minimum Spanning Tree* (EMST) using the data in the XYZ format obtained during phase one. We used *The Computational Geometry Algorithms Library* [105]; namely, we trivially adapted the example at [35]. The Mars Elevation (`mrs.emst.dem`) was obtained from [89] and processed similarly. Prior to building EMST, `eu.emst.dem` and `mrs.emst.dem` datasets were sampled u.a.r. to have the sizes shown in Table 5.1. In the case of `mrs.emst.dem`, the data has already been supplied in the XYZ format, sparing us the projection and translation steps.

5.4.3 Space Performance and Construction Costs

Each data structure we implement (be it ever `nv-`, `ext-`, or `hpw`-family), taken individually, answers all three types of queries (`PM`, `PC`, and `PR`). Hence, we consider space consumption first.

The upper part of the Table 5.3 shows the space usage of our data structures. The structures

	Dataset	$\mathtt{nv}$	$\mathtt{nv}^{\mathtt{L}}$	$\mathtt{hpw}^{\dagger}$	$\mathtt{ext}^{\dagger}$	$\mathtt{nv}^{\mathtt{c}}$	$\mathtt{ext}^{\mathtt{c}}$	$\mathtt{ext}^{\mathtt{p}}$	$\mathtt{hpw}^{\mathtt{c}}$	$\mathtt{hpw}^{\mathtt{p}}$
space	eu.mst.osm	406.3	972.1	3801	5943	21.71	59.85	75.74	21.71	34.42
	eu.mst.dmcs	406.4	974.0	4274	6768	34.46	82.16	106.0	29.69	48.77
	eu.emst.dem	394.1	988.5	3342	4613	19.64	45.41	59.15	19.64	31.66
	mrs.emst.dem	386.7	1005	3579	5383	17.35	51.71	66.02	17.35	28.80
peak/time	eu.mst.osm	491.0/1	987.9/5	3785/28	9586/47	21.71/1	295.0/23	295.0/23	1347/62	1347/61
	eu.mst.dmcs	439.8/1	1002/4	4403/19	12382/37	29.69/1	399.7/18	399.7/18	1360/42	1360/42
	eu.emst.dem	401.0/2	1021/10	3460/47	5286/67	19.64/1	287.6/32	287.6/32	1333/115	1333/115
	mrs.emst.dem	392.4/1	1016/5	3719/30	6027/46	17.35/1	269.3/22	269.3/22	1337/69	1337/69

TABLE 5.3: Storage space and construction characteristics. (upper) Space occupancy of our data structures, in bits per node (bpn), when loaded into memory; (lower) peak memory usage (**m** bpn) and time (t in seconds) during construction is shown as $\mathbf{m}/t$.

$\mathtt{nv}/\mathtt{nv}^{\mathtt{L}}$ are lighter than $\mathtt{ext}^{\dagger}/\mathtt{hpw}^{\dagger}$, as expected. Adding fast LCA support (in our implementation of the solution of [12]) doubles the space requirement for `nv`, whereas succinctness ($\mathtt{nv}^{\mathtt{c}}$) uses up to 20 times less space than `nv`. The difference between $\mathtt{ext}^{\dagger}$ and $\mathtt{hpw}^{\dagger}$, in turn, is in explicit storage of the 0-views for each of the $\Theta(n \lg \sigma)$ nodes occurring during tree extraction. In $\mathtt{hpw}^{\dagger}$, by contrast, $\mathtt{rank}_0$ is induced from $\mathtt{rank}_1$ (via subtraction) – hence the difference in the empirical sizes of the otherwise $\Theta(n \lg \sigma)$-word data structures.

The succinct $\mathtt{nv}^{\mathtt{c}}$'s empirical space occupancy is close to the information-theoretic minimum given by $\lg \sigma + 2$ (Table 5.1). The structures $\mathtt{ext}^{\mathtt{c}}/\mathtt{ext}^{\mathtt{p}}$ occupy about three times as much, which is consistent with the design of our practical solution (Section 5.3.2). It is interesting to note that the data structure $\mathtt{hpw}^{\mathtt{c}}$ occupies space close to bare succinct storage of the input alone ($\mathtt{nv}^{\mathtt{c}}$). Entropy-compression significantly impacts both families of succinct structures, `hpw` and `ext`, saving up to 20 bits per node when switching from plain bitvector to a compressed one. Compared to pointer-based solutions ($\mathtt{nv}/\mathtt{nv}^{\mathtt{L}}/\mathtt{hpw}^{\dagger}/\mathtt{ext}^{\dagger}$), we note that $\mathtt{ext}^{\mathtt{c}}/\mathtt{ext}^{\mathtt{p}}/\mathtt{hpw}^{\mathtt{c}}/\mathtt{hpw}^{\mathtt{p}}$ still allow usual navigational operations on T, whereas the former shed this redundancy, to save space, after preprocessing.

Overall, the succinct $\mathtt{hpw}^{\mathtt{p}}/\mathtt{hpw}^{\mathtt{c}}/\mathtt{ext}^{\mathtt{p}}/\mathtt{ext}^{\mathtt{c}}$ perform very well, being all well-under 1 gigabyte for the large datasets we use. This suggests scalability: when trees are so large as not to fit into main memory, it is clear that the succinct solutions are the method of choice.

The lower part in Table 5.3 shows peak memory usage (**m**, in bits per node) and construction time (t, in seconds), as $\mathbf{m}/t$. The structures $\mathtt{ext}^{\mathtt{p}}/\mathtt{ext}^{\mathtt{c}}$ are about three times faster than $\mathtt{hpw}^{\mathtt{p}}/\mathtt{hpw}^{\mathtt{c}}$ to build, and use four times less space at peak. This is expected, as `hpw` builds two different structures (HPD and then WT). This is reversed for $\mathtt{ext}^{\dagger}/\mathtt{hpw}^{\dagger}$; time-wise, as $\mathtt{ext}^{\dagger}$ performs more memory allocations during construction (although our succinct structures are

	Dataset	nv	nv^{L}	ext†	hpw†	nv^{c}	ext^{c}	ext^{p}	hpw^{c}	hpw^{p}	
median	eu.mst.osm	658	475	4.22	6.10	7078	85.3	51.1	111	51.2	
	eu.mst.dmcs	566	412	5.16	6.28	6556	84.6	54.8	120	54.7	
	eu.emst.dem	710	436	4.44	5.10	9404	106	81.9	96.7	54.9	
	mrs.emst.dem	472	298	4.93	4.53	7018	124	97.0	88.3	49.5	
counting	eu.mst.osm	238	140	6.88	18.4	3553	247	167	139	56.9	large
	eu.mst.dmcs	204	121	7.31	19.7	3300	253	178	142	57.3	
	eu.emst.dem	338	195	5.97	11.5	4835	215	168	105	55.9	
	mrs.emst.dem	232	174	5.25	8.40	3614	206	164	91	49.3	
	eu.mst.osm	244	143	5.47	17.8	3555	213	146	129	54.2	medium
	eu.mst.dmcs	209	124	6.94	18.4	3297	224	160	133	56.5	
	eu.emst.dem	339	195	4.55	10.0	4840	178	140	100	54.9	
	mrs.emst.dem	237	143	5.91	8.74	3613	199	154	89.7	48.9	
	eu.mst.osm	239	139	5.25	15.4	3551	190	132	119	53.9	small
	eu.mst.dmcs	209	123	5.25	18.9	3300	206	148	126	55.2	
	eu.emst.dem	347	200	3.92	9.34	4832	154	124	94.9	53.2	
	mrs.emst.dem	238	144	4.82	7.41	3615	178	133	84.2	47.6	

Table 5.4: Average time to answer a path median or a path counting query , in μs. A fixed set of 10^6 randomly generated path median and path counting queries is used. Path counting is given in `large`, `medium`, and `small` configurations.

flattened into a heap layout, `ext`† stores pointers to T_0/T_1; this is less of a concern for `hpw`†, whose very purpose is tree linearisation).

5.4.4 Path Median Queries

The upper section of Table 5.4 records the mean time for a single median query (in μs) averaged over a fixed set of 10^6 queries, randomly generated using the approach in Section 5.4.1.

Our discussion focuses on the most salient features of the data.

Comparison of `hpw` and `ext`. Succinct structures `hpw`^{c}/`hpw`^{p}/`ext`^{c}/`ext`^{p} perform well on these queries, with a slow-down of at most 20-30 times from their pointer-based counterparts. Using entropy-compression degrades the speed of `hpw` almost twice. Overall, the families `hpw` and `ext` seem to perform at the same order of magnitude. This is surprising, as in theory `hpw` should be a factor of $\lg n$ slower. There can be several contributing factors; two stand out.

First, take the number of chains in HPD. The discrepancy between the theoretical and practical performance of `hpw` and `ext` is explained partly by the small number of segments in HPD, averaging 9 ± 2 for our queries. (The number of unary-degree nodes in our datasets is 35%-56%, which makes smaller number of heavy-path segments prevalent. We did not use trees with few unary-degree nodes in our experiments, as the height of such trees are not large enough to make constructing data structures for path queries worthwhile.) When the queries are partitioned by the number of chains in the HPD, the curves for `ext`^{c}/`ext`^{p} remain flat whereas those for `hpw`^{c}/`hpw`^{p} grow linearly. Let us study Figure 5.5 and take `eu.mst.dmcs` as an example. When the query path is partitioned into 9 chains, `ext`^{p} is only slightly faster than `hpw`^{p}, but when the query path contains 19 chains, `ext`^{p} is about 2.3 times so. This suggests to favour the `ext` family over `hpw` whenever performance in the worst case is important.

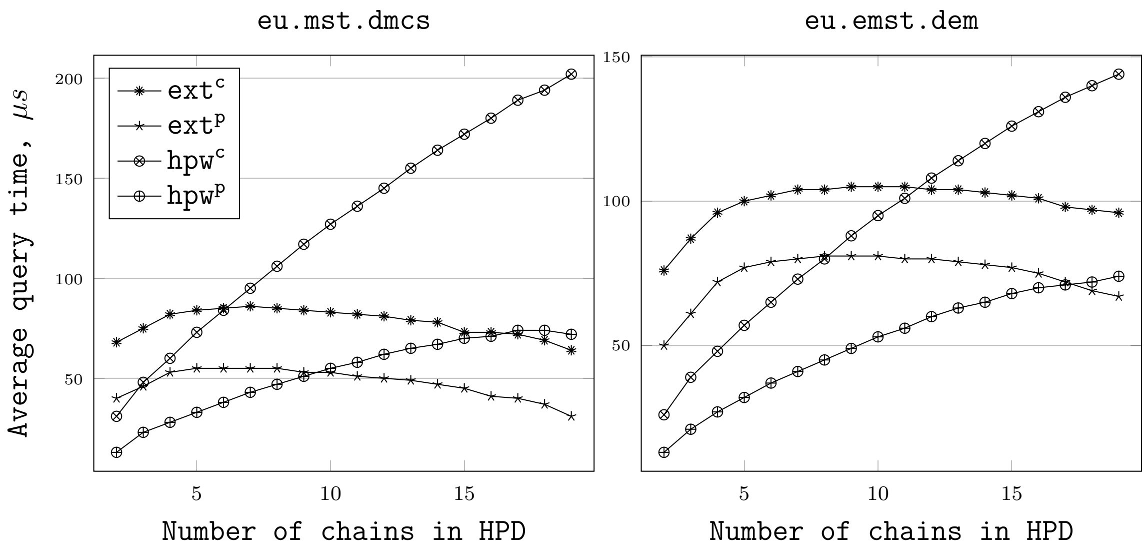

FIGURE 5.5: Average time to answer a path median query, controlled for the number of segments in heavy-path decomposition, in μs. Random fixed query set of size 10^6.

Second, consider the usage patterns for `rank`/`select`. Navigational operations in the (succinct) `ext` and `hpw`, despite of similar theoretical worst-case guarantees, involve different patterns of using the `rank`/`select` primitives. For one, `hpw`^{p}/`hpw`^{c} does not call LCA during the search – mapping of the search ranges when descending down the recursion is accomplished by a single `rank` call, whereas `ext`^{p}/`ext`^{c} computes LCA at each level of descent (for its own analog of `rank` – the view computation in Algorithm 1). Now, LCA is a non-trivial combination of `rank`/`select` calls. The difference between `ext`^{p}/`ext`^{c} and `hpw`^{p}/`hpw`^{c} should therefore become pronounced in a large enough tree; with tangible HPD sizes, the constants involved in (albeit theoretically $\mathcal{O}(1)$) LCA calls are overcome by $\lg n$.

Overall evaluation. Naïve structures `nv`/$\texttt{nv}^{\texttt{L}}$/$\texttt{nv}^{\texttt{c}}$ are visibly slower in `PM` than in `PC` (considered in Section 5.4.5), as expected — for `PM`, having collected the nodes encountered, we also call a selection algorithm. In `PC`, by contrast, neither insertions into a container nor a subsequent search for median are involved. Navigation and weights-uncompression in $\texttt{nv}^{\texttt{c}}$ render it about 10 times slower than its plain counterpart. The $\texttt{nv}^{\texttt{L}}$ being little less than twice faster than its LCA-devoid counterpart, `nv`, is explained by the latter effectively traversing the query path twice — once to locate the LCA, and once again to answer the query proper. Any succinct solution is about 4-8 times faster than the fastest naïve, $\texttt{nv}^{\texttt{L}}$.

5.4.5 Path Counting Queries

The lower section in Table 5.4 records the mean time for a single counting query (in μs) averaged over a fixed set of 10^6 randomly generated queries, for `large`, `medium`, and `small` setups.

Structures `nv`/$\texttt{nv}^{\texttt{L}}$/$\texttt{nv}^{\texttt{c}}$ are insensitive to κ (i.e. the output size), as the bottleneck is in physically traversing the path.

Succinct structures $\texttt{hpw}^{\texttt{p}}$/$\texttt{hpw}^{\texttt{c}}$ and $\texttt{ext}^{\texttt{p}}$/$\texttt{ext}^{\texttt{c}}$ exhibit decreasing running times as one moves from the `large` to the `small` configuration – as the query weight-range shrinks, so does the chance of branching during the traversal of the implicit range tree. The fastest (uncompressed) $\texttt{hpw}^{\texttt{p}}$ and the slowest (compressed) $\texttt{ext}^{\texttt{c}}$ succinct solutions differ by a factor of 4, which is intrinsically larger constants in $\texttt{ext}^{\texttt{c}}$'s implementation compounded with slower `rank/select` primitives in compressed bitvectors, at play. The uncompressed $\texttt{hpw}^{\texttt{p}}$ is about 2-3 times faster than $\texttt{ext}^{\texttt{p}}$, the gap narrowing towards the `small` setup. The slowest succinct structure, $\texttt{ext}^{\texttt{c}}$, is nonetheless competitive with the `nv`/$\texttt{nv}^{\texttt{L}}$ already in `large` configuration, with the advantage of being insensitive to tree topology.

In $\texttt{ext}^{\dagger}$-$\texttt{hpw}^{\dagger}$ pair, $\texttt{hpw}^{\dagger}$ is 2-3 times slower. This is predictable, as the inherent $\lg n$-factor slow-down in $\texttt{hpw}^{\dagger}$ is no longer offset by differing memory access patterns – following a pointer "downwards" (i.e. $0/1$-view in $\texttt{ext}^{\dagger}$ and $\texttt{rank}_{0/1}$ in $\texttt{hpw}^{\dagger}$) each require a single memory access.

5.4.6 Path Reporting Queries

Table 5.5 records the mean time for a single reporting query (in μs) averaged over a fixed set of 10^6 randomly generated queries, for the `large`, `medium`, and `small` setups.

Structures $\texttt{hpw}^{\texttt{c}}$/$\texttt{hpw}^{\texttt{p}}$/$\texttt{ext}^{\texttt{c}}$/$\texttt{ext}^{\texttt{p}}$ recover each reported node's weight in $\mathcal{O}(\lg \sigma)$ time. Thus, when $\lg n \ll \kappa$, the query time for both `ext` and `hpw` families become $\mathcal{O}(\kappa \cdot \lg \sigma)$. (At

Dataset	κ	nv	$\mathtt{nv}^{\mathtt{L}}$	$\mathtt{ext}^{\dagger}$	$\mathtt{hpw}^{\dagger}$	$\mathtt{nv}^{\mathtt{c}}$	$\mathtt{ext}^{\mathtt{c}}$	$\mathtt{ext}^{\mathtt{P}}$	$\mathtt{hpw}^{\mathtt{c}}$	$\mathtt{hpw}^{\mathtt{P}}$	
eu.mst.osm	9,840	356	256	184	70.7	3766	▨	▨	▨	▨	large
eu.mst.dmcs	9,163	309	224	147	66.8	3485	▨	▨	▨	▨	large
eu.emst.dem	14,211	389	241	140	77.5	4926	▨	▨	▨	▨	large
mrs.emst.dem	10,576	267	178	89.2	55.1	3668	▨	▨	▨	▨	large
eu.mst.osm	1,093	322	222	43.7	28.8	3706	▨	▨	▨	▨	medium
eu.mst.dmcs	1,090	277	196	34.0	29.7	3434	▨	▨	▨	▨	medium
eu.emst.dem	1,464	354	206	32.1	20.1	4880	▨	▨	▨	▨	medium
mrs.emst.dem	1,392	250	151	22.1	15.6	3639	▨	▨	▨	▨	medium
eu.mst.osm	182	311	212	13.8	19.0	3685	1965	485	795	226	small
eu.mst.dmcs	236	271	193	13.2	21.0	3529	2518	632	1043	292	small
eu.emst.dem	215	353	203	10.2	12.7	4873	1276	378	590	205	small
mrs.emst.dem	117	242	145	8.88	9.57	3632	881	278	475	162	small

TABLE 5.5: Average time to answer a path reporting query, in μs. A set of 10^6 randomly generated path reporting queries is used. The queries are given in `large`, `medium`, and `small` configurations. Average output size for each group is given in column κ.

this juncture, a caveat is in order: The design of `hpw` [98], described in Section 5.3.1, allows a PR-query to only return the *index in the array C* — not the original preorder identifier of the node, as does the `ext`.) When κ is large, therefore, these structures are not suitable for use in PR, as `nv`/$\mathtt{nv}^{\mathtt{L}}$/$\mathtt{nv}^{\mathtt{c}}$are clearly superior ($\mathcal{O}((1+\kappa)\lg n)$ *vs* $\mathcal{O}(\kappa)$), and we confine the experiments for $\mathtt{ext}^{\mathtt{c}}$/$\mathtt{ext}^{\mathtt{P}}$/$\mathtt{hpw}^{\mathtt{c}}$/$\mathtt{hpw}^{\mathtt{P}}$ to the `small` setup only (bottom-right corner in Table 5.5).

We observe that the succinct structures $\mathtt{ext}^{\mathtt{P}}$ and $\mathtt{hpw}^{\mathtt{P}}$ are competitive with `nv`/$\mathtt{nv}^{\mathtt{L}}$, in the `small` setting: informally, time saved in locating the nodes to report is used to uncompress the nodes' weights (whereas in `nv`/$\mathtt{nv}^{\mathtt{L}}$ the weights are explicit). Between the succinct `ext` and `hpw`, clearly `hpw` is faster, as `select` on a sequence as we go up the wavelet tree tend to have lower constant factors than the counterpart operation on BP.

Structures $\mathtt{hpw}^{\dagger}$ and $\mathtt{ext}^{\dagger}$ exhibit same order of magnitude in query time, with the former being sometimes about 2 times faster on non-`small` setups. Among the two somewhat intertwined reasons, one is that $\mathtt{hpw}^{\dagger}$ returns an index to the permuted array, as noted previously. (Converting to the original id would necessitate an additional memory access.) Secondly, in the implicit range tree during the 2D search in $\mathtt{hpw}^{\dagger}$, when the current range is contained within the query interval, we start reporting the node weights by merely incrementing a counter — position in the WT sequence. By contrast, in such situations $\mathtt{ext}^{\dagger}$ iterates through the nodes being reported calling `parent` for the current node, which is one additional memory access

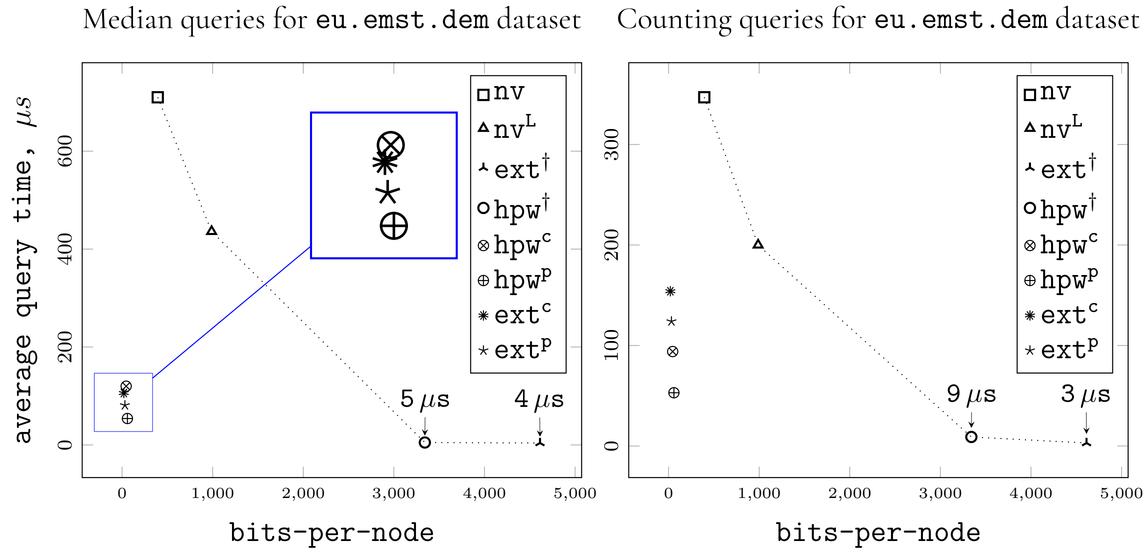

FIGURE 5.6: Visualization of some of the entries in Table 5.4. Inner rectangle magnifies the mutual configuration of succinct data structures `hpw`$^{\mathtt{p}}$/`hpw`$^{\mathtt{c}}$ and `ext`$^{\mathtt{p}}$/`ext`$^{\mathtt{c}}$. The succinct naïve structure `nv`$^{\mathtt{c}}$ is not shown.

compared to `hpw`† (at the scale of μs, this matters). Indeed, operations on trees tend to be little more expensive than similar operations on sequences. That said, we note that in the `small` setup, `ext`† is uniformly better than `hpw`†: the additive term influences the total running time more than the reporting part.

Structures `nv`/`nv`$^{\mathtt{L}}$/`nv`$^{\mathtt{c}}$ are less sensitive to the query weight-range's magnitude, since they simply scan the path along with pushing into a container. The differences in running time in Table 5.5 between the configurations are thus accounted for by the container operations' cost. Naïve structures' query times for `PR` being dependent solely on the query path's length, they are unfeasible for large-diameters trees (whereas they may be suitable for shallow ones, e.g. originating from "small-world" networks).

OVERALL EVALUATION. We visualize in Figure 5.6 some typical entries in Table 5.4 to illustrate the structures clustering along the space/time trade-offs: `nv`/`nv`$^{\mathtt{L}}$ (upper-left corner) are lighter in terms of space, but slow; pointer-based `ext`†/`hpw`† are very fast, but space-heavy. Between the two extremes of the spectrum, the succinct structures `ext`$^{\mathtt{c}}$/`ext`$^{\mathtt{p}}$/`hpw`$^{\mathtt{c}}$/`hpw`$^{\mathtt{p}}$, whose mutual configuration is shown magnified in the inner rectangle, are space-economical and yet offer fast query times.

5.5 Conclusion

We have designed and experimentally evaluated recent algorithmic proposals in path queries in weighted trees, by either faithfully replicating them or offering practical alternatives. Our data structures include both plain pointer-based and succinct implementations. Our succinct realizations are themselves further specialized to be either plain or entropy-compressed.

We measure both query time and space performance of our data structures on large practical sets. We find that the succinct structures we implement present an attractive alternative to plain pointer-based solutions, in scenarios with critical space- and query time-performance and reasonable tolerance to slow-down. Some of the structures we implement ($\texttt{hpw}^{c}$) occupy space equal to bare compressed storage ($\texttt{nv}^{c}$) of the object and yet offer fast queries on top of it, while another structure ($\texttt{ext}^{c}/\texttt{ext}^{p}$) occupies space comparable to $\texttt{nv}^{c}$, offers fast queries and low peak memory in construction. While `hpw` succinct family performs well in average, thus representing an attractive trade-off between query time and space occupancy, `ext` is robust to the structure of the underlying tree, and is therefore recommended when strong worst-case query-time guarantees are vital.

Our design of the practical succinct structure based on tree extraction (`ext`) results in theoretical space occupancy of $3n \lg \sigma + o(n \lg \sigma)$ bits, which explains its somewhat higher empirical space cost when compared to the succinct `hpw` family. At the same time, verbatim implementation of the space-optimal solution by He et al. [70] draws on components that are likely to be cumbersome in practice. For the path query types considered in this study, therefore, realization of the theoretically time- and space-optimal data structure – or indeed some feasible alternative thereof – remains an interesting open problem in algorithm engineering.

Chapter 6

Conclusion

We conclude the thesis by stating the results achieved, along with the highlights on the techniques that enabled them (Section 6.1). Then in Section 6.2 we discuss possible directions of research, and technical difficulties overcoming which would bring new results or improve the existing ones.

6.1 Results and Discussion

Orthogonal range searching is a classical discipline in computer science. We study trees whose nodes are assigned d-dimensional weight vectors, as generalizations of $(d+1)$-dimensional point-sets in $\mathbb{R}^{d+1}$ (or the rank space $[n]^{d+1}$). Since any two nodes in a tree uniquely define a path, weighted trees do indeed generalize point-sets. In addition, when a tree degenerates into a single path, both objects become identical.

For a formal setting, let $d \geq 0$ is an integer constant, and $0 < \epsilon < 1$ be an arbitrarily small constant. The input object is an ordinal tree T on n nodes, each of which is assigned a d-dimensional weight vector (when $d = 0$, the tree T is unweighted).

This thesis comprises three main lines of research on preprocessing such a tree for efficient online path queries.

The first line studies the generalizations of the **classical orthogonal range searching** problems – dominance reporting, range successor, range counting, and range reporting – to the case of the weighted trees. We create a general framework for semigroup path sum problem, and apply it to extend our base-case data structures for the above problems to higher dimensions.

We study the ancestor dominance reporting problem as a generalization of the well-known dominance reporting problem. In T, a node dominates some other node *iff* the former is an ancestor of the latter, and the former's weight vector dominates that of the latter. Out data structure uses $\mathcal{O}(n \lg^{d-2} n)$ words of space and has $\mathcal{O}(\lg^{d-1} n + k)$ query time for $d \geq 2$. When $d = 2$, this matches the space bound for 3D dominance reporting of [2, 17], while still providing efficient query support. When $d \geq 3$, we also achieve a trade-off of $\mathcal{O}(n \lg^{d-2+\epsilon} n)$ words of space, with query time of $\mathcal{O}(\lg^{d-1} n/(\lg\lg n)^{d-2} + k)$.

To achieve these results, we work in the *layers of maxima* [25, 83] paradigm for dominance reporting, and provide efficient means of iterating through such layer, i.e. enumerating the 2-maximal nodes.

Next, for the same T, we study the path successor problem. Our data structure uses $\mathcal{O}(n \lg^{d-1} n)$ words and and executes queries in $\mathcal{O}(\log^{d-1+\epsilon} n)$ time, where $\epsilon > 0$ is an arbitrarily small constant. When $d = 1$, these bounds match the first trade-off for the corresponding range successor problem in 2D, given by Nekrich and Navarro [95].

Explicitly maintaining several tree extractions for various subranges of the weight alphabet is clearly space-prohibitive. We propose a way of doing so fairly efficiently in linear space in the special case of ranges that are the intervals represented by the nodes of a (binary) range tree, which may be sufficient for some applications.

We next solve the path counting problem, in $\mathcal{O}(n(\frac{\lg n}{\lg\lg n})^{d-1})$ words and $\mathcal{O}((\frac{\lg n}{\lg\lg n})^{d})$ query time. This matches the best bound for the corresponding range counting problem in $d+1$ dimensions [74].

The solution was due to a careful choice of the parameters in the two-level partitioning using tree covering (Section 2.3.2) and the ability to encode the partial answers in accordingly smaller space with shrinking partition sizes.

Finally, we solve the path reporting problem in $\mathcal{O}(n \lg^{d-1+\epsilon} n)$ words and $\mathcal{O}(\frac{\lg^{d-1} n}{(\lg\lg n)^{d-2}} + k)$ query time, for $d \geq 2$. When $d = 2$, the space matches that of the corresponding result of Chan et al. [21] on 3D range reporting, while the first term in the query complexity is slowed down by a sub-logarithmic factor.

This result was achieved by directly applying our general framework of semigroup path sum problem to the currently-best result on path reporting on 1D-weighted trees, by [19].

The second line of research is the study of the **categorical path counting** problem. In this problem, T's nodes are assigned categories. We show a conditional lower bound on the hardness of the categorical path counting problem, and provide a data structure whose query time matches the lower bound within polylogarithmic factors. Further, we extend the solution to the case of trees weighted with d-dimensional weight vectors, matching the best currently-known data structures within polylogarithmic factors, both in space and time, when $d \geq 2$. We further solve the above two problems in approximate mode; the data structures we propose are simple to the degree of being implementable.

Balancing pre-computation against explicit traversals with on-the-fly computations proves to be an remarkably useful template for data-structuring. It is certainly not new and has been applied in various contexts such as path queries proper [36] and categorical counting in $\mathbb{R}^d$ [76]. We use the currently best results in path range emptiness queries and another useful tree partitioning technique [36] to apply the template to exact categorical counting. The same high-level idea is at work in one of our approximate solutions, too. It employs sketching and yet another tree mark-up technique. The ability to update sketches fast, a feature that could have been overlooked in a static data structure, was aptly used in the "explicit traversals" part of the template.

We conclude the thesis by the study of the **practical performance** of the data structures for path queries. For a 1D-weighted tree T, we implement the currently best data structures for the path median, path counting, and path reporting problems. These problems generalize respectively the well-known range median problem in 1D arrays, as well as the 2D path counting and path reporting problems. We implement our data structures in either succinct or plain pointer-based form. Through a series of experiments on large datasets, we show that succinct data structures can serve as viable alternatives to traditional, pointer-based implementations, being both fast and small enough. Our succinct implementations are further divided into entropy-compressed and "plain" succinct forms, and we study the effect of entropy-compression on space and time.

6.2 Future Directions

Being dependent on the output size, k, range searching problems of the reporting variety seem to be the first candidates for closing the gap between the computation on trees and point-sets.

The challenge thus very often is in matching the additive term of the query time. Our solution to the ancestor dominance problem is exponentially worse in the additive term, when compared to the state-of-the-art in dominance reporting in 3D ($\mathcal{O}(\lg n + k)$ versus $\mathcal{O}(\lg\lg n + k)$). An obvious question is, can we do better? In particular, can one generalize *shallow cuttings* [93, 2, 3], which have been instrumental for the currently best results in dominance reporting [2, 17] to trees?

The same question applies to the categorical version of path reporting – a natural extension of the path reporting problem to the categorical case, where one needs to enumerate all the distinct categories, each at most $\mathcal{O}(1)$ times, on a given query path. Nekrich [94] used an ingenious data structure called *path range trees* for categorical range reporting and approximate categorical range counting. Roughly, the *modus operandi* is to build a range tree by the x-coordinate and to store certain, controlled amount of the lowest points (i.e. with the smallest y-coordinate) pertaining to the nodes that "hang inwards" in the range tree (for these range-tree nodes, the constraint on the x-coordinate is lifted, as illustrated in Figure 3.5). The snag in porting this method to ordinal trees is precisely in the "lowest points" part – the entire bottom part (or, alternatively, the crown) of the tree would qualify as such, and there is no obvious way of limiting its size.

Indeed, the "lowest points" trick is omnipresent in the currently best solutions and trade-offs (see [21, 110] to name a few). The other best trade-off in the path successor problem due to Zhou [110] (which is $\mathcal{O}(n \lg\lg n)$ words of space and $\mathcal{O}(\lg\lg n)$ time; see Section 3.1.1) is harder to achieve in trees copying the author's blueprint only, owing to its use of sparsification via the lowest points.

And then there is the *block-shrinking* technique of [36] that enabled a $\sqrt{w}$-fold speedup in solving certain types of path frequency queries (here, $w = \Omega(\lg n)$ is the word size in the word-RAM). For example, for the path mode problem, they built a linear-space data structure with $\mathcal{O}(\sqrt{\frac{n}{w}} \lg\lg n)$ query time. The idea is to use the tree mark-up scheme of Lemma 4.1 recursively, with ever smaller block sizes. They observe that the information one needs to explicitly store with each recursive mark-up (block-shrinking) is expressed in the number of bits that is asymptotically smaller than the corresponding block size t. Roughly speaking, primarily this reduced space usage for the pre-computed part (as in the table M in Section 4.2.3) from $(n/t)^2 \lg t$ bits to $(n/t)^2 \lg t'$ bits, with $t' \ll t$. In the case of categorical path counting, however, the partial answer (i.e. the number of distinct categories) can be $\Theta(t)$, so the block-shrinking speedup is not directly applicable. If one is ever to shave off a factor of $\sqrt{w}$ from our query time for categorical path counting (Theorem 4.2), the techniques are likely to be quite different.

Furthermore, *sketches* seem to be relatively new to path queries, and practical performance of our data structures would be interesting to evaluate experimentally, as Cormode et al. [28] did on "real-world" streams. From a theoretical point of view, one can not but feel that our solution in Section 4.4 is too coarse, and certainly rather straightforward. The next logical step after the realization that the sketches behave as regular numbers is to try packing them into fewer number of bits, in order to allow for multiple levels of decomposition. (For example, this is done in Section 3.6 for storing answers pertaining to mini-trees.) It turns out, however, that the p-stable distributions [28] that underlie the sketches by necessity assume unbounded values (the proof is beyond the scope of the thesis).

Bibliography

[1] Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical Compressed Suffix Trees. *Algorithms*, 6(2):319–351, 2013.

[2] Peyman Afshani. On Dominance Reporting in 3D. In *ESA*, pages 41–51, 2008.

[3] Peyman Afshani and Konstantinos Tsakalidis. Optimal deterministic shallow cuttings for 3-d dominance ranges. *Algorithmica*, 80(11):3192–3206, 2018.

[4] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range Searching in Categorical Data: Colored Range Searching on Grid. In *ESA*, pages 17–28, 2002.

[5] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows - Theory, Algorithms and Applications*. Prentice Hall, 1993.

[6] Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel-Aviv University, 1987.

[7] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *ALENEX*, pages 84–97, 2010.

[8] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct Trees in Practice. In *ALENEX*, pages 84–97, 2010.

[9] Diego Arroyuelo, Francisco Claude, Reza Dorrigiv, Stephane Durocher, Meng He, Alejandro López-Ortiz, J. Ian Munro, Patrick K. Nicholson, Alejandro Salinger, and Matthew Skala. Untangled monotonic chains and adaptive range search. *Theor. Comput. Sci.*, 412(32):4200–4211, 2011.

[10] Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. *Theory Comput.*, 8(1):69–94, 2012.

[11] Lilian Beckert and Valentin Buchhold. personal communication.

[12] Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *LATIN*, volume 1776, pages 88–94, 2000.

[13] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[14] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing Trees of Higher Degree. *Algorithmica*, 43(4):275–292, 2005.

[15] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[16] Nieves R. Brisaboa, Guillermo de Bernardo, Roberto Konow, Gonzalo Navarro, and Diego Seco. Aggregated 2d range queries on clustered points. *Inf. Syst.*, 60:34–49, 2016.

[17] Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Trans. Algorithms*, 9(3):22:1–22:22, 2013.

[18] Timothy M. Chan. Speeding up the four russians algorithm by about one more logarithmic factor. In *SODA*, pages 212–217, 2015.

[19] Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications. *Algorithmica*, 78(2):453–491, 2017.

[20] Timothy M. Chan, Qizheng He, and Yakov Nekrich. Further results on colored range searching. In *SoCG*, volume 164, pages 28:1–28:15, 2020.

[21] Timothy M. Chan, Kasper Green Larsen, and Mihai Pǎtraşcu. Orthogonal range searching on the RAM, revisited. In *SoCG*, pages 1–10, 2011.

[22] Timothy M. Chan and Yakov Nekrich. Better data structures for colored orthogonal range reporting. In *SODA*, pages 627–636, 2020.

[23] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[24] Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.

[25] Bernard Chazelle and Herbert Edelsbrunner. Linear Space Data Structures for Two Types of Range Search. *Discrete & Computational Geometry*, 2:113–126, 1987.

[26] Francisco Claude, J. Ian Munro, and Patrick K. Nicholson. Range Queries over Untangled Chains. In *SPIRE*, pages 82–93, 2010.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[28] Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. Comparing Data Streams Using Hamming Norms (How to Zero In). *IEEE Trans. Knowl. Data Eng.*, 15(3):529–540, 2003.

[29] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[30] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[31] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS Succinct Tree Representation. In *WEA*, pages 134–145, 2006.

[32] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On Cartesian Trees and Range Minimum Queries. *Algorithmica*, 68(3):610–625, 2014.

[33] Camil Demetrescu, Andrew Goldberg, and David Johnson. DIMACS'09. `http://users.diag.uniroma1.it/challenge9/download.shtml`. Accessed: 2019-07-27.

[34] Postgis developer community. Postgis spatial and geographic objects for postgresql. `http://postgis.net/`. Accessed: 2019-07-27.

[35] GDAL devloper community. Euclidean MST Example. `https://doc.cgal.org/latest/BGL/BGL_triangulation_2_2emst_8cpp-example.html`. Accessed: 2019-07-27.

[36] Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-space data structures for range frequency queries on arrays and trees. *Algorithmica*, 74(1), 2016.

[37] Hicham El-Zein, J. Ian Munro, and Yakov Nekrich. Succinct Color Searching in One Dimension. In *ISAAC*, pages 30:1–30:11, 2017.

[38] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *CPM*, pages 130–140, 1996.

[39] Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.

[40] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *PODS*, pages 181–190, 2008.

[41] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[42] Michael L. Fredman and Dan E. Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[43] José Fuentes and Gonzalo Navarro. Experimental datasets graphs, trees, parentheses. `https://users.dcc.uchile.cl/~jfuentess/datasets/trees.php`. Accessed: 2018-03-14.

[44] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and Related Techniques for Geometry Problems. In *STOC*, pages 135–143, 1984.

[45] Travis Gagie, Meng He, and Gonzalo Navarro. Tree Path Majority Data Structures. In *ISAAC*, volume 123, pages 68:1–68:12, 2018.

[46] Travis Gagie and Juha Kärkkäinen. Counting Colours in Compressed Strings. In *CPM*, pages 197–207, 2011.

[47] Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE*, pages 1–6, 2009.

[48] Arnab Ganguly, J. Ian Munro, Yakov Nekrich, Rahul Shah, and Sharma V. Thankachan. Categorical Range Reporting with Frequencies. In *ICDT*, pages 9:1–9:19, 2019.

[49] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation, 2019. URL: `https://gdal.org`.

[50] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[51] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, 2(4):510–534, 2006.

[52] Geofabrik GmbH. OSM Europe Maps. `http://download.geofabrik.de/europe.html`. Accessed: 2019-07-27.

[53] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *SEA*, pages 326–337, 2014.

[54] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *SIAM J. Comput.*, 9(4):551–570, 1961.

[55] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994.

[56] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *STACS*, pages 517–528, 2009.

[57] Roberto Grossi and Søren Vind. Colored range searching in linear space. In *SWAT*, volume 8503, pages 229–240, 2014.

[58] PTV Group. KIT roadgraphs. `https://i11www.iti.kit.edu/information/roadgraphs`. Accessed: 07/12/2018.

[59] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *J. Algorithms*, 19(2):282–317, 1995.

[60] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[61] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

[62] Torben Hagerup. Parallel preprocessing for path queries without concurrent reading. *Inf. Comput.*, 158(1):18–28, 2000.

[63] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[64] Meng He and Serikzhan Kazi. Path and Ancestor Queries over Trees with Multidimensional Weight Vectors. In *ISAAC*, pages 45:1–45:17, 2019.

[65] Meng He and Serikzhan Kazi. Path Query Data Structures in Practice. In *SEA*, volume 160, pages 27:1–27:16, 2020.

[66] Meng He and Serikzhan Kazi. Data Structures for Categorical Path Counting Queries. submitted, under review.

[67] Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct ordinal trees based on tree covering. *ACM Trans. Algorithms*, 8(4):42:1–42:32, 2012.

[68] Meng He, J. Ian Munro, and Gelin Zhou. Path queries in weighted trees. In *ISAAC*, pages 140–149, 2011.

[69] Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, 2014.

[70] Meng He, J. Ian Munro, and Gelin Zhou. Data structures for path queries. *ACM Trans. Algorithms*, 12(4):53:1–53:32, 2016.

[71] David A. Hutchinson, Anil Maheshwari, and Norbert Zeh. An external memory data structure for shortest path queries. *Discret. Appl. Math.*, 126(1):55–82, 2003.

[72] Kazuki Ishiyama and Kunihiko Sadakane. A succinct data structure for multidimensional orthogonal range searching. In *DCC*, pages 270–279, 2017.

[73] Guy Jacobson. Space-efficient Static Trees and Graphs. In *FOCS*, pages 549–554, 1989.

[74] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *ISAAC*, pages 558–568, 2004.

[75] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

[76] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient Colored Orthogonal Range Counting. *SIAM J. Comput.*, 38(3):982–1011, 2008.

[77] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[78] Ying Kit Lai, Chung Keung Poon, and Benyun Shi. Approximate colored range and point enclosure queries. *J. Discrete Algorithms*, 6(3):420–432, 2008.

[79] Kasper Green Larsen and Freek van Walderveen. Near-optimal range reporting structures for categorical data. In *SODA*, pages 265–276, 2013.

[80] Tom Leighton. Methods for message routing in parallel machines. In *STOC*, pages 77–96, 1992.

[81] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Trans. Algorithms*, 4(3):28:1–28:13, 2008.

[82] Christos Makris and Athanasios K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Inf. Process. Lett.*, 66(6):277–283, 1998.

[83] Christos Makris and Konstantinos Tsakalidis. An improved algorithm for static 3D dominance reporting in the pointer machine. In *ISAAC*, pages 568–577, 2012.

[84] Carsten Möller. OSMPO converter and routing engine. `http://osm2po.de`. Accessed: 2019-07-27.

[85] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012.

[86] J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[87] David R. Musser. Introspective Sorting and Selection Algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

[88] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[89] NASA. MOLA Mars Orbiter Laser Altimeter data from NASA Mars Global Surveyor. `https://planetarymaps.usgs.gov/mosaic/Mars_MGS_MOLA_DEM_mosaic_global_463m.tif`. Accessed: 10/01/2019.

[90] Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[91] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[92] Gonzalo Navarro and Alberto Ordóñez Pereira. Faster Compressed Suffix Trees for Repetitive Collections. *ACM Journal of Experimental Algorithmics*, 21(1):1.8:1–1.8:38, 2016.

[93] Yakov Nekrich. A data structure for multi-dim. range reporting. In *SoCG*, pages 344–353, 2007.

[94] Yakov Nekrich. Efficient range searching for categorical and plain data. *ACM Trans. Database Syst.*, 39(1):9:1–9:21, 2014.

[95] Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *SWAT*, pages 271–282, 2012.

[96] NGA and NASA. SRTM Shuttle Radar Topography Mission. `http://srtm.csi.cgiar.org/srtmdata/`. Accessed: 10/01/2019.

[97] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[98] Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct representations of weighted trees supporting path queries. *J. Discrete Algorithms*, 17:103–108, 2012.

[99] R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. In *IEEE Transactions on Systems, Man and Cybernetics*, pages 17–30, 1989.

[100] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

[101] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), 2007.

[102] A. Rényi and G. Szekeres. On the height of trees. *Journal of the Australian Mathematical Society*, 7(4):497–507, 1967.

[103] Teodor Sigaev and Oleg Bartunov. ltree module for PostreSQL RDBMS. `https://www.postgresql.org/docs/current/ltree.html`. Accessed: 10/01/2020.

[104] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[105] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019. URL: `https://doc.cgal.org/4.14/Manual/packages.html`.

[106] Dekel Tsur. Succinct representation of labeled trees. *Theor. Comput. Sci.*, 562:320–329, 2015.

[107] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[108] Dan E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[109] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012.

[110] Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016.