Space Efficient Data Structures in the Word-RAM and Bitprobe Models

by

Patrick Kevin Nicholson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

This thesis studies data structures in the word-RAM and bitprobe models, with an emphasis on space efficiency. In the word-RAM model of computation the space cost of a data structure is measured in terms of the number of w-bit words stored in memory, and the cost of answering a query is measured in terms of the number of read, write, and arithmetic operations that must be performed. In the bitprobe model, like the word-RAM model, the space cost is measured in terms of the number of bits stored in memory, but the query cost is measured solely in terms of the number of bit accesses, or *probes*, that are performed.

First, we examine the problem of succinctly representing a partially ordered set, or *poset*, in the word-RAM model with word size $\Theta(\lg n)$ bits. A succinct representation of a combinatorial object is one that occupies space matching the information theoretic lower bound to within lower order terms. We show how to represent a poset on n vertices using a data structure that occupies $n^2/4 + o(n^2)$ bits, and can answer precedence (i.e., less-than) queries in constant time. Since the transitive closure of a directed acyclic graph is a poset, this implies that we can support reachability queries on an arbitrary directed graph in the same space bound. As far as we are aware, this is the first representation of an arbitrary directed graph that supports reachability queries in constant time, and stores less than $\binom{n}{2}$ bits. We also consider several additional query operations.

Second, we examine the problem of supporting range queries on strings of n characters (or, equivalently, arrays of n elements) in the word-RAM model with word size $\Theta(\lg n)$ bits. We focus on the specific problem of answering *range majority queries*: i.e., given a range, report the character that is the majority among those in the range, if one exists. We show that these queries can be supported in constant time using a linear space (in words) data structure. We generalize this result in several directions, considering various frequency thresholds, geometric variants of the problem, and dynamism. These results are in stark contrast to recent work on the similar *range mode problem*, in which the query operation asks for the mode (i.e., most frequent) character in a given range. The current best data structures for the range mode problem take $\tilde{O}(\sqrt{n})$ time per query for linear space data structures.

Third, we examine the deterministic membership (or dictionary) problem in the bitprobe model. This problem asks us to store a set of n elements drawn from a universe $[1, u]$ such that membership queries can be always answered in t bit probes. We present several new fully explicit results for this problem, in particular for the case when $n = 2$, answering an open problem posed by Radhakrishnan, Shah, and Shannigrahi [ESA 2010]. We also present a general strategy for the membership problem that can be used to solve many related fundamental problems, such as rank, counting, and emptiness queries.

Finally, we conclude with a list of open problems and avenues for future work.

Acknowledgements

First and foremost, I would like to thank my supervisor Ian Munro for his guidance and support. It has been an honour to work with and learn from him during my graduate studies. He has provided me with an excellent working environment, and as a result I have enjoyed my time at Waterloo immensely.

My thesis examination committee (Timothy Chan, Joseph Cheriyan, Danny Krizanc and Alex López-Ortiz) provided helpful feedback on the initial draft of my thesis, and for that I thank them. I would also like to give a big thank you to our department's administrative coordinator, Wendy Rush, for all the help she has provided me during my graduate studies; especially when I was in the process of submitting my thesis.

Early on during my graduate studies, Meng He taught me a great deal about succinct data structures, and got me interested in many problems in this area, so I would like to thank him for doing that. Later on, during the final year of my studies, I had the good fortune to work with Venkatesh Raman and Moshe Lewenstein, who were both visiting Waterloo on sabbatical. I learned a lot from our collaboration during this time, and was exposed to many interesting topics that I greatly enjoyed researching.

I would also like to thank my other coauthors (not mentioned above): Diego Arroyuelo, Francisco Claude, Reza Dorrigiv, Stephane Durocher, Amr Elmasry, Bob Fraser, Travis Gagie, Srinivasa Rao, Alejandro Salinger, Diego Seco, Matthew Skala, and Norbert Zeh. I would like to give extra thanks to my office mates: Francisco Claude, Bob Fraser, Shahin Kamali, Alejandro Salinger, Diego Seco, Konstantinos Tsakalidis, and Gelin Zhou. A big thank you also goes to my family and friends who supported me during my graduate studies. Last, but certainly not least, I would like to thank my fiancée Niina for all the support she has provided me during my time in Waterloo, which was absolutely essential during my time here.

Table of Contents

List of Figures **viii**

1 Introduction **1**

1.1 Motivation 1

1.2 Models of Computation 2

1.3 Succinct Data Structures 4

1.4 Outline of Thesis and Summary of Contributions 5

1.5 Preliminaries 7

1.5.1 Fundamental Operations on Strings and Entropy 7

1.5.2 Prefix-Free Codes 10

1.5.3 Bit Strings 10

1.5.4 Strings with Larger Alphabets 12

1.5.5 Other Combinatorial Objects 14

1.5.6 A Brief Note About Randomization 14

2 Succinct Posets **16**

2.1 Introduction 16

2.2 Definitions 17

2.3 Previous work 19

2.4 Our Contributions 23

2.5 The Data Structure 27

2.5.1 Flattening a Poset 27
2.5.2 Compressing Flat Posets 32
2.6 Extension to Transitive Reductions and Transitive Relations 38
2.7 Issues Relating to Construction 40
2.8 Additional Operations 43
2.8.1 Another Representation That Yields More Operations 44
2.8.2 Meet, Join, and Boolean Matrix Multiplication 48
2.9 Simultaneous Representation 50
2.10 Summary and Concluding Remarks 51

3 Range Majority Queries **52**
3.1 Introduction 52
3.2 Previous Work 53
3.3 Our Contributions 59
3.4 Subsequent (and Concurrent) Work 60
3.5 Static Range Majority Data Structure 63
3.5.1 Quadruple Decomposition 64
3.5.2 Candidates 65
3.5.3 Data Structures for Counting 65
3.6 Generalization to Static Range α-Majority Queries 67
3.6.1 Handling Large Alphabets 67
3.7 Parameterized Query and Trade-offs 68
3.8 Applications to Static Geometric Problems 73
3.8.1 Static Range Majority for Coloured Points in One Dimension 73
3.8.2 Static Range Majority in Higher Dimensions 74
3.9 Approach for the Dynamic Geometric Problem 75
3.9.1 Lower Bound 76
3.9.2 Assumptions about Colours 77

3.9.3 Dynamic Tree Structure 77
3.9.4 Supporting Queries 78
3.9.5 Supporting Updates 82
3.9.6 Speedup for Integer Coordinates 84
3.9.7 Dynamic String 87
3.9.8 Higher Dimensions 88
3.10 Summary and Concluding Remarks 88

4 Explicit Bitprobe Data Structures **90**
4.1 Introduction 90
4.2 Previous Results 93
4.2.1 Deterministic Schemes for Membership 93
4.2.2 Rank, Range Counting, and Emptiness 99
4.3 Our Contributions 100
4.3.1 Membership 100
4.3.2 Rank, Range Counting, and Emptiness 101
4.4 Membership Queries 101
4.4.1 Fully Explicit Adaptive Schemes for Two Elements 101
4.4.2 Limitations of the Blocking Scheme Approach for $n \geq 3$ 107
4.4.3 Fully Explicit Adaptive Schemes for $n \geq 3$ 111
4.5 Rank Queries 117
4.6 Range Counting and Emptiness Queries 119
4.7 Summary and Concluding Remarks 121

5 Conclusions and Open Problems **122**
5.1 Succinct Posets 122
5.2 Range Majority Queries 123
5.3 Explicit Bitprobe Data Structures 124

References **126**

List of Figures

2.1 Various diagrams of the same poset. 18

2.2 An antichain decomposition of our example poset. 29

2.3 Illustration of the key lemma for compressing the poset. 33

2.4 A graph of the function $F(\varepsilon_0) = 1.42\varepsilon_0 + 0.42\varepsilon_0 \ln(1/\varepsilon_0) + 1$. 40

2.5 The reinterpretation of $\texttt{EdgeArray}_\ell$. 44

3.1 Blocks, quadruples, and query ranges. 64

4.1 Decision trees for adaptive and non-adaptive schemes. 91

4.2 Dividing the bits into blocks. 102

4.3 The various cases for our $(2, u, 7u^{2/5}, 3)$-scheme.) 103

4.4 Various types of (j, k)-decompositions. 113

Chapter 1

Introduction

1.1 Motivation

In many settings, modern software must deal with high volumes of data in an efficient manner. For example, when a user types a query into an Internet search engine, the software running on the server is able to quickly find web pages that are relevant to the query, and return them to the user. The software must search through many terabytes of data in order to accomplish this goal. How does the software organize the data in order to do this efficiently? The answer is by using efficient *data structures*.

This thesis examines several abstract data structure problems, with an emphasis on space efficiency. That is, we wish to design data structures that occupy as little, or close to as little, memory as possible. We also want to support *query operations* on the data as efficiently as possible. Ideally, we would like the query operations to take some fixed amount of time, regardless of the size of the data set.

Before giving a high level description of the contributions of this thesis, we require some definitions. In particular, we need to define the notion of computation (Section 1.2), and also discuss issues of what it means for a data structure to be "space efficient" (Section 1.3). We then return in Section 1.4 to the question of what this thesis is about, before diving into the preliminary results that we use throughout the thesis in Section 1.5.

1.2 Models of Computation

Since the architectures of modern computing devices change over time, it behooves researchers to allow some of the hardware specific details of a computer to be abstracted, so that the core data structure problem to be addressed can be attacked more readily. As such, there are several theoretical models of how a computer operates, each with its own interesting properties and drawbacks. We provide a non-exhaustive list briefly summarizing some existing models of computation. We do this in order to give a flavour for the various models, and to also make the point that the ones studied in this thesis are not the only that exist. For each model, we include a reference—for the interested reader who wants more details—to the paper or book that we used as the basis for the summary.

- **Comparison Model** [106]: In the comparison model, a data structure query can be represented as a rooted tree. The tree need not be binary (e.g., we might have three-way comparisons), but we assume it is to simplify our definition. The root node stores the binary comparison that is made first, given the query element, and the right and left children represent how the query proceeds, given that the comparison returns true or false, respectively. The query algorithm eventually ends up in a leaf of the tree, where the answer to the query is stored. The cost of the query is equal to the number of comparisons required to reach the leaf node.

- **Pointer Machine Model** [147, 16]: In a pointer machine[1], a data structure is represented as a directed graph with fixed out-degree. The interpretation of the structure of the graph is dependent on the underlying problem we are trying to model. For example, the nodes of the graph may represent the data elements or records that we wish to store and query. A query specifies a set of input nodes in the graph, and asks us to return a set of output nodes: i.e., the correct answer to the query. This is done by traversing the edges (or pointers) of the graph, and the time complexity of the query can be measured by counting the number of pointers that were traversed. Similarly, we can measure the space cost of the data structure by counting the number of nodes in the graph.

[1]We note that the second reference [16] recommends the use of the terminology *pointer algorithm* instead of pointer machine to avoid confusion. However, pointer machine appears to be the standard terminology at this point in time.

- **Word-RAM Model** [88]: One of the main drawbacks of pointer machine is that it does not capture the notion of *random access memory* (RAM) that is supported in modern computer hardware. In the word-RAM model with word size w, we assume that our computer can perform a set of unit cost operations on w-bit numbers, and that our memory consists of an infinite number of cells numbered with indices $\{1, 2, \dots\}$. The content of each cell is an integer in the range $\{0, \dots, 2^w - 1\}$, called a *word*. For two words x_1 and x_2, we can compute $x_1 \oplus x_2 \mod 2^w$, where $\oplus$ can be any one of the following operations: addition (+), subtraction (−), multiplication (×), integer division (/), bit-wise Boolean arithmetic, and left and right bit-wise shifting. The time complexity of a data structure query counts the number of these operations used, plus the number of read and write operations to memory. The space complexity is the maximum index of a word that is read or written during any query to the data structure. There are many variants of the word-RAM model that consider restricted operations sets: e.g., they do not permit unit cost multiplication or integer division [88]. However, we note that the operations we permit are somewhat standard in the literature, and also realistic in the sense that they are supported efficiently on most modern computing architectures.

- **Cell-Probe Model** [159]: The cell-probe model is more general than the word-RAM model, and is usually used in the context of proving data structure lower bounds. It considers memory to be an infinite number of w-bit cells, and is only concerned with counting the number of cells of the data structure that need to be read (*probed*), or modified, in order to answer a query. The amount of intermediate computation allowed by the query in order to determine where the next probe should occur is unbounded in the cell-probe model. This means that a lower bound proved in the cell-probe model holds in the word-RAM model, *regardless* of the operations permitted!

- **Bitprobe Model** [110, 30]: The bitprobe model is a special case of the cell probe model that occurs when the cell size, $w = 1$; i.e., the cells contain a single bit. Interestingly, it is older than the cell-probe model, and was formalized by Minski and Papert in their 1969 book “Perceptrons” [110]. A *perceptron* is a model of a computer that answers decision problems based on whether the weighted sum of the truth values of a set of predicates exceeds a given threshold. Since perceptrons were originally examined in a slightly different context than we consider here, the bitprobe model we refer to—which is equivalent to the cell probe model with one bit cells—is formally defined by Buhrman et al. [30]. The concern in this model is studying the trade-off between the number of bits that must be read from the data structure in order to answer a query, and the overall number of bits occupied by the data structure.

1.3 Succinct Data Structures

Since the word-RAM and cell-probe models deal with individual bits of information, a natural question that arises is, “How many bits are required to store a data structure representing an arbitrary combinatorial object of type X, such that we can answer queries of type Y about X efficiently?” As a starting point, it is instructive to first ask the easier question, “How many bits are required to store an arbitrary combinatorial object of type X, and be able to distinguish it from other objects of that type?” If we can answer this question, then it provides a lower bound on the space that *any* data structure representing X must occupy.

Information Theory Lower Bound: Consider the case where X is an arbitrary rooted binary tree with n internal nodes, each of which have both left and right children. If we use N denote the number of distinct trees of this type, it is known that $\lg N = \lg\left(\binom{2n}{n}/(n+1)\right) \approx \lg(4^n/(n^{3/2}\sqrt{\pi})) = 2n - o(n)$: N is the n-th Catalan number.[2] Therefore, we need $2n - o(n)$ bits just to uniquely identify the tree we wish to store. This is known as the *information theory lower bound*, and it holds regardless of what operations we support. Note that this is significantly fewer bits than explicitly storing a pointer for the left and right children of each node. Such a pointer-based representation would require $\Theta(n \lg n)$ bits.

An area of research that has been active since the late eighties is that of *succinct data structures* [98]. A succinct data structure is a data structure that represents a combinatorial object using space matching the information theory lower bound, *to within lower order terms*, while also supporting efficient query operations. Returning to the example, there are existing succinct data structures for binary trees that occupy $2n + o(n)$ bits of space, and support a wide variety of efficient navigation operations [54]. We note that although the area of succinct data structures started with Jacobson [98], many of the ideas used in the area are much older. In fact, supporting efficient query operations is the only major difference between the area of succinct data structures and “the oldest mathematical subject” [160], *combinatorial enumeration*.

[2]Throughout this thesis we use $\lg N$ to denote $\log_2 N$ unless otherwise specified.

Perhaps the first use of the word "succinct" in the context of space efficient data structures was by Turán in his 1984 paper "On the succinct representations of graphs" [152], which showed how to represent unlabelled planar graphs on n vertices using $12n$ bits. By our definition of succinctness above, this is *not truly* a succinct data structure, as the space it occupies is greater than the information theory lower bound by a constant factor [76], rather than just a lower order term. Furthermore, the only operations considered by Turán were *encode* and *decode*: the first converts a standard adjacency list representation of a planar graph into the succinct data structure, and the second converts the succinct data structure back into the standard adjacency lists representation. We use the terminology *succinct representation* to describe a succinct data structure that supports only the operations encode and decode, both in polynomial time.

Succinct data structures and representations of *many* types of combinatorial objects have been studied. We return in Section 1.5 to survey the area and previous results.

1.4 Outline of Thesis and Summary of Contributions

We are now ready to outline the main results of this thesis. The common theme is the design of space efficient data structures that provide constant time query operations. That is, operations in which the time complexity remains fixed regardless of the size of the input data. We examine three data structure problems, that all involve notions of succinctness in some way, though the techniques used vary widely between the chapters. In light of this, we will defer detailed technical discussion of related work and techniques to each of the individual chapters. However, many preliminary data structures and concepts are used throughout the thesis, and we define these in the next section.

In Chapter 2 we design a succinct data structure for representing a partially ordered set, or *poset*. Posets are fundamental combinatorial objects that appear in many different contexts in computer science. Unlike previous work [87, 144, 71, 85, 139, 55], we consider the problem of representing and performing query operations on *arbitrary* posets. The main query operation that we support, given two elements, is to report whether one precedes the other. This is equivalent to succinctly representing the transitive closure graph of the poset, and the same method can also be used to succinctly represent the transitive reduction graph. For an n element poset, the data structure occupies $n^2/4 + o(n^2)$ bits, in the worst case, and can support precedence queries (i.e., less than queries) in $O(1)$ time, in the word-RAM model with word size $w = \Theta(\lg n)$

bits. Contrast this with the trivial data structure for representing an arbitrary (not necessarily transitive) directed acyclic graph, which is computed by topologically sorting the vertices and storing an upper triangular bit matrix to represent the edges: this uses $n^2/2 + \Theta(n \lg n)$ bits. Thus, our data structure occupies roughly half as many bits for the case of a poset: i.e., when the directed acyclic graph is a transitive closure or reduction. To compress the poset, we describe an interesting compression technique that borrows ideas from the area of extremal graph theory, and also makes use of ideas from the area of succinct data structures. We further consider issues of how to construct the data structure. Finally, we consider several additional query operations and show that they can be supported in constant time, or give evidence suggesting why they are difficult to support efficiently.

In Chapter 3 we focus on supporting *range queries* on strings (or, equivalently, arrays), the various types of which have received a great deal of attention in the last few years [18, 23, 108, 63, 130, 89, 131, 28, 68, 27, 62, 101]. In particular, we design data structures for the *range majority problem*. The majority of a string of length n is the character that appears more than $n/2$ times, if such a character exists. Given a string $\mathfrak{A}$ of length n, the range majority problem is to preprocess the string, such that later, given a query range $[i, j]$ where $1 \leq i \leq j \leq n$, we can efficiently return the majority of the substring $\mathfrak{A}[i, j]$ (if it exists).[3] The idea to study this problem came up because the closely related *range mode problem* [130, 131, 81] appears to be difficult: i.e., compute the *mode*—the character that appears at least as often as any other character—of an arbitrary range in a string. In fact, for linear space data structures, it is not known how to achieve $O(n^{1/2-\varepsilon})$ time queries, for any constant $\varepsilon > 0$. In contrast, we describe a linear space data structure in the word-RAM model with word size $\Theta(\lg n)$ bits, that answers range majority queries in *constant* time. The main technique is a surprisingly simple tree decomposition, in conjunction with standard ideas from the area of succinct data structures. We further generalize this problem by defining range α-majority queries, in which we wish to return all the characters in the substring $\mathfrak{A}[i, j]$ with frequency greater than $\alpha(j - i + 1)$. We show that range α-majority queries can be answered in $O(1/\alpha)$ time using $O(n(\lg(1/\alpha) + 1))$ words of space, for any fixed $\alpha \in (0, 1)$. Other trade-offs, compression, and dynamism are also discussed.

In Chapter 4 we consider the membership problem in the bitprobe model. Previous work by Buhrman, Miltersen, Radhakrishnan and Venkatesh [30] studied the membership problem in the bitprobe model, presenting both randomized and deterministic schemes for storing a bit vector

[3]We formally define the notation just used in Section 1.5.

of length u containing n one bits, such that membership queries can be answered in t bit probes. They showed a separation between the power of *adaptive probes*—where the query algorithm can decide where to probe next based on the bits read during prior probes—and non-adaptive probes. We focus on deterministic adaptive schemes that, unlike many previous *non-explicit* results[4], are *fully explicit*: i.e., the data structure can be efficiently constructed *and* the query algorithm can compute which bits to probe in time polynomial in $\lg u$. Focusing on the case when $n = 2$, we describe the first non-trivial fully explicit schemes that use $t \geq 3$ adaptive probes. These answer a problem of Radhakrishnan, Shah, and Shannigrahi [134], who asked for a fully explicit scheme matching the $O(u^{2/5})$ space bound of their non-explicit scheme for three adaptive probes, and even improve upon their non-explicit scheme when $t > 3$. We also describe a recursive scheme for $n \geq 3$ that not only improves upon previous fully explicit schemes for a wide range of input parameters, but can be used to solve a number of related problems that generalize set membership, such as the rank, range counting and emptiness problems.

Finally, in Chapter 5 we conclude by presenting a list of open problems and avenues for future related research.

1.5 Preliminaries

In this section we state some preliminary lemmas that we will use throughout the remainder of this thesis.

1.5.1 Fundamental Operations on Strings and Entropy

We now define some operations on strings that are used throughout the thesis:

Fundamental Operations on Strings: Given a string $\mathfrak{S}$ of length U, we use the notation $\mathfrak{S}[i]$ to denote the i-th character in the string, and $\mathfrak{S}[i_1, i_2]$ to denote the substring of $\mathfrak{S}$ beginning at position i_1, and ending at position i_2, inclusively, where $1 \leq i_1 \leq i_2 \leq U$. We use the set cardinality notation $|\mathfrak{S}|$ to refer to the length of $\mathfrak{S}$; i.e., $|\mathfrak{S}| = U$. The characters of $\mathfrak{S}$ are said to be *drawn from an alphabet* $\Sigma = \{1, \ldots, \sigma\} = [1, \sigma]$. That is, each character $\mathfrak{S}[i] \in [1, \sigma]$ for all $1 \leq i \leq U$. We define the following fundamental operations:

[4]We give formal definitions of all the emphasized terms in Chapter 4.

- $\texttt{access}(\mathfrak{S}, i)$: Return the character $\mathfrak{S}[i]$, for any $i \in [1, U]$.
- $\texttt{rank}_j(\mathfrak{S}, i)$: Return the number of occurrences of the character j in the substring $\mathfrak{S}[1, i]$, for any $i \in [1, U]$ and $j \in [1, \sigma]$.
- $\texttt{select}_j(\mathfrak{S}, i)$: Return the position of the i-th occurrence of character j in the string $\mathfrak{S}$, for any $i \in [1, \texttt{rank}_j(\mathfrak{S}, U)]$ and $j \in [1, \sigma]$.

These operations are fundamental in many succinct data structures, and as such are widely used in this thesis. For example, consider the string $\mathfrak{S} = \text{ABCABABCACBC}$. Based on the previous definitions, $\texttt{access}(\mathfrak{S}, 7) = \text{B}$, $\texttt{rank}_\text{A}(\mathfrak{S}, 6) = 3$, and $\texttt{select}_\text{C}(\mathfrak{S}, 3) = 10$.

Another concept that is important when discussing space efficiency is that of the entropy of a string.

Zeroth Order Entropy: The *zeroth order entropy* of a string $\mathfrak{S}$, denoted $\mathsf{H}_0(\mathfrak{S})$, is a measure of the compressibility of a string. Assume the characters that comprise $\mathfrak{S}$ come from the alphabet $\Sigma = [1, \sigma]$, and are drawn according to some fixed probability distribution. Let p_i be the probability of drawing character i according to this distribution, for each $i \in \Sigma$. The zeroth order entropy of $\mathfrak{S}$ is defined as follows (interpreting $\lg 0 = 0$):

$$\mathsf{H}_0(\mathfrak{S}) = \sum_{i=1}^{\sigma} p_i \lg(1/p_i) \ . \tag{1.5.1}$$

If all the characters in the string $\mathfrak{S}$ are known, and the total length is U, we define the *zeroth order empirical entropy* of the string based on the empirical probability distribution of its characters; i.e., the distribution we get by computing the relative frequencies of the characters ($\{p_i = \texttt{rank}_i(\mathfrak{S}, U)/U\}$). As an example, consider the string

$$\mathfrak{S} = \text{AABAAAAAAAACAAADAAAAAAAEAAAFAAAAAAAGAAHA} \ .$$

Since the size of the alphabet, $\sigma = 8$, we might expect to spend three bits per character to write down $\mathfrak{S}$, if we did not know any additional information about the empirical probability distribution with which each of its characters occur. However, knowing *only* these empirical probabilities, the zeroth order empirical entropy provides a lower bound on how well $\mathfrak{S}$ can be compressed. In particular it indicates that we need at least

$$\frac{33}{40}\lg\left(\frac{40}{33}\right)+7\left(\frac{1}{40}\lg(40)\right)\approx 1.1604 \text{ bits per character on average.}$$

For bit strings, the following bound can be used to relate zeroth order empirical entropy to binomial coefficients:

Lemma 1.5.1 (Theorem 9.2,[113]). *Let $\mathfrak{S}$ be a bit string of length U containing N one bits. Then,*

$$\frac{2^{U\mathsf{H}_0(\mathfrak{S})}}{U+1}\leq\binom{U}{N}\leq 2^{U\mathsf{H}_0(\mathfrak{S})} \tag{1.5.2}$$

Finally, though we do not use it in any technical way, we also define higher order entropy, as we do mention it:

***k*-th Order Entropy:** The idea of zeroth order entropy can be naturally generalized to higher order entropy, where the probability of drawing a certain character is not independent, but rather depends on the previous k characters that were drawn. This concept is especially useful in the empirical setting, where the string is fixed, and the conditional probabilities can be estimated, as in the zeroth order case, by examining the string.

Let $\mathfrak{S}$ be a string of length U, drawn from an alphabet $\Sigma=[1,\sigma]$. Given a string $\mathfrak{S}'\in\Sigma^k$, where Σ^k denotes the set of all strings of length k drawn from the alphabet Σ, we use the notation $\textsc{NextChar}(\mathfrak{S},\mathfrak{S}')$ denote the concatenation of the single characters following the occurrences of $\mathfrak{S}'$ in $\mathfrak{S}$. For example, if $\mathfrak{S}=$ ABABBAABA and $\mathfrak{S}'=$ AB, then $\textsc{NextChar}(\mathfrak{S},\mathfrak{S}')=$ ABA. The *k-th order empirical entropy* of $\mathfrak{S}$ is defined as follows:

$$\mathsf{H}_k(\mathfrak{S})=\frac{1}{U}\sum_{\mathfrak{S}'\in\Sigma^k}|\textsc{NextChar}(\mathfrak{S},\mathfrak{S}')|\mathsf{H}_0(\textsc{NextChar}(\mathfrak{S},\mathfrak{S}'))\ . \tag{1.5.3}$$

Based on the above definitions, it should be clear that $\mathsf{H}_k(\mathfrak{S})\leq\mathsf{H}_0(\mathfrak{S})$ for any $k\geq 1$. If each character in the string $\mathfrak{S}$ is drawn from a fixed probability distribution where each draw is independent, then the zeroth order empirical entropy is a lower bound for compression. However, for certain kinds of real-world data, such as English text, compressors that achieve close to k-th order empirical entropy bounds perform far better than ones that achieve close to the zeroth order, due to the fact that many real-world texts are highly repetitive. For more information on this phenomena, see the experimental study of Ferragina et al. [59].

1.5.2 Prefix-Free Codes

When we say *code* we are referring to a bit string that is used to represent a character drawn from some non-binary alphabet. A set of codes is *prefix-free* if no code in the set is a prefix of any other. For example, the set of codes $\{0, 10, 110\}$ is prefix-free, but the set $\{0, 00\}$ is not. This prompts the following definitions; note that we use the operator $X \cdot Y$ to denote the concatenation of X and Y.

Lemma 1.5.2 (Huffman Code [96] (see also Section 1.2. of [65]))**.** *Let $\mathfrak{S}$ be a string of length U, with characters drawn from the alphabet $\Sigma = [1, \sigma]$. There is a prefix-free encoding of each character $\mathtt{HUFF}(i)$, for $i \in \Sigma$ with the property that the string $\mathtt{HUFF}(\mathfrak{S}) = \mathtt{HUFF}(\mathfrak{S}[1]) \cdot \mathtt{HUFF}(\mathfrak{S}[2]) \cdot \ldots \cdot \mathtt{HUFF}(\mathfrak{S}[U])$ occupies no more than $U(\mathsf{H}_0(\mathfrak{S}) + 1)$ bits.*

Lemma 1.5.3 (Elias γ-Code [51])**.** *Let $\mathfrak{S}$ be a string of length U, with characters drawn from the alphabet $\Sigma = [1, \sigma]$. Further suppose that the characters are sorted in non-decreasing order of frequency, i.e., character i appears at least as frequently in $\mathfrak{S}$ as $i + 1$, and that, for simplicity, σ is a power of two. There is a prefix-free encoding of each character $\gamma(i)$, for $i \in \Sigma$, with the following properties:*

1. *the string $\gamma(\mathfrak{S}) = \gamma(\mathfrak{S}[1]) \cdot \gamma(\mathfrak{S}[2]) \cdot \cdots \cdot \gamma(\mathfrak{S}[U])$, occupies $O(U(\mathsf{H}_0(\mathfrak{S}) + 1))$ bits of space;*
2. *the length of the codes for characters $[2^i, 2^{i+1} - 1]$ is $\Theta(i + 1)$ bits, for $0 \leq i \leq \lg \sigma - 1$.*

1.5.3 Bit Strings

The fundamental operations on strings are heavily studied in the context of succinct data structures, as many combinatorial objects can be decomposed and represented by sets of strings. Consider a bit string $\mathfrak{S}$ of U bits. Jacobson [98] showed that `rank` and `select` can be supported on $\mathfrak{S}$ in $O(\lg U)$ bit accesses, by representing $\mathfrak{S}$ using a data structure that occupies $U + o(U)$ bits. Later, Clark and Munro [36] showed that, on a word-RAM with word size $\Theta(\lg U)$ bits, these operations can be supported in $O(1)$ time, while still using $U + o(U)$ bits. One natural question that arises is whether it is possible to use less space if the bit string is sparse; e.g., if $\mathsf{H}_0(\mathfrak{S})$ is much less than 1. Brodnik and Munro [29], Pagh [126], and Pătraşcu [128] showed that it is possible to represent a bit string $\mathfrak{S}$ of length U containing N one bits using space

$$\lg\binom{U}{N} + o\left(\lg\binom{U}{N}\right) \leq U\mathsf{H}_0(\mathfrak{S}) + o(U\mathsf{H}_0(\mathfrak{S})) \text{ bits}, \tag{1.5.4}$$

while supporting `access` in $O(1)$ time on a word-RAM with word size $\Theta(\lg U)$ bits. Note that the right hand side of Equation 1.5.4 follows from Lemma 1.5.1. These results improved the space bound of a well-known technique by Fredman, Komlós and Szemerédi [64], now frequently referred to as *FKS-hashing*, that occupies $O(N \lg U)$ bits of space and provides $O(1)$ time `access` on a word-RAM with word size $\Theta(\lg U)$ bits.

Access and Membership: At this point we make explicit the connection between supporting the `access` operation on a bit string, and the *static membership* or *dictionary* problem. In the static membership problem, we are given a set $\mathcal{E}$ of n elements drawn from a universe $[1, u]$, and asked to support queries of the form "Is $x \in \mathcal{E}$?" for any $x \in [1, u]$. Given such a set, the *characteristic* bit string $\mathfrak{S}$ representing $\mathcal{E}$ is a bit string of length u, such that $\mathfrak{S}[i] = 1$ iff $i \in \mathcal{E}$. Clearly, supporting `access` on this string is equivalent to solving the static membership problem.

Parameterized Rank and Select: Several papers have examined the difficulty of supporting `rank` and `select` [136, 78, 128, 83], in addition to `access`, while still achieving a space bound parameterized in terms of the number of 1 bits. We make use of the following result, noting that the lower order term in the space bound is not the strongest result of this type (c.f., [128]), but it is simple and sufficient for our needs:

Lemma 1.5.4 (Fully Indexable Dictionary [136]). *Let $\mathfrak{S}$ be a string of U bits, containing N one bits. In the word-RAM model with word size $\Theta(\lg U)$ bits, there is a data structure, called a* fully indexable dictionary*, of size $\lg\binom{U}{N} + O((U \lg\lg U)/\lg U) \leq U H_0(\mathfrak{S}) + O((U \lg\lg U)/U)$ bits that supports the operations* `access`$(\mathfrak{S}, i)$, `rank`$_j(\mathfrak{S}, i)$, *and* `select`$_j(\mathfrak{S}, i)$ *in $O(1)$ time, where $j \in [0, 1]$ and $i \in [1, U]$. The data structure can be constructed in $O(U)$ time*[5]*.*

Since the leading term in the space bound of the previous lemma is somewhat awkward—either as a binomial coefficient or in terms of entropy—we can rewrite it in the following way, derived using Stirling's approximation:

[5] The construction time is not explicitly stated by Raman, Raman, and Rao, but each of the (constant number of) indices that comprise this data structure can be constructed in $O(U)$ time, so the observation follows.

Lemma 1.5.5 (Section 4.6.4 of [90])**.** $\lg \binom{U}{N} \leq N \lg(eU/N) + O(1)$ *for integers* $U \geq N \geq 1$.

We note that if the full power of a fully indexable dictionary is not required, we can settle for a reduced operation set in exchange for a smaller lower-order space term (see [136]):

Lemma 1.5.6 (**Indexable Dictionary** [136])**.** *Let* $\mathfrak{S}$ *be a string of* U *bits, containing* N *one bits. In the word-RAM model with word size* $\Theta(\lg U)$ *bits, there is a data structure, called an* indexable dictionary*, of size* $\lg \binom{U}{N} + o(N) + O(\lg \lg u)$ *bits that supports the operations* $\mathtt{access}(\mathfrak{S}, i)$ *and* $\mathtt{select}_1(\mathfrak{S}, i)$ *in* $O(1)$ *time. The query* $\mathtt{rank}_1(\mathfrak{S}, i)$ *can be performed in* $O(1)$ *time for indices storing one bits, i.e., only for the case when* $\mathtt{access}(\mathfrak{S}, i) = 1$.

Finally, we note that several authors have examined the problem of determining how much *redundancy* is necessary to support these operations (see [69, 77, 80], and references therein). More recently, for constant time rank and select, the representation of Pătraşcu [128] has been shown to be optimal, to within the constant in the lower order term, by Pătraşcu and Viola [132].

1.5.4 Strings with Larger Alphabets

For strings with larger alphabet sizes, we make use of the following data structure, called a *wavelet tree*:

Lemma 1.5.7 (**Wavelet Tree** [82, 121])**.** *Let* $\mathfrak{S}$ *be a string of* U *characters drawn from the alphabet* $\Sigma = [1, \sigma]$*. In the word-RAM model with word size* $\Theta(\lg U)$ *bits, there is a data structure of size* $U(H_0(\mathfrak{S}) + 2)(1 + o(1)) + O(\sigma \lg U)$ *bits that supports the operations* $\mathtt{access}(\mathfrak{S}, i)$*,* $\mathtt{rank}_j(\mathfrak{S}, i)$*, and* $\mathtt{select}_j(\mathfrak{S}, i)$ *in* $O(\lg \sigma)$ *time, where* $j \in [1, \sigma]$ *and* $i \in [1, U]$*. The data structure can be constructed in time* $O(U \lg \sigma)$ *time.*

The idea behind the wavelet tree is to encode the bits of the characters in $\mathfrak{S}$ in a different manner than explicitly storing them as a string. The tree is structured as follows. Every node in the tree has two children, referred to the left and right children. Each node represents a *subrange* of the alphabet, $\mathcal{R} = [i, j] \subseteq [1, \sigma]$, where $1 \leq i \leq j \leq \sigma$. The root node represents the entire range of the alphabet. The left and right children of a node represent disjoint ranges $\mathcal{R}_\ell$ and $\mathcal{R}_r$ such that $\mathcal{R}_\ell \cup \mathcal{R}_r = \mathcal{R}$. The leaves of the tree represent a range consisting of a single character.

If the characters in $\mathfrak{S}$ are represented in the usual way, i.e., by using $\lg \sigma$ bits, then the tree will have height $\lg \sigma + 1$[6].

The root node stores a bit string $\mathfrak{B}$, where $\mathfrak{B}[i]$ represents the character $\mathfrak{S}[i]$, and is a 1 if $\mathfrak{S}[i] \in \mathcal{R}_r$ and a 0 otherwise. The left and right children store bit strings representing the subsequence of $\mathfrak{S}$ with characters in their respective ranges. If each of these bit strings are stored in a data structure supporting $\texttt{rank}_j$ and $\texttt{select}_j$ on the bit string (see the previous section) for $j \in [0, 1]$, then it is not difficult to support $\texttt{rank}_j$ and $\texttt{select}_j$ on $\mathfrak{S}$ in $O(\lg \sigma)$ time, for $j \in [1, \sigma]$. Note that these bit strings need not be explicitly in each node, but rather we can store the strings in each level of the tree as one *concatenated string*. By carefully traversing the path from the root node to a given internal node, v, at level ℓ, we can compute the offset of v's bit string within the concatenated string for level ℓ. This circumvents any issues of having to worry about the lower order space terms that the data structures for the bit strings occupy, as the strings become shorter, deeper in the tree.

One final detail is that it is possible to represent the characters in $\mathfrak{S}$ using any set of prefix-free codes [121]. Doing this can cause the tree to take on some of the properties of the encoding. For example, as we have described it, the wavelet tree occupies $U \lg \sigma(1 + o(1))$ bits. However, in Lemma 1.5.7 the space bound is stated as $U(\mathsf{H}_0(\mathfrak{S}) + 2)(1 + o(1)) + O(\sigma \lg U)$ bits. There are many ways to achieve this compression. One way is to represent the characters using a set of Huffman codes (Lemma 1.5.2), rather than the usual $(\lg \sigma)$-bit encoding [121]. Note that we require $O(\sigma \lg U)$ extra bits to store the decoded character in each leaf of the wavelet tree. We mention this method of applying prefix-free codes to boost the compression of the wavelet tree as we use it later.

There are several results that improve the time required by the wavelet tree for these operations. Golynski, Munro, and Rao [79] showed the following.

Lemma 1.5.8 ([79])**.** *Let $\mathfrak{S}$ be a string of U characters drawn from the alphabet $\Sigma = [1, \sigma]$. In the word-RAM model with word size $\Theta(\lg U)$ bits,*

1. *there is a data structure of size $U\mathsf{H}_0(\mathfrak{S}) + O(U)$ bits that supports the operations $\texttt{rank}_j(\mathfrak{S}, i)$, and $\texttt{select}_j(\mathfrak{S}, i)$ in $O(\lg \lg \sigma)$ time, where $j \in [1, \sigma]$ and $i \in [1, U]$; and,*

[6]We follow the convention that a single node has height 1.

2. *there is a data structure of size* $U \lg \sigma(1+o(1))$ *bits that supports the operations* $\texttt{access}(\mathfrak{S}, i)$ *and* $\texttt{rank}_j(\mathfrak{S}, i)$ *in* $O(\lg\lg\sigma)$ *time, and* $\texttt{select}_j(\mathfrak{S}, i)$ *in* $O(1)$ *time, where* $j \in [1, \sigma]$ *and* $i \in [1, U]$.

Ferragina et al. [60] showed how to support all three operations in the same space as the wavelet tree, but in time $O(1 + \lg\sigma / \lg\lg U)$ by increasing the branching factor of the original wavelet tree. Other work has also examined decreasing the space further by using higher order compression, as well as trade-offs between the schemes of Golynski et al. and Ferragina et al. [140, 12, 15]. Finally, Belazzougui and Navarro discuss lower bounds for performing `rank` queries on strings with larger alphabets [15].

We also make use of the following lemma, that provides optimal storage of, and access to, strings that have characters drawn from alphabets with non-power-of-two sizes.[7]

Lemma 1.5.9 (From [46])**.** *Consider a string* $\mathfrak{S}$ *of* U *characters from an alphabet* $\Sigma = [1, \sigma]$*. In the word-RAM model, with word size* $\Theta(\log U)$ *bits, we can represent* $\mathfrak{S}$ *using* $\lceil U \lg \sigma \rceil$ *bits, and support* `access` *in* $O(1)$ *time.*

1.5.5 Other Combinatorial Objects

We give a brief list of other combinatorial objects that have been made succinct. These objects include planar graphs [98, 10], trees (labelled, unlabelled, cardinal, ordinal, etc.) [98, 118, 54, 93], arbitrary directed and undirected graphs [56, 54], permutations and functions [117], partial orders [55] (see also Chapter 2), binary relations [13], succinct text indexes [90], and indexes for the information retrieval [37]. Several papers, following the work of Turán [152], have presented succinct representations of various combinatorial objects, such as general undirected graphs [120, 35].

1.5.6 A Brief Note About Randomization

Finally, in a few places, we briefly mention randomized algorithms. A *Las Vegas* randomized algorithm is one where the answer produced is guaranteed to be correct, but the running time may depend on random choices. A *Monte Carlo* randomized algorithm is one where the running

[7]The optimality follows from a recent result of Viola [153].

time does not depend on random choices, but the answer may be incorrect or the algorithm may fail with some probability.

Chapter 2

Succinct Posets

2.1 Introduction

Partially ordered sets, or *posets*, are useful for modelling relationships between objects, and appear in many different areas, such as natural language processing, machine learning, and database systems (see, for instance, [138, p.4]). As problem instances in these areas are ever-increasing in size, developing more space efficient data structures for representing posets is becoming an increasingly important problem.

By a constructive enumeration argument, Kleitman and Rothschild [105] showed that the number of size n posets is $2^{n^2/4+O(n)}$. Thus, the information theory lower bound indicates that representing an arbitrary poset requires $\lg(2^{n^2/4+O(n)}) = n^2/4 + O(n)$ bits.[1] This naturally raises the question of how a poset can be represented using only $n^2/4 + o(n^2)$ bits, *and* support efficient query operations.

It is well-known that, since a poset is a special kind of directed acyclic graph, we can represent it using an upper triangular bit matrix. The bit matrix approach occupies $n^2/2 + o(n^2)$ bits of space, and can support precedence (i.e., less than) queries in constant time: only a single bit must be examined. However, the leading coefficient is twice the information theory lower bound for representing a poset, so it raises the question of whether it is possible to do better.

[1]In this chapter we use $\lg x$ to denote $\lceil \log_2 x \rceil$: all logs hide the ceiling operator.

The purpose of this chapter is to close this gap by describing a *succinct* data structure for representing arbitrary posets. We note that a preliminary version of the content of this chapter appeared in the 20th Annual European Symposium on Algorithms (ESA 2012) as the paper "Succinct Posets" [116], and was joint work with J. Ian Munro.

We give a detailed description of our results in Section 2.4, but first provide some definitions in Section 2.2 and then highlight some of the previous work related to this problem in Section 2.3.

2.2 Definitions

A poset P, is a reflexive, antisymmetric, transitive binary relation $\preceq$ on a set S of n vertices[2], denoted $P = (S, \preceq)$. Let s_1 and s_2 be two vertices in S. For convenience we write $s_1 \prec s_2$ if $s_1 \preceq s_2$ and $s_1 \neq s_2$. If $s_1 \prec s_2$, we say s_1 *precedes* s_2. We refer to queries of the form, "Does s_1 precede s_2?" as *precedence queries*. If neither $s_1 \preceq s_2$ or $s_2 \preceq s_1$ hold, then s_1 and s_2 are *incomparable*, and we denote this as $s_1 \parallel s_2$. The set of *predecessors* of vertex s_1 is the set of vertices $\{s_2 : s_2 \prec s_1\}$. Similarly, the set of *successors* of vertex s_1 is the set of vertices $\{s_2 : s_1 \prec s_2\}$.

Each poset $P = (S, \preceq)$ is uniquely described by a directed acyclic graph, or *DAG*, $G_{\mathtt{c}} = (S, E_{\mathtt{c}})$, where $E_{\mathtt{c}} = \{(s_1, s_2) : s_1 \prec s_2\}$ is the set of edges. The DAG $G_{\mathtt{c}}$ is the *transitive closure graph* of P. Note that a precedence query for vertices s_1 and s_2 is equivalent to the query, "Is the edge (s_1, s_2) in $E_{\mathtt{c}}$?" Alternatively, let $G_{\mathtt{r}} = (S, E_{\mathtt{r}})$ be the DAG such that $E_{\mathtt{r}} = \{(s_1, s_2) : s_1 \prec s_2, \nexists_{s_3 \in S}, s_1 \prec s_3 \prec s_2\}$, i.e., the minimal set of edges that imply all the edges in $E_{\mathtt{c}}$ by transitivity. The DAG $G_{\mathtt{r}}$ also uniquely describes P, and is called the *transitive reduction graph* of P.

Posets are often illustrated using a *Hasse diagram*, which displays all the edges in the transitive reduction, and indicates the direction of an edge (s_1, s_2) by drawing vertex s_1 above s_2. In fact, we use the terminology *above* and *below* to refer to edges in the Hasse diagram, or transitive reduction; for example if s_1 is above s_2, then $(s_2, s_1) \in E_{\mathtt{r}}$. We refer to vertices that have no outward edges in the transitive reduction as *sinks*, and vertices that have no inward edges in the transitive reduction as *sources*. See Figure 2.1 for an example. Since all these concepts

[2]All posets we discuss are finite.

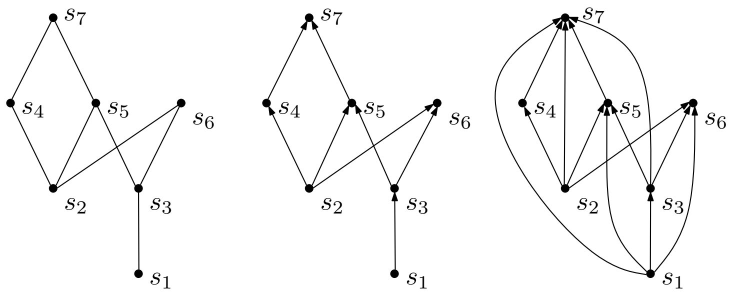


Figure 2.1: A Hasse diagram of a poset (left), the transitive reduction (centre), and the transitive closure (right). Vertices s_1 and s_2 are sources, and vertices s_6 and s_7 are sinks. In all our diagrams, paths from sinks to sources go upward.

are equivalent, we may freely move between them when discussing a poset, depending on which representation is the most convenient.

A *chain* of a poset, $P = (S, \preceq)$, is a totally ordered subset $\mathcal{C} = \{c_1, ..., c_k\} \subseteq S$; i.e., $c_i \prec c_j$ iff $i < j$, for $1 \leq i < j \leq k$. An *antichain* is a subset $A = \{a_1, ..., a_k\} \subseteq S$, such that each $a_i \parallel a_j$, for $i \neq j$. The *height* of a poset is the number of vertices in its maximum chain[3], and the *width* of a poset is the number of vertices in its maximum antichain. The following theorem is fundamental in partial order theory, and relates the size of minimal decompositions of chains and antichains to the width and height of the partial order:

Theorem 2.2.1 (Dilworth's Theorem and its dual [150])**.** *If $P = (S, \preceq)$ is a poset of width d, then there is a decomposition of S into vertex disjoint sets $\mathcal{C}_1 \cup ... \cup \mathcal{C}_d = S$, where $\mathcal{C}_i$ is a chain. Dually, if h is the height of P, then there is a decomposition of S into vertex disjoint sets $A_1 \cup ... \cup A_d = S$, where A_i is an antichain.*

Given a poset $P = (S, \preceq)$, a *linear extension* $\mathcal{L} = (S, \preceq_1)$ of P is a total ordering of S, i.e., a chain defined on the vertices in S with respect to the relation $\preceq_1$, such that if $s_i \prec s_j$ for some $i, j \in [1, n]$, then $s_i \prec_1 s_j$. However, note that the converse is not necessarily true: we cannot determine whether $s_i \prec s_j$ just based on the fact that $s_i \prec_1 s_j$. Let $\mathcal{L}^\star = \{\mathcal{L}_1 = (S, \preceq_1), ..., \mathcal{L}_k = (S, \preceq_k)\}$ denote a set of linear extensions of a poset $P = (S, \preceq)$. $\mathcal{L}^\star$ is said to be a *realizer* of P, if, for all $s_i, s_j \in S$, $s_i \prec s_j$ implies $s_i \prec_{k'} s_j$ for all $1 \leq k' \leq k$, *and* $s_i \prec_{k'} s_j$ for all $1 \leq k' \leq k$ implies $s_i \prec s_j$. The *order dimension* of a poset P is the cardinality of the smallest realizer for P.

[3]We note that the height is often defined as the number of vertices in its maximum chain minus one, but we find our definition slightly more convenient for our purposes in this chapter.

Based on the previous definitions, consider any chain $\mathcal{C} = \{s_1, ..., s_k\}$ of P. In the transitive closure, each edge (s_i, s_j) is present, where $1 \le i < j \le k$. Similarly, in the transitive reduction, *no* edge (s_i, s_j) is present, unless $j = i + 1$ for each $1 \le i < k$; many theorems about posets use the fact that no *forbidden polygon* exists in the transitive reduction, consisting of directed edges $(s_i, s_{i+1}), ..., (s_{j-1}, s_j), (s_i, s_j)$.[4] For example, Kleitman and Rothschild [104] use this property to count the number of distinct posets on n vertices.

Given a poset $P = (S, \preceq)$, and two vertices $s_1, s_2 \in S$, the *join* of s_1 and s_2 is the set of all vertices $s_3 \in S$ such that $s_1 \preceq s_3$, $s_2 \preceq s_3$, and there exists no $s_4 \in S$ such that $s_1 \preceq s_4 \prec s_3$ and $s_2 \preceq s_4 \prec s_3$. For example, the join of s_2 and s_3 in Figure 2.1 is $\{s_5, s_6\}$, and the join of s_3 and s_4 is $\{s_7\}$. The *meet* is defined symmetrically, as the set of all vertices $s_3 \in S$ such that $s_3 \preceq s_1$, $s_3 \preceq s_2$, and there exists no $s_4 \in S$ such that $s_3 \preceq s_4 \prec s_1$ and $s_3 \preceq s_4 \prec s_2$. The join can be thought of as a generalization of the *lowest common ancestor* of two nodes, s_1 and s_2, in a rooted tree [18]: i.e., the deepest node whose induced subtree contains both s_1 and s_2. Note that if the meet and join are unique for all pairs of vertices, then the poset is a *lattice*.

For a graph $G = (V, E)$, we use $E(H)$ to denote the set of edges $\{(s_1, s_2) : (s_1, s_2) \in E, s_1 \in H, s_2 \in H\}$, where $H \subseteq V$. Similarly, we use $G(H)$ to denote the subgraph of G induced by H, i.e., the subgraph with vertex set H and edge set $E(H)$. If $(s_1, s_2) \in E$, or $(s_2, s_1) \in E$, we say that s_2 is a *neighbour* of s_1 in G. We divide the neighbours of s_1 into two classes: *in-neighbours* and *out-neighbours*. The in-neighbours of s_1 are the set of vertices $\{s_2 : (s_2, s_1) \in E\}$, and the out-neighbours of s_1 are the set of vertices $\{s_2 : (s_1, s_2) \in E\}$. Thus, in the graph G_{c}, the in-neighbours of s_1 are the predecessors of s_1, and the out-neighbours of s_1 are the successors of s_1.

Finally, we use $K_{q,q}$ to refer to a balanced biclique with parts of size q, i.e., an *undirected* graph $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2| = q$, and $E = \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\}$.

2.3 Previous work

One way of storing a poset is by representing either its transitive closure graph, or transitive reduction graph, using an adjacency matrix. If we topologically order the vertices of this graph, then we can use an upper triangular matrix to represent the edges, since the graph is a DAG.

[4]Note that a forbidden polygon is *different* from a *directed cycle*: a forbidden polygon is acyclic.

Such a matrix occupies $\binom{n}{2}$ bits, and can, in a single bit probe, be used to report whether an edge exists in the graph between two vertices. Thus, this simple approach achieves a space bound that is roughly two times the information theory lower bound.

Since a poset is a special case of a DAG, the succinct graph representation of Farzan and Munro [56] can be used to represent posets. Given a poset with n vertices and m relations (i.e., edges in the transitive closure graph), this representation occupies

$$\lg \binom{\binom{n}{2}}{m}(1+o(1)) \text{ bits,} \tag{2.3.1}$$

provided m is either $o(n^{\varepsilon})$, or $\omega(n^{2-\varepsilon})$ for any constant $\varepsilon > 0$. In the alternative case, when m does not satisfy these bounds, they gave a representation that occupies

$$(1+\varepsilon)\lg \binom{\binom{n}{2}}{m} \text{ bits, for any constant } \varepsilon > 0. \tag{2.3.2}$$

We note that both of these representations support precedence queries in constant time, as well as listing all predecessors (resp. successors) of a vertex in the transitive closure in constant time per vertex reported. One might ask, "What is the value of m for a typical poset?" As it turns out, for a poset selected uniformly at random from the set of all possible posets, we have $m = n^2/8 + \Theta(n^{7/8})$ with high probability [105]. Thus, applying Equation 1.5.1 and Lemma 1.5.1, we get that the space (ignoring lower order terms) is $0.8113\binom{n}{2} \approx 0.4053n^2$ bits, with high probability. Thus, this representation is smaller than the plain adjacency matrix, especially when m is small, but still fails to match the information theory lower bound in general.

A very different representation, called the *ChainMerge* structure, was proposed by Daskalakis et al. [41], and occupies $\Theta(dn)$ *words* of space, where d is the width of the poset. The data structure essentially stores a minimum cardinality chain decomposition of the poset, and for each pair of chains, structural information on how the vertices relate to one another. The ChainMerge structure, like the other representations mentioned so far, supports precedence queries in constant time.

Recently, Farzan and Fischer [55] presented a data structure that represents a poset using $2dn(1 + o(1)) + (1 + \varepsilon)n \lg n$ bits, where d is the width of the poset, and $\varepsilon > 0$ is an arbitrary positive constant. This data structure supports precedence queries in constant time, and many other operations in time proportional to the width of the poset. These operations are best expressed in terms of the transitive closure and reduction graphs, and include:

1. reporting all k_{out} predecessors (resp. successors) of a vertex in $O(d + k_{\text{out}})$ time;
2. reporting all vertices above (resp. below) an arbitrary vertex in $O(d^2)$ time;
3. reporting an arbitrary neighbour of a vertex in the transitive reduction in $O(d)$ time;
4. reporting whether an edge exists between two vertices in the transitive reduction in $O(d)$ time;
5. reporting all k_{out} vertices that, for two vertices s_1 and s_2, are preceded by s_1 and precede s_2 in $O(d + k_{\text{out}})$ time;
6. reporting the set of vertices in the join (resp. meet) of a set of k_{in} vertices in $O(d^2 + k_{\text{in}})$ time;
7. and, reporting an arbitrary vertex in the join (resp. meet) of a set of k_{in} vertices in $O(d \min\{k_{\text{in}}, d\})$ time.

The basic idea of their data structure is to encode the ChainMerge structure of Daskalakis et al. [41] using bit strings, and to answer queries by using `rank` and `select` queries on these bit strings. We note that since both structures rely on finding a minimum chain decomposition, they require $O(d^2 n \lg(n/d))$ construction time [41].

Since the data structure of Farzan and Fischer [55] is adaptive on width, it is appropriate for posets where the width is small relative to n. However, if we select a poset of n vertices uniformly at random from the set of all possible n vertex posets, then it will have width $n/2 + o(n)$ with high probability [105, 146]. Thus, this representation will occupy $n^2 + o(n^2)$ bits, which is roughly *four times* the information theory lower bound. Furthermore, the time to construct the data structure of Farzan and Fischer is $O(n^3)$. We note that this can be improved to $\tilde{O}(n^{5/2})$ [57] since finding a minimum chain decomposition can be done by solving the max-flow problem on an unit capacity bipartite network [58].[5] Finally, with the exception of constant time precedence queries, all other operations described above take $\Theta(n)$ time for such a poset.

[5]We use the $\tilde{O}(x)$ or *soft-Oh* notation to hide factors polylogarithmic in x.

Other Related Work

For subclasses of posets, space efficient representations have been developed, though they are not succinct. These subclasses include: lattices [86, 143, 144], distributive lattices [87], Hasse diagrams representable by certain subclasses of planar DAGs [71] and forests [73]. There has been work on finding space efficient encodings of posets based on the size of their order dimension, or related measures of dimension that generalize order dimension, including: boolean dimension [72, 70, 71], encoding dimension [85], string dimension [74], and rectangle dimension [112]. However, it is well-known that it is an NP-complete problem to determine whether the order dimension of a poset is 3 or larger [158][6], and similar NP-completeness proofs exist for several of these alternative proposed measures of dimension [139]. As far as we are aware, boolean dimension [71] is the only alternative proposal that is known to be computable in polynomial time, and for arbitrary posets does not yield a succinct representation. Even approximating the order dimension of a partial order to within a factor of $O(n^{0.5-\varepsilon})$ for any constant $\varepsilon > 0$ is NP-hard [94], though we are not aware of such hardness of approximation results for these similar measures.

There is also work on representing arbitrary binary relations in a compact form [13, 11]. Since a poset is a special case of a binary relation, i.e., a transitive one, these representations can be used to represent a poset. However, for an arbitrary poset, these representations overshoot the information theory lower bound for posets by a $\lg n$ factor, and do not support constant time precedence queries.

For an arbitrary DAG, the *reachability relation* between vertices is a poset: i.e., given two vertices, s_1 and s_2, the relation of whether there a directed path from s_1 to s_2 in the DAG. Clearly, if we can answer precedence queries in an arbitrary poset, then we can answer reachability queries in an arbitrary DAG. We note that there is a great deal of work in the area of developing reachability (and distance) oracles for specific classes of directed graphs, such as planar directed graphs [148]. There is also work on distance oracles for undirected graphs, in both the approximate case, such as the seminal work of Thorup and Zwick [149] (see also the hundreds of papers by which it is referenced), and in the exact case [61]. However, none of the previous work has considered how to achieve succinct space for arbitrary directed graphs. For example, Thorup and Zwick [149, Proposition 5.2] mention that $\Omega(n^2)$ bits are required for reachability in a directed graph, but do not consider constant factors. This lower bound was later parameterized in terms

[6]A typical poset on n vertices has order dimension about $n/4$ [150, p.181].

of m, the number of edges in the directed graph, to $\Omega(nm^{1/2})$ bits by Cohen et al. [38], but again, no discussion of the constant is presented. There are numerous papers from the database community about answering reachability queries in directed graphs, starting with the paper of Agrawal, Borgida, and Jagadish [1]. Many different techniques have been proposed, such as tree cover, path-tree cover, 2-hop and 3-hop indexing (see Jin et al. [100] for a brief survey). These techniques are aimed at directed graphs where $m = o(n^2)$, and do not provide a succinct representation in the case of arbitrary directed graphs.

Finally we note that other than precedence queries, representations of posets have been explored that support efficient counting of linear extensions, and generation of a random linear extension [43]. There is also work on developing labelling schemes for graphs, so that reachability and adjacency can be tested efficiently by comparing graph labels [142].

2.4 Our Contributions

Our results hold in the word-RAM model of computation with word size $\Theta(\lg n)$ bits. Our main result is summarized in the following theorem:

Theorem 2.4.1. *Let $P = (S, \preceq)$ be a poset, where $|S| = n$. There is a succinct data structure for representing P that occupies $n^2/4 + O(n^2 \lg\lg n / \lg n)$ bits, and can support precedence queries in constant time: i.e., given vertices $s_1, s_2 \in S$, report whether $s_1 \preceq s_2$.*

Note that, unlike—for instance—the structure of a labelled tree, the structure of a poset requires far more bits to represent than a labelling of its vertices. Thus, in order to refer to the vertices in the poset, we assume each vertex has a label; i.e, an integer associated with it, drawn from the range $[1, n]$. We further note that we are also free to relabel the vertices, as an entire set of labels will require only $\Theta(n \lg n)$ bits. Consequently, we will always refers to vertices by their labels, so often when we refer to "vertex s_1", it means "the vertex in S with label s_1", depending on context.

Theorem 2.4.1 implies that we can, in constant time, answer queries of the form, "Is the edge (s_1, s_2) in the transitive closure graph of P?" In fact, we can also apply the same representation to support, in constant time, queries of the form, "Is the edge (s_1, s_2) in the transitive reduction graph of P?" However, at present it seems as though we can only support efficient queries in one or the other, *not both* simultaneously using succinct space. For this reason we focus on the closure, since it is more interesting, but state the following theorem:

Theorem 2.4.2. *Let $G_r = (S, E_r)$ be the transitive reduction graph of a poset, where $|S| = n$. There is a succinct data structure for representing G_r that occupies $n^2/4 + O(n^2 \lg\lg n/\lg n)$ bits, and, given vertices $s_1, s_2 \in S$, can report whether $(s_1, s_2) \in E_r$ in constant time.*

We also show that both of these data structures can be constructed efficiently. We have the following result:

Theorem 2.4.3. *Given the transitive closure (resp. reduction) graph of a poset P, the data structure of Theorem 2.4.1 (resp. Theorem 2.4.2) can be constructed for P in time $O(n^{2+\varepsilon})$, for any* constant $\varepsilon \in (0, 1.42]$. *Note that the smaller the constant ε is chosen to be, the larger the constant factor in the lower order space term—$O(n^2 \lg\lg n/\lg n)$—becomes.*

Reachability in Directed Graphs: Theorem 2.4.1 implies that there is a data structure that occupies $n^2/4 + o(n^2)$ bits, and can support reachability queries in a DAG, in constant time. As far as we are aware, this is the first reachability oracle for arbitrary DAGs that uses strictly less space than an upper triangular matrix, and supports reachability queries in constant time. We can even strengthen this observation by noting that for an *arbitrary directed graph G*, the *condensation* of G—the graph that results by contracting each strongly connected component into a single vertex [39, Section 22.5]—is a DAG. Given vertices s_1 and s_2, if s_1 and s_2 are in the same strongly connected component, then s_2 is reachable from s_1. Otherwise, we can apply Theorem 2.4.1 to the condensation of G. Thus, we get the following corollary:

Corollary 2.4.1. *Let G be a directed graph with n vertices and Φ strongly connected components. There is a data structure that occupies $\Phi^2/4 + O(\Phi^2 \lg\lg \Phi/\lg \Phi) + O(n \lg \Phi)$ bits and, given two vertices of G, s_1 and s_2, can report whether s_2 is reachable from s_1 in constant time.*

Note that even in the worst case when $\Phi = n$, the space bound of the previous corollary is roughly a quarter of the space required to represent an arbitrary directed graph. Switching back to the terminology of order theory, the previous corollary generalizes Theorem 2.4.1 to the larger class of binary relations known as *quasi-orders*: i.e., binary relations that are reflexive and transitive, but not necessarily antisymmetric. In fact, reflexivity does not restrict the binary relation very much, so we can further generalize Theorem 2.4.1 to arbitrary *transitive binary relations*.

Theorem 2.4.4. *Let $T = (S, \preceq)$ be a transitive binary relation $\preceq$ on a set of vertices S, where $|S| = n$. There is a succinct data structure for representing T that occupies $n^2/4 + O(n^2 \lg\lg n/\lg n)$ bits, and can support precedence queries in constant time: i.e., given two vertices $s_1, s_2 \in S$, report whether $s_1 \preceq s_2$.*

Additional operations: We also ask the question of what additional operations can be supported by our representation. We show that with a few minor changes, the data structure can support efficient listing of both the predecessors and successors of an arbitrary vertex. We note that these changes reveal a trade-off in the leading term of the space bound, but also make the lower order term slightly worse. In particular, we have the following:

Theorem 2.4.5. *Let $P = (S, \preceq)$ be a poset on n vertices and m relations. Using a data structure of size $\min\{n^2/4, \lg\binom{\binom{n}{2}}{m}\} + O(n^2/\sqrt{\lg n})$ bits, we can: answer precedence queries in constant time; list the predecessors of an arbitrary vertex in constant time per vertex reported; and, list the successors of an arbitrary vertex in constant time per vertex reported.*

Overview of the data structure and techniques used: The main idea behind our succinct data structure is an algorithm for compressing a poset so that it occupies space matching the information theory lower bound, to within lower order terms. The difficulty is ensuring that we are able to query the compressed structure efficiently. Our first attempt at designing a compression algorithm was essentially a reverse engineered version of an enumeration proof by Kleitman and Rothschild [104]. However, though the algorithm achieved the desired space bound, there was no obvious way to answer queries on the compressed data due to one crucial compression step[7]. Though there are several other enumeration proofs (cf., [105, 26]), they appeal to a similar strategy, making them difficult to use as the basis for a data structure. This led us to develop an alternate compression algorithm, that borrows techniques from extremal graph theory. In particular, we make use of algorithmic approaches to solving the *Zarankiewicz Problem*. This problem asks, "What is the minimum number of edges a bipartite graph must have before it is

[7]Further explanation: In their enumeration argument (see Lemma 3) they count the set B_{n+1}, which contains partial orders that have a source vertex s_1 that is below every vertex in some set Q, of size $\sqrt{n}$, where Q has the property that at least $n/2$ vertices are below some vertex in Q. This provides an efficient way to encode the relations with s_1, as any vertex below a vertex in Q cannot be above s_1, as it would form a forbidden polygon; in this case a triangle. Thus, we only need to store the set Q, as well as a bit for each vertex *not* covered by a vertex in Q, of which there are at most $n/2 - \sqrt{n}$. However, creating a data structure using this property seems difficult, as the data is implicit in terms of the neighbours of Q, which are not explicitly stored.

guaranteed to contain a balanced biclique $K_{q,q}$ as a subgraph?" There is a well-known result by Kövári, Sós, and Turán [107] that gives a bound for any q, but does not provide much intuition on how to find such a biclique. We make use of a recent result by Mubayi and Turán [114] that allows us to find such bicliques efficiently.

It would seem conceptually simpler to present our compression algorithm as having two steps. In the first step, we preprocess the poset, removing edges in its transitive closure graph, to create a new poset where the height is not too large. We refer to what remains as a *flat* poset. We then make use of the fact that, in a flat poset, either balanced biclique subgraphs of the transitive closure graph—containing $\Omega(\lg n/\lg\lg n)$ vertices—must exist, or the poset is relatively sparsely connected. In the former case, the connectivity between these balanced biclique subgraphs and the remaining vertices is shown to be space efficient to encode using the fact that all edges implied by transitivity are in the transitive closure graph. Once we encode the edges, we can remove the vertices in the biclique from the poset and apply the same idea recursively. In the latter case, we can directly apply techniques from the area of succinct data structures to compress the poset.

After discussing the data structure and how it is used, we return to the question of how efficiently it can be constructed. One issue is that Mubayi and Turán's analysis of the running time for their algorithm only examines the time taken to compute a single biclique. Since our construction algorithm computes many bicliques, we describe a simple way of efficiently computing a set of vertex disjoint bicliques using Mubayi and Turán's algorithm. We note that this idea is not entirely original, as batch computation of bicliques that cover the *edges* of a graph, called *clique stripping*, has been studied previously by Feder and Motwani [57]. Clique stripping finds applications in speeding up existing algorithms on graphs by transforming the input into an equivalent graph that has fewer edges. Mubayi and Turán appear to have overlooked this related work, and we note that Feder and Motwani's algorithm can be used to find balanced bicliques, though they do not discuss this feature. The main difference between our construction algorithm and the clique stripping algorithm is that our biclique covering is vertex disjoint rather than edge disjoint.

Remark 2.4.1. *For the particular application of compressing directed graphs that represent social networks, experimental results have found that heuristic approaches for compression based on finding large bicliques work quite well in practice [95]. Though our result is not aimed at a specific application, we find it interesting that our similar approach to directed graph compression can be shown to be both efficient to compute, and optimal for arbitrary directed graphs.*

Road Map: In the next section, we describe our succinct data structure for representing the transitive closure graph of a poset (Theorem 2.4.1). Then in Section 2.6 we prove Theorems 2.4.2 and 2.4.4. In Section 2.7 we examine how to construct the data structure of Theorem 2.4.1, and prove Theorem 2.4.3. In Section 2.8 we briefly discuss additional operations. In particular, we show how to efficiently support the listing the predecessors *and* successors of an arbitrary vertex, and also show that any data structure representing a poset of n vertices that supports meet and join queries efficiently must have construction time equal to the cost of performing boolean matrix multiplication on two arbitrary $n \times n$ matrices. In Section 2.9 we return to the question of representing both the transitive reduction and closure simultaneously, and show how, like an adjacency matrix, we can use Lemma 1.5.9 to reduce the cost of a *simultaneous representation.*

2.5 The Data Structure

In this section we describe a succinct data structure for representing posets.

2.5.1 Flattening a Poset

Let $\gamma \geq 1$ be a parameter, to be fixed later; the reader would not be misled by thinking that we will eventually set $\gamma = \lg n$. We call a poset *γ-flat* if it has height no greater than γ. In this section, we describe a preprocessing algorithm for posets, transforming a poset into a γ-flat poset, without losing any information about its original structure. After describing this preprocessing algorithm, we develop a compression algorithm for flat posets. Using the preprocessing algorithm together with the compression algorithm, and an additional index, yields a succinct data structure for posets.

Height-Based Antichain Decomposition

Let $P = (S, \preceq)$ be an arbitrary poset with transitive closure graph $G_{\mathsf{c}} = (S, E_{\mathsf{c}})$. We define the *height of a vertex* $s_1 \in S$ to be the cardinality of the maximum length chain that has s_1 as the *maximum vertex*; i.e., no vertex is above s_1 in the chain. Thus, all sources are of height 1. Let $h(P)$ denote the height of P; we use h for brevity when no confusion is possible. We use A_i to denote the set of all the vertices of height i, $1 \leq i \leq h$, and $\mathcal{A}$ to denote the set $\{A_1, ..., A_h\}$.

Each set A_i is an antichain, since if $s_1 \prec s_2$ then s_2 has height strictly greater than s_1. An equivalent recursive definition is that A_1 is the set of sources of P, and A_i is the set of sources of the poset $P \setminus \cup_{j=1}^{i-1} A_j$, for $1 < i \leq h(P)$; i.e., we recursively remove the sources of P to form the next antichain.

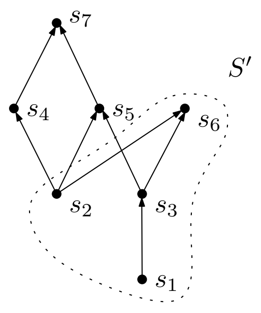
Remark 2.5.1. *There are many ways that a poset can be decomposed into antichains. We choose the height-based method described above mainly because it is the simplest method of which we are aware. We also note that, though the above method returns a decomposition into a minimum number of antichains by Theorem 2.2.1, the lengths of these antichains are rather arbitrary. For the purposes of our data structure, we wish to have antichains that are also large relative to n. Instead of trying to come up with a method of decomposing antichains to maximize antichain length, we instead merge consecutive antichains together, changing the structure of the poset, so that we end up with a poset that has long antichains.*

As a next step, we compute a linear extension $\mathcal{L}$ of the poset P in the following way, using $\mathcal{A}$. The linear extension $\mathcal{L}$ is such that all vertices in A_i are ordered before those in A_{i+1} for all $1 \leq i < h$, and the vertices within the same antichain A_i are ordered arbitrarily within $\mathcal{L}$. We write $S(i)$ to denote the vertex ordered i-th in $\mathcal{L}$. Similarly, given any subset $S' \subseteq S$, we use the notation $S'(x)$ to denote the vertex ordered[8] x-th among the vertices in the subset S', according to the linear extension $\mathcal{L}$, where $1 \leq x \leq |S'|$. Thus, for some fixed subset S', we can imagine a bit string $\mathfrak{B}$ of length n, where bit i is a 1 if $S(i) \in S'$, and 0 otherwise. Then based on the previous definitions, we have $S'(i) = S(\texttt{select}_1(\mathfrak{B}, i))$. We illustrate these concepts in Figure 2.2. Later, this particular linear extension will be used extensively, when we output the structure of the poset as a bit string.

Merging Antichains

We now describe a preprocessing algorithm to transform an arbitrary poset P into a γ-flat poset $\tilde{P}$. We assume P is not γ-flat, otherwise we are done. Given two consecutive antichains A_i and A_{i+1}, we define a *merge step* to be the operation of replacing A_i and A_{i+1} by a new antichain $A'_i = A_i \cup A_{i+1}$, and outputting and removing all the edges between vertices in A_i and A_{i+1} in the transitive closure of P, i.e., $E_{\mathtt{c}}(A_i \cup A_{i+1})$. We say that A_{i+1} is the *upper antichain*, A_i is

[8]We are specifically using the term "ordered" to avoid overloading the term "rank".



$A_1 = \{s_1, s_2\}$	$S'(1) = s_2$
$A_2 = \{s_3, s_4\}$	$S'(2) = s_1$
$A_3 = \{s_5, s_6\}$	$S'(3) = s_3$
$A_4 = \{s_7\}$	$S'(4) = s_6$

$$\mathcal{L} = \{s_2, s_1, s_3, s_4, s_6, s_5, s_7\}$$

Figure 2.2: The antichain decomposition of the poset from Figure 2.1. The set S' is the set of vertices surrounded by the dotted line. Note that $\mathcal{L}$ is only one of many possible linear extensions.

the *lower antichain*, and refer to the new antichain A'_i as the *merged antichain*. Each antichain A_j where $j > i + 1$ becomes antichain A'_{j-1} in the *residual decomposition*, after the merge step.

To represent the edges that were removed, let $\mathfrak{S}$ be a bit string, storing $\binom{n}{2}$ bits, one for each possible edge in the transitive closure graph of P. Thus $\mathfrak{S}$ can be thought of as an upper triangular adjacency matrix (of bits). Each time an edge $(S(x), S(y))$ is removed during a merge step, we record a bit in $\mathfrak{S}$ representing the pair (x, y) in $\mathfrak{S}$. Obviously, we do not wish to store $\mathfrak{S}$ in its raw form, as it would occupy $\binom{n}{2}$ bits, so we soon discuss how to represent $\mathfrak{S}$ so that it occupies subquadratic space.

```
Algorithm  FLATTEN($\mathcal{A}, i$): where $i$ is the index of an antichain in $\mathcal{A}$.
  if $i \geq |\mathcal{A}|$ then
    EXIT
  end if
  if $|A_i| + |A_{i+1}| \leq 2n/\gamma$ then
    Perform a merge step on $A_i$ and $A_{i+1}$
  else
    $i \leftarrow i + 1$
  end if
  FLATTEN($\mathcal{A}, i$)
```

The Flatten Algorithm

There are many possible ways that we could apply merge steps to the poset in order to make it γ-flat. The simple method we choose is presented in algorithm FLATTEN. Let $\tilde{\mathcal{A}}$ be the residual antichain decomposition that remains after executing FLATTEN$(\mathcal{A}, 1)$, and $\tilde{P}$ be the resulting poset. The number of antichains in $\tilde{\mathcal{A}}$ is at most γ, and therefore the resulting poset $\tilde{P}$ is γ-flat. We make the following further observation:

Lemma 2.5.1. FLATTEN$(\mathcal{A}, |\mathcal{A}|)$ *removes* $O(n^2/\gamma)$ *edges from* G_c.

Proof. Consider the decomposition $\mathcal{A}$, where $h = |\mathcal{A}|$. Let $n_1, ..., n_h$ denote the number of vertices in $A_1, ..., A_h$, and n_{k_0,k_1} to denote $\sum_{i=k_0}^{k_1} n_i$. We use induction to prove the inequality:

$$\sum_{i=k_0}^{k_1-1} \left(\left(\sum_{j=k_0}^{i} n_j \right) n_{i+1} \right) \leq \frac{n_{k_0,k_1} \left(n_{k_0,k_1} - 1\right)}{2}, \tag{2.5.1}$$

where $1 \leq k_0 < k_1 \leq h$. Note that this summation describes the maximum number of edges that are deleted between pairs of vertices in a residual antichain after the FLATTEN algorithm terminates. The induction is over the parameter k_1, fixing $k_0 = 1$, since the value of k_0 is irrelevant. The base case, $k_1 = 2$ holds since $(n_1 + n_2)(n_1 + n_2 - 1)/2 \geq n_1 n_2$ for all integers $n_1, n_2 \geq 1$. Assume the inequality holds for all $k \in [2, k_1 - 1]$. We have:

$$\sum_{i=1}^{k_1-1} \left(\left(\sum_{j=1}^{i} n_j \right) n_{i+1} \right) = \sum_{i=1}^{k_1-2} \left(\left(\sum_{j=1}^{i} n_j \right) n_{i+1} \right) + \left(\sum_{j=1}^{k_1-1} n_j \right) n_{k_1} \tag{2.5.2}$$

$$\leq n_{1,k_1-1} \left(\frac{n_{1,k_1-1} - 1}{2} + n_{k_1} \right) \text{ (by the inductive step)} \tag{2.5.3}$$

$$= (n_{1,k_1} - n_{k_1}) \left(\frac{n_{1,k_1} - 1 + n_{k_1}}{2} \right) \tag{2.5.4}$$

$$= n_{1,k_1} \left(\frac{n_{1,k_1} - 1}{2} \right) + \frac{n_{1,k_1} n_{k_1}}{2} - \frac{n_{k_1} n_{1,k_1}}{2} - \frac{n_{k_1}(n_{k_1} - 1)}{2} \tag{2.5.5}$$

$$\leq n_{1,k_1} \left(\frac{n_{1,k_1} - 1}{2} \right). \tag{2.5.6}$$

For each of the at most γ antichains in $\tilde{\mathcal{A}}$, there are two cases. Either the antichain was not created as the result of merge steps, or the antichain has size at most $2n/\gamma$, and is the result of some sequence of merge steps. Thus, the previous inequality implies that FLATTEN removes no more than $O(n_{k_0,k_1}^2)$ edges for such an antichain, where $n_{k_0,k_1} = O(n/\gamma)$. Therefore, the total number of edges removed during the merging steps is $O((n/\gamma)^2\gamma) = O(n^2/\gamma)$. □

Next, we show how to answer precedence queries on vertices that end up in the same antichain in the residual decomposition:

Lemma 2.5.2. *Let* $\texttt{S-Acc}(U, N)$ *denote the space cost of storing a bit string of length* U *bits, containing* N *ones, such that the operation* `access` *can be performed in constant time, and* $\texttt{T-Acc}(U, N)$ *denote its construction time. For any* $\gamma \geq \lg n$*, there is a data structure of size* $\texttt{S-Acc}(\binom{n}{2}, n^2/\gamma) + \Theta(n \lg n)$ *bits that, given vertices* s_1 *and* s_2*, can determine in constant time whether* s_1 *precedes* s_2*, if both* s_1 *and* s_2 *belong to the same antichain in the residual antichain decomposition* $\tilde{\mathcal{A}}$*, created by executing* $\textsc{Flatten}(P, |\mathcal{A}|)$*. The data structure can be constructed in* $O(\texttt{T-Acc}(\binom{n}{2}, n^2/\gamma) + n^2)$ *time, given access to an adjacency matrix representation of* $G_{\mathtt{c}}$*, in addition to an array storing the linear extension* $\mathcal{L}$*, and the number of vertices in each antichain in* $\mathcal{A}$*.*

Proof. Recall that each merge step writes the removed edges to the bit string $\mathfrak{S}$. By Lemma 2.5.1, at most $O(n^2/\gamma)$ edges are deleted by the $\textsc{Flatten}$ algorithm. Thus, at most $O(n^2/\gamma)$ ones appear in $\mathfrak{S}$, which has length $\binom{n}{2}$. We also store an array of size $\Theta(n \lg n)$ bits that, given the label of a vertex s_1, returns x and i such that $S(x) = s_1$, and $s_1 \in A_i$.

Suppose we are given two vertices s_1 and s_2 that we wish to determine the relationship between. We use the array to find x, y, i, and j such that $S(x) = s_1$, $S(y) = s_2$, $s_1 \in A_i$ and $s_2 \in A_j$. If $i \neq j$, then we return "DIFFERENT ANTICHAINS", which means that our data structure cannot provide the answer. Without loss of generality, assume $x < y$; if $x = y$, then return $s_1 = s_2$. We examine the bit corresponding to the pair (x, y) in $\mathfrak{S}$, and return $s_1 \prec s_2$ if it is a 1 and $s_1 \parallel s_2$ otherwise.

In terms of construction time, the edges written to $\mathfrak{S}$ can be determined in $O(n^2/\gamma)$ time by carefully scanning through the array storing $\mathcal{L}$. $\square$

Remark 2.5.2. *If we choose* $\gamma \geq \lg n$*, and use Lemma 1.5.4 to represent bit strings, then the data structure of the previous lemma occupies* $O(n^2 \lg\lg n / \lg n)$ *bits, by Lemma 1.5.5, and can be constructed in* $O(n^2)$ *time. We note that other representations of bit strings can be used to achieve an asymptotically smaller space bound when* $\gamma = \omega(\lg n)$ *(see Section 1.5.3). However, as we shall see later, this data structure is not the bottleneck that causes the lower order term of Theorem 2.4.1 to be of order* $O(n^2 \lg\lg n / \lg n)$*.*

2.5.2 Compressing Flat Posets

In this section we describe a compression algorithm for flat posets that, in the worst case, matches the information theory lower bound to within lower order terms. We begin by stating the following lemma, which is an algorithmic solution to the Zarankiewicz problem:

Lemma 2.5.3 (Mubayi and Turán [114]). *There is a constant ς_{min} such that, given a graph $G = (V, E)$ with $|V| \geq \varsigma_{min}$ vertices and $8|V|^{3/2} \leq |E| \leq |V|^2/2$ edges, we can find a balanced biclique $K_{q,q}$, where $q = \Theta(\lg |V| / \lg(|V|^2/|E|))$, in time $\Theta(|E|)$.*

For our purposes in this section, Lemma 2.5.3 will suffice, but we will examine the problem of computing such bicliques in more detail in Section 2.7.

Let $\tilde{P}$ be a $\lg n$-flat poset, $G_{\mathsf{c}} = (S, E_{\mathsf{c}})$ be its transitive closure, and $\tilde{\mathcal{A}} = \{A_1, ..., A_h\}$ be its height-based antichain decomposition (discussed in the last section), which contains $h \leq \lg n$ antichains. We now prove our key lemma, which is crucial for the compression algorithm.

Lemma 2.5.4 (Key Lemma). *Consider the subgraph $G_\Upsilon = G_{\mathsf{c}}(A_i \cup A_{i+1})$ for some $1 \leq i < h$, and ignore the edge directions so that G_Υ is undirected. Suppose G_Υ contains a balanced biclique subgraph with vertex set χ, and $|\chi| = \tau$; i.e., $G(\chi)$ is a $K_{\tau/2,\tau/2}$. Then there are at most $2^{\tau/2+1} - 1$ ways that the vertices in χ can have edges to a vertex in $S \setminus (A_i \cup A_{i+1})$.*

Proof. Each vertex $v \in S \setminus (A_i \cup A_{i+1})$ is in A_j, where either $j > i + 1$ or $j < i$. Without loss of generality, consider the case where $j > i + 1$. If any vertex $v_0 \in \chi \cap A_{i+1}$ has an edge to v, then *all* vertices in $\chi \cap A_i$ have an edge to v; see Figure 2.3 for an illustration. Thus, the vertices in $\chi \cap A_{i+1}$ can have edges to v in $2^{\tau/2} - 1$ ways, or the vertices in $\chi \cap A_i$ can have edges to v in $2^{\tau/2} - 1$ ways, or the vertices in χ can have no edges to v. In total, there are $2^{\tau/2+1} - 1$ ways edges can go to v from vertices in χ. □

In more explicit terms, consider a biclique in the lowest two antichains $\chi \subset G_{\mathsf{c}}(A_1 \cup A_2)$ where $\tau = |\chi|$, and any vertex $v \in S \setminus (A_1 \cup A_2)$. Let $x_0 \, x_1 \, \ldots \, x_{\tau/2}$ be a bit string, where bit x_0 is a *control bit* that indicates whether there is any edge from vertices in $\chi \cap A_2$ to v in E_{c}. If $x_0 = 1$, then x_i indicates whether $(\chi(i + \tau/2), v) \in E_{\mathsf{c}}$, for $1 \leq i \leq \tau/2$; we need not consider vertices in $\chi \cap A_1$, since they *all* have edges to v. Otherwise, x_i indicates whether $(\chi(i), v) \in E_{\mathsf{c}}$. Thus, by checking 2 bits, we can determine whether an edge between $\chi(i)$ and v exists, for any $1 \leq i \leq \tau$.

Algorithm Compress-Flat$(\hat{P}, \hat{n}, \hat{\mathcal{A}}, \hat{h})$: where $\hat{P} = (\hat{S}, \preceq)$ is a $\lg n$-flat poset of $\hat{n} \le n$ vertices, and $\hat{\mathcal{A}} = \{\hat{A}_1, ..., \hat{A}_{\hat{h}}\}$ is a decomposition of the vertices in $\hat{P}$ into $\hat{h}$ antichains.

```
1: if $\hat{h} = 1$ then
2:     EXIT
3: else if $|\hat{A}_1 \cup \hat{A}_2| \ge \varsigma_{\min}$ and $|E_{\mathsf{c}}(\hat{A}_1 \cup \hat{A}_2)| \ge (\hat{n}/\lg \hat{n})^2$ then
4:     Apply Lemma 2.5.3 to the subgraph $G_{\mathsf{c}}(\hat{A}_1 \cup \hat{A}_2)$. This computes a balanced biclique with
       vertex set $\chi \subset \hat{A}_1 \cup \hat{A}_2$ such that $\Omega(\lg \hat{n}/\lg\lg \hat{n}) = \tau = |\chi| \le \Theta(\lg \hat{n})$.
5:     Let $H \leftarrow \hat{S} \setminus \chi$. Output an array EdgeArray, where EdgeArray$[k] \in [0, 2^{\tau/2+1} - 1]$ and
       indicates how $H(k)$ relates to $\chi$ (see Lemma 2.5.4 and discussion in subsequent paragraph).
6:     Let $\hat{A}_1 \leftarrow \hat{A}_1 \setminus \chi$, and $\hat{A}_2 \leftarrow \hat{A}_2 \setminus \chi$.
7:     Compress-Flat$(\hat{P} \setminus \chi, \hat{n} - \tau, \hat{\mathcal{A}}, \hat{h})$
8: else
9:     Perform a merge step on $\hat{A}_1$ and $\hat{A}_2$
10:    Set $\hat{h} \leftarrow \hat{h} - 1$
11:    Compress-Flat$(\hat{P}, \hat{n}, \hat{\mathcal{A}}, \hat{h})$
12: end if
```

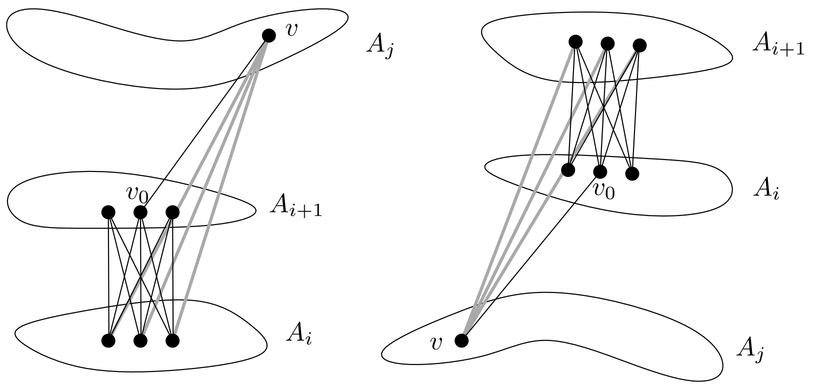


Figure 2.3: Illustration of Lemma 2.5.4. Left: the case where $j > i + 1$. If there is an edge $(v_0, v) \in E_{\mathsf{c}}$, then all the dark gray edges must exist. Right: the case where $j < i$. If there is an edge $(v, v_0) \in E_{\mathsf{c}}$, then all the dark gray edges must exist.

Furthermore, since there are no edges between pairs of vertices $v \in A_j$ and $v_0 \in \chi \cap A_j$, for $j \in [1, 2]$, the same encoding can be used for vertices in $A_1 \cup A_2 \setminus \chi$. If $v \in A_2 \setminus \chi$, then we set x_0 to 0, indicating that any edges to v come from vertices in $\chi \cap A_1$. Otherwise, we set x_0 to 1, indicating that any edge must *go from* v to vertices in $\chi \cap A_2$, *and* no vertex in $\chi \cap A_1$. Note that, we must first know whether v is in antichain A_1 or not to properly interpret the bits $x_0 \dots x_{\tau/2}$.

Lemma 2.5.5. *Consider the undirected graph formed by the vertices and edges in $G_\Upsilon = G_c(A_1 \cup A_2)$. If $|E_c(A_1 \cup A_2)| > n^2/\lg^\varsigma n$ for any constant $\varsigma > 0$, then the undirected graph G_Υ contains a $K_{\tau/2,\tau/2}$ as a subgraph, where $\Omega(\lg n/\lg\lg n) = \tau \le \Theta(\lg n)$.*

Proof. In order to have $n^2/\lg^\varsigma n$ edges, for any constant $\varsigma > 0$, G_Υ must contain at least $\Theta(n/\lg^{\varsigma/2} n)$ vertices. In this extreme case, $\tau = \Theta(\lg(n/\lg^{\varsigma/2} n)) = \Theta(\lg n)$, for sufficiently large n, by Lemma 2.5.3. In the other extreme case, there are $\Theta(n)$ vertices in $A_1 \cup A_2$, and thus $\tau = \Omega(\lg n/\lg\lg^\varsigma(n)) = \Omega(\lg n/\lg\lg n)$, by Lemma 2.5.3. □

Consider the algorithm Compress-Flat. The main idea is to apply Lemma 2.5.5 to the bottom two consecutive antichains the antichain decomposition, if they have many edges—defined on line 3—between them in the transitive closure graph. If they have few edges between them, then we apply a merge step. The algorithm terminates when only one antichain remains. We refer to the case on lines 4-7 as the *dense case*, and the case on lines 9-11 as the *sparse case*. We now prove that the size of the output of Compress-Flat matches the information theory lower bound to within lower order terms. Note that Compress-Flat, as described, is not truly a compression algorithm, as an additional index—that we describe later—will be needed to interpret the output.

Lemma 2.5.6. *Let $\texttt{S-Acc}(U, N)$ denote the space cost of storing a bit string of length U bits, containing N ones, such that the operation* `access` *can be performed in constant time. The output of* Compress-Flat$(\tilde{P}, n, \tilde{A}, h)$ *is no more than $n^2/4 + O(n^2 \lg\lg n/\lg n) + \texttt{S-Acc}(\binom{n}{2}, n^2/\lg n)$ bits.*

Proof. In the base case (line 2), the lemma trivially holds since nothing is output. Next we show that the total output from all the sparse cases does not exceed $\texttt{S-Acc}(\binom{n}{2}, n^2/\lg n)$ bits. Recall that a merge step records a bit in the string $\mathfrak{S}$ corresponding to each edge that was removed. Since there can be at most $\lg n$ merge steps— $\tilde{P}$ has height at most $\lg n$—and each merge step removes at most $n^2/\lg^2 n$ edges, at most $n^2/\lg n$ bits are written to the bit string $\mathfrak{S}$.[9] Thus, we spend $\texttt{S-Acc}(\binom{n}{2}, n^2/\lg n)$ bits to store the output from the sparse cases.

We now prove the lemma by strong induction for the dense cases. Let $\mathcal{S}(n)$ denote the number of bits output by Compress-Flat$(\tilde{P}, n, \tilde{A}, h)$ in the dense cases. We can assume $\mathcal{S}(n_0) \leq n_0^2/4 + \varsigma_0(n_0^2 \lg\lg n_0)/\lg n_0$ for all $3 \leq n_0 < n$, where $\varsigma_0 > 0$ is a sufficiently large constant.[10] All additional self-delimiting information—for example, storing the length of the strings output on lines 5-8— occupies no more than $\varsigma_1 \lg n$ bits for some constant $\varsigma_1 > 0$. Finally, recall that $\varsigma_2 \lg n/\lg\lg n \leq \tau \leq \varsigma_3 \lg n$ for some constants $\varsigma_2, \varsigma_3 > 0$, by Lemma 2.5.5. We have:

$$\begin{aligned}
\mathcal{S}(n) &= \lg(2^{\tau/2+1})\,(n-\tau) + \varsigma_1 \lg n + \mathcal{S}(n-\tau) \\
&< \left(\frac{\tau}{2}+1\right)(n-\tau) + \varsigma_1 \lg n + \left(\frac{1}{4} + \frac{\varsigma_0 \lg\lg n}{\lg(n-\tau)}\right)\left(n^2 - 2n\tau + \tau^2\right) \\
&< \frac{n^2}{4} + \frac{\varsigma_0 n^2 \lg\lg n}{\lg(n-\tau)} - (\varsigma_5 - \varsigma_4)n \quad (\varsigma_5 < 2\varsigma_0\varsigma_2,\ \varsigma_4 > 1).
\end{aligned}$$

Note that through our choice of ς_0 and ς_4, we can ensure that $\varsigma_5 - \varsigma_4$ is a positive constant. This rightmost term can absorb the discrepancy between the $\lg(n-\tau)$ term in the denominator, and the desired $\lg n$ term. Thus, the induction holds. □

Remark 2.5.3. *The previous lemma contains the bottleneck that causes the lower order term of Theorem 2.4.1 to be $O(n^2 \lg\lg n/\lg n)$ bits. For example, if it were possible to find balanced biclique subgraphs containing $\omega(\lg n)$ vertices in graphs with n vertices and $o(n^2/\lg^2 n)$ edges, then it would be possible to improve the lower order term to $o(n^2 \lg\lg n/\lg n)$. However, since we want to make the bicliques as large as possible without causing the representation of the sparse cases, or the index of Lemma 2.5.2, to exceed $O(n^2 \lg\lg n/\lg n)$ bits, we choose $\gamma = \lg n$ and set $\varsigma = 2$ in Lemma 2.5.5.*

Next, we show how to support precedence queries on a $\lg n$-flat poset, using the output of the Compress-Flat algorithm. If vertex s_1 is removed in the dense case, we say s_1 is *associated* with the output on lines 6-9.

[9] We note that we can use the same bit string to record these edges as the one used in the index of Lemma 2.5.2

[10] Note that the 3 comes from the fact that $\lg\lg 2 = 0$.

Lemma 2.5.7. *Let* $\texttt{S-Rank}(U, N)$ *and* $\texttt{S-Acc}(U, N)$ *denote the space cost of storing a bit string of length* U *bits, containing* N *ones, such that the operations* $\texttt{rank}_1$ *and* `access` *can be performed in constant time, respectively. Let* $\tilde{P}$ *be a* $\lg n$*-flat poset on* n *vertices, with antichain decomposition* $\tilde{A} = \{A_1, ..., A_h\}$*. There is a data structure of size* $n^2/4 + O(n^2 \lg\lg n / \lg n) + \texttt{S-Acc}(\binom{n}{2}, n^2/\lg n) + \texttt{S-Rank}(k, k)$ *bits, where* $k = O(n^2 \lg\lg n / \lg n)$*, that, given vertices* s_1 *and* s_2*, can report whether* s_1 *precedes* s_2 *in constant time.*

Algorithm PRECEDENCE-QUERY(s_1, s_2): Reports the relation between the vertices s_1 and s_2.

```
1: Determine $i$, $j$, $x$ and $y$, such that $A_i, A_j \in \tilde{\mathcal{A}}$, $s_1 \in A_i$, $s_2 \in A_j$, $S(x) = s_1$, and $S(y) = s_2$.
   {We assume without loss of generality that $i \leq j$.}
2: if $i = j$ then
3:     REPORT $s_1 \not\prec s_2$
4: else if the bit corresponding to pair $(x, y)$ in $\mathfrak{S}$ is a 1 then
5:     REPORT $s_1 \prec s_2$
6: else if $\texttt{Record}[s_2]\texttt{.id} = \texttt{Record}[s_1]\texttt{.id}$ then
7:     if $\texttt{Record}[s_1]\texttt{.id} \neq \infty$ and $\texttt{Record}[s_2]\texttt{.top} \neq \texttt{Record}[s_1]\texttt{.top}$ then
8:         REPORT $s_1 \prec s_2$
9:     else
10:        REPORT $s_1 \parallel s_2$
11:    end if
12: else if $\texttt{Record}[s_2]\texttt{.id} < \texttt{Record}[s_1]\texttt{.id}$ then
13:    Let $\ell = \texttt{Record}[s_2]\texttt{.id}$.
14:    if $\texttt{Record}[s_2]\texttt{.top} = 1$ then
15:        Locate EdgeArray using $\texttt{Record}[s_2]\texttt{.pnt}$.
16:        REPORT the relation by checking $\texttt{EdgeArray}[\texttt{Map}_\ell(x)]$ (see discussion below Lemma 2.5.4)
17:    else
18:        REPORT $s_1 \parallel s_2$
19:    end if
20: else if $\texttt{Record}[s_1]\texttt{.id} < \texttt{Record}[s_2]\texttt{.id}$ then
21:    Let $\ell = \texttt{Record}[s_1]\texttt{.id}$.
22:    Locate EdgeArray using $\texttt{Record}[s_1]\texttt{.pnt}$.
23:    REPORT the relation by checking $\texttt{EdgeArray}[\texttt{Map}_\ell(y)]$ (see discussion below Lemma 2.5.4).
24: end if
```

Proof. We augment the output of COMPRESS-FLAT with additional data structures in order to answer queries efficiently. We denote the first set of vertices removed in a dense case as χ_1, the second set as χ_2, and so on. Let χ_κ denote the last set of vertices removed in a dense case. We next prove that $\kappa = O(n \lg\lg n / \lg n)$. Consider the following recurrence:

$$\mathcal{S}(n) = \begin{cases} 0 & \text{if } n < \varsigma_{\min}, \text{ where } \varsigma_{\min} \text{ is the constant from Lemma 2.5.5,} \\ \mathcal{S}\left(n - \dfrac{\varsigma_6 \lg n}{\lg\lg n}\right) + 1 & \text{for some constant } \varsigma_6 > 0, \text{ otherwise.} \end{cases}$$

The maximum value of κ is $\mathcal{S}(n)$, provided we select ς_6 to be the constant factor hidden by the big-Omega term from Lemma 2.5.5. We note that, following the analysis of Mubayi and Turán [114, Thm. 1], $\varsigma_{\min}$ is fixed such that each biclique removed has at least 2 vertices (i.e., at least a single edge is returned). Thus, $\mathcal{S}(\sqrt{n}) < \sqrt{n}$. Furthermore, since

$$\mathcal{S}(n) < \frac{(n - \sqrt{n}) \lg\lg(\sqrt{n})}{\varsigma_6 \lg(\sqrt{n})} + \mathcal{S}(\sqrt{n}) < \frac{2n \lg\lg n}{\varsigma_6 \lg n} + \sqrt{n}, \qquad (2.5.7)$$

we get $\kappa = O(n \lg\lg n / \lg n)$.

Let $S_\ell = S/(\cup_{i=1}^{\ell} \chi_i)$, for $1 \le \ell \le \kappa$. We define $\mathtt{Map}_\ell(x) = |\{s : s \in S_\ell, s = S(y), y \le x\}|$, i.e., the number of vertices in S_ℓ that are ordered at most x-th in S according to the linear extension $\mathcal{L}$. We next discuss how to compute $\mathtt{Map}_\ell(x)$ it in constant time, for any $1 \le \ell \le \kappa$ and $1 \le x \le n$, using a data structure of size $\mathtt{S\text{-}Rank}(k, k)$ bits, where $k = O(n^2 \lg\lg n / \lg n)$. Define $\mathtt{MapString}_\ell$ to be a bit string, where $\mathtt{MapString}_\ell[x] = 1$ iff $S(x) \in S_\ell$, for $x \in [1, n]$. Overall, these data structures occupy the claimed space bound since $\kappa = O(n \lg\lg n / \lg n)$, and each bit string has length at most n. To compute $\mathtt{Map}_\ell(x)$ we return $\mathtt{rank}_1(\mathtt{MapString}_\ell, x)$.

Consider a vertex s_1 removed during the dense case as part of the biclique χ_k. We store an array of records of size $\Theta(n \lg n)$ bits, denoted `Record`, where:

1. $\mathtt{Record}[s_1]$.`id` is the value k such that $s_1 \in \chi_k$, or ∞ if s_1 was never removed;
2. $\mathtt{Record}[s_1]$.`size` is the number of vertices in χ_k;
3. $\mathtt{Record}[s_1]$.`pos` is the value x such that $\chi_k(x) = s_1$;
4. $\mathtt{Record}[s_1]$.`top` is a bit indicating whether s_1 was in $\hat{A}_2$, when χ_k was removed (we do not need to explicitly store this, as we can infer it by comparing the fields `size` and `pos`);

5. `Record`$[s_1]$`.pnt` is a pointer to data associated with s_1: `EdgeArray`;

6. `Record`$[s_1]$`.dt` indicates the number of vertices in $\hat{A}_1$ immediately after χ_k was removed.

As in Lemma 2.5.2, we store an $\Theta(n \lg n)$ bit array that, in constant time, can return the order of an arbitrary vertex s in the linear extension $\mathcal{L}$, as well as the index of the antichain that contains s in $\tilde{\mathcal{A}}$. Thus, for vertices s_1 and s_2 we can return i, j, x and y such that $A_i, A_j \in \tilde{\mathcal{A}}$, $s_1 \in A_i$, $s_2 \in A_j$, $S(x) = s_1$, and $S(y) = s_2$. The overall claimed space bound holds by Lemma 2.5.6. We present the algorithm Precedence-Query that shows how to use these indices to answer a query to the data structure. The algorithm has several cases, but each case can clearly be computed in $O(1)$ time. Note that we may be required to determine whether the vertex s_1 was in $\hat{A}_1$ when a biclique χ_ℓ was removed in order to interpret the bits stored in `EdgeArray`. This can be done by comparing `Map`$_\ell(x)$ to `Record`$[s_2]$`.dt`. □

We now prove our main theorem:

Proof of Theorem 2.4.1. The theorem follows by combining Lemmas 2.5.2 (with γ set to $\lg n$) and 2.5.7, and by using the data structure of Lemma 1.5.4 to represent the bit strings. Thus, `S-Acc`$(\binom{n}{2}, n^2/\gamma) = O(n^2 \lg\lg n/\lg n)$, and `S-Rank`$(k,k) = O(n^2 \lg\lg n/\lg n)$, since the value $k = O(n^2 \lg\lg n/\lg n)$. This yields an overall space bound of $n^2/4 + O(n^2 \lg\lg n/\lg n)$. Given two vertices s_1 and s_2, we check the index of Lemma 2.5.2 to determine if both s_1 and s_2 were put in the same residual antichain by the Flatten algorithm. If they were, then the index of Lemma 2.5.2 is capable of answering whether $s_1 \prec s_2$. If not, we defer the question to the index of Lemma 2.5.7. □

2.6 Extension to Transitive Reductions and Transitive Relations

The proof of Theorem 2.4.2 is very similar to that of Theorem 2.4.1, with a few minor differences. In this section we describe those differences in detail. First, we observe that there is a simpler way of reducing the height of the transitive reduction, compared to Lemma 2.5.2.

Lemma 2.6.1. *Let $G_r = (S, E_r)$ be the transitive reduction graph of a poset $P = (S, \preceq)$. For a given vertex $s_1 \in S$, let $\Gamma(s_1)$ denote the set of neighbours of s_1 in G_r. The height of the poset, h, is no more than $2n/(\min_{s_1 \in S} |\Gamma(s_1)|)$.*

Proof. Let $\mathcal{C}$ denote a maximum length chain of vertices in S. Consider each vertex $s_1 \in (S \setminus \mathcal{C})$. It cannot be the case that there are edges (s_1, s_2) and (s_1, s_3) in $E_{\mathtt{r}}$, where $s_2, s_3 \in \mathcal{C}$, since that would mean there is a forbidden polygon in $E_{\mathtt{r}}$. By the same argument, there cannot be edges (s_2, s_1) and (s_3, s_1) in $E_{\mathtt{r}}$. Thus, each vertex $s_1 \in (S \setminus \mathcal{C})$ can have edges to at most 2 vertices in $\mathcal{C}$; at most one above and at most one below. $\mathcal{C}$ can therefore have no more than $2n/(\min_{s_1 \in S} |\Gamma(s_1)|)$ vertices. □

The previous lemma can be used as a flattening algorithm for the transitive reduction graph. We recursively remove all vertices that have less than $n/(2 \lg n)$ neighbours in the transitive reduction graph, and record the neighbours using a bit string, compressed by the data structure of Lemma 1.5.4. As a result, the remaining poset has height at most $\lg n$. Let S' denote the set of removed vertices, and suppose we are given two vertices $s_1, s_2 \in S$, and asked to report their relation. If s_1 *or* s_2 are in S', then we can do this in constant time, using an index of size $\mathtt{S\text{-}Acc}(\binom{n}{2}, O(|S'|n/\lg n)) + \Theta(n \lg n)$ bits. Since $S' \leq n$, this index can be used as a replacement for Lemma 2.5.2. The remaining ingredient is to swap Lemma 2.5.4 with the following similar lemma:

Lemma 2.6.2 (Key Lemma for Transitive Reduction Graphs). *Consider the subgraph $G_\Upsilon = G_r(A_i \cup A_j)$ for some $1 \leq i < j \leq h$, and ignore the edge directions so that G_Υ is undirected. Suppose G_Υ contains a balanced biclique subgraph with vertex set χ, and $|\chi| = \tau$: i.e., $G(\chi)$ is a $K_{\tau/2,\tau/2}$. Then there are at most $2^{\tau/2+1} - 1$ ways that the vertices in χ can have edges to each vertex in $S \setminus (A_i \cup A_{i+1})$.*

Proof. Each vertex $v \in S \setminus (A_i \cup A_{i+1})$ is in A_j, where, either $j > i + 1$ or $j < i$. Without loss of generality, consider the case where $j > i + 1$. If any vertex $u \in \chi \cap A_i$ has an edge to v, then *none* of the vertices in $\chi \cap A_{i+1}$ have an edge to v; otherwise there is a forbidden polygon in the transitive reduction graph. Thus, the vertices in $\chi \cap A_{i+1}$ can have edges to v in $2^{\tau/2} - 1$ ways, or the vertices in $\chi \cap A_i$ can have edges to v in $2^{\tau/2} - 1$ ways, or the vertices in χ can have no edges to v. In total, there are $2^{\tau/2+1} - 1$ ways to have edges between v to χ. □

Thus, Lemma 2.5.7 holds for testing whether a relation is in the transitive reduction, and this completes the proof of Theorem 2.4.2. Next, we prove Theorem 2.4.4:

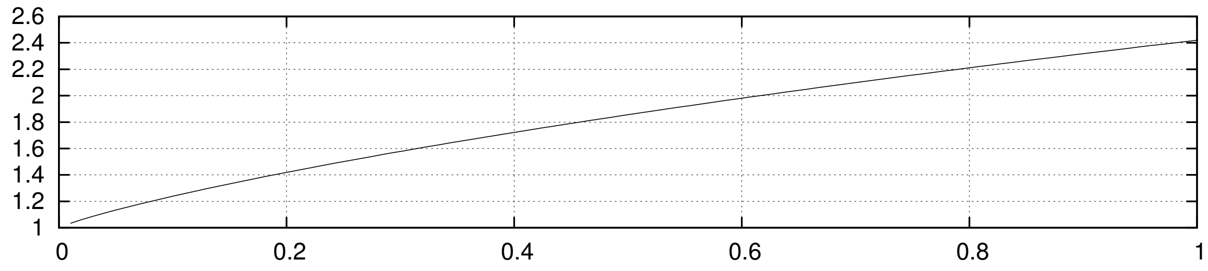


Figure 2.4: A graph of the function $F(\varepsilon_0) = 1.42\varepsilon_0 + 0.42\varepsilon_0 \ln(1/\varepsilon_0) + 1$. *When* $\varepsilon_0 = 1$ *the exponent is* 2.42, *whereas when* $\varepsilon_0 = 1/5$ *(the value selected by Mubayi and Turán [114]) the exponent is less than* 1.419.

Proof of Theorem 2.4.4. Given a transitive binary relation, $T = (S, \preceq)$, we store a bit string $\mathfrak{S}_R$, where $\mathfrak{S}_R[i] = 1$ iff $S(i) \preceq S(i)$. Thus, by using n bits, we can report whether $s_1 \preceq s_1$ in constant time, for any $s_1 \in S$. We define a quasiorder $P = (S, \preceq')$, where $s_1 \preceq' s_2$ if $s_1 \preceq s_2$, for all *distinct vertices* $s_1, s_2 \in S$. We represent the quasiorder P using Corollary 2.4.1. Given $s_1, s_2 \in S$, if $s_1 = s_2$, and $S(i) = s_1$, then we query $\mathfrak{S}_R$ and report "$s_1 \preceq s_2$" iff $\mathfrak{S}_R[i] = 1$, otherwise, we query the representation of P to determine whether s_1 precedes s_2. $\square$

2.7 Issues Relating to Construction

The algorithm Compress-Flat takes $\tilde{O}(n^3)$ time to execute. This follows from the fact that each biclique takes time $\Theta(|E|) = \Theta(n^2)$ to find by Lemma 2.5.3, and the total number of bicliques removed during the construction of the data structure is $O(n \lg\lg n/ \lg n)$ by Inequality 2.5.7. The time to construct the additional indices required by Theorem 2.4.1, described in Lemmas 2.5.7 and 2.5.2, are $O(n^2)$, and are therefore dominated by the cost of executing Compress-Flat.

In this section, we examine the bottleneck of constructing our data structure, Lemma 2.5.3, more carefully. After working through the algorithm, we improve the running time to $O(n^{2+\varepsilon})$ for any constant $\varepsilon \in (0, 1.42]$. Note that this running time assumes we are given the transitive closure graph of P as input. We begin with a more specific restatement of Lemma 2.5.3 (Theorem 2 of Mubayi and Turán [114]):

Lemma 2.7.1. *Let* $q = \lfloor \ln(n/2)/\ln(2en^2/m)\rfloor$, $r = \lfloor qn^2/m \rfloor$, *and* $q' = \lfloor \varepsilon_0 q \rfloor$ *for some constant* $\varepsilon_0 \in (0, 1]$. *Let* $G = (V, E)$ *be a graph where* $|V| = n \geq \varsigma_{\min}$ *for some sufficiently large constant* $\varsigma_{\min}$, *and* $|E| = m$, *where* $8n^{3/2} \leq m \leq n^2/2$. *If we are given a list of the* r *ver-*

tices of highest degree in G, we can find a $K_{q',q'}$ in G in time $\tilde{O}(n^{(1+\varepsilon_0(1+(\frac{1}{\ln(4e)})(1+\ln(\frac{1}{\varepsilon_0}))))}) = O(n^{1.42\varepsilon_0+0.42\varepsilon_0\ln(1/\varepsilon_0)+1})$.

Proof. Given the r vertices of highest degree, the above algorithm FIND-BIPARTITE(G, r, q') can be implemented to run in time $O(\binom{r}{q'}nq')$ [114, p.175][11]. Following *exactly* the same analysis of Mubayi and Turán [114, p.176] (using ε_0 instead of $1/5$), we get:

$$\binom{r}{q'} \leq \left(\frac{1}{\varepsilon_0}\right)^{\varepsilon_0 q} e^{\varepsilon_0 q}\left(\frac{n^2}{m}\right)^{\varepsilon_0 q} = \left(\frac{1}{\varepsilon_0}\right)^{\varepsilon_0 q} (e^q e^{q\ln\left(\frac{n^2}{m}\right)})^{\varepsilon_0} \tag{2.7.1}$$

We examine the previous equation in parts. Since $m < n^2/2$, it is implied that:

$$e^q \leq n^{1/\ln 4e}. \tag{2.7.2}$$

This in turn implies

$$\left(\frac{1}{\varepsilon_0}\right)^{\varepsilon_0 q} = e^{\varepsilon_0 q \ln\left(\frac{1}{\varepsilon_0}\right)} = (e^q)^{\varepsilon_0\ln\left(\frac{1}{\varepsilon_0}\right)} \leq \left(n^{\frac{1}{\ln(4e)}}\right)^{\varepsilon_0\ln\left(\frac{1}{\varepsilon_0}\right)}. \tag{2.7.3}$$

Thus, we get

$$\binom{r}{q'}nq' \leq n^{(1+\varepsilon_0(1+\left(\frac{1}{\ln(4e)}\right)(1+\ln\left(\frac{1}{\varepsilon_0}\right))))}q', \tag{2.7.4}$$

since $q < \ln(n)/\ln\left(\frac{n^2}{m}\right)$. $\square$

Algorithm FIND-BIPARTITE(G, r_1, q_1): where graph $G = (V, E)$ satisfies $|E| \geq 8|V|^{3/2}$.

```
1: $V_1 \leftarrow$ the $r_1$ vertices of highest degree in $G$
2: for all subsets $\chi_1 \subseteq V_1$ with $|\chi_1| = q_1$ do
3:     $\chi_2 \leftarrow \bigcap\{\Gamma(v) \setminus V_1 : v \in \chi_1\}$
4:     if $|\chi_2| \geq q_1$ then
5:         $\chi_2 \leftarrow$ an arbitrary $q_1$ vertex subset of $\chi_2$.
6:         RETURN $\chi_1 \cup \chi_2$
7:     end if
8: end for
```

[11]Note that we have copied the algorithm FIND-BIPARTITE for completeness: we have not modified it in any way (other than notation) from the original source [114, p.175].

We include a graph of the function in the exponent from the previous lemma in Figure 2.4. Using the previous lemma, we prove the following.

Lemma 2.7.2. *Let $G = (V, E)$ be a graph, with $|V| = n$ and $|E| = m$. Let $H \subseteq V$ be a set of vertices such that $|E(H)| \geq (n/\lg n)^2$. A set, $\mathcal{K} = \{\chi_1, ..., \chi_\kappa\}$, can be computed, such that each χ_i is a set of vertices that form a balanced biclique subgraph of H, and each pair in the set $\mathcal{K}$ is vertex disjoint. Let $n_0 = n$, $n_i = |V(G \setminus \bigcup_{j=1}^{\kappa} \chi_j)|$, $m_0 = m$, and $m_i = |E(G \setminus \bigcup_{j=1}^{\kappa} \chi_j)|$. $\mathcal{K}$ also satisfies the following additional properties:*

1. *Either $|E(H \setminus \bigcup_{j=1}^{\kappa} \chi_j)| < (n_\kappa/\lg n_\kappa)^2$ or $|V(H \setminus \bigcup_{j=1}^{\kappa} \chi_j)| < \varsigma_{\min}$, where $\varsigma_{\min}$ is the constant from Lemma 2.7.1: i.e., the number of edges in the subgraph H after removing all the balanced bicliques in $\mathcal{K}$ is less than $(n_\kappa/\lg n_\kappa)^2$, or the number of vertices in H is reduced to a constant.*

2. *$\varsigma_2 \lg n_{i-1}/\lg\lg n_{i-1} \leq |\chi_i| \leq \varsigma_3 \lg n_{i-1}$, where ς_2 and ς_3 are constants, and $1 \leq i \leq \kappa$: i.e., the bicliques have size roughly the logarithmic in the number of vertices remaining the graph G.*

Overall, the time required to compute $\mathcal{K}$ is $O(n^{2+\varepsilon})$, where $\varepsilon > 0$ is a constant, but affects ς_2 and ς_3. That is, the smaller we choose ε to be, the smaller the constants ς_2 and ς_3 become.

Proof. The idea for the proof of this lemma follows came from the following comment by Mubayi and Turán [114, p.177]: "... apart from finding the r largest degree vertices in [the start of the FIND-BIPARTITE algorithm], the running time is actually sublinear in [the number of edges in G]." Thus, finding subsequent bicliques efficiently is as simple as keeping track of the largest degree vertices in G.

We represent G using adjacency lists for each vertex, and represent the adjacency lists using balanced binary search trees. We also store a heap that contains all vertices in H, sorted in descending order of degree.

Let $i = 1$. We extract $r = \lfloor q n_{i-1}^2/m \rfloor$ vertices from the heap; i.e., the r vertices of highest degree in H. We then execute FIND-BIPARTITE(H, r, q'), where $q = \lfloor \ln(n_{i-1}/2)/\ln(2e n_{i-1}^2/m_{i-1}) \rfloor$ and $q' = \lfloor \varepsilon_0 q \rfloor$, for

$$\varepsilon_0 < 4e^{2+W_{-1}\left[-\varepsilon \ln(4e)/(4e^2)\right]}, \tag{2.7.5}$$

where W_{-1} is the function defining the lower branch of the Lambert W-function. This ensures $\varepsilon > \varepsilon_0[1 + (1/\ln[4e])(1 + \ln[1/\varepsilon_0])]$. Recall that FIND-BIPARTITE(H, r, q') finds a biclique subgraph of H. Let χ_i denote the vertex set of this subgraph. We have $\varsigma_2 \lg n_{i-1}/\lg\lg n_{i-1} \leq |\chi_i| \leq \varsigma_3 \lg n_{i-1}$ by Lemma 2.5.5; since the constant is affected by ε_0, the size of the auxiliary data structures will be proportional to $1/\varepsilon_0$ by Inequality 2.5.7. For each $v \in \chi_i$, we update the adjacency lists of all vertices adjacent to v, and also update their keys in the heap to reflect the removal of v. Next, we increment i, then we repeat the whole process on the subset of vertices $H \setminus \bigcup_{j=1}^{i-1} \chi_i$ to get the next χ_i. We stop once $|E(H \setminus \bigcup_{j=1}^{i} \chi_i)| < (n_i/\lg n_i)^2$ or $|H| \leq \varsigma_{\min}$ where $\varsigma_{\min}$ is the constant from Lemma 2.7.1.

The preprocessing time to construct the adjacency lists, compute the degree of all vertices, and store them in a heap is $O(n^2 \lg n)$, since there are at most $\binom{n}{2}$ edges in H. By Lemma 2.7.1, executing FIND-BIPARTITE(H, r, q') requires time $o(n^{1+\varepsilon})$. Updating the heap and structure of the adjacency lists for H after each biclique is removed requires $O(n \lg^2 n)$ time, since each biclique contains $O(\lg n)$ vertices that potentially have edges to $O(n)$ vertices. In order to terminate, κ can be no larger than $O(n \lg\lg n/\lg n)$, since $V(H) \leq n$ and by the recurrence in Inequality 2.5.7. Therefore, the running time is $O(n^{2+\varepsilon})$; all polylogarithmic factors are absorbed by the ε. □

We now prove Theorem 2.4.3:

Proof of Theorem 2.4.3. Computing the antichain decomposition of P is equivalent to topologically sorting the transitive closure graph of P, which takes $O(n^2)$ time. Furthermore, the index of Lemma 2.5.2 can be constructed in $O(n^2)$ time, provided we use the bit string representation of Lemma 1.5.4. We now consider the execution of COMPRESS-FLAT. The sparse cases require time $O(n^2)$ overall since they examine each of the $O(n^2)$ possible edges in constant time per edge. We apply Lemma 2.7 during the dense cases, with parameter equal to some constant $\varepsilon' < \varepsilon$, so that the running time is $O(n^{2+\varepsilon'})$. Since there are at most $\lg n$ merge steps, this means we must apply Lemma 2.7 at most $\lg n$ times, and therefore the total running time of COMPRESS-FLAT is $O(n^{2+\varepsilon'} \lg n) = O(n^{2+\varepsilon})$. □

2.8 Additional Operations

Until now, we have only considered the operation of answering precedence queries on two vertices in the poset. In this section we consider some additional operations.

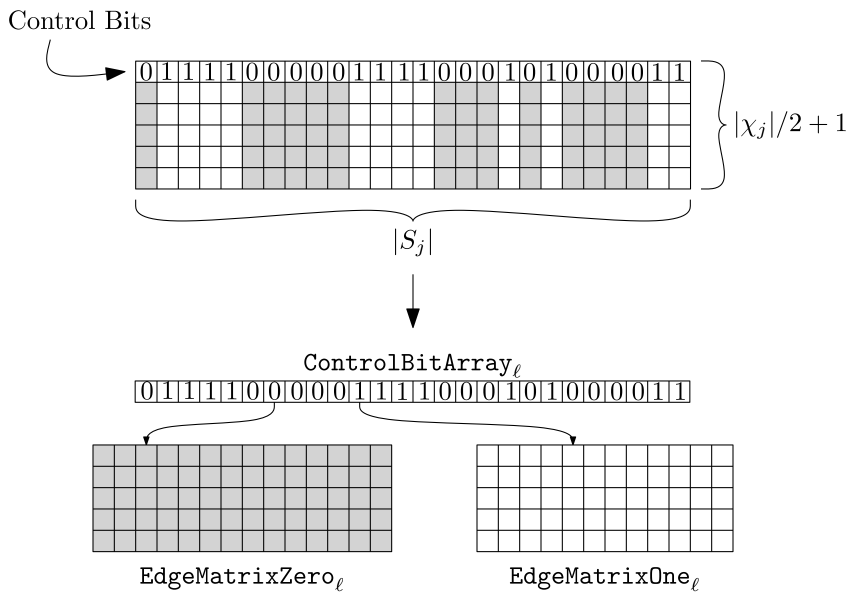


Figure 2.5: We can reinterpret `EdgeArray`$_\ell$ *as a bit string and two matrices* `EdgeMatrixZero`$_\ell$ *and* `EdgeMatrixOne`$_\ell$*.*

2.8.1 Another Representation That Yields More Operations

Let us turn our attention to the array `EdgeArray`$_\ell$, output during the dense case to represent the edges between biclique χ_ℓ and the rest of the vertices in the poset, for a fixed $\ell \in [1, \kappa]$. Recall the discussion following Lemma 2.5.4. We can think of `EdgeArray`$_\ell$ as having two *sections*. As before, let $S_\ell = S \setminus \cup_{i=1}^{\ell} \chi_i$. The first section is a bit string of $|S_\ell|$ *control bits*, each corresponding to an vertex in $s_1 \in S_\ell$, indicating whether an vertex in the *top part* of χ_ℓ has an edge to (or possibly from) s_1. The second section can be interpreted as a $|\chi_\ell|/2 \times |S_\ell|$ bit matrix, `EdgeMatrix`$_\ell$, that, together with the control bits, indicate which edges are present between the vertices in χ_ℓ and those in S_ℓ.

We modify how these two sections are stored. First, we store the control bits in a separate bit string `ControlArray`$_\ell$, where `ControlArray`$_\ell[x]$ is the control bit indicating how χ_ℓ has edges to the vertex $S_\ell(x)$. Second, we divide `EdgeMatrix`$_\ell$ into two bit matrices `EdgeMatrixZero`$_\ell$ and `EdgeMatrixOne`$_\ell$. `EdgeMatrixZero`$_\ell$ is a $|\chi_\ell|/2 \times$ `rank`$_0$(`ControlArray`$_\ell, |S_\ell|$) matrix, consisting of those columns in `EdgeArray`$_\ell$ whose control bit was a zero. The bits in column i of `EdgeMatrixZero`$_\ell$ correspond to the bits in column `select`$_0$(`ControlArray`$_\ell, i$) of `EdgeMatrix`$_\ell$.

`EdgeMatrixOne`$_\ell$ is the analogous array constructed for the columns with control bits set to one. See Figure 2.5 for an illustration, and compare this to the description of the wavelet tree following Lemma 1.5.7.

We now state a lemma, that rewords of a result of Farzan and Munro [56, 54]:

Lemma 2.8.1 (Restatement of Theorem 5.8 [54]). *A $U_1 \times U_2$ bit matrix containing N ones can be represented by a data structure that occupies $\lg \binom{U_1 U_2}{N} + O(U_1 U_2/\sqrt{\lg U})$ bits of space, such that* `access` *can be computed for any entry in constant time, and* `select`$_1$ *can be supported on any row* or *column in constant time.*

We represent each of the matrices `EdgeMatrixZero`$_\ell$ and `EdgeMatrixOne`$_\ell$ using Lemma 2.8.1, and each `ControlArray`$_\ell$ using Lemma 1.5.4, for $1 \le \ell \le \kappa$. Let m be the number of edges in the transitive closure graph of P, and m_ℓ be the number of 1 bits in `EdgeMatrix`$_\ell$. Note that $\sum_{\ell=1}^{\kappa} |n_\ell||\chi_\ell|/2 \le n^2/4$ by Lemma 2.5.7, and $\sum_{\ell=1}^{\kappa} m_\ell \le m$, so the space for the matrices is:

$$\min\left\{n^2/4, \sum_{\ell=1}^{\kappa} \lg \binom{|n_\ell||\chi_\ell|/2}{m_\ell}\right\} + O\left(\sum_{\ell=1}^{\kappa} \frac{n|\chi_\ell|}{\sqrt{\lg n}}\right) \le \min\left\{n^2/4, \lg \binom{\binom{n}{2}}{m}\right\} + O(n^2/\sqrt{\lg n}). \tag{2.8.1}$$

The space for the control bit arrays is at most $O(n^2 \lg\lg n/\lg n)$ by Inequality 2.5.7. Combining this with Lemma 2.5.7, the overall space bound becomes $\min\{n^2/4, \lg \binom{\binom{n}{2}}{m}\} + O(n^2/\sqrt{\lg n})$. Next, we show how to use this new representation to list the predecessors and successors of an arbitrary vertex, proving Theorem 2.4.5:

Proof of Theorem 2.4.5. We store the data structure of Theorem 2.4.1, modified as per the discussed above. Recall that there are two parts to the data structure of Theorem 2.4.1: a "sparse" adjacency matrix $\mathfrak{S}$ which stores the edges removed during merge steps, and a collection of control bit arrays and matrices (discussed above). We store two copies of the adjacency matrix $\mathfrak{S}$, represented using the data structure of Lemma 1.5.4. One is in row-major order, and the other is in column-major order. This allows us to support all three desired operations in constant time, on those edges output during the sparse cases. Alternatively, we could store one copy of the adjacency matrix, represented using Lemma 2.8.1; asymptotically it will make no difference as these adjacency matrices do not dominate the space cost. We now discuss how to support the three types of operations on edges removed during dense cases:

Precedence Queries: Recall the discussion following Lemma 2.5.4, that stated precedence queries on edges output during a dense case check *at most two* bits in $\texttt{EdgeArray}_\ell$, for some $\ell \in [1, \kappa]$. One was a control bit, and the other was a bit in what we now call $\texttt{EdgeMatrix}_\ell$. The control bit for $\texttt{EdgeArray}_\ell[x]$ is now stored in $\texttt{ControlArray}_\ell[x]$, and the other bits are accessible in column $\texttt{rank}_0(\texttt{ControlArray}_\ell, x)$ of $\texttt{EdgeMatrixZero}_\ell$, if $\texttt{access}(\texttt{ControlArray}_\ell, x) = 0$, or column $\texttt{rank}_1(\texttt{ControlArray}_\ell, x)$ of $\texttt{EdgeMatrixOne}_\ell$, otherwise.

Predecessor Queries: First we define the query $\texttt{Map}_\ell^{-1}(y)$ which returns the value x such that $\texttt{Map}_\ell(x) = y$. This query can be implemented by computing $\texttt{select}_1(\texttt{MapString}_\ell, y)$. This can be done in constant time, since each $\texttt{MapString}_\ell$ is represented using the index of Lemma 1.5.4.

For each vertex in $s_1 \in S$ we store a data structure that indicates, for each biclique removed in the dense case, χ_1, ..., χ_κ, whether χ_ℓ contained a vertex that precedes s_1. This data structure is stored as a bit string $\texttt{PredStr}_{s_1}$, where $\texttt{PredStr}_{s_1}[i] = 1$ iff χ_ℓ contains a vertex that precedes s_1. We store $\texttt{PredStr}_{s_1}$ using the data structure of Lemma 1.5.4, so that we can support $\texttt{select}$ queries in constant time on $\texttt{PredStr}_{s_1}$. We also store, for each biclique χ_ℓ, the list of vertices in χ_ℓ sorted in order according to $\mathcal{L}$. These data structures occupy at most $O(\kappa n + n \lg n) = O(n^2 \lg\lg n / \lg n)$ bits by Inequality 2.5.7 and Lemma 1.5.4.

Let $S(x) = s_1$. We perform $\texttt{select}$ queries on $\texttt{PredStr}_{s_1}$, to find a biclique χ_ℓ that contains vertices that precede s_1. There are two cases:

1. *Type-I Predecessors:* If $\ell = \texttt{Record}[s_1].\texttt{id}$ and $\texttt{Record}[s_1].\texttt{top} = 1$, then we report all the vertices in the lower part of χ_ℓ; vertices $\chi_\ell(1), ..., \chi_\ell(|\chi_\ell|/2)$. We also need to report any vertices that precede s_1 and were in the bottom antichain when χ_ℓ was removed. Note that the control bits for vertices that precedes s_1 in this case are all 1, and form a prefix of $\texttt{ControlArray}_\ell$. Thus, we perform $\texttt{select}_1$ queries on row

$$\texttt{Record}[s_1].\texttt{pos} - \texttt{Record}[s_1].\texttt{size}/2 \qquad (2.8.2)$$

   of $\texttt{EdgeMatrixOne}_\ell$. Suppose a 1 appears in column y. If $y \geq \texttt{select}_0(\texttt{ControlArray}_\ell, 1)$, then we are done, as we have exhausted the prefix of the control bits representing vertices in the bottom antichain. Note that $\texttt{select}_0(\texttt{ControlArray}_\ell, 1) = \texttt{Record}[s_1].\texttt{dt}$. Otherwise, we report the vertex $S(\texttt{Map}_\ell^{-1}(y))$, as it precedes s_1. Overall, this takes $O(1)$ time per vertex reported.

2. *Type-II Predecessors:* If $s_1 \notin \chi_\ell$, we examine $\texttt{ControlArray}_\ell[\texttt{Map}_\ell(x)]$. There are two subcases:

   (a) If $\texttt{ControlArray}_\ell[\texttt{Map}_\ell(x)] = 1$, then all vertices in the bottom part of χ_ℓ precede s_1, and we report them in $O(1)$ time per vertex. We also perform $\texttt{select}_1$ queries on column $\texttt{rank}_1(\texttt{ControlArray}_\ell, \texttt{Map}_\ell(x))$ of $\texttt{EdgeMatrixOne}_\ell$ to determine the vertices in the top part of χ_ℓ that precede s_1.

   (b) If $\texttt{ControlArray}[\texttt{Map}_\ell(x)] = 0$, then none of the vertices in the top part of χ_ℓ precede s_1, and we perform $\texttt{select}_1$ queries on column $\texttt{rank}_0(\texttt{ControlArray}_\ell, \texttt{Map}_\ell(x))$ of $\texttt{EdgeMatrixZero}_\ell$ to determine the vertices in the bottom part of χ_ℓ that precede s_1.

   Since each `rank` query in `ControlArray` takes constant time, and each `select` query on columns of the matrices takes constant time, the time to list a single predecessor is constant.

Successor Queries: Similar to the predecessor query, for each vertex in $s_1 \in S$, we store a data structure that indicates, for each biclique removed in the dense case $\chi_1, \ldots, \chi_\kappa$, whether χ_ℓ contained a vertex that *succeeds* s_1. This data structure is represented as a bit string $\texttt{SuccStr}_{s_1}$, where $\texttt{SuccStr}_{s_1}[i] = 1$ iff χ_ℓ contains a vertex that succeeds s_1. We store $\texttt{SuccStr}_{s_1}$ using the data structure of Lemma 1.5.4, so that we can support `select` queries in constant time on $\texttt{SuccStr}_{s_1}$. This occupies at most $O(n^2 \lg\lg n/\lg n)$ bits by Inequality 2.5.7 and Lemma 1.5.4.

First we find all of the bicliques that contain successors of s_1 in the following way.

1. *Type-I Successors:* Let $S(x) = s_1$. We identify all the bicliques that contain successors of s_1 by performing $\texttt{select}_1$ queries on $\texttt{SuccStr}_{s_1}$. Once a biclique χ_ℓ is identified with this property, we perform $\texttt{select}_1$ queries on column $\texttt{Map}_\ell(x)$ of $\texttt{EdgeMatrixOne}_\ell$. Note that $\texttt{Map}_\ell(x) = \texttt{rank}_1(\texttt{ControlArray}_\ell, \texttt{Map}_\ell(x))$, since s_1 must be in A_1 in order to have successors in χ_ℓ. If a 1 appears in row y, then we report $\chi_\ell(y + |\chi_\ell|/2)$; since we store the vertices of χ_ℓ explicitly, this takes $O(1)$ time. Thus, we can report all type-I successors of s_1 in $O(1)$ time per vertex reported.

2. *Type-II Successors:* We examine the biclique χ_ℓ which contains s_1, if one exists. There are two cases:

(a) If s_1 is in the bottom part of χ_ℓ, then we report all the vertices in the top part of χ_ℓ. We scan through row $\texttt{Record}[s_1]\texttt{.pos}$ in $\texttt{EdgeMatrixZero}_\ell$, using $\texttt{select}_1$ queries identify columns in which a 1 occurs. Suppose a 1 occurs in column y. We report the vertex $S(\texttt{Map}_\ell^{-1}(y))$. We also report vertices corresponding to columns in $\texttt{EdgeMatrixOne}_\ell$ that are successors of s_1: i.e., all columns of $\texttt{EdgeMatrixOne}_\ell$ except for the first

$$\texttt{rank}_1(\texttt{ControlArray}_\ell, \texttt{select}_0(\texttt{ControlArray}_\ell, 1)), \tag{2.8.3}$$

as these columns to not contain any successors.

(b) If s_1 is in the top part of χ_ℓ, then we use $\texttt{select}$ queries on row $\texttt{Record}[s_1]\texttt{.pos} - \texttt{Record}[s_1]\texttt{.size}/2$ of $\texttt{EdgeMatrixOne}_\ell$ to report the vertices that are successors of s_1. As before, note that we skip the first

$$\texttt{rank}_1(\texttt{ControlArray}_\ell, \texttt{select}_0(\texttt{ControlArray}_\ell, 1)) \tag{2.8.4}$$

columns of $\texttt{EdgeMatrixOne}_\ell$, as these correspond to *predecessors* rather than *successors*. None of the vertices corresponding to columns in $\texttt{EdgeMatrixZero}_\ell$ are successors of s_1.

Since each vertex reported corresponds to a constant number of `rank` and `select` queries, each vertex can be reported in constant time.

$\square$

2.8.2 Meet, Join, and Boolean Matrix Multiplication

In this section we discuss the meet and join operations, and show that any data structure that supports meet and join operations on a poset can be used to solve boolean matrix multiplication. We note that the theorem presented in this section was already proved implicitly by Bender et al. [18], who consider a problem equivalent to that of computing the join of all pairs of vertices in a poset. Nonetheless, we state the theorem in the context of data structures for representing posets, and present a slightly different proof.

Definition 2.8.1. *Given a poset $P = (S, \preceq)$, and two vertices $s_1, s_2 \in S$, an* existential join *query reports **true** if there exists a vertex $s_3 \in S$ such that $s_1 \preceq s_3$ and $s_2 \preceq s_3$, and **false***

otherwise. The existential meet *query is defined symmetrically; i.e., replace all $\preceq$ with $\succeq$ in the previous sentence.*

For our purposes, we focus only on join queries, since meet is symmetric. Any data structure for representing posets, $\mathcal{D}$, that supports join queries can trivially support existential join queries. We give a reduction from boolean matrix multiplication on $n \times n$ matrices to the problem of constructing $\mathcal{D}$, and answering a series of n^2 existential join queries. This shows that either the construction time for the data structure $\mathcal{D}$ is as large as boolean matrix multiplication, or the time to execute an existential join query is quite expensive.

At the time of writing, the best bound for $n \times n$ boolean matrix multiplication is $\Theta(\mathsf{M}(n)) = \Theta(n^{2.3727})$ [157]. We have shown that our data structure can be constructed in asymptotically less time than this. This implies that the time cost for an existential join query is $\Omega(\mathsf{M}(n)/n^2) = \Omega(n^{0.3727})$, unless our meet/join algorithm improves upon the best known algorithm for boolean matrix multiplication.

Theorem 2.8.1. *Let $\mathbf{A}$ and $\mathbf{B}$ be $n \times n$ boolean matrices, and $N_{\mathbf{A}}$ and $N_{\mathbf{B}}$ be the number of 1 entries in matrices $\mathbf{A}$ and $\mathbf{B}$, respectively. We can compute $\mathbf{C}$, the boolean matrix product of $\mathbf{A}$ and $\mathbf{B}$, in time $O(n^2) + \eta(3n, N_{\mathbf{A}} + N_{\mathbf{B}}) + n^2\Delta$, where $\eta(n, m)$ denotes the cost to construct a data structure on a poset P containing n vertices, and m relations, and Δ denotes the time requires to perform an existential join query on P.*

Proof. We begin by defining a poset $P = (S, \preceq)$ as follows. P is a height 2 poset that has $2n$ source vertices, partitioned into two sets $\mathbf{X_A} \cup \mathbf{X_B}$, and n sink vertices, denoted $\mathbf{Y}$.

Each row, i in $\mathbf{A}$, represents a vertex $x_i \in \mathbf{X_A}$, and each column j in $\mathbf{B}$ represents a vertex $x_j \in \mathbf{X_B}$. The vertices in $\mathbf{Y}$ are denoted $y_1, ..., y_n$. If $\mathbf{A}_{i,k} = 1$ then x_i is below y_k, otherwise it is not. Similarly, if $\mathbf{B}_{k,j} = 1$, then x_j is below y_k, otherwise it is not. This completes the description of P. We note that we can construct both an adjacency matrix and adjacency list representation of P in $O(n^2)$ time, given $\mathbf{A}$ and $\mathbf{B}$ as input. Suppose we construct $\mathcal{D}$, to represent P, and then execute the following series of n^2 existential join queries. The solutions to the queries are recorded in matrix $\mathbf{C}$, and $\mathbf{C}_{i,j} = 1$ if the existential join query for vertices x_i and x_j returns **true**, and $\mathbf{C}_{i,j} = 0$ otherwise. It is clear that $\mathbf{C} = \mathbf{AB}$, since $\mathbf{C}_{i,j} = 0$ unless there exists some k such that $\mathbf{A}_{i,k} = \mathbf{B}_{k,j} = 1$. □

2.9 Simultaneous Representation

Given two vertices s_1 and s_2 in S, we define an *extended precedence* query to be the question, "Is $s_1 \prec s_2$, *and* if so, is the relation in the transitive reduction of P?" As discussed in Section 2.4, we do not have a method for representing a poset using $n^2/4 + o(n^2)$ bits that supports constant time extended precedence queries.

However, we observe that a single adjacency matrix of ternary digits, or *trits* can be used to support such queries. Each of the $\binom{n}{2}$ possible relations (edges) in the poset are recorded as follows: a 0 indicates the edge is not present in either the closure or reduction, a 1 indicates the edge is present in the reduction, and a 2 indicates the edge is present only in the closure. By representing the adjacency matrix of trits using Lemma 1.5.9, the space cost is $\lceil \binom{n}{2} \log_2 3 \rceil + O(n \lg n) \approx 0.7925n^2 + O(n \lg n)$ bits.

We can apply the same trick to the index of Lemma 2.5.2, as well as the sequences output during the execution of the Compress-Flat, in order to achieve the same savings for Theorem 2.4.1. Consider the following replacement for Lemma 2.5.4:

Lemma 2.9.1. *Consider a balanced biclique with vertex set χ, from the subgraph $G_c(A_i \cup A_{i+1})$ (ignoring edge directions so that the graph is undirected) where $1 \leq i < h$. Let $\chi_i = A_i \cap \chi$, $\chi_{i+1} = A_{i+1} \cap \chi$, and let $S' = S \setminus (A_i \cup A_{i+1})$. We can answer extended precedence queries between S' and the vertices in χ using $\lceil |S'|(\log_2 3|\chi|/2) \rceil + |S'|$ bits.*

Proof. Consider, without loss of generality, a vertex $v \in A_j$ where $j > i + 1$. There are two cases. If any vertex $v_0 \in \chi_{i+1}$ exists such that $v_0 \prec v$ then all vertices in χ_i have edges in the transitive closure (and not the transitive reduction) to v. For each edge to v from the vertices in χ_{i+1}, the edge can either not exist, be in the transitive reduction *and* the closure, or just the closure. Thus, we can describe the edges to v from the vertices in χ using a string of trits of length $|\chi|/2$. If no vertex $v_0 \in \chi_{i+1}$ exists such that $v_0 \prec v$, then we apply the same encoding scheme described in the previous case, except that it is stored with respect to the vertices in χ_i. To distinguish between cases requires one extra bit. We store the concatenated string of $|\chi|/2$ trits for all vertices in S' using Lemma 1.5.9, and also separately maintain a sequence of S' bits to indicate which case we are in for each block of trits. $\square$

Plugging the above lemma into the framework for Theorem 2.4.1, we get the following:

Theorem 2.9.1. *There is a data structure that can represent a poset $P = (S, \preceq)$ using:*

$$(\log_2 3)n^2/4 + O(n^2 \lg\lg n/\lg n) \approx 0.39624n^2 + O(n^2 \lg\lg n/\lg n) \ \textit{bits,}$$

and support extended precedence queries on P in constant time.

2.10 Summary and Concluding Remarks

We have presented the first succinct data structure for arbitrary posets. For a poset of n vertices, our data structure occupies $n^2/4 + o(n^2)$ bits and supports precedence queries in constant time. This is equivalent to supporting constant time queries of the form, “Is the edge (s_1, s_2) in the transitive closure graph of P?” With minor modifications, we have shown that the data structure can also be used to answer queries of the form, “Is the edge (s_1, s_2) in the transitive reduction graph of P?”, using the same amount of space.

Currently the best algorithm for computing the transitive closure of a partial order matches that of boolean matrix multiplication for two $n \times n$ matrices. We have shown that, given the transitive closure as input, our data structure can be constructed in asymptotically less time than this. Thus, the bottleneck for constructing our data structure (like most data structures for representing posets) is the computation of the transitive closure.

We have considered several additional operations, such as predecessor and successor queries, as well as meet and join queries. We have shown that our representation can be used to efficiently support the listing of predecessors and successors of an arbitrary vertex in constant time per vertex reported, and discussed why supporting meet and join queries efficiently appears to be difficult.

Finally, we have discussed *extended precedence queries*, which allow us to determine whether a relation (or edge) is present and whether it is in the transitive reduction or closure of P. Our data structure for supporting these kinds of queries is not succinct, but still occupies less space than the standard upper triangular bit matrix.

Chapter 3

Range Majority Queries

3.1 Introduction

The *majority* of a string $\mathfrak{A}[1,n]$ is the character, if any, that occurs more than $n/2$ times in $\mathfrak{A}$. The *majority problem* is to determine whether a given string has a majority, and if so, to report that character. This problem is fundamental to data analysis and has been well studied. Linear time deterministic and randomized algorithms for this problem, such as the Boyer-Moore voting algorithm or the Misra-Gries algorithm [24, 111], have been known since the late 1970s, and are sometimes included in the curricula of introductory courses on algorithms. Interestingly, the Misra-Gries algorithm, which cites and generalizes the result of Boyer and Moore [24], was published almost a decade earlier [24, Section 5.8].

In this chapter, we consider a natural data structure counterpart to this problem. We are interested in designing a data structure that represents a string $\mathfrak{A}$ of length n to answer *range majority queries*: given a query range $\mathcal{R} = [a, b]$ where $1 \le a \le b \le n$, return the majority of the substring $\mathfrak{A}[a, b]$ if it exists. Here we define the majority of a substring $\mathfrak{A}[a, b]$ as the character whose *frequency* in $\mathfrak{A}[a, b]$, i.e., the number of occurrences of the character in $\mathfrak{A}[a, b]$, is more than half of the size of the range $[a, b]$.

We further generalize this problem by defining the *α-majorities* of a substring $\mathfrak{A}[a, b]$ to be the characters whose frequencies are more than $\alpha(b - a + 1)$, i.e., α times the size of the range $[a, b]$, for $0 < \alpha < 1$. Thus an *α-majority query* on string $\mathfrak{A}[1, n]$ can be defined as: given a query range $[a, b]$ where $1 \le a \le b \le n$, return all the α-majorities of the substring $\mathfrak{A}[a, b]$.

3.2 Previous Work

A Comment Regarding Chronology: This chapter is based on results that appear in the papers "Range Majority in Constant Time and Linear Space" [48, 49] (joint work with Stephane Durocher, Meng He, J. Ian Munro, and Matthew Skala), "Finding Frequent Elements in Compressed 2D Arrays and Strings" [66] (joint work with Travis Gagie, Meng He, and J. Ian Munro), and "Dynamic Range Majority Data Structures" [52] (joint work with Amr Elmasry, Meng He, and J. Ian Munro). Several relevant papers have been published both concurrently and subsequently to these papers. Thus, we have divided the related work into previous work (this section) and subsequent and concurrent work (Section 3.4), which follows the section describing the results discussed in this chapter (Section 3.3).

Computing the Mode, Majority, and Plurality of a String: The mode of a multiset $\mathfrak{M}$ of n items can be found in $O(n \lg n)$ time by sorting $\mathfrak{M}$ and counting the frequency of each character. Note that, unlike the *range queries* we focus on in this chapter, these sorting based results find the mode of the *entire string*, and they do it one time. The decision problem of determining whether the frequency m of the mode exceeds one reduces to the *element uniqueness problem*, resulting in a lower bound of $\Omega(n \lg n)$ time in the algebraic decision tree model [17] (a model that generalizes the comparison model). Better bounds have been obtained by parameterizing in terms of m: Munro and Spira [119] and Dobkin and Munro [45] described an $O(n \lg(n/m))$ time algorithm and corresponding lower bound of $\Omega(n \lg(n/m))$ time.

The Boyer-Moore algorithm can be thought of a *streaming algorithm*; i.e., an algorithm that performs some computation using a limited number of passes through the data, and limited extra memory. In this case, the algorithm uses two passes and constant extra words of memory. The first pass of the Boyer-Moore algorithm is used to determine a single element that is a *candidate* for being the majority. The second pass through the string is used to *verify* that the candidate is actually a majority, by explicitly counting its occurrences. Generalizing this, Misra and Gries [111] gave an $O(n(1 + \lg(1/\alpha)))$ time algorithm for computing an α-majority for $\alpha \in (0, 1]$. The Misra and Gries algorithm has been rediscovered several times since 1982, notably by Demaine, López-Ortiz, and Munro [44], and also Karp, Shenker, and Papadimitriou [102]. The problem of computing α-majorities has also recently been studied in the approximate setting, using the term *heavy hitters* instead of α-majorities [40].

The *plurality* of a multiset $\mathfrak{M}$ is a unique mode of $\mathfrak{M}$. That is, every multiset has a mode, but it might not have a plurality. The mode algorithms mentioned above can verify the uniqueness of the mode without any asymptotic increase in time. Numerous results have established bounds on the number of comparisons required for computing a majority, α-majority, mode, or plurality (e.g., [2, 4, 45, 119]).

Source	Problem	Space (words)	Query Time
[130]	mode	$O(n^2/\lg n)$	$O(1)$
[131]	mode	$O(n^2 \lg\lg n/\lg^2 n)$	$O(1)$
[108]	mode	$O(n^{2-2\varepsilon})$	$O(n^{\varepsilon} \lg n)$
[130]	mode	$O(n^{2-2\varepsilon})$	$O(n^{\varepsilon})$
[50]	mode	$O(n)$	$O(\sqrt{n})$
[50]	mode	$O(n)$	$O(\|b-a+1\|)$
[50]	mode	$O(n)$	$O(\sigma)$
[50]	mode	$O(n)$	$O(m)$
[23]	c-approx. mode‡	$O(n/(c-1))$	$O(\log\log_c n)$
[81]	c-approx. mode†	$O(n/(c-1))$	$O(\lg(1/(c-1)))$
[81]	mode	s	$\Omega(\lg n/\lg(sw/n))$

Table 3.1: Word-RAM upper bounds and cell-probe lower bounds for the static range mode problem on a query range $\mathcal{R} = [a, b]$. The size of the alphabet of $\mathfrak{A}$ is denoted by σ, and m denotes the frequency of the mode of $\mathfrak{A}$. For the entry with the †, $1 < c \leq 2$, whereas for the entry with the ‡, $1 < c$. Note that $\varepsilon \in (0, 1/2]$.

Range Mode and Frequency Queries: A *range mode query* for range $[a, b]$ returns a character in $\mathfrak{A}[a, b]$ that occurs at least as frequently as any other character. Krizanc et al. [108] described data structures that provide constant time range mode queries using $O(n^2 \lg\lg n/\lg n)$ space, and $O(n^{\epsilon} \lg n)$ time queries using $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in (0, 1/2]$. They also describe data structures to answer these kinds of queries on trees rather than strings (or arrays). Petersen and Grabowski [131] improved the first bound to constant time and $O(n^2 \lg\lg n/\lg^2 n)$ space. Petersen [130] and Durocher and Morrison [50] improved the second bound to $O(n^{\epsilon})$ time and $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in (0, 1/2]$. Durocher and Morrison [50] described four $O(n)$ space data structures that return the mode of a query range $[a, b]$ in $O(\sqrt{n})$, $O(\sigma)$, $O(m)$, and $O(|b-a+1|)$ time, respectively, where σ denotes the number of distinct characters, and m denotes the frequency of the mode of $\mathfrak{A}$. Greve et al. [81] proved a lower bound of $\Omega(\lg n/\lg(sw/n))$ query time for any range mode query data structure that uses s memory cells of w bits. The opinion of Greve et al. [81] is that the true lower bound is larger than the bound they proved, and they note that improving the bound would require radically new techniques for proving cell-probe lower bounds. Research has also focused on the c-approximate range mode problem [23, 81]: given a range $[a, b]$, return an element that occurs at least $1/c$-times as often as the mode in $\mathfrak{A}[a, b]$. See Table 3.1.

A *k-frequency query* for range $[a, b]$ determines whether any character in $\mathfrak{A}[a, b]$ occurs with frequency *exactly* k. Greve et al. [81] also studied this problem, and noted that when k is fixed a straightforward linear space data structure exists for determining whether any character has frequency *at least* k in constant time. Determining whether any character has frequency *exactly* k requires a different approach. For any fixed $k > 1$, they described how to support range k-frequency queries in $O(\lg n/ \lg\lg n)$ time. When k is given at query time, Greve et al. showed a lower bound of $\Omega(\lg n/ \lg\lg n)$ time applies to either query: exactly k or at least k.

Static α-majority: The best previous result applicable to the range α-majority problem is that of Karpinski and Nekrich [103, Theorem 2]. They studied the problem in a geometric setting, in which points on the real line are assigned colours, and the goal is to find *τ-dominating colours*: that is, given a range $\mathcal{R}$, return all the colours that are assigned to at least a τ fraction of the points in $\mathcal{R}$. If we treat each character in a string $\mathfrak{A}[1, n]$ as a coloured point in a bounded universe $[1, n]$, their data structure can be used to represent $\mathfrak{A}$ in $O(n/\alpha)$ space to support range α-majority queries in $O((\lg\lg n)^2/\alpha)$ time. See Table 3.2 for a summary. Note that if the number of colours is a fixed constant, then it is possible to answer queries using $O(\lg\lg n)$ time, by storing a static y-fast trie for each colour [156], and performing range counting queries for each colour. Thus, we assume the number of colours is $\omega(1)$.

A trivial method, that is much simpler than any of approaches discussed so far is that of *random sampling*. To answer a range α-majority query, we could sample ℓ characters in the specified range, and use the samples to guess at the answer. This *Monte Carlo* based approach yields the following lemma:

Lemma 3.2.1. *Suppose ℓ characters are sampled from $\mathcal{R} = [a, b]$, independently and uniformly at random, and let $\mathfrak{M}$ be the multiset of sampled characters. For a given character j, let $|\mathfrak{M}_j|$ denote the frequency of j in $\mathfrak{M}$. The probability that there exists a character j such that j is an*

Source	Exact	Input	Space (words)	Query
[103]	Yes	real	$O(n/\alpha)$	$O(\lg n \lg\lg n/\alpha)$
[103]	Yes	real	$O(n \lg\lg n/\alpha)$	$O(\lg n/\alpha)$
[103]	Yes	integer	$O(n/\alpha)$	$O(\lg\lg n \lg\lg u/\alpha)$
[103]	Yes	string	$O(n/\alpha)$	$O((\lg\lg n)^2/\alpha)$
[103]	Yes	d-dim	$O((n \lg^{d-1} n)/\alpha)$	$O(\lg^d n/\alpha)$
Random sampling	No	string	$O(n)$	$O(\lg\lg\sigma/\alpha)$

Table 3.2: Word-RAM results for the static range α-majority problem. For the "input" column: String *refers to an string $\mathfrak{A}$ of characters (or colours) of length n. The characters in $\mathfrak{A}$ can be considered to be ordered by their indices in $\mathfrak{A}$.* Integer *refers to points on a line with word-sized integer coordinates, i.e., integers in the range $[1, u]$, where $u < 2^w$.* Real *refers to points on the real line where the only assumption is that we can compare two points in constant time to determine their order, and that each point occupies only a constant number of words ($O(w)$ bits).* d-dim *refers to points in $d \geq 2$ dimensions (d is a constant) whose coordinates are comparable in constant time.*

α-majority for $\mathcal{R}$ and $|\mathfrak{M}_j| \leq (\alpha\ell)/10$ is no more than

$$\frac{1}{\alpha}\exp\left(-\frac{\alpha\ell\left(\frac{9}{10}\right)^2}{2}\right). \tag{3.2.1}$$

Proof. There are no more than $1/\alpha$ characters that are α-majorities for $\mathcal{R}$. Then for a fixed character j' that is an α-majority, the probability that we sample j' less than $(\alpha\ell)/10$ times is

$$\exp\left(-\frac{\alpha\ell\left(\frac{9}{10}\right)^2}{2}\right) \tag{3.2.2}$$

by Chernoff bounds [113, Theorem 4.5 (2)]. Since there are at most $1/\alpha$ such α-majorities, we can apply the union bound [113, Lemma 1.2] to complete the lemma. □

Using the previous lemma, it is trivial to prove the following theorem:

Theorem 3.2.1. *Let $\mathfrak{A}$ be a string of length n. Assume that a character can be sampled— independently and uniformly at random— from an arbitrary range $\mathcal{R} = [a, b]$ in $\mathfrak{A}$ in $O(1)$ time. There is a data structure that occupies $\Theta(n \lg n)$ bits that, given an arbitrary range $\mathcal{R}$ and $\alpha \in (0, 1)$, can return a set $\mathfrak{L}$ of characters in $O(\lg\lg\sigma/\alpha)$ time with the following properties:*

1. *for all $j \in \mathfrak{L}$, j is an α-majority in $\mathcal{R}$, and,*

2. *the probability that there is a character* $j_0 \notin \mathfrak{L}$ *that is an* α*-majority for* $\mathcal{R}$ *is* $1/(\alpha\Theta(\lg n))$.

Proof. In addition to the string $\mathfrak{A}$, we store a copy of the data structure of Lemma 1.5.8, which can support $\mathtt{rank}_j(\mathfrak{A}, i)$ queries in $O(\lg\lg\sigma)$ time. Given a query range $\mathcal{R}$, we sample $\ell = \lg\lg\sigma/\alpha$ characters from $\mathcal{R}$, in $O(1)$ time per sample. Using an auxiliary array of size $\sigma\lg n$ bits, we can count the number of occurrences of each character that was sampled, and determine a *set of candidates* that contains all characters that were sampled at least $\lg\lg\sigma/10$ times. For each candidate character j we perform rank queries to determine its actual number of occurrences in $\mathcal{R}$. If j appears more than $|\mathcal{R}|\alpha$ time, we report it. □

Thus, it is trivial to improve upon previous deterministic results if we allow some probability of failing to return *all* the α-majorities. As we will see, it is possible to achieve the same query time as Theorem 3.2.1 (and even improve upon it), without the need for a Monte Carlo query algorithm.

Dynamic α-majority: In the dynamic case, consider the problem of storing a set of coloured points on a line, subject to insertions and deletions. It turns out that handling updates efficiently comes at the cost of increasing the query time to $\omega(1)$. Husfeldt and Rauhe [97, Proposition 11] proved a lower bound of $\Omega(\lg n/\lg\lg n)$ query time for any dynamic range α-majority data structure that has $O(\mathtt{polylog}(n))$ update time, when $1/\alpha = O(1)$; we give details in Section 3.9.1. If the number of colours is a fixed constant, and the points have integer coordinates, then it is trivial achieve $\Theta(\lg n/\lg\lg n)$ query and amortized update time—which is optimal for constant α—by using existing dynamic one-dimensional range counting data structures [7].

For a summary of upper bounds, see Table 3.3. Karpinski and Nekrich [103] considered a dynamic version of the range α-majority problem, again using the terminology τ-dominating colours. As in the static case, they make use of a tree based decomposition of the point set. Lai et al. [109] also studied the problem, but in the approximate setting. That is, their query returns all the α-majorities, but may also return false positive $(\alpha(1-\varepsilon))$-majorities, for some fixed $\varepsilon \in (0, 1)$. They presented two data structures based on sketching techniques [40] that require $1/\alpha = O(1)$. In the case when the total number of colours assigned to the points is such that $\lg\sigma = \Theta(\lg n)$, their data structures are outperformed by Karpinski and Nekrich's data structure, which returns the *exact* solution.

Source	Exact	Input	Space (words)	Query	Insert	Delete
[103]	Yes	real	$O\left(\frac{n}{\alpha}\right)$	$O\left(\frac{\lg^2 n}{\alpha}\right)$	$O\left(\frac{\lg^2 n}{\alpha}\right)$	$O\left(\frac{\lg^2 n}{\alpha}\right)*$
[103]	Yes	real	$O\left(\frac{n \lg n}{\alpha}\right)$	$O\left(\frac{\lg n}{\alpha}\right)$	$O\left(\frac{\lg n}{\alpha}\right)*$	$O\left(\frac{\lg n}{\alpha}\right)*$
[103]	Yes	d-dim	$O\left(\frac{n \lg^{d-1} n}{\alpha}\right)$	$O\left(\frac{\lg^{d+1} n}{\alpha}\right)$	$O\left(\frac{\lg^{d+1} n}{\alpha}\right)$	$O\left(\frac{\lg^{d+1} n}{\alpha}\right)*$
[109]†	No	real	$O(nh)$	$O(h \lg n)$	$O(h \lg n)$	$O(h \lg n)$
[109]†	No	real	$O(n)$	$O\left(h \lg n + h^2\right)$	$O\left(h \lg n + h^2\right)$	$O\left(h \lg n + h^2\right)$
[97]‡	—	—	—	$\Omega\left(\frac{\lg n}{\lg((t_i+t_d) w \lg n)}\right)$	t_i	t_d

Table 3.3: Word-RAM results for the dynamic α-majority problem. The column "Exact" indicates whether the query is guaranteed to yield the correct result, or whether there is a probability of receiving either an incomplete list of α-majorities, or a list containing false positives. For the entries marked with "" the running times are amortized. For the entries marked with †, the value $h = O((\lg \sigma/\varepsilon) \lg(\lg \sigma/(\alpha\delta)))$ depends on the threshold α, the approximation factor ε, the total number of colours σ, and the probability of failure δ. For sources marked with a ‡, the bound follows assuming $1/\alpha = O(1)$.*

Other Range Queries on Strings (and Arrays) For a comprehensive treatment of *range array queries* see the recent survey by Skala [141]; we mention only a few key results here.

The *range median* and more general *k-selection* problems have been studied heavily in recent years, in both the static and dynamic setting [23, 108, 89, 130, 131, 68, 75, 28, 27, 101, 91, 33]. Given an array $\mathfrak{A}$, the k-selection problem asks us to return the k-th smallest character stored in an arbitrary range of the string; median is the special case where k is half the length of the range. Matching upper and lower bounds for linear space static data structures have been found [101, 33]. For exact static range medians in constant time, there have been several iterations of near-quadratic space data structures [108, 130, 131]. We note that Gagie et al. [68] point out that the static problem can be solved trivially using a wavelet tree (Lemma 1.5.7) in $O(\lg \sigma)$ time, where σ is the number of distinct characters in the input string n. We mention k-selection since, as we will see later, it can be used to find α-majorities. Note that a special case of this problem in which $k = 1$—also called the *range minimum* problem—has been studied extensively [47].

More broadly related are the problems of *orthogonal range counting* and *orthogonal range reporting*, in which we wish to count the number of (resp. report the) points stored in an arbitrary orthogonal rectangle on an $n \times n$ grid [34, 22]. There are other data structure problems that deal with coloured points. In *coloured range reporting problems* [84], we are interested in reporting the set of distinct colours assigned to the points contained in an axis-aligned rectangle. Similarly, in *the coloured range counting problem* we are interested in returning the number of such distinct colours. Gupta et al. [84], Bozanis et al. [25], and, more recently, Gagie et al. [68] and Gagie and

Kärkkäinen [67] studied these problems and presented several interesting results.

Finally, a problem that is related—but structurally distinct from the range majority problem—is the problem the *α-significant presence colour problem* [42]. That is, preprocess a set of points on a line, such that, given a query range $\mathcal{R}$, we can efficiently return all of the colours c, such that c appears at least αt_c times in Q, where t_c is the total number points assigned colour c in the entire data set. The colours returned for a query range are called α-significant colours.

3.3 Our Contributions

The results in this chapter hold under the word-RAM model, with word size $w = \Theta(\lg n)$ bits. We give a brief outline of each section:

In Section 3.5 we present a data structure for answering range majority queries in the static case. It occupies $O(n \lg n)$ bits, i.e., a linear number of words, and answers range majority queries in constant time. The data structure is conceptually simple and based on the idea that, for query ranges above a certain size threshold, only a small set of *candidate* characters need be considered in order to determine the majority. In order to verify the frequency of these characters efficiently, we present a novel decomposition technique that uses wavelet trees (see Lemma 1.5.7).

In Section 3.6 we generalize our data structure to answer range α-majority queries in the static case, for any fixed $\alpha \in (0, 1]$. Note that although α is fixed at construction time, it is not necessarily a constant. For example, setting $\alpha = 1/\lg n$ is permitted. Our structure occupies $O(n \lg n(\lg(1/\alpha) + 1))$ bits and answers range α-majority queries in $O(1/\alpha)$ time. In order to generalize our data structure when $1/\alpha$ is large, i.e., when $1/\alpha = \omega(1)$, we make use of *batched* queries over wavelet trees. In light of the lower bounds for range mode and k-frequency, it is interesting that a linear space data structure can answer range α-majority queries in constant time for fixed constant values of α.

In Section 3.7 we discuss the parameterized problem in the static case, where we are asked to find the β-majorities where $\beta \in (\alpha, 1]$ is specified at query time. We show that a minor modification to our α-majority data structure can return the β-majorities for any $\beta \in (\alpha, 1]$ in time $O(1/\beta)$. We also discuss further trade-offs for the range α-majority problem, and refine our space analysis to consider the zeroth order entropy of the string $\mathfrak{A}$. We present two data structures: one that can answer β-majority queries in $O(1/\beta)$ time, that occupies $O(n(\min\{\lg(1/\alpha), \mathsf{H}_0(\mathfrak{A})\} +$

$1)\lg n)$ bits of space, and another that answers α-majority queries in $O(\lg\lg\sigma/\alpha)$ time, and occupies $O(n(\mathsf{H}_0(\mathfrak{A})+1))$ bits of space.

In Section 3.8 we discuss the applications of our data structure for range α-majority queries to the coloured range searching problems for *static* d-dimensional point sets, defined by Karpinski and Nekrich [103], and present improved data structures for these problems. In particular, for any $d \geq 2$, we present data structures that can answer range α-majority queries on a set of n points in d-dimensions. The data structures occupy $O(n\lg^{d-1} n)$ words of space, and take time $O(\lg^d n/\alpha)$ time to report the α-majority colours in an orthogonal query range.

Finally, in Section 3.9 we examine the dynamic geometric problem, where we wish to store a set of coloured points on a line, answer range α-majority queries on the points (treating colours as characters), and also support insertions and deletions. We present a linear space—$O(n\lg n)$ bit— data structure that supports range α-majority queries in $O(\lg n/\alpha)$ time, and insertions and deletions in $O(\lg n/\alpha)$ amortized time. When the coordinates of points are integers drawn from a bounded universe $[1,u]$, we improve the query time to $O(\lg n/(\alpha\lg\lg n))$, which is optimal (i.e., asymptotically matches a known cell-probe lower bound) for constant values of α and data structures with polylogarithmic update. We extend this result to d-dimensions, for $d \geq 2$, using range trees as in the static case, yielding an $O(n\lg^{d-1} n)$ word data structure that can answer queries in $O(\lg^{d+1} n/\alpha)$ time, and perform updates in $O(\lg^d n/\alpha)$ amortized time. This improves the update time of the previous solution [103], as well as the space bound (by a factor of α).

3.4 Subsequent (and Concurrent) Work

Independently of the results presented in this chapter, Wei and Yi [154] studied the approximate version of the α-majority problem, and showed that if false positives are allowed, α-majority queries can be answered in $O(\lg n + 1/\alpha)$ time on a set of coloured points on a line. Their approach is also based on a tree decomposition, where the nodes of a tree store sketching data structures [40], similar to the approach of Lai et al. [109]. In the dynamic case, the cost of updates is $O(\mu\lg n\lg(1/\varepsilon))$ amortized time, where μ is the cost of updating the sketch stored in a tree node. We note that this result was obtained independently of ours, and that both our techniques and the main technique they develop, called *exponential decomposability*, are similar. By combining Theorem 4 of their paper with standard range counting data structures, it is not difficult to get a dynamic data structure that occupies linear space, answers queries in $O(\lg n/\alpha)$ time, and supports updates in $O((\lg n \lg(1/\alpha))/\alpha)$ amortized time for the non-approximate version of the problem that we study. We achieve a slightly better update time, and also improve the query time for the case when the point's coordinates are integers. On the other hand, their structure is part of a more general framework that supports other kinds of aggregate queries.

See Tables 3.4 and 3.5 for a summary of the most recent results for the exact versions of the range α-majority problem.

Source	Input	Space (w-bit words)	Query
Thm. 3.6.1	string	$O(n(\lg(1/\alpha)+1))$	$O(1/\alpha)$
Thm. 3.7.1	string	$O(n(\min\{\mathsf{H}_0(\mathfrak{A}), \lg(1/\alpha)\}+1))$	$O(1/\beta)$
Thm. 3.7.2	string	$O((n(\mathsf{H}_0(\mathfrak{A})+1)+\sigma\lg n)/w)$	$O((\lg\lg\sigma)/\alpha)$
[14]	string	$O(n)$	$O(1/\alpha)$
[14]	string	$O(n\lg\lg\sigma)$	$O(1/\beta)$
[14]	string	$O(n)$	$O((1/\beta)\lg\lg(1/\beta))$
[14]	string	$(n(\mathsf{H}_0(\mathfrak{A})+1)(1+o(1)))/w$	$O((1/\beta)\lg\lg\sigma)$
Thm. 3.8.2	d-dim points	$O(n\lg^{d-1} n)$	$O(\lg^d n/\alpha)$
[122]	2-dim grid	$O(n)$	$O(\lg^3 n/\alpha)$
[122]	2-dim grid	$O(n\lg n)$	$O(\lg^2 n/\alpha)$
[122]	2-dim grid	$O(n\lg n)$	$O(\lg^2 n/\beta)$
[155]	2-dim points	$O(n\lg^{\varepsilon} n\lg(1/\alpha))$	$O(\lg n/\alpha)$

Table 3.4: Up to date Word-RAM results for the static α-majority problem. β denotes a parameter specified at query time, rather than α which is fixed at construction time. We use ε to denote an arbitrarily small constant larger than 0. "2-dim grid" refers to points on an $n \times n$ integer grid, whereas "2-dim points" refers to two dimensional points, but does not assume the points have integer coordinates.

Navarro and Russo [123] (Nekrich was later added to the journal version [122]), studied the problem of answering range queries on $n \times n$ grids; i.e., two dimensional point sets. They reduce the problem of answering α-majority queries to that of range selection queries: selecting the i-th smallest character (or colour) in the range, for $i \in \{1, \alpha\sigma, 2\alpha\sigma, ..., \sigma\}$ will yield all α-majorities. Thus, if a range selection query takes `T-Sel` time, then they can return the α-majorities in time $O($`T-Sel`$/\alpha)$.

For both the static and dynamic *one-dimensional case* of the range α-majority problem this strategy is inferior to our approach. In the static case this approach yields a α-majority query time of $O((1/\alpha)(\lg n/\lg\lg n))$ [28], and in the dynamic case this yields an α-majority query time of $O((1/\alpha)(\lg n/\lg\lg n)^2)$ [91, 92], and an $O((\lg n/\lg\lg n)^2)$ amortized update time. However, for the two dimensional static case, their approach yields several trade-offs. They get a linear space

Source	Input	Space (words)	Query	Insert	Delete
Thm. 3.9.1	points	$O(n)$	$O\left(\frac{\lg n}{\alpha}\right)$	$O\left(\frac{\lg n}{\alpha}\right)$ *	$O\left(\frac{\lg n}{\alpha}\right)$ *
Thm. 3.9.2	integer	$O(n)$	$O\left(\frac{\lg n}{(\alpha \lg\lg n)}\right)$	$O\left(\frac{\lg n}{\alpha}\right)$ *	$O\left(\frac{\lg n}{\alpha}\right)$ *
Thm. 3.9.3	string	$O(n)$	$O\left(\frac{\lg n}{(\alpha \lg\lg n)}\right)$	$O\left(\frac{\lg n}{\alpha}\right)$ *	$O\left(\frac{\lg n}{\alpha}\right)$ *
Thm. 3.9.4	d-dim points	$O\left(n \lg^{d-1} n\right)$	$O\left(\frac{\lg^{d+1} n}{\alpha}\right)$	$O\left(\frac{\lg^{d} n}{\alpha}\right)$ *	$O\left(\frac{\lg^{d} n}{\alpha}\right)$ *
[122]	2-dim grid	$O(n \lg n)$	$O\left(\frac{\lg^{d+1} n}{\beta \lg\lg n}\right)$	$O\left(\frac{\lg^{d+1} n}{\lg\lg n}\right)$	$O\left(\frac{\lg^{d+1} n}{\lg\lg n}\right)$
[53]	d-dim points	$O\left(n \lg^{d-1} n\right)$	$O\left(\frac{\lg^{d} n}{\alpha}\right)$	$O\left(\frac{\lg^{d} n}{\alpha}\right)$ *	$O\left(\frac{\lg^{d} n}{\alpha}\right)$ *

Table 3.5: Up to date Word-RAM results for the dynamic α-majority problem. For the entries marked with "" the running times are amortized. β denotes a parameter specified at query time, rather than α which is fixed at construction time.*

(in words) data structure that takes $O(\lg^3 n/\alpha)$ time to query, or an $O(n \lg n)$ word data structure that takes $O(\lg^2 n/\alpha)$ time to query (and thus, asymptotically matches our result). They also get a slight improvement by describing an $O(n \lg n)$ word data structure that takes $O(\lg^2 n/\beta)$ time to query, where the data structure has the additional feature that it is parameterized. In the two dimensional dynamic case, their data structure occupies asymptotically the same space bound as ours, but has a slightly better query time (which is also parameterized), and slightly worse update time (though the update operations are deamortized).

For two-dimensional range α-majority queries, Wilkinson [155] showed how to use approximate range counting in order to improve the query time for α-majority queries on 2-dimensional point sets by a $(\lg n)$-factor, and also improved the space bound by a $(\lg^{\varepsilon} n)$-factor, for any $\varepsilon > 0$. Elmasry et al. [53] showed that the same technique could be applied to the dynamic case, yielding a $(\lg n)$-factor improvement in query time over the result we present here. We also mention that the related problem of supporting range α-majority queries over a two-dimensional array has been considered by Gagie et al. [66].

A very recent paper of Belazzougui, Gagie, and Navarro [14] shows how to improve the space bound for non-constant α-majority queries to linear, regardless of α. Furthermore, they give additional improvements for the parameterized case, subsuming all the results we present in this chapter.

Other related subsequent work: For the range mode problem, Chan et al. [31] gave a linear space data structure that can answer range mode queries in $O(\sqrt{n/\lg n})$ time. As before, define

$\mathsf{M}(n)$ to be the time required to compute the boolean matrix product of two $n \times n$ matrices. Chan et al. showed that any data structure that can solve n arbitrary range mode queries on a string of size n can be used to compute the boolean matrix product of two $\sqrt{n} \times \sqrt{n}$ matrices. This shows that any range mode data structure must take $O(\mathsf{M}(\sqrt{n})) = O(\sqrt{n}^{2.3727}) = O(n^{1.1864})$ time [157] to construct, or that arbitrary queries must take time $\Omega(\mathsf{M}(\sqrt{n})/n) = \Omega(n^{0.1864})$, unless the data structure simultaneously yields an improved algorithm for boolean matrix multiplication.

Finally, Chan et al. [32] introduced and studied the related problem of *α-minority queries*: an α-minority of a query range is any character that appears less than an α fraction of the number of characters in the range. They also give an alternative data structure for computing β-majorities in the parameterized case. We note that Belazzougui, Gagie, and Navarro [14] also present improved data structures for the α-minority problem.

3.5 Static Range Majority Data Structure

In this section we describe a static linear space data structure that supports range majority queries in constant time. To provide some intuition, suppose we partition the input string $\mathfrak{A}[1, n]$ into four contiguous equally-sized blocks. If we are given a query range that contains one of these four blocks, then it is clear that a majority for this query must have frequency strictly greater than $n/8$ in $\mathfrak{A}$. Thus, at most seven characters need be considered when computing the majority for queries that entirely contain one of these four fixed blocks.

Of course, not all queries contain one of these four blocks. Therefore, we decompose the string into *multiple levels* in order to support arbitrary queries (Sections 3.5.1 and 3.5.2). Using this decomposition in conjunction with succinct data structures [98], we design a linear space data structure that answers range majority queries in constant time (Section 3.5.3). The data structure works by counting the frequency of a constant number of *candidate* characters in order to determine the majority for a given query.

From this point on we make the assumption that the characters stored in the input string $\mathfrak{A}$ are drawn from the alphabet $[1, \sigma]$, where $\sigma \leq n$ is the number of distinct characters in $\mathfrak{A}$. If this is not the case, then we can apply the well known technique of *reduction to rank space* [34, 5] as a preprocessing step, and store an auxiliary array of size σ to invert a character in $[1, \sigma]$ to its original value. This preprocessing step takes $O(n \lg \sigma)$ time using a balanced binary search tree.

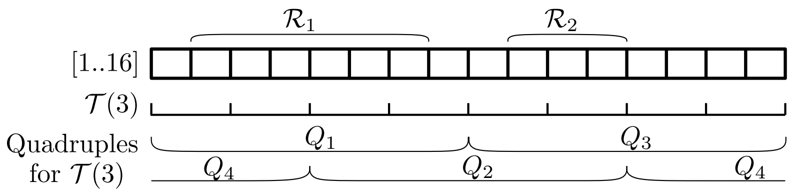


Figure 3.1: An example where $n = 16$. Blocks in $\mathcal{T}(3)$ have size 2, and each of the 4 quadruples contain 4 blocks. Query ranges $\mathcal{R}_1$ and $\mathcal{R}_2$ are associated with quadruples $\mathcal{Q}_1$ and $\mathcal{Q}_3$ respectively.

3.5.1 Quadruple Decomposition

The first stage of our decomposition is to construct a notional complete binary tree $\mathcal{T}$ over the range $[1, n]$, in which each node represents a subrange of $[1, n]$. Let the root of $\mathcal{T}$ represent the entire range $[1, n]$. For a node corresponding to range $[a, b]$, its left child represents the left half of its range, i.e., the range $[a, \lfloor (a + b)/2 \rfloor]$, and its right child represents the right half, i.e., the range $[\lfloor (a + b)/2 \rfloor + 1, b]$. For simplicity, we can assume that n is a power of 2, since, if it is not, we can pad the string with extra characters until it is. This padding will at most double the size of n. Each leaf of the tree represents a range of size 1, which corresponds to a single character of the string $\mathfrak{A}$. We refer to ranges represented by the nodes of $\mathcal{T}$ as *blocks*. Note that the tree $\mathcal{T}$ is for illustrative purposes only, so we need not store it explicitly.

The tree $\mathcal{T}$ has $\lg n + 1$ levels, which are numbered 0 through $\lg n$ from top to bottom. For each level ℓ, $\mathcal{T}$ partitions $\mathfrak{A}$ into 2^ℓ blocks of size $n/2^\ell$. Let $\mathcal{T}(\ell)$ denote the set of blocks at level ℓ in $\mathcal{T}$.

The second stage of our decomposition consists of arranging adjacent blocks within each level $\mathcal{T}(\ell)$, $2 \leq \ell \leq \lg n$, into groups. Each group consists of four blocks and is called a *quadruple*. Formally, we define a quadruple $\mathcal{Q}_q$ to be a range $[a, b]$ at level $\ell \geq 2$ of size $4n/2^\ell$, where $a = 2(q - 1)n/2^\ell + 1$ and $b = 2(q - 1)n/2^\ell + 4n/2^\ell$, for $1 \leq q \leq 2^{\ell-1} - 1$. In other words, each quadruple at level ℓ contains exactly four consecutive blocks, and its starting position is separated from the starting position of the previous quadruple by two blocks. To handle border cases, we also define an extra quadruple $\mathcal{Q}_{2^{\ell-1}}$ which contains both the first two and last two blocks in $\mathcal{T}(\ell)$. Thus, at level ℓ there are $2^{\ell-1}$ quadruples, and each block in $\mathcal{T}(\ell)$ is contained in two quadruples. These definitions are summarized in Figure 3.1.

3.5.2 Candidates

Based on the decomposition from the previous section, we observe the following:

Observation 3.5.1. *For every query range $\mathcal{R}$ there exists a unique level ℓ such that:*

1. *$\mathcal{R}$ contains at least one and at most two consecutive blocks in $\mathcal{T}(\ell)$; and,*
2. *if $\mathcal{R}$ contains two blocks, then the nodes representing these blocks are not siblings in the tree $\mathcal{T}$.*

Let $\mathcal{Q}$ be a quadruple consisting of four consecutive blocks, B_1 through B_4 from $\mathcal{T}(\ell)$, where ℓ is the level referred to in the previous observation. We *associate* $\mathcal{R}$ with $\mathcal{Q}$ if $\mathcal{R}$ contains B_2 or B_3; for convenience we also say that $\mathcal{R}$ is associated with level ℓ. Note that $\mathcal{R}$ may contain both B_2 and B_3; see $\mathcal{R}_1$ in Figure 3.1. We proved the following lemma in the discussion at the beginning of Section 3.5:

Lemma 3.5.1. *There exists a set $\mathfrak{L}$ of at most 7 characters such that, for any query range $\mathcal{R}$ associated with quadruple $\mathcal{Q}$, the majority for $\mathcal{R}$ is in $\mathfrak{L}$.*

For a quadruple $\mathcal{Q}$, we define the set of *candidates* for $\mathcal{Q}$ to be the characters in $\mathfrak{L}$. Next, we describe how the sets of candidates can be computed efficiently.

Lemma 3.5.2. *The sets of candidates for all the quadruples can be identified in $O(n \lg n)$ time.*

Proof. Recall that the characters in $\mathfrak{A}$ are drawn from the alphabet $\{1, \ldots, \sigma\}$, where $\sigma \leq n$. We can count the frequencies of all the characters in quadruple $\mathcal{Q}$ in $O(|\mathcal{Q}|)$ time, in a single pass over the characters in $\mathcal{Q}$, using an auxiliary array of size σ. When the count of a character exceeds $|\mathcal{Q}|/8$, we add it to the set of candidates for $\mathcal{Q}$. This implies that the sets of candidates for all the quadruples in all of the $\lg n + 1$ levels of $\mathcal{T}$ can be found in $O(n \lg n)$ time. $\square$

3.5.3 Data Structures for Counting

We now describe the data structures stored for each level ℓ of the tree $\mathcal{T}$, for $2 \leq \ell \leq \lg n$. Given a quadruple $\mathcal{Q}_q$ in level ℓ, for $1 \leq q \leq 2^{\ell-1}$ we store the set of (at most seven) candidates for $\mathcal{Q}_q$ in a list $\mathfrak{L}_q$. Let Υ_q be a string of length $|\mathcal{Q}_q|$, where the i-th character in Υ_q is f iff the i-th character in $\mathcal{Q}_q$ is $\mathfrak{L}_q[f]$ for some $f \in [1, 7]$, and a unique character—we use 0 to denote this character in our discussion later—otherwise. Let Υ be the concatenation of the strings Υ_1 through $\Upsilon_{2^{\ell-1}}$. We use a *wavelet tree* [82] to represent Υ, which has alphabet size $\sigma_\Upsilon = \max_{q\in[1,2^{\ell-1}]} |\mathfrak{L}_q| + 1 \le 8$. This representation uses $n \lg \sigma_\Upsilon(1 + o(1))$ bits to provide constant time support for the operation $\mathtt{rank}_f(\Upsilon, i)$, which returns the number of occurrences of the character f in $\Upsilon[1, i]$.

Theorem 3.5.1. *Given a string $\mathfrak{A}[1, n]$, there exists an $O(n \lg n)$ bit data structure that supports range majority queries on $\mathfrak{A}$ in $O(1)$ time, and can be constructed in $O(n \lg n)$ time.*

Proof. Given a query range $\mathcal{R} = [a, b]$, we first find the level ℓ and the index q of the quadruple $\mathcal{Q}_q$ with which $\mathcal{R}$ is associated. This can be reduced to finding the length of the longest common prefix of the $(\lg n)$-bit binary representations of a and b, which can be done in constant time in the following way. Let $z = \mathtt{MSB}(n/(b-a+1))$, where $\mathtt{MSB}(x)$ returns the position of the most significant bit of x. If $(a - 1)2^z \mod n = 0$, or $(b - 1)2^z \mod n = 0$, or $\lceil(a - 1)2^z/n\rceil \neq \lfloor(b - 1)2^z/n\rfloor$, then $\ell = z$; otherwise, $\ell = z + 1$. Let $q' = \lfloor(a - 1)2^{\ell-1}/n\rfloor$. The quadruple $\mathcal{Q}_q$ associated with $\mathcal{R}$ is either $q = q'/2$ or $q = (q' - 1)/2$ depending on whether the size of the blocks at level ℓ divide into the starting position $a - 1$. Note that we interpret $\mathcal{Q}_0$ to mean $\mathcal{Q}_{2^{\ell-1}}$. Since we can support the $\mathtt{MSB}$ operation in $O(1)$ time using a precomputed table of size $o(n)$ bits [115], we can compute both ℓ and q in $O(1)$ time.

Next, we show how to answer queries associated with a quadruple at level ℓ, for $2 \le \ell \le \lg n$; the case in which $0 \le \ell \le 1$ can be handled similarly. The representation of quadruple $\mathcal{Q}_q$ in Υ begins at $s = 4(q - 1)n/2^\ell + 1$. We normalize the values a and b to the starting position s, so let $t = 2(q - 1)n/2^\ell + 1$. For each f in $[1, |\mathfrak{L}_q|]$, we count the frequency of $\mathfrak{L}_q[f]$ in $[a, b]$ using $\mathtt{rank}_f(\Upsilon, s+b-t) - \mathtt{rank}_f(\Upsilon, s+a-1-t)$, or, equivalently, $\mathtt{rank}_f(\Upsilon, t+b) - \mathtt{rank}_f(\Upsilon, t+a-1)$. We then report $\mathfrak{L}_q[f]$ if it is a majority. Since Υ has a constant sized alphabet, this process takes $O(1)$ time.

In addition to the input string, we store the lists $\mathfrak{L}_q$ for each of the $O(n)$ quadruples, and each list requires $O(\lg \sigma) = O(\lg n)$ bits. For each of the $\lg n + 1$ levels in $\mathcal{T}$ we store a wavelet tree on an alphabet of size $\sigma_\Upsilon \le 8$, requiring $O(n \lg n)$ bits. To answer queries in constant time, we require $o(n)$ bits of additional space for a lookup table to determine ℓ and q. Thus, the additional space requirements beyond the input string are $O(n \lg n)$ bits. In terms of construction time, we can build the lists of candidates in $O(n \lg n)$ time by Lemma 3.5.2, and the wavelet trees require $O(n)$ time to construct per level (by Lemma 1.5.7). Thus, the overall construction time is $O(n \lg n)$. □

Remark 3.5.1. *In most cases, the upper bound on the number of candidates stored for each quadruple is significantly greater than the number of candidates actually stored by the data structure. Therefore, we expect that the constant factor in the $O(n)$ space term is not very large in practice.*

3.6 Generalization to Static Range α-Majority Queries

In this section we generalize the data structure from Theorem 3.5.1 to report α-majorities, for some fixed $\alpha \in (0, 1]$, supplied at construction time. Using the same arguments presented in the beginning of Section 3.5, it is clear that if a character is an α-majority for any query associated with quadruple $\mathcal{Q}$, then the character must appear more than $\alpha|\mathcal{Q}|/4$ times in $\mathcal{Q}$. This implies that the set of candidate α-majorities has size less than $4/\alpha$.

3.6.1 Handling Large Alphabets

If the number of candidates, $|C| = 4/\alpha$, is non-constant, then we require the following lemma about executing batched rank queries on a wavelet tree.

Lemma 3.6.1. *A string $\mathfrak{S}[1, n]$ over an alphabet $[1, \sigma_\Upsilon]$, where $\sigma_\Upsilon \leq n$, can be represented using a wavelet tree such that given an index i, the results of $\texttt{rank}_f(\mathfrak{S}, i)$ for all $f \in [1, \sigma_\Upsilon]$ can be computed in $O(\sigma_\Upsilon)$ time.*

Proof. If we use the wavelet tree in the standard way to perform $\texttt{rank}_f(\mathfrak{S}, i)$ for a particular character, f, the algorithm consists of two steps. First, in the bit vector representing the leaf, v, corresponding to f (and one other character), we locate the bit that corresponds to $\mathfrak{S}[i]$. This step is done by performing constant-time rank queries on bit vectors, representing internal nodes of the wavelet tree, on the path from the root to the leaf v. The second step is to perform a rank query in constant time on the bit vector representing v, to compute the number of 1s up to and including the bit corresponding to $\mathfrak{S}[i]$. By performing batch processing of the queries $\texttt{rank}_f(\mathfrak{S}, i)$ for f in $1, 2, \ldots, \sigma_\Upsilon$, as there are $\sigma_\Upsilon - 1$ nodes in a wavelet tree, the first step for all f

requires $O(\sigma_\Upsilon)$ constant-time rank queries on bit vectors stored in internal nodes. Therefore, with careful tuning, we can perform the first step of all the $\texttt{rank}_f(\mathfrak{S}, i)$ queries in $O(\sigma_\Upsilon)$ time. The second step requires a constant time rank query for each leaf, so it uses $O(\sigma_\Upsilon)$ time in total. $\square$

With the above observation we present the following theorem:

Theorem 3.6.1. *Given a string $\mathfrak{A}[1, n]$ and any fixed $\alpha \in (0, 1]$, there is a data structure that occupies $O(n \lg n(\lg(1/\alpha) + 1))$ bits, supports range α-majority queries on $\mathfrak{A}$ in $O(1/\alpha)$ time, and can be constructed in $O(n \lg n(\lg(1/\alpha) + 1))$ time.*

Proof. From Theorem 3.5.1 and Lemma 3.6.1 the query time follows, so we focus on analyzing the space. We observe that if $\alpha < 1/4$, then we need not keep data structures at level $\lg n$ in $\mathcal{T}$, since every distinct character contained in a query range, $\mathcal{R}$, associated with this level is a $(1/4 - \varepsilon)$-majority for $\mathcal{R}$, for $0 < \varepsilon < 1/4$. Instead, we perform a linear scan of the query range in $O(1/\alpha)$ time, returning all the distinct characters. This can be done by using the auxiliary array, which has size σ.

Continuing this argument, we observe that we only require the list $\mathfrak{L}_q$, that stores candidates for quadruple q, if q represents a range of size larger than $O(1/\alpha)$. Since there are $O(n\alpha)$ quadruples larger than $O(1/\alpha)$, the lists require $O(n\alpha(1/\alpha \lg n)) = O(n \lg n)$ bits in total. The overall space required for the wavelet tree data structures is $O(n \lg(1/\alpha + 1) \lg n)$ bits, and this term dominates the overall space requirements.

We can construct the sets of candidates for all quadruples of size larger than $O(1/\alpha)$ in $O(n \lg n)$ time using the same technique described in Lemma 3.5.2. To construct the wavelet trees requires $O(n(\lg(1/\alpha) + 1))$ time per level (by Lemma 1.5.7), for an overall time bound of $O(n \lg n(\lg(1/\alpha)+1))$. Thus, the construction time is dominated by the wavelet tree construction, and requires $O(n \lg n(\lg(1/\alpha) + 1))$ time overall. $\square$

3.7 Parameterized Query and Trade-offs

In this section, we describe some minor changes to the data structure of Theorem 3.6.1 that allow us to answer β-majority queries for any $\beta \in [\alpha, 1]$, where β is specified at query time (recall that α is fixed at construction time). This data structure occupies $O(n(\min\{\lg(1/\alpha), \mathsf{H}_0(\mathfrak{A})\} + 1) \lg n)$

bits of space, and can support β-majority queries in time $O(1/\beta)$. Recall that $\mathsf{H}_0(\mathfrak{A})$ is the zeroth-order empirical entropy of the string $\mathfrak{A}$, and so $\mathsf{H}_0(\mathfrak{A}) \leq \lg \sigma$. Thus, we not only generalize the query operation, but also further refine the space analysis.

We begin with a lemma that describes how to compress a string, and support access operations. We do not attempt to optimize constant factors here, as the space occupied by the data structures that apply this lemma are not succinct (this lemma is not the space bottleneck):

Lemma 3.7.1. *Let $\mathfrak{S}$ be a string of U characters, drawn from the alphabet $\Sigma = [0, \sigma]$, where $\sigma \leq U$. There is a data structure for representing $\mathfrak{S}$ that occupies $O(U(H_0(\mathfrak{S}) + 1) + \sigma \lg \sigma)$ bits of space, and supports the operation* `access`$(\mathfrak{S}, i)$ *for any $1 \leq i \leq U$ in $O(1)$ time.*

Proof. This lemma can be proved using any encoding scheme that asymptotically achieves zeroth order entropy compression (Huffman codes, Elias γ-codes, etc.), and has the property that all code lengths are no longer than $\Theta(\lg U)$ bits. Without loss of generality, we use Elias γ-codes to achieve the bound. We compute $\gamma(\mathfrak{S})$ and store it in a bit string. We also store a bit string $\mathfrak{B}$ of length $|\gamma(\mathfrak{S})|$, which marks the start of each encoded character with a 1. Storing $\mathfrak{B}$ in a the data structure of Lemma 1.5.6 allows us to access the Elias γ-code of any element in $\mathfrak{S}$ using a $O(1)$ time via a single select operation, and increases the overall space by no more than a constant factor.

We also store a lookup table as a bit string $\mathfrak{B}'$ in the following way: $\mathfrak{B}'[i] = 1$ iff i is a valid code used in the representation of $\mathfrak{S}$. Since Elias γ-codes have length $O(\lg \sigma)$ by Lemma 1.5.3 (item 2), the length of $\mathfrak{B}'$ is polynomial in σ. We also store a table, storing the decoded characters in $\mathfrak{S}$, sorted in the order of the ranks in $\mathfrak{B}'$ of the Elias gamma codes for the characters. Thus, by storing $\mathfrak{B}'$ in the data structure of Lemma 1.5.6, and applying Lemma 1.5.5, these extra data structures occupy $O(\sigma \lg \sigma)$ bits. Given a character's γ-code, i, we compute $\mathtt{rank}_1(\mathfrak{B}', i)$. We then index into the table at this rank to return decoded character. Thus, we can decode any γ-code in constant time. □

In the data structure of Theorem 3.6.1, the candidate lists for each quadruple are unsorted. We add the constraint that these lists be sorted in descending order of frequency within the quadruple (breaking ties arbitrarily). The observation is that, since $\beta \geq \alpha$, all β-majorities for the given quadruple will be a prefix of the candidate list: in particular, the first $\lceil 4/\beta \rceil$ characters in the list. This alone will not be enough to speed up the process of verifying which candidates are β-majorities. For that we need the following lemma:

Lemma 3.7.2. *Let $\mathfrak{S}$ be a string of U characters, drawn from the alphabet $\Sigma = [0, \sigma]$, where, ignoring the special character 0, character i is the i-th most frequent character in $\mathfrak{S}$, and $\sigma \leq U$. There is a* skewed wavelet tree *$\mathcal{T}$ representing the string $\mathfrak{S}$, that has the following properties:*

1. *The leaf representing character i has depth $\Theta(\lg(i+2))$ in $\mathcal{T}$ (the root has depth 1).*
2. *For $1 \leq \sigma' \leq \sigma$, the leaves representing $1, \ldots, \sigma'$ can be traversed in $O(\sigma')$ time.*
3. *The wavelet tree $\mathcal{T}$ occupies $O(U(H_0(\mathfrak{S})+1)(1+o(1)) + \sigma \lg U)$ bits.*

Proof. An example of a wavelet tree $\mathcal{T}$ with these properties is one where the implicit sequence of bits along the path to the leaf representing character i is the Elias γ-code for $i+1$. All three properties of the wavelet tree follow from Lemma 1.5.3. $\square$

We now prove the main theorem of this section:

Theorem 3.7.1. *Let $\mathfrak{A}$ be a string of length n drawn from the characters $\Sigma = [1, \sigma]$, where $\sigma \leq U$ and $\alpha \in (0, 1]$ be some fixed parameter. There is a data structure that occupies*

$$O(n \min(\{\lg(1/\alpha), H_0(\mathfrak{A})\} + 1) \lg n) \tag{3.7.1}$$

bits, that, given a range $\mathfrak{A}[a, b]$ and a fraction $\beta \in [\alpha, 1]$, can return the β-majorities of the substring $\mathfrak{A}[a, b]$ in $O(1/\beta)$ time.

Proof. We represent the string using the access scheme of Lemma 3.7.1. Thus, we can support `access` in $O(1)$ time on a compressed version of $\mathfrak{A}$. Next we consider the data structure of Theorem 3.6.1 and make some changes to speed it up.

Recall that we store the concatenation of the strings represented by the quadruples at level ℓ, in ascending order of quadruple starting position, as a string Υ_ℓ, for $0 \leq \ell \leq \lg(1/\alpha)$. Furthermore, recall that $|\Upsilon_\ell| = O(n)$. Using the wavelet tree from Lemma 3.7.2 to represent these strings will allow us to count the frequency of candidate characters in time $O(1/\beta)$. However, we do not have concatenated strings for quadruples of size smaller than $1/\alpha$, and thus can only return

β-majorities for these smaller quadruples in time $O(1/\alpha)$ via a linear-time scan of the relevant quadruple. We now explain how to construct $\Upsilon_{\ell'}$, for levels ℓ', where $\lg(1/\alpha) < \ell' \leq \lg n$, in order to verify the frequency of candidates in these smaller blocks in $O(1/\beta)$ time.

Imagine that we had constructed $\lg(1/\alpha)$ copies of the data structure we have described thus far, for values of α' drawn from the set $\{1/2, 1/4, \ldots, \alpha\}$. Let $\mathcal{D}_{\alpha'}$ denote the data structure constructed for parameter α'. Let α'' be the smallest value such that $\mathcal{D}_{\alpha''}$ stores candidate lists at level ℓ'. The string we store in our data structure to represent level ℓ' matches that of $\mathcal{D}_{\alpha''}$, and we also store the candidate lists from $\mathcal{D}_{\alpha''}$ for quadruples at ℓ'.

Space for candidate lists: By our previous lemmas, the space occupied by the candidate lists in the upper levels of the tree (above level $\lg(1/\alpha)$) is at most $O(n \lg n)$ bits. We show that the space bound for the remaining candidate lists (i.e., the additional ones described in the previous paragraph) is $O(n(\lg(1/\alpha) + 1)(\min\{\mathsf{H}_0(\mathfrak{A}), \lg(1/\alpha)\} + 1)$ bits. Since these remaining candidate lists are in the bottom $O(\lg(1/\alpha) + 1)$ levels of the tree, it suffices to bound the space for a single level as $O(n(\min\{\mathsf{H}_0(\mathfrak{A}), \lg(1/\alpha)\} + 1))$ bits.

This bound can be achieved by combining the two following methods, and using the one that occupies less space. The first method is to store the candidate lists in compressed form, using the Elias γ-codes for $\mathfrak{A}$: the same ones stored for the access scheme of Lemma 3.7.1. To store all the lists at level ℓ' cannot take more than $O(n(\mathsf{H}_0(\mathfrak{A}) + 1))$ bits, since each candidate appears at least once in its respective quadruple, and exactly once in the candidate list. The second method is to store the index of the first occurrence of the candidate within its respective quadruple (i.e., the offset from the start of the quadruple). This method requires no more than $n(\lg(1/\alpha) + 1)$ bits to store level ℓ', since each quadruple has size at most $1/\alpha$ in ℓ'. Thus, we achieve the claimed bound of $O(n(\min\{\mathsf{H}_0(\mathfrak{A}), \lg(1/\alpha)\} + 1))$. In either case, we can convert a rank in a candidate list into the candidate character in $O(1)$ time.

Space for wavelet trees: We now argue that representing the strings, for all levels, using Lemma 3.7.2 requires no more than $O(n(\min\{\lg(1/\alpha), \mathsf{H}_0(\mathfrak{A})\} + 1) \lg n)$ bits.

Consider the string Υ which corresponds to some quadruple, and therefore some substring $\mathfrak{A}[a, b]$. Thus, there is a many-to-one mapping from $\mathfrak{A}[a, b] \rightarrow \Upsilon$, which replaces the characters in $\mathfrak{A}[a, b]$ with either their ranks by frequency, or by the special character 0. Applying such a many-to-one mapping from a string's alphabet to a smaller alphabet cannot increase its zeroth order empirical entropy, so $\mathsf{H}_0(\Upsilon) \leq \mathsf{H}_0(\mathfrak{A}')$.

Moreover, since each string Υ_ℓ is the concatenation of strings represented by quadruples, and *each character in* $\mathfrak{A}$ *appears in exactly two quadruples*, it is clear that $\mathsf{H}_0(\Upsilon_\ell) \leq \mathsf{H}_0(\mathfrak{A})$ for $1 \leq \ell \leq \lg n$. The final observation is that $\mathsf{H}_0(\Upsilon_\ell) \leq O(\lg(1/\alpha))$, since the alphabet size is at most $O(1/\alpha)$. Since there are $O(\lg n)$ strings, the space bound for these strings is $O(n(\min\{\log(1/\alpha), \mathsf{H}_0(\mathfrak{A})\} + 1)\log n)$ bits overall.

Verifying the candidates for a quadruple at level ℓ involves traversing the first $O(1/\beta)$ leaves of the skewed wavelet tree representing Υ_ℓ, which takes $O(1/\beta)$ time by Lemma 3.7.2. At each leaf, which represents a candidate, at most two $O(1)$ time rank queries on the bit string stored in the leaf can be used to determine the frequency of the candidate in a range, in $O(1/\beta)$ time overall. $\square$

Remark 3.7.1. *As discussed earlier, the recent paper of Belazzougui et al. [14] presents several new trade-offs that improve Theorem 3.7.1 in terms of both time and space.*

We also have the following trade-off, which results in a more space efficient data structure, but increases the query time by a factor of $\lg\lg n$. The following structure also has the disadvantage that it does not allow parameterized queries:

Theorem 3.7.2. *Let* $\mathfrak{A}$ *be a string of length* n*, drawn from the set of characters* $\Sigma = [1, \sigma]$*, where* $\sigma \leq U$ *and* $\alpha \in (0, 1]$ *be some fixed parameter. There is a data structure that occupies*

$$O(n(H_0(\mathfrak{A}) + 1)) + O(\sigma \lg n) \tag{3.7.2}$$

bits, that, given a range $[a, b]$*, the* α*-majorities of the substring* $\mathfrak{A}[a, b]$ *can be returned in time* $O((\lg\lg\sigma)/\alpha)$*.*

Proof. We represent the string $\mathfrak{A}$ in the data structure of Lemma 1.5.8, which occupies $n\mathsf{H}_0(\mathfrak{A}) + o(n)$ bits of space, and can support `rank` queries in time $O(\lg\lg\sigma)$ time. We also store the string $\mathfrak{A}$ using the access scheme of Lemma 3.7.1. Finally, we store the candidate lists for each quadruple representing more than $1/\alpha$ characters, compressed using their codes in $\mathfrak{A}$, and the access scheme of Lemma 3.7.1. Overall, these candidate lists occupy at most $O(n(\mathsf{H}_0(\mathfrak{A}) + 1))$ bits. Given a range $[a, b]$, we perform two `rank` queries per candidate to count the occurrences of the at most $O(1/\alpha)$ candidates in the list associated with $[a, b]$. This takes $O((\lg \lg \sigma)/\alpha)$ time. $\square$

Remark 3.7.2. *Belazzougui et al. [14] improve the space bound of Theorem 3.7.2 by using recent results [15] on representing strings in order to support support* `access` *and* `rank` *efficiently. They also show how to parameterize the query time.*

3.8 Applications to Static Geometric Problems

In this section we describe some consequences of Theorem 3.6.1 to the geometric versions of the range α-majority problem described by Karpinski and Nekrich [103]. These problems have applications to database problems, in which we would like to identify attributes that are frequently associated with points in a query range [103]. In the next subsections we describe these geometric problems, which deal with coloured points instead of characters in a string.

3.8.1 Static Range Majority for Coloured Points in One Dimension

We are given a set, $\mathcal{P}$, of points in one dimension, where each point $p \in \mathcal{P}$ is assigned a colour c from a set, Σ, of colours. Let $\texttt{Colour}(p) = c$ denote the colour of p. We are also given a fixed parameter $\alpha \in (0, 1]$, which defines the threshold for determining whether a colour is to be considered frequent. Let $\mathcal{P}(\mathcal{R})$ be the set $\{p \mid p \in \mathcal{R}, p \in \mathcal{P}\}$, and $\mathcal{P}(\mathcal{R}, c)$ be the set $\{p \mid p \in \mathcal{P}(\mathcal{R}), \texttt{Colour}(p) = c\}$. Our goal is to design a data structure that, given query range $\mathcal{R}$, can return the set of colours Σ such that for each colour $c \in \Sigma$, the size of the set $|\mathcal{P}(\mathcal{R}, c)| > \alpha|\mathcal{P}(\mathcal{R})|$. To be consistent with our original formulation of the problem, we refer to a colour $c \in \Sigma$ as an *α-majority* for $\mathcal{R}$, and the query $\mathcal{R}$ as an *α-majority query*, though they are also referred to as *α-dominating colours* [103].

To solve this problem, we apply the reduction to rank space technique [34, 5] to the coordinates of the points, and store a string representing the left-to-right sequence of colours in the data structure from Theorem 3.6.1. We store the original coordinates of the points in any linear space data structure that supports predecessor queries. Given a range query on the line, we can use the predecessor search data structure to remap the query to a range query on the string. Thus, we can support range α-majority queries on the line using the data structures from the previous section. We present the following theorem:

Theorem 3.8.1. *Given a set $\mathcal{P}$ of n points in one dimension and a fixed $\alpha \in (0, 1]$, there is an $O(n(\lg(1/\alpha) + 1))$ word data structure that supports range α-majority queries on $\mathcal{P}$ in $O(\texttt{pred}(\mathcal{P}) + 1/\alpha)$ time, where $\texttt{pred}(\mathcal{P})$ is the time required to do a one-dimensional predecessor search on the coordinates of the points in $\mathcal{P}$ (within the allotted space bound).*

Remark 3.8.1. *The previous theorem implies that if we only assume the coordinates of the points can be compared in constant time, then we can answer range α-majority queries in $O(\lg n + 1/\alpha)$ time by storing the coordinates of the points in a balanced binary search tree. If the points have integer coordinates, drawn from a bounded universe $[1, u]$, then we can store their coordinates in an exponential search tree [8] and answer range α-majority queries in $O(\sqrt{\lg n / \lg\lg n} + 1/\alpha)$ time and occupies linear space. Alternatively, we can store their coordinates in a y-fast trie [156], which yields a query time of $O(\lg\lg u + 1/\alpha)$, and also occupies linear space.*

3.8.2 Static Range Majority in Higher Dimensions

In the same manner as Karpinski and Nekrich [103], we can extend Theorem 3.8.1 to higher dimensions using the well-known range tree technique of Bentley [19]. The main problem with moving to higher dimensions is that we cannot use the wavelet tree to verify the frequency of candidates when the number of dimensions, $d \geq 2$. However, we can use the small list of candidates generated by the data structure in conjunction with any d-dimensional range counting data structure, such as that of Chazelle [34]. In particular, we store a counting structure that stores all the points (regardless of colour), and one that stores the points of each colour individually. By removing the wavelet trees from the data structure of Theorem 3.6.1, we remove the $(\lg(1/\alpha)+1)$ term from the space bound. Thus, this stripped down one-dimensional data structure uses $O(n)$ *words* of space, and can return, for any range α-majority query $\mathcal{R}$ on $\mathfrak{A}$, a list of $O(1/\alpha)$ characters that contains all the α-majorities for $\mathcal{R}$ in $O(1/\alpha)$ time.

Theorem 3.8.2. *Given a set $\mathcal{P}$ of n points in d-dimensions, for any constant $d \geq 2$, and a fixed $\alpha \in (0, 1]$, there is an $O(n \lg^{d-1} n)$* word *data structure that supports range α-majority queries on $\mathcal{P}$ in $O((\lg^d n)/\alpha)$ time.*

Proof. Using range trees, we can convert any d-dimensional range α-majority query into $\Theta(\lg n)$ $(d-1)$-dimensional range α-majority queries and d-dimensional range counting queries. In particular, let $\mathfrak{T}_{\texttt{count}}(n, d)$ denote the cost of a d-dimensional range counting query on a static set of

n points, and $\mathfrak{T}_{\texttt{maj}}(n,d)$ denote the cost of a d-dimensional range α-majority on a static set of n points. Thus, we have:

$$\mathfrak{T}_{\texttt{maj}}(n,1) = O(1/\alpha) + O(\lg n) \tag{3.8.1}$$

$$\mathfrak{T}_{\texttt{maj}}(n,d) = \Theta(\lg n)\mathfrak{T}_{\texttt{maj}}(n,d-1) + \Theta((\lg n)/\alpha)\mathfrak{T}_{\texttt{count}}(n,d), \tag{3.8.2}$$

since we can extract and verify the frequency of the $O((\lg n)/\alpha)$ candidates from the $O(\lg n)$ nodes representing the range spanned by the d-th coordinate of the query range. Since d-dimensional static orthogonal range counting can be done in $O(\lg^{d-1} n)$ time [34] for $d \geq 2$, we have:

$$\mathfrak{T}_{\texttt{maj}}(n,d) = O((\lg^d n)/\alpha). \tag{3.8.3}$$

The space occupied by the two-dimensional data structure will be $O(n \lg n)$ words, since each point will be duplicated in $\Theta(\lg n)$ one-dimensional range majority data structures. This space cost dominates that of the linear space two-dimensional counting data structures, and increases by a $(\lg n)$-factor with each additional dimension. □

Remark 3.8.2. *As in the one-dimensional case, we can exploit word-level parallelism to improve the time complexity of queries in the case where the points in $\mathcal{P}$ have integer coordinates. Using the data structure of JaJa et al. [99], the query time for the two dimensional data structure can be improved to $O((\lg^2 n)/(\alpha \lg\lg n))$. As we commented in the "Subsequent Work" section, the two dimensional result has been improved by Wilkinson [155], who used approximate range counting data structures to avoid having to perform an expensive exact range counting query for each candidate α-majority. Navarro, Nekrich, and Russo [122] match the result of Theorem 3.8.2, achieving the same query time and space bounds (to within constant factors), but additionally have a parameterized query.*

3.9 Approach for the Dynamic Geometric Problem

In this section we devise an approach for the dynamic geometric problem. As before, we are given a set, $\mathcal{P}$, of n points, where each point $p \in \mathcal{P}$ is assigned a colour c from a set, Σ, of colours. We denote the colour of p as $\texttt{Colour}(p) = c$. We are also given a fixed parameter $\alpha \in (0,1)$, that defines the threshold for determining whether a colour is to be considered frequent. Our goal is to design a *dynamic range α-majority data structure* that can perform the following operations in addition to the α-majority query:

- INSERT(p, c): Insert a point p with colour c into $\mathcal{P}$.
- DELETE(p): Remove the point p from $\mathcal{P}$.

3.9.1 Lower Bound

As discussed in the previous work section, there is a cell-probe lower bound for the dynamic range α-majority problem, based on a reduction to a special case of a dynamic partial sums problem. The *partial sum problem for threshold functions* [97] is as follows: maintain n bits $x_1, ..., x_n$ subject to updates and *threshold queries*. An update consists of flipping the bit at a specified index. The answer to query `threshold`(i) is "yes" if and only if $\sum_{j=1}^{i} x_j \geq f(i)$, where $f(i)$ is an integer function such that $f(i) \in \{0, ..., \lceil i/2 \rceil\}$. Husfeldt and Rauhe [97] proved a lower bound on the query time t_q for a data structure that can answer threshold queries with update time t_u.

Any data structure for dynamic α-majority can be used to solve the partial sum problem for threshold functions. In particular, we can treat the problem as involving n points with integer coordinates $1, ..., n$, with each point having one of two colours. A flip operation can be implemented as a deletion followed by an insertion. Thus, we can state their lower bound in terms of our problem, denoting the cell size of our machine as w:

Lemma 3.9.1 (Follows from [97], Prop. 4)**.** *Let t_u and t_q denote the update and query times, respectively, for any dynamic α-majority data structure. Then,*

$$t_q = \Omega\left(\frac{\lg(\min\{\alpha n, (1-\alpha)n\})}{\lg(t_u w \lg(\min\{\alpha n, (1-\alpha)n\}))}\right).$$

This bound implies that, for constant values of α and word size $\Theta(\lg n)$ bits, $O(\lg n / \lg \lg n)$ query time for integer point sets is optimal for any data structure with polylogarithmic update time.

3.9.2 Assumptions about Colours

In the following sections, we assume that we can compare "colours" in constant time. In order to support a dynamic set of colours, we employ the techniques described by Gupta et al. [84]. These techniques allow us to maintain a mapping from the set of colours to integers in the range $[1, 2n]$, where n is the number of points *currently* in our data structure. This mapping allows us to use a colour as an array index, which speeds up certain parts of our query and update algorithms.

For the dynamic problems discussed, the mapping is maintained using a method similar to *global rebuilding* to ensure that the integer identifiers of the colours do not grow too large [84, Section 2.3]. When a coloured point is inserted, we first determine whether we have already assigned an integer to that colour. By storing the set of known colours in a balanced binary search tree, this can be checked in $O(\lg |\Sigma|)$ time, where $|\Sigma|$ is the number of distinct colours currently assigned to points in our data structure. Since $|\Sigma| \leq n$, this cost will be negligible compared to the update time of our data structure. Therefore, from this point on, we assume that we are dealing with integers in the range $[1, 2n]$ when we discuss colours.

3.9.3 Dynamic Tree Structure

In one-dimension we can interchange the notion of points and x-coordinates in $\mathcal{P}$, since they are equivalent. Depending on the context we may use either term. Our basic data structure is a modified *weight balanced B-tree* [9]. We prove several interesting combinatorial properties of α-majorities in order to provide more efficient support for queries compared to previous data structures.

We begin by defining a weight-balanced B-tree:

Definition 3.9.1 (Arge and Vitter [9]). *$\mathcal{T}$ is a* weight-balanced B-tree *with branching parameter f and leaf parameter g, where $f > 4$ and $g > 0$ are integers, if the following holds:*

1. *All leaves of $\mathcal{T}$ are on the same level and have weight between g and $2g - 1$; in the context of our problem, they store between g and $2g - 1$ points.*

2. *An internal node of height h (leaves have height 0) has weight less than $2f^h g$.*

3. *Except for the root, an internal node of height h has weight larger than $(1/2)f^h g$.*

4. The root has more than one child.

Let $\mathcal{T}$ be a weight-balanced B-tree with branching parameter $f = 8$ and leaf parameter $g = 1$ such that each leaf represents an x-coordinate in $\mathcal{P}$. From left to right the leaves are sorted in ascending order of the x-coordinate that they represent. Let $\mathcal{T}(v)$ be the subtree rooted at node v. Each internal node v in the tree represents a range $\mathcal{R}(v) = [x_{\min}, x_{\max}]$, where $x_{\min}$ is the x-coordinate represented by the leftmost leaf in $\mathcal{T}(v)$, and $x_{\max}$ is the x-coordinate represented by the rightmost leaf in $\mathcal{T}(v)$. If a node is h levels above the leaf level, we say that this node is of *height* h. By Properties 2 and 3 of weight-balanced B-trees, the range represented by an internal node of height h (with the exception of the root) contains more than $8^h/2$ points and less than $2(8^h)$ points, and the degree of each internal node is between 2 and 32 [9, Lem. 3.4].

3.9.4 Supporting Queries

Given a query $\mathcal{R}' = [x'_a, x'_b]$, we perform a top-down traversal on $\mathcal{T}$ to map $\mathcal{R}'$ to the range $\mathcal{R} = [x_a, x_b]$, where x_a and x_b are the points in $\mathcal{P}$ with x-coordinates that are the successor and the predecessor of x'_a and x'_b, respectively. We call the query range $\mathcal{R}$ *general* if $\mathcal{R}$ is not represented by a single node of $\mathcal{T}$. We first define the notion of *representing* a general query range[1] by a set of nodes:

Definition 3.9.2. *Given a general query range $\mathcal{R} = [x_a, x_b]$, $\mathcal{R}$ induces a set, I, of nodes in the tree $\mathcal{T}$, satisfying the following two conditions.*

1. *The range represented by the parent of each node in I is not entirely contained in $\mathcal{R}$.*

2. *For all $p \in \mathcal{P}(\mathcal{R})$, there is exactly one node $v \in I$ with $p \in \mathcal{R}(v)$.*

We say that I is the set of nodes in the tree $\mathcal{T}$ representing *$\mathcal{R}$.*

For each node $v \in \mathcal{T}$, we keep a list, $\mathfrak{L}(v)$, of k *candidate* colours, i.e., the k most frequent colours in the range $\mathcal{R}(v)$ represented by v, breaking ties arbitrarily. Later, we will fix a value for k. Let $\mathfrak{L}^\star = \cup_{v \in I}\mathfrak{L}(v)$, i.e., the union of all the candidate lists among the nodes representing the query range $\mathcal{R}$. For each colour $c \in \Sigma$, we keep a separate range counting data structure, F_c,

[1]In alternate terminology, we are defining the *canonical nodes* that span the query range

containing all points $p \in \mathcal{P}$ with colour c, and also a range counting data structure, F, containing all of the points in $\mathcal{P}$. Let μ be the total number of points in the range $[x_a, x_b]$, which can be determined by querying F. For each $c \in \mathfrak{L}^\star$, we query F_c with the range $[x_a, x_b]$ letting `occ` be the result. If `occ` $> \alpha\mu$, then we report that c is an α-majority.

It is clear that I contains at most $\Theta(\lg n)$ nodes [9, Corollary 5]. Furthermore, if a colour c is an α-majority for $\mathcal{R}$, then it must be an α-majority for at least one of the ranges in I [103, Observation 1]. If we set $k = \lceil 1/\alpha \rceil$ and store $\lceil 1/\alpha \rceil$ colours in each internal node as candidate colours, then, by the procedure just described, we will perform a range counting query on $\Theta((\lg n)/\alpha)$ colours. If we use balanced search trees for our range counting data structures, then this takes $\Theta((\lg^2 n)/\alpha)$ time overall. However, in the sequel we show how to improve this query time by exploiting the fact that the nodes in I that are closer to the root of $\mathcal{T}$ contain more points in the ranges that they represent.

We shall prove useful properties of a general query range $\mathcal{R}$ and the set, I, of nodes representing it in Lemmas 3.9.2, 3.9.3, 3.9.4, and 3.9.5. In these lemmas, μ denotes the number of points in $\mathcal{R}$, and $i_1, i_2, ...$ denote the distinct values of the heights of the nodes in I, where $i_1 > i_2 > ... \geq 0$. We first give an upper bound on the number of points contained in the ranges represented by the nodes of I of a given height:

Lemma 3.9.2. *The total number of points in the ranges represented by all the nodes in I of height i_j is less than $\mu(\min\{1, 31(8^{1-j})\})$.*

Proof. Since $\mathcal{R}$ is general and contains at least one node of height i_1, μ is greater than the minimum number of points that can be contained in a node of height i_1, which is $8^{i_1}/2$. The nodes of I whose height is i_j, $j \neq 1$, are siblings and must have at least one sibling that is not in I. The number of points contained in the interval represented by this sibling is greater than $8^{i_j}/2$. Therefore, the number, μ_j, of points in the ranges represented by the nodes of I of height i_j is less than $2(8^{i_j+1}) - 8^{i_j}/2 = (31/2)(8^{i_j})$. Thus, $\mu_j/\mu < 31(8^{i_j - i_1}) < 31(8^{1-j})$. □

We next use the above lemma to bound the number of points whose colours are not among the candidate colours stored in the corresponding nodes in I.

Lemma 3.9.3. *Suppose we are given a node $v \in I$ of height i_j and a colour c. Let $n_v^{(c)}$ denote the number of points with colour c in $\mathcal{R}(v)$, the range covered by v, if c is not among the first*

$k_j = \lceil k/2^{j-1} \rceil$ *most frequent candidate colours in the candidacy list of* v*, and* $n_v^{(c)} = 0$ *otherwise. Then* $\sum_{v \in I} n_v^{(c)} < 5.59\mu/(k+1)$*.*

Proof. If c is not among the first k_j candidate colours stored in v, then the number of points with colour c in $\mathcal{R}(v)$ is at most $1/(k_j+1)$ times the number of points in $\mathcal{R}(v)$. Thus,

$$\begin{aligned}
\sum_{v \in I} n_v^{(c)} &< \sum_{j=1}^{2} \frac{\mu}{k_j+1} + \sum_{j \geq 3} \frac{(31(8^{1-j}))\,\mu}{k_j+1} \\
&< \frac{\mu}{k+1}\left(1 + 2 + 31\left(\frac{2^2}{8^2} + \frac{2^3}{8^3} + \cdots\right)\right) \\
&< \frac{5.59\mu}{k+1}
\end{aligned}$$

□

We next consider the nodes in I that are closer to the leaf level. Let I_t denote the nodes in I that are at one of the top $t = \lceil(\lg(1/\alpha)/3 + 2.05\rceil$—not necessarily consecutive—levels of the nodes in I. We prove the following property:

Lemma 3.9.4. *The number of points contained in the ranges represented by the nodes in* $I \setminus I_t$ *is less than* $\alpha\mu/2$*.*

Proof. By Lemma 3.9.2, the number of points contained in the ranges represented by the nodes in $I \setminus I_t$ is less than:

$$\begin{aligned}
31\mu \sum_{j \geq t+1} 8^{1-j} &< 31\mu\left(\frac{1}{8^t} + \frac{1}{8^{t+1}} + \cdots\right) \\
&< 31\mu\left(\frac{8}{7}\left(\frac{1}{8^t}\right)\right)
\end{aligned}$$

Since $t \geq \lg(1/\alpha)/3 + 2.05$, the above value is less than $\alpha\mu/2$. □

With the above lemmas, we can choose an appropriate value for k, guaranteeing the following property, that is critical to achieve improved query time:

Lemma 3.9.5. *When $k = \lceil 11.18/\alpha \rceil - 1$, any α-majority colour, c, of the query range $\mathcal{R}$ is among the union of the first $\lceil k/2^{j-1} \rceil$ candidates stored in each node of height i_j representing a range in I_t.*

Proof. The total number of points with colour c in the ranges represented by the nodes in $I \setminus I_t$ is less than $\alpha\mu/2$ by Lemma 3.9.4. By Lemma 3.9.3 and our choice for the value of k, less than $\alpha\mu/2$ points in the ranges represented by the nodes in I_t for which c is not a candidate can have colour c. □

For each node $v \in \mathcal{T}$, we keep a semi-ordered list of the k candidate colours in the range $\mathcal{R}(v)$ represented by v. The order on the colours for any candidacy list is maintained such that the most frequent $\lceil k/2^{j-1} \rceil$ colours come first, for all $j = 2, 3, \ldots$, arbitrarily ordered within their positions. Note that such a semi-ordering can be obtained in $O(k)$ time by repeated median queries. That is, by using a linear time median finding algorithm [21], we can partition the list so that the first half of the list contains the $k/2$ most frequent colours, and then recurse on the first half of the list until the list has 1 element. In total, this takes $O(k + k/2 + k/4 + \cdots) = O(k)$ time.

By setting $k = \lceil 11.18/\alpha \rceil - 1$, Lemma 3.9.5 implies that the colours that we have checked are the only possible α-majority colours for the query. Furthermore, Lemma 3.9.4 implies that we need only check the nodes on the top $O(\lg(1/\alpha))$ levels in I. Let I_t denote the set of nodes in these levels. We present the following lemma:

Lemma 3.9.6. *There is a data structure that occupies $O(n)$ words, and can be used to answer a range α-majority query in $O((\lg n)/\alpha)$ time.*

Proof. To support α-majority queries, we only consider the nontrivial case in which the query range $\mathcal{R}$ is general. By Lemma 3.9.5, the α-majorities can be found by examining the first $\lceil k/2^{j-1} \rceil$ candidate colours stored in each node representing a range in I_t. Thus, there are at most $O(\lceil 1/\alpha \rceil + \lceil 1/(2\alpha) \rceil + \lceil 1/(4\alpha) \rceil + \cdots + \lceil 1/(2^{t-1}\alpha) \rceil) = O(1/\alpha)$ relevant colours to check. Let $\mathfrak{L}_t$ denote the set of these colours. For each $c \in \mathfrak{L}_t$ we query our range counting data structures F_c and F in $\Theta(\lg n)$ time to determine whether c is an α-majority. Thus, the overall query time is $O((\lg n)/\alpha)$.

There are $\Theta(n)$ nodes in the weight-balanced B-tree. Therefore, one would expect the space to be $\Theta(n/\alpha)$ words, since each node stores $\Theta(1/\alpha)$ colours. We use the same pruning technique as in the static case, on the lower levels of the tree in order to reduce the space to $O(n)$ words overall. If a node v covers less than $1/\alpha$ points, then we need not store $\mathfrak{L}(v)$, since every colour in $\mathcal{T}(v)$ is an α-majority for $\mathcal{R}(v)$. Instead, during a query, we can traverse the leaves of $\mathcal{T}(v)$ in order to determine the unique colours. To make this efficient, we store an array D consisting of $2n$ counters, each with $\Theta(\lg n)$ bits, to count the frequencies of the colours in $\mathcal{R}(v)$. As mentioned in Section 3.3, we can map a colour to an index of the array D, which allows us to increment a frequency counter in $O(1)$ time. Thus, we can extract the unique colours in $\mathcal{R}(v)$ in $O(|\mathcal{T}(v)|) = O(1/\alpha)$ time. The number of tree nodes whose subtrees have at least $1/\alpha$ leaves is $O(n\alpha)$. Thus, we store $O(k) = O(1/\alpha)$ words in $O(n\alpha)$ nodes, and the total space used by our weight balanced B-tree $\mathcal{T}$ is $O(n)$ words. The only other data structures we make use of are the array D and the range counting data structures F and F_c for each $c \in \Sigma$, and together these occupy $O(n)$ words. $\square$

3.9.5 Supporting Updates

We next establish how much time is required to maintain the list $\mathfrak{L}(v)$ in node v under insertions and deletions. We begin by observing that it is not possible to lazily maintain the list of the top $k = \lceil 11.18/\alpha \rceil - 1$ most frequent colours in each range: many of these colours could have low frequencies, and the list $\mathfrak{L}(v)$ would have to be rebuilt after very few insertions or deletions. To circumvent this problem, we relax our requirements on what is stored in $\mathfrak{L}(v)$, only guaranteeing that *all* of the β-majorities of the range $\mathcal{R}(v)$ must be present in $\mathfrak{L}(v)$, where $\beta = \lceil 11.18/\alpha \rceil^{-1}$. With this alteration, we can still make use of the lemmas from the previous section, since they depend only on the fact that there are no colours $c \notin \mathfrak{L}(v)$ with frequency greater than $\beta|\mathcal{T}(v)|$. The issue now is how to maintain the β-majorities of $\mathcal{R}(v)$ during insertions and deletions of colours.

Karpinski and Nekrich noted that if we store the $(\beta/2)$-majorities for each node v in $\mathcal{T}$, then it is only after $|\mathcal{T}(v)|\beta/2$ deletions that we must rebuild $\mathfrak{L}(v)$ [103]. For the case of insertions and deletions, their data structure performs a range counting query at each node v along the path from the root of $\mathcal{T}$ to the leaf representing the inserted or deleted colour c. This counting query is used to determine if the colour c should be added to, or removed from, the list $\mathfrak{L}(v)$.

In contrast, our strategy is to be lazy during insertions and deletions, waiting as long as possible before recomputing $\mathfrak{L}(v)$, and to avoid performing range counting queries for each node in the update path. Note that, with each colour stored in the list $\mathfrak{L}(v)$ we store a counter that will keep track of the frequency of the colour in $\mathcal{R}(v)$.

Lemma 3.9.7. *Suppose the list $\mathfrak{L}(v)$ for node v contains the $\lceil 2/\beta \rceil$ most frequent colours in the range $\mathcal{R}(v)$, breaking ties arbitrarily. Let ℓ be the number of points contained in $\mathcal{R}(v)$. Only after $\lceil \beta\ell/2 \rceil$ insertions or deletions into $\mathcal{T}(v)$ can a colour $c \notin \mathfrak{L}(v)$ possibly become a β-majority for the range spanned by node v.*

Proof. Since we initially store the k most frequently appearing colours in the range $\mathcal{R}(v)$ in $\mathfrak{L}(v)$, any colour not in $\mathfrak{L}(v)$ can appear at most $\ell/(k+1)$ times. Thus, $\beta\ell/2$ updates are required before a colour $c \notin \mathfrak{L}(v)$ can become a β-majority for the range spanned by node v. □

By Lemma 3.9.7, our lazy updating scheme only requires each list $\mathfrak{L}(v)$ to have size $O(1/\alpha)$. This leads to the following theorem:

Theorem 3.9.1. *Given a set $\mathcal{P}$ of n points in one dimension and a fixed $\alpha \in (0, 1)$, there is an $O(n)$ space data structure that supports range α-majority queries on $\mathcal{P}$ in $O((\lg n)/\alpha)$ time, and insertions and deletions in $O((\lg n)/\alpha)$ amortized time.*

Proof. The query time follows from Lemma 3.9.6. In order to get the desired space, we combine Lemmas 3.9.6 and 3.9.7, implying that each list $\mathfrak{L}(v)$ contains $O(1/\alpha)$ colours. This allows us to use the same pruning technique described in Lemma 3.9.6 in order to reduce the space to $O(n)$.

When an update occurs, we follow the path from the root of $\mathcal{T}$ to the updated node v_0. Suppose, without loss of generality, that the update is an insertion of a point of colour c. For each vertex v on the path, if v contains a list $\mathfrak{L}(v)$, we check whether c is in $\mathfrak{L}(v)$. If it is, then we increment the count of colour c. This takes $O(1/\alpha)$ time. We also increment a counter stored in node v that keeps track of the number of updates into $\mathcal{T}(v)$ that have occurred since $\mathfrak{L}(v)$ was rebuilt. Thus, modifying the lists and counters along the path requires $O((\lg n)/\alpha)$ time in the worst case.

Next, we examine the costs of maintaining the lists $\mathfrak{L}(v)$. The list $\mathfrak{L}(v)$ can be rebuilt in $O(|\mathcal{T}(v)|)$ time, using the array D. Note that D can be maintained under updates using the same scheme described in Section 3.3. First, we use D to compute the frequency of all the colours in $\mathcal{R}(v)$ in $\Theta(|\mathcal{T}(v)|)$ time. Let k be the value from Lemma 3.9.7. Since there are at most $O(|\mathcal{T}(v)|)$ colours, we can use a linear time selection algorithm to find the k-th most frequent colour in D, and then find the top k most frequent colours via a linear scan in $O(|\mathcal{T}(v)|)$ time. We can then enforce the necessary semi-ordering on this list in $O(k) = O(1/\alpha)$ time, as described in Section 3.9.4. Thus, each leaf in $\mathcal{T}(v)$ pays $O(1)$ cost every $\Theta(|\mathcal{T}(v)|\alpha)$ insertions, or $O(1/\alpha)$ amortized cost per insertion. Since each update may cause $O(\lg n)$ lists to be rebuilt, this increases the cost to $O((\lg n)/\alpha)$ amortized time per update.

We use standard local rebuilding techniques to keep the tree $\mathcal{T}$ balanced, rebuilding the lists in nodes that are merged or split during an update. Since a node v will only be merged or split after $O(|\mathcal{T}(v)|)$ updates by the properties of weight-balanced B-trees, these local rebuilding operations require $O((\lg n)/\alpha)$ amortized time. Finally, we can update F_c and F during an insertion or deletion of a point of colour c in $O(\lg n)$ time. Thus, updates require $O((\lg n)/\alpha)$ amortized time overall, and are dominated by the costs of maintaining the lists $\mathfrak{L}(v)$ in each node v. $\square$

3.9.6 Speedup for Integer Coordinates

In this section, we describe how to improve the query time of the data structure from Theorem 3.9.1 from $O((\lg n)/\alpha)$ to $O(\lg n/(\alpha \lg\lg n))$, for the case in which the x-coordinates of the points in $\mathcal{P}$ are integers drawn from the universe $[1, u]$, where $1 \leq n \leq u \leq 2^w$.

We make use of the following lemma, discussed as a final remark in a paper of Andersson et al. [7]:

Lemma 3.9.8 (Augmented Fusion Tree [7])**.** *In the word-RAM model with word size $\Theta(\lg n+\lg u)$ bits, a set $\mathcal{E}$ of n elements from a bounded universe $[1, u]$ can be stored in a data structure of size $O(n)$ words, such that, given a range $\mathcal{R} = [x_a, x_b]$, the number of elements contained in $\mathcal{E} \cap \mathcal{R}$ can be reported in $O(\lg n/\lg\lg n)$ time. Inserting or deleting an element into $\mathcal{E}$ is supported in $O(\lg n/\lg\lg n)$ amortized time.*

In order to achieve $O(\lg n/(\alpha \lg\lg n))$ query time, we implement all the range counting data structures as the augmented fusion tree from Lemma 3.9.8. That is, the data structures F, and F_c for each $c \in C$. Immediately, we get that we can perform a query in $O(\lg n/(\alpha \lg\lg n) + \lg n)$ time: $O(\lg n/(\alpha \lg\lg n))$ time for the range counting queries, and $O(\lg n)$ time to find the nodes in I_t. We now discuss how to remove the additive $O(\lg n)$ term, which involves modifying our weight-balanced B-tree to support dynamic *lowest common ancestor queries*. Recall the lowest common ancestor of two nodes, v_1 and v_2, in a rooted tree is the deepest node whose induced subtree contains both v_1 and v_2. To identify the top $O(\lg(1/\alpha) + 1)$ levels of I, we use the following lemma:

Lemma 3.9.9. *The weight-balanced B-tree $\mathcal{T}$ can be augmented in order to support lowest common ancestor queries in $O(\sqrt{\lg n})$ time without changing the $O((\lg n)/\alpha)$ amortized time required for updates.*

Proof. Let the first ancestor of a node v_0 be the parent of v_0, and the ℓ-th ancestor of v_0 be the parent of the $(\ell - 1)$-th ancestor of v_0 for $\ell > 1$. In order to support lowest common ancestor queries between two nodes z_a and z_b, denoted $\text{LCA}(z_a, z_b)$, we add three pointers to each node $v_0 \in \mathcal{T}$: pointers to the leaves representing both the minimum and maximum x-coordinates in $\mathcal{T}(v_0)$, and a pointer to the ℓ-th ancestor of v_0; we will fix the value of ℓ later. We can search for the $\text{LCA}(z_a, z_b)$ by setting $v = z_a$ and following the pointer to the ℓ-th ancestor of v, denoted v'. By checking the maximum x-coordinate to see if $\mathcal{R}(v')$ contains z_b, we can determine whether v' is an ancestor of $\text{LCA}(z_a, z_b)$ or a descendant of $\text{LCA}(z_a, z_b)$ in constant time. If v' is a descendant of $\text{LCA}(z_a, z_b)$, then we set v to v' and v' to the ℓ-th ancestor of v'. If v' is an ancestor of $\text{LCA}(z_a, z_b)$, then we backtrack and walk up the path from v to v' until we find $\text{LCA}(z_a, z_b)$. Overall, it takes $O(h_0/\ell + \ell)$ time to find node $z = \text{LCA}(z_a, z_b)$, if z is at height h_0 in $\mathcal{T}$. By setting $\ell = O(\sqrt{\lg n})$ we get $O(\sqrt{\lg n})$ time. Furthermore, the pointers we added to $\mathcal{T}$ can be updated in $O(\lg n)$ amortized time during an insertion or deletion. Whenever we merge or split a node v_0, we have $O(|\mathcal{T}(v_0)|)$ time to fix all of the pointers into v_0, by properties of weight-balanced B-trees. The pointers out of v_0 can be fixed in $O(\lg n)$ worst case time. □

Although Lemma 3.9.9 is weaker than other results (cf. [151]), it is simple and sufficient for our needs. We next present the following theorem:

Theorem 3.9.2. *Given a set $\mathcal{P}$ of n points in one dimension with integer coordinates and a fixed $\alpha \in (0, 1)$, there is an $O(n)$ space data structure that supports range α-majority queries on $\mathcal{P}$ in $O(\lg n/(\alpha \lg\lg n))$ time, and both insertions and deletions into $\mathcal{P}$ in $O((\lg n)/\alpha)$ amortized time.*

Proof. Suppose we are given a query range $[x_a, x_b]$. Applying Lemma 3.9.9 to the weight-balanced B-tree $\mathcal{T}$, we claim that we can identify the top ℓ levels of I– that are *not* necessarily from consecutive levels in $\mathcal{T}$– using $O(\ell)$ least common ancestor operations. To show this, we describe a recursive procedure $\textsc{FindTop}(z_a, z_b, \ell)$ for identifying the top ℓ levels of I. We assume that we have acquired pointers to z_a and z_b, the leaves of $\mathcal{T}$ that represent the x-coordinates of the successor of x_a and predecessor of x_b, respectively. To do this, we add a pointer from each leaf in the augmented fusion tree F to its corresponding leaf in $\mathcal{T}$. Given a query, we initially perform a successor query for x_a and predecessor query for x_b in F, and follow these extra pointers to z_a and z_b, respectively. We assume that $z_a \neq z_b$, otherwise the query is trivially answered by reporting the colour stored in z_a.

Let $z = \mathrm{LCA}(z_a, z_b)$, and $\texttt{child}(z, i)$ denote the i-th child of z. Let z_l and z_r denote the leftmost and rightmost leaves in $\mathcal{T}(z)$. In constant time we can determine children $\texttt{child}(z, j)$ and $\texttt{child}(z, k)$ of z which are on the path to z_a and z_b, respectively. Note that $k - j > 0$, otherwise z is not the $\mathrm{LCA}(z_a, z_b)$. We say we are in the *good case* when $z_a = z_l$, $z_b = z_r$, and/or $k - j > 1$. When we are in the good case, either $\texttt{child}(z, j)$, $\texttt{child}(z, k)$, and/or $\texttt{child}(z, j+1), \ldots, \texttt{child}(z, k-1)$ are in the top level of I, and we set $\ell' = \ell - 1$. Otherwise, if $k - j = 1$ and $z_a \neq z_l$ and $z_b \neq z_r$, then we are in the *bad case*. In the bad case we have not found the top level of I, and we set $\ell' = \ell$. In both cases (good or bad), let $z_{b'}$ be the leaf in $\texttt{child}(z, k)$ representing the minimum x-coordinate in $\mathcal{T}(\texttt{child}(z, k))$, and $z_{a'}$ be the leaf in $\texttt{child}(z, j)$ representing the maximum x-coordinate in $\mathcal{T}(\texttt{child}(z, j))$. We recurse if $\ell' > 0$, calling $\textsc{FindTop}(z_a, z_{a'}, \ell')$ if $z_a \neq z_{a'}$ and $\textsc{FindTop}(z_{b'}, z_b, \ell')$ if $z_b \neq z_{b'}$.

We observe that the procedure $\textsc{FindTop}(z_a, z_b, \ell)$ uses $O(\ell)$ least common ancestor queries. This is because if a call to FINDTOP is in the bad case, then the subsequent recursive call(s) will be in the good case by choice of $z_{a'}$ and $z_{b'}$, and only the initial call to FINDTOP can make two recursive calls. Using FINDTOP, we can identify the top $O(\lg \frac{1}{\alpha})$ levels of I in $O(\sqrt{\lg n} \lg \frac{1}{\alpha})$ time, replacing the $O(\lg n)$ additive term. This factor is strictly asymptotically less than the time required to perform the range-counting queries, which is $O(\lg n / (\alpha \lg \lg n))$.

By Lemma 3.9.9, the we can support the lowest common ancestor operation without increasing the update time of $\mathcal{T}$ as stated in Theorem 3.9.1. The extra pointers we added from the leaves of F to the leaves of $\mathcal{T}$ can also be updated without affecting the bound from Theorem 3.9.1, since during any insertion/deletion of a point p, the two leaves corresponding to p in both F and $\mathcal{T}$ must be located. Therefore, the total update time follows from Theorem 3.9.1. □

3.9.7 Dynamic String

In this section we extend our results to dynamic strings. In the dynamic string problem, we wish to support the following operations on a string $\mathfrak{A}$ of length n, where each $\mathfrak{A}[i]$ stores a character drawn from the alphabet $\Sigma = [1, \sigma]$, for $1 \leq i \leq n$:

- $\textsc{Insert}(i, c)$: Insert the character c between the characters $\mathfrak{A}[i-1]$ and $\mathfrak{A}[i]$. This shifts the characters in positions i to n to positions $i+1$ to $n+1$, respectively.
- $\textsc{Delete}(i)$: Delete the character $\mathfrak{A}[i]$. This shifts the character in positions $i+1$ to n to positions i to $n-1$, respectively.
- $\textsc{Modify}(i, c)$: Set the character $\mathfrak{A}[i]$ to c.
- $\textsc{Query}(a, b)$: Let $|\mathfrak{A}[a, b]|_c$ denote the number of occurrences of character c in the range $\mathfrak{A}[a, b]$. Report the set of characters $\Sigma^\star$ such that for each $c \in \Sigma^\star$, $|\mathfrak{A}[a, b]|_c > \alpha|j - i + 1|$. As before, we refer to a character $c \in \Sigma^\star$ as an *α-majority* in the range $\mathfrak{A}[a, b]$, and the query as a *range α-majority query.*

We now prove the following theorem:

Theorem 3.9.3. *Given a string $\mathfrak{A}[1, n]$ and a fixed $\alpha \in (0, 1)$, there is an $O(n)$* word *data structure that supports range α-majority queries on $\mathfrak{A}$ in $O((\lg n)/(\alpha \lg\lg n))$ time,* $\textsc{Insert}$ *in $O((\lg n)/\alpha)$ amortized time,* $\textsc{Delete}$ *in $O((\lg n)/\alpha)$ amortized time, and* $\textsc{Modify}$ *in $O((\lg n)/\alpha)$ amortized time.*

Proof. We maintain our data structure $\mathcal{T}$ from Theorem 3.9.2, using standard techniques to augment the tree slightly, so that instead of maintaining integer coordinates we maintain the relative ordering of the characters in the string $\mathfrak{A}$. We also replace our range counting data structures with the recent data structure of Navarro and Nekrich [124], which allows for efficent range counting of characters via `rank` queries in $O(\lg n / \lg\lg n)$ time. Their dynamic string data structure also supports updates in $o(\lg n)$ time, yielding the theorem. $\square$

3.9.8 Higher Dimensions

In this section we generalize our dynamic results to higher dimensions:

Theorem 3.9.4. *Given a set $\mathcal{P}$ of n points in d-dimensions, for any constant $d \geq 2$, and a fixed $\alpha \in (0, 1]$, there is an* $O(n \lg^{d-1} n)$ word *data structure that supports range α-majority queries on $\mathcal{P}$ in $O((\lg^{d+1} n)/\alpha)$ time, and insertions and deletions in $O((\lg^{d} n)/\alpha)$ amortized time.*

Proof. We use the dynamic counting data structure of Chazelle [34], which occupies $O(n \lg^{d-2} n)$ space and supports counting queries in $O(\lg^{d} n)$ time, as well as insertions and deletions. Combining this data structure with Theorem 3.9.1 we apply the range tree technique of Bentley [19] as in the static case, and analyze the costs of the two dimensional structure. In two dimensions we spend $O((\lg^{3} n)/\alpha)$ time to answer a query, and our data structure occupies $O(n \lg n)$ words, since each point is duplicated in $O(\lg n)$ one dimensional range majority structures. Update time is $O((\lg^{2} n)/\alpha)$, since we must update each of the $O(\lg n)$ range majority structures in $O((\lg n)/\alpha)$ time per structure. This dominates the cost of updating the counting structures by a factor of $1/\alpha$. Each additional dimension beyond the second adds a $(\lg n)$-factor to the space, query, and update cost. $\square$

3.10 Summary and Concluding Remarks

We have presented an $O(n \lg n)$ bit data structure that answers range majority queries in constant time, and an $O(n \lg n(\lg(1/\alpha) + 1))$ bit data structure that answers range α-majority queries in $O(1/\alpha)$ time, for any fixed $\alpha \in (0, 1]$. This result is interesting in light of the nearly logarithmic cell-probe lower bounds of Greve et al. for the closely related problems of range mode and range k-frequency [81].

Our data structure is based on an interesting tree decomposition method used to preprocess a string such that each query is associated with a short list of candidate α-majorities. Then, using techniques from the area of succinct data structures, each character in this short list is efficiently checked in order to determine whether it is an α-majority for the given query. We have also described a parameterized version of the query, where at query time the user specifies a $\beta \in [\alpha, 1]$ along with an arbitrary range, and asks for the β-majorities in that range to be reported. We have shown that this query is solvable in $O(1/\beta)$ time, using no more space than the original

α-majority structure. Furthermore, we showed how to reduce the space of these data structures using standard compression techniques. We also discussed a more space efficient variant of the data structure that uses $O((\lg\lg n)/\alpha)$ time to report the α-majorities in a given range. We have given applications of our data structure to the higher dimensional geometric problems described by Karpinski and Nekrich [103], improving the space bounds for these. Finally, we have discussed dynamic versions of the problem for both the case of coloured points on a line and strings. We have given data structures in both cases that have optimal query time, as well as generalized our structures to the case of points in higher dimensions.

Chapter 4

Explicit Bitprobe Data Structures

4.1 Introduction

In this chapter we examine the static membership problem in the bitprobe model (see Section 1.2). According to Buhrman et al. [30], it is natural to study the problem of membership in the bitprobe model, since to answer a query we need only one bit, indicating yes or no, rather than $O(\lg u)$. By the same argument, there are other problems besides membership that are natural to study in the bitprobe model. Consider the following problems, that, like membership, ask us to store a subset $\mathcal{E}$ of n elements from an integer universe $[1, u]$:

1. *Range Emptiness Problem*: return whether $[x_1, x_2] \cap \mathcal{E} = \emptyset$ for $1 \leq x_1 \leq x_2 \leq u$.
2. *Rank Problem*: return the cardinality of $[1, x] \cap \mathcal{E}$ for any $x \in [1, u]$.
3. *Range Counting Problem*: return the cardinality of $[x_1, x_2] \cap \mathcal{E}$ for $1 \leq x_1 \leq x_2 \leq u$.

As with membership, the number of bits required to encode an answer to these types of queries does not depend on u. For emptiness—a generalization of membership—we require only one bit, whereas for rank and counting we require no more than $\lceil \lg(n+1) \rceil$ bits. In fact, for rank and counting, it might be natural to ask how much space a data structure that answers queries using $O(\lg(k+1)+1)$ bit probes must occupy, where k is the value of the answer. Though we mainly focus on the membership problem, it happens that many of the techniques we develop for

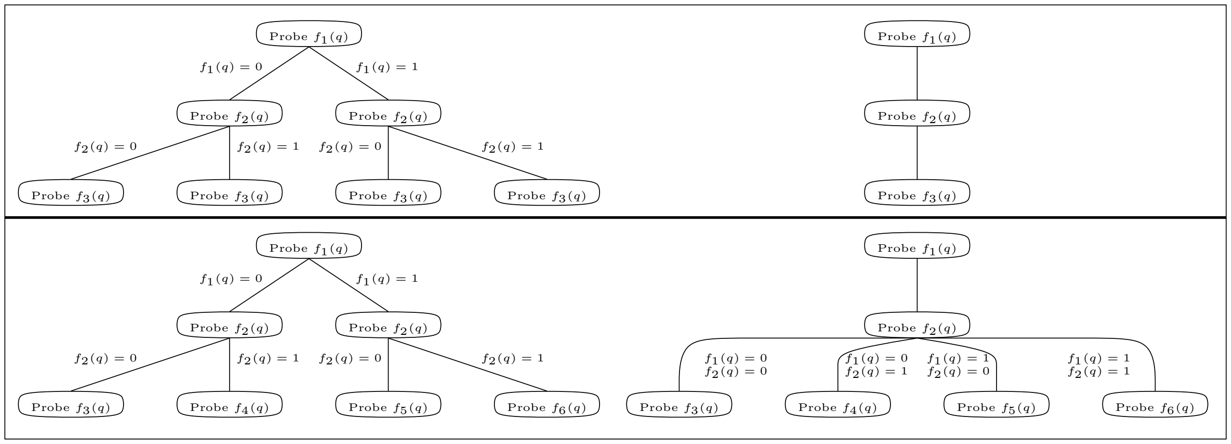


Figure 4.1: We use $f_i(q)$ to denote some function of the query element q. Top panel: an example of a decision tree associated with a non-adaptive scheme (left); we can interpret it as being a path (right) as the same locations are examined regardless of the intermediate bits probed. Bottom panel: an example of a decision tree associated with an adaptive scheme (left). The final probe uses the bits returned by the first two probes to determine which function to use; we can think of decision tree as a path, until the last probe, where a four-way branch takes place (right).

membership are rather generic, and can be modified to develop efficient bitprobe data structures for the additional listed problems. We note that, as in the other chapters, the results appearing in this chapter appear in a (currently unpublished) paper "Explicit Data Structures in the Bitprobe Model", which is joint work with Moshe Lewenstein, J. Ian Munro, and Venkatesh Raman.

Following the convention of previous work [30, 133, 134] we use the notation (n, u, s, t)-*scheme* to refer to a data structure that uses s bits of memory to store any n element subset $\mathcal{E}$ of a universe of size u, such that queries can be always be answered using at most t probes. For example, a bit string together with direct access is a $(n, u, u, 1)$-scheme for the membership problem. There are several ways of categorizing a (n, u, s, t)-scheme, which we now define:

- **Deterministic or randomized:**

  In a deterministic scheme the answer provided by the query algorithm must always be correct, and neither the storage scheme nor the query algorithm has access to random bits. In a randomized scheme the query algorithm may use random bits to decide which locations to probe in the data structure, and is also permitted to answer incorrectly with some failure probability, denoted ε. However, in a randomized scheme the storage scheme (i.e., the method to compute the data structure) must be deterministic. Given that the number of probes to be made is bounded, randomized schemes are Monte Carlo rather than Las Vegas. We focus only on deterministic schemes, but note that randomized schemes for membership have been studied [125].

- **Non-adaptive or adaptive:**

  In *non-adaptive schemes* the locations of the data structure to be probed are fixed based only on the query. In contrast, in *adaptive schemes* the location of only the first probe is fixed based on the query, whereas the locations of subsequent probes can also depend upon the bits returned by prior probes. Thus, we can think of an adaptive scheme as a binary decision tree of depth t, where the root represents the first bit read, and the left/right children represent the location of which bit to read next, depending on whether the root represents a zero/one, respectively. Note that to be adaptive, the scheme need only have two nodes at the same level in the tree where the locations probed differ. See Figure 4.1 for an example.

  Unlike previous work, we further refine the distinction between different adaptive schemes. Let us define a *ρ-adaptive* scheme as one where only the bottom ρ levels in the tree probe different locations. Thus, in terms of the decision tree in Figure 4.1 (bottom panel), only the lowest level in the tree contains nodes where the locations probed differ, so the scheme is 1-adaptive.

  Remark 4.1.1. *In response to the above definition a natural question might be, "If a scheme makes ρ adaptive probes, must they be the final ρ probes? Why not have a scheme perform some adaptive probes before some later non-adaptive probes?" Such a scheme would be perfectly valid, though we suspect that deferring adaptivity to the final ρ probes does not limit it in any way, as the query algorithm would have access to additional bits of information.*

  From the above discussion we note that any non-adaptive scheme can simulate a ρ-adaptive t-probe scheme, at the cost of doing exponentially more probes (i.e., $2^t - 2^{t-\rho-1}$ in total): simply probe each unique locations specified in the decision tree and examine the bits to determine the path that would have been followed by the adaptive scheme.

- **Non-explicit or explicit:** A *non-explicit* scheme can be thought of as an existential proof. It shows that there is a storage scheme that achieves the desired space bounds, such that membership queries can be answered in the desired number of probes, but does not provide intuition as to *how* to construct it efficiently, or how the query algorithm would compute which bits to probe. On the other hand, explicit schemes are divided into two categories:

1. *Explicit storage scheme*: an explicit storage scheme is one that, given $\mathcal{E}$, can compute the data structure of s bits in time polynomial in s.
2. *Explicit query scheme*: an explicit query scheme is one where the locations of the bit probes to be performed by the query algorithm are computable in time polynomial in t and $\lg u$, for any query element $q \in [1, u]$.

We use the terminology *fully explicit scheme* to describe a scheme that is both an explicit storage scheme *and* an explicit query scheme.

We present several new data structures for the bitprobe model that are deterministic, adaptive, and fully explicit. In particular, we are interested in the case where the number of probes, t, is small and does not depend on the size of the universe, u. We discuss our results in detail in Section 4.3, but first review the previous work in this area, and introduce some definitions.

4.2 Previous Results

We begin with the problem of membership and discuss deterministic upper and lower bounds. We then discuss the well-studied special case where $n = 2$. After that we discuss some properties of the previous results, and introduce some new terminology. Finally, we discuss previous results for rank, range counting and emptiness queries in the bitprobe models. Note that in the remainder of this chapter, we ignore floor and ceiling operators, except in cases where asymptotic behaviour is affected.[1]

4.2.1 Deterministic Schemes for Membership

When discussing previous work, we adopt the notation of Radhakrishnan, Shah, and Shannigrahi [134] and use $s_N(n, u, t)$ to denote the minimum space s such that there exists a deterministic non-adaptive (n, u, s, t)-scheme. We use $s_A(n, u, t)$ analogously for adaptive schemes.

[1]Note also that in this chapter there is no implicit ceiling operator on $\lg x$.

The first effort to address the worst-case behaviour of the membership problem in the bitprobe model was that of Buhrman et al. [30]. They showed that

$$\binom{u}{n} \leq \max_{i \leq nt} \binom{2s_A(n,u,t)}{i}. \tag{4.2.1}$$

As pointed out by Alon and Feige [3] this bound can be written as:

$$s_A(n,u,t) = \Omega(tn^{1-1/t}u^{1/t}), \tag{4.2.2}$$

when $n \leq u^{1-\varepsilon}$, for a constant $\varepsilon > 0$. The bound is proved via an information theoretic argument. Note that this bound implies that the trivial $(n,u,u,1)$-scheme is optimal to within constant factors, when $n \leq u^{1-\varepsilon}$, for some constant $\varepsilon > 0$. It also implies (Corollary 1.1 in [30]) that FKS-hashing makes an optimal number of probes (to within a constant factor) for schemes that use $\Theta(n \lg u)$ bits of space, when $n \leq u^{1-\varepsilon}$, for some constant $\varepsilon > 0$. For the case when $n = \Theta(u)$, which is not covered by the lower bound of Inequality 4.2.1, Viola [153] has shown that, for all sufficiently large u divisible by 3, $s_A(u/3,u,t) \geq \lg \binom{u}{u/3} + u/2^{O(t)} - \lg u$. Pagh [127] asymptotically improved upon the number of probes required by FKS-hashing by describing a scheme that uses $\Theta(\lg(u/n)+1)$ probes.

When both t and n are very small relative to u, the primary concern is with the exponent of u. In particular, the goal is to make the exponent as close to the Equation 4.2.2 lower bound of $1/t$ as possible, even though—as we shall see—it is not possible even when $n = 2$. For upper bounds, Buhrman et al. showed

$$s_N(n,u,t) = O(ntu^{4/(t+1)}) \text{ for odd } t, \text{ and} \tag{4.2.3}$$

$$s_A(n,u,t) = O(nt'u^{1/t'}), \tag{4.2.4}$$

where $t' = t - \Theta(\lg n + \lg \lg u)$ and $t' > 0$.[2] The first bound (Equation 4.2.3) is non-explicit and relies on the existence of a special kind of expander graph, whereas the second is fully explicit and generalizes FKS-hashing. The first bound shows that non-adaptivity is not too restrictive, since the exponent in the space bound for the non-adaptive scheme is only about four times larger than that of the lower bound in Equation 4.2.2. Compare this with our earlier discussion

[2]Our quoted bound matches the one stated in the proof of Theorem 7 (part 2), rather than the bound stated in the theorem itself.

Type	Bound	Constraints	Reference
Lower Bound	$\binom{u}{n} \leq \max_{i \leq nt} \binom{2s_A(n,u,t)}{i}$		[30]
Lower Bound	$s_A(n,u,t) = \Omega(tn^{1-1/t}u^{1/t})$	$n \leq u^{1-\varepsilon}$ for $\varepsilon > 0$	[30, 3]
Lower Bound	$s_A(u/3,u,t) \geq \lg \binom{u}{u/3} + u/2^{O(t)} - \lg u$	u divisible by 3	[153]
Lower Bound	$s_N(n,u,3) = \Omega(n^{1/2}u^{1/2}/\lg^{1/2} u)$	Valid when $n > 16 \lg u$	[3]
Upper Bound†	$s_A(n,u,\Theta(\lg u)) \leq \lg \binom{u}{n} + o(\lg \binom{u}{n})$		[29, 126, 128]
Upper Bound†	$s_A(n,u,\Theta(\lg(u/n)) \leq \Theta(\lg \binom{u}{n})$		[127]
Upper Bound†	$s_A(n,u,t) = O(nt'u^{1/t'})$	$t' = t - \Theta(\lg n + \lg\lg u)$ and $t > \Theta(\lg n + \lg\lg u)$	[30]
Upper Bound†	$s_A(n,u,t) = O\left(t^n u^{1/(t-n+1)}\right)$	$t > n \geq 2$	[134]
Upper Bound	$s_A(n,u,t) = O\left(ntu^{1/(t-(n-1)(t-1)2^{1-t})}\right)$	$t > n \geq 2$	[134]
Upper Bound	$s_N(n,u,t) = O(ntu^{4/(t+1)})$	t is odd	[30]
Upper Bound†	$s_A(n,u,\lceil \lg\lg n \rceil + 2) = o(u)$	$n = O(u^{1/\lg\lg u})$	[133]
Upper Bound†	$s_A(n,u,\lceil \lg(n+1) \rceil + 1) \leq (n + \lceil \lg(n+1) \rceil)\sqrt{u}$		[133]
Upper Bound	$s_A(n,u,2) = O((un \lg(\lg(u)/n))/\lg u)$	$n < \lg u$	[3]
Upper Bound	$s_A(n,u,3) = O(n^{1/3}u^{2/3})$		[3]
Upper Bound	$s_N(n,u,4) = O(n^{1/3}u^{2/3})$		[3]
Upper Bound†	$s_N(n,u,\Theta(\sqrt{n \lg n})) = O(\sqrt{un \lg n})$		[137]

Table 4.1: Summary of results for deterministic membership schemes in the bitprobe model. A dagger (†) in the "Type" column indicates that the scheme is fully explicit.

of simulating an adaptive scheme using a non-adaptive scheme that proved non-adaptivity was no more than exponentially worse. The exponent in the second bound (Equation 4.2.4) matches that of Equation 4.2.2, after subtracting $\Theta(\lg n + \lg\lg u)$ probes. Note that the $\lg\lg u$ term comes from reading prime numbers stored by the data structure.

Radhakrishnan, Raman, and Rao [133] focused on explicit deterministic constructions for small numbers of probes, describing a fully explicit scheme that, for $n = O(u^{1/\lg\lg u})$, shows $s_A(n,u,\lceil \lg\lg n \rceil + 2) = o(u)$. They also gave the following fully explicit bound: $s_A(n,u,\lceil \lg(n+1) \rceil + 1) \leq (n + \lceil \lg(n+1) \rceil)\sqrt{u}$. The idea is to divide the universe into buckets of size $\sqrt{u}$, and assign $\lceil \lg(n+1) \rceil + 1$ bits to each bucket, indicating the rank of the largest element stored in the bucket. Rao [137] described several non-adaptive schemes. His main non-adaptive result is a fully explicit scheme showing that: $s_N(n,u,\Theta(\sqrt{n \lg n})) = O(\sqrt{un \lg n})$.

Later, Alon and Feige [3] improved the first result of Radhakrishnan, Raman, and Rao, for the case when $n < \lg u$, by providing an explicit storage scheme showing $s_A(n,u,2) = O((un \lg(\lg(u)/n))/\lg u)$. The storage scheme makes use of an explicit construction of regular bipartite graphs of high girth, combined with an application of Hall's Theorem. They also describe non-explicit schemes showing that $s_A(n,u,3) = O(n^{1/3}u^{2/3})$, and $s_N(n,u,4) = O(n^{1/3}u^{2/3})$, based on similar graph and hypergraph theoretic techniques. They note that if the number of probes is increased to a larger constant, these non-explicit schemes can be converted into explicit storage schemes. Finally, they showed that if $n > 16 \lg u$, $s_N(n,u,3) = \Omega(n^{1/2}u^{1/2}/\lg^{1/2} u)$. This lower bound is proved using techniques from extremal hypergraph theory.[3]

Radhakrishnan, Shah, and Shannigrahi [134] presented two results. The first is a fully explicit scheme that, for $t > n \geq 2$, shows

$$s_A(n,u,t) = O\left(t^n u^{1/(t-n+1)}\right). \tag{4.2.5}$$

The second result is a non-explicit scheme that, for $t > n \geq 2$, shows

$$s_A(n,u,t) = O\left(ntu^{1/(t-(n-1)(t-1)2^{1-t})}\right). \tag{4.2.6}$$

Equation 4.2.5 is proved using a recursive unary encoding scheme, whereas Equation 4.2.6 is proved by a probabilistic argument. The probabilistic argument essentially sends all the elements in $\mathcal{E}$ into one table from a set of 2^{t-1} possible tables. Each table has its own hash function, which is assumed to assign elements to table entries uniformly at random. The argument shows that false positives can be avoided by deleting at most half the elements from the universe. Thus, by remapping to a universe of size $2u$, we can store a universe of size u using this scheme. Note that although both Equations 4.2.5 and 4.2.6 approach the optimal space bound exponent for fixed n and sufficiently large t, the second bound is significantly stronger when n is close to t. Thus, there is a *significant gap* between existing explicit and non-explicit schemes. For the non-explicit scheme of Equation 4.2.6, we note that the first $t-1$ probes are non-adaptive: the scheme is 1-adaptive. This is in contrast to the explicit scheme of Equation 4.2.5 that is $(t-1)$-adaptive.

We return later in this section to introduce some terminology related to these schemes, but first discuss a special case of the membership problem.

[3]See also a survey by Blue [20] of the techniques used by Alon and Feige for non-adaptive upper and lower bounds.

Scheme	Lower Bound	Upper Bound	Constraints	Reference
$s_A(2,u,1)$	$\Omega(u)$	$O(u)\dagger$		[30]
$s_A(2,u,2)$	$\Omega(u^{4/7})$	$O(u^{2/3})\dagger$	$\Omega(u^{2/3})$ lower bound for a restricted class	[134, 133]
$s_A(2,u,3)$	$\Omega(u^{1/3})$	$O(u^{2/5})$		[30, 134]
$s_N(2,u,2)$	$\Omega(u)$	$O(u)\dagger$		[30]
$s_N(2,u,3)$	$\Omega(\sqrt{u})$	$O(\sqrt{u})\dagger$		[30, 134]
$s_A(2,u,t)$	$\Omega(u^{1/t})$	$O(t^2 u^{1/(t-1)})\dagger$		[134]
$s_A(2,u,t)$	$\Omega(u^{1/t})$	$O\left(tu^{1/(t-(t-1)2^{1-t})}\right)$		[134]

Table 4.2: Summary of results for deterministic membership schemes in the bitprobe model, for the special case when $n = 2$. A dagger (†) indicates that the scheme is fully explicit.

The case when $n = 2$:

If $n = 1$ then the scheme we discussed earlier can achieve the bound $s_A(1,u,t) \leq tu^{1/t}$, which matches the exponent-of-u in the lower bound of Equation 4.2.2. The idea is to store t characteristic bit vectors; one for each $1/t$-th fraction of the bits of the sole element to be stored.

The first non-trivial special case of the membership problem that has been studied heavily, though is still not very well understood, is when $t = 2$. Equation 4.2.2 implies $s_A(2,u,1) = \Omega(u)$, which raises the question of how the bound behaves for other values of t. We summarize the results for this special case in Table 4.2.

Buhrman et al. [30] showed that for non-adaptive schemes, adding a second probe does not improve the space bound over one probe, i.e., $s_N(2,u,2) = \Omega(u)$. However, if adaptivity is permitted, then it is possible to get a $o(u)$ space bound. This shows a strict separation between the power of adaptive and non-adaptive probes. However, rather surprisingly, even the 2-probe case is still not completely settled! For upper bounds, Radhakrishnan, Raman, and Rao [133] showed that $s_A(2,u,2) = O(u^{2/3})$ via a subtle fully explicit scheme. They also proved a matching lower bound for a restricted class of schemes, but could not show it in general. Later, Radhakrishnan, Shah, and Shannigrahi [134] showed that $s_A(2,u,2) = \Omega(u^{4/7})$ by modelling the problem as a graph, and making a forbidden subgraph argument. Interestingly, this is the only lower bound for adaptive schemes that beats the bound of Equation 4.2.2 when $n \ll u$. They also conjectured that the true lower bound asymptotically matches the $O(u^{2/3})$ upper bound.

For $t = 3$, the complexity of non-adaptive schemes is settled asymptotically [30, 134]: it is known that $s_N(2,u,3) = \Theta(\sqrt{u})$. However, for adaptive schemes there are no lower bounds other than that of Equation 4.2.2. Plugging $n = 2$ and $t = 3$ into Equation 4.2.6 implies there is a non-explicit scheme with $s_A(2, u, 3) = O(u^{2/5})$, whereas the fully explicit scheme of Equation 4.2.5 only yields the bound $s_A(2, u, 3) = O(u^{1/2})$. For general values of $t \geq 3$ in the $n = 2$ case, the scheme of Equation 4.2.6 yields the following bound:

$$s_A(2, u, t) = O\left(tu^{1/(t-(t-1)2^{1-t})}\right). \tag{4.2.7}$$

Radhakrishnan, Shah, and Shannigrahi [134] pointed out that by using limited independence their scheme can be turned into explicit storage scheme. However, they left finding a fully explicit scheme that matches their non-explicit bound in this case as an open problem.

Blocking Schemes

We observe that the functions used to determine the locations to probe on the first $t-1$ probes for the non-explicit scheme of Equation 4.2.6 have a particular format. In particular, if the space bound of the non-explicit scheme is $\Theta(u^c)$ for some constant $c > 0$—let us treat t as a constant to simplify the discussion—then the scheme divides the bits of the query element q into *blocks* $B_1(q), ..., B_t(q)$, where the first $t-1$ blocks consist of $c\lceil \lg u \rceil$ bits, and the t-th block is potentially smaller. We refer to such a scheme that divides the bits of the query element into blocks, such that the location of probe i, $1 \leq i \leq t-1$ is specified by (the number represented by) $B_i(q)$, as *blocking*. Note that this definition makes no claims about the function used to determine the final probe. Thus, in the terminology of this section, the non-explicit scheme of Equation 4.2.6 is blocking, whereas the explicit scheme of Equation 4.2.5 is not.

Explicit Schemes and Cramér's Conjecture

There are two separate issues that must be addressed in order to make an the non-explicit scheme of Radhakrishnan, Shah, and Shannigrahi (Equation 4.2.6) explicit. The first is that the analysis makes use of a non-trivial re-mapping phase, where as many as half the elements in the universe are deleted because they have certain bad properties. Thus, implementing the scheme seems to force the query algorithm to do significant preprocessing—i.e., polynomial in s work—in order to determine where in the data structure to probe when presented with a query element.

The second (and perhaps more important) issue is that the location of the t-th probe is assumed to be computed using a set of independent hashing functions. Since the query algorithm only knows n, u, and t *a priori*, this raises the question of how such a set of hash functions can be computed. For example, if the hash functions use prime numbers of size $\Theta(p)$ then these primes must be acquired by the query algorithm. This can either be done by computing them using a pre-defined deterministic strategy that matches the one used by the data structure, or by reading them in $\Theta(\lg p)$ bit probes from the data structure. Indeed, the FKS-based scheme of Buhrman et al. does the latter, which accounts for the $\lg \lg u$ term in the exponent in their space bound. If we choose the former strategy, we make the observation that computing a prime number of size $\Theta(p)$ is a well studied problem, and currently all deterministic methods that take $O(\texttt{polylog}(p))$ time for this problem rely on Cramér's Conjecture [145] (or similar conjectures). Informally, Cramér's Conjecture states that the difference between consecutive primes is polylogarithmic in the size of the primes. Without assuming any such conjecture the best time bound for deterministic algorithms is $\tilde{O}(p^{0.525})$ [145]. Thus, unless we assume Cramér's conjecture, we cannot make use of arbitrary prime numbers of size $\Theta(u^{\varepsilon})$, for some constant $\varepsilon > 0$, in an explicit scheme.

4.2.2 Rank, Range Counting, and Emptiness

Unlike the membership problem, there appear to be no previous *upper bounds* for the *static* rank, counting, and emptiness problems *specifically designed* for the bitprobe model. This is in contrast to the dynamic versions of these problems, where techniques mentioned above can be used [129, 135]. However, in the static case, there are well-known word-RAM solutions that can solve all three problems simultaneously (by reducing them to rank queries). Note that a trivial solution based on balanced search trees occupies $\Theta(n \lg u)$ bits, and uses $\Theta(\lg n \lg u)$ bit probes to answer a query. To improve the number of bit probes to $\Theta(\lg u)$, the *fully indexable dictionary* of Lemma 1.5.4 can be used, which occupies $\lg \binom{u}{n} + O(u \lg \lg u / \lg u)$ bits of space.[4] For the range emptiness problem, Alstrup, Brodal, and Rauhe [6] describe a data structure that answers queries in $\Theta(\lg u)$ bit probes, and occupies $O(n \lg u)$ bits.

For a survey of other data structures in the bitprobe model see the recent survey by Nicholson, Raman, and Rao [125].

[4] As noted in Section 1.5.3, the $O((u \lg \lg u)/ \lg u)$ term can be improved to $O(u/\texttt{polylog}(u))$ [128], and other trade-offs are available [83].

4.3 Our Contributions

4.3.1 Membership

In Section 4.4, we present new explicit deterministic schemes for membership in the bitprobe model. We have two main results, one for the special case when $n = 2$, and one for the more general case for $n \geq 3$.

Result for $n = 2$:

We answer the open problem of Radhakrishnan, Shah, and Shannigrahi [134] and provide a fully explicit scheme demonstrating $s_A(2, u, 3) = O(u^{2/5})$. Furthermore, we generalize this scheme to $t \geq 3$ probes, yielding the following fully explicit bound:

$$s_A(2, u, t) \leq (2^t - 1)u^{1/(t-2^{2-t})}. \tag{4.3.1}$$

For $t > 3$, and $2^t = o(u^\varepsilon)$ for any constant $\varepsilon > 0$, the exponent of u in this bound is smaller than that of Equation 4.2.7. Thus, we not only make the scheme fully explicit, but show how to improve the best previous exponent for the case of $n = 2$. We note that the non-explicit scheme in Equation 4.2.7 uses adaptivity on only the final probe, whereas our scheme uses adaptivity on every probe, with the exception of the first. We believe this added adaptivity is the reason we are able to sidestep the issues discussed in Section 4.2.1. We provide a commentary on what we believe are the limitations of our approach, and why it appears difficult to generalize this method to larger values of n in Section 4.4.2.

Result for $n \geq 3$:

We describe a fully explicit adaptive scheme for $n \geq 3$, and $2^t = o(u^\varepsilon)$ for any constant $\varepsilon > 0$, that significantly improves upon the bounds of Equation 4.2.5, and also works for a wider range of values for t. In particular, for $n \geq 3$ and $t \geq 2\lfloor \lg(n) \rfloor + 1$ we get the following bound:

$$s_A(n, u, t) \leq (2^t - 1)u^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})}. \tag{4.3.2}$$

It is important to note that the exponent in Inequality 4.3.2 is not only significantly better than that of Equation 4.2.5, but is also applicable in the range $t \in [2\lfloor \lg(n) \rfloor + 1, n]$; both the fully explicit and non-explicit scheme of Radhakrishnan et al. [134] require $t > n$. We also note that this scheme improves the exponent of the FKS-based scheme of Buhrman et al. [30] (Equation 4.2.4) by removing the dependence on $\lg \lg u$ from the exponent of u.

4.3.2 Rank, Range Counting, and Emptiness

In Section 4.5, we modify the fully explicit membership data structure of Inequality 4.3.2 to solve the rank problem, achieving the following bound, for $t \geq \lceil \lg(n+1) \rceil + 2\lfloor \lg(n-2) \rfloor + 2$:

$$s_A(n,u,t) \leq (2^t - 1)\lceil \lg(n+1) \rceil u^{1/(t-\lceil \lg(n+1) \rceil - 2\lfloor \lg(n-2) \rfloor - 1)}. \tag{4.3.3}$$

As far as we are aware this is the first non-trivial bit probe scheme for the rank problem. Since two rank queries can be used to solve range counting and emptiness queries, in Section 4.6 we observe that the same space bound applies to those problem, though with double the number of probes. Finally, we observe some trade-offs for the range emptiness problem, and show that even a constant number of probes suffice to reduce the quadratic (in u) space required by a one probe scheme to linear. In Section 4.7 we conclude with a brief summary and a few remarks.

4.4 Membership Queries

4.4.1 Fully Explicit Adaptive Schemes for Two Elements

In this section we prove Inequality 4.3.1. We begin by explaining a special case of the equation in detail.

Theorem 4.4.1. *There is an explicit adaptive* $(2, u, 7u^{2/5}, 3)$*-scheme for the membership problem. The scheme is* 2*-adaptive, and blocking.*

Proof. We define a *subblock* to be $\lceil \lg u \rceil / 5$ consecutive bits. A *block* is two consecutive subblocks. Let $z \in [0, u-1]$ be an integer[5], and z_i be the i-th bit in the binary representation of z, for

[5]For notational convenience, we remap the universe $[1, u]$ to $[0, u-1]$ in the following proofs.

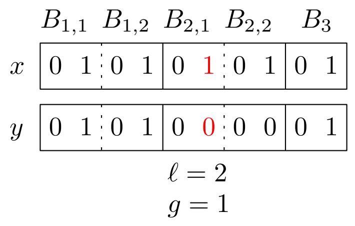


Figure 4.2: Given integers $x = 341$, and $y = 321$, and a universe $[0, 1023]$, we divide the bits of x and y into blocks.

$1 \leq i \leq \lceil \lg u \rceil$, where z_1 is the most significant bit. We divide the binary representation of z into blocks, $B_1(z), ..., B_3(z)$, where $B_j(z)$ are bits $z_{(j-1)\chi+1}, z_{(j-1)\chi+2}, ..., z_{\min(j\chi, \lceil \lg u \rceil)}$, where $1 \leq j \leq t$ and $\chi = 2\lceil \lg u \rceil / 5$. The block $B_3(z)$ is special in that it is not a complete block: i.e., it will only consist of one subblock, rather than two. Finally, we use $B_{j,k}(z)$ to denote the k-th subblock of the j-th block of z, for $1 \leq k \leq 2$. An illustration of these definitions can be found in Figure 4.2. Note that in the following description when we refer to a particular block or subblock, we are referring to the binary number represented by the bits contained in the block, following the convention that the leftmost bit is the most significant bit.

Our scheme stores 7 tables. Each table is denoted $\mathcal{T}_{\mathcal{S}}$, where $\mathcal{S}$ is a bit string in $\{\epsilon, 0, 1, 00, 01, 10, 11\}$, and ϵ represents the empty string. Each table occupies $u^{2/5}$ bits. Thus, the total space is $7u^{2/5}$.

We begin by describing the algorithm for searching the data structure. Let q be the element we are searching for:

1. We probe table $\mathcal{T}_\epsilon$ at location $B_1(q)$, and are given bit r_1.
2. We probe table $\mathcal{T}_{r_1}$ at location $B_2(q)$, and are given bit r_2.
3. We read the bit r_3 by probing table $\mathcal{T}_{r_1 r_2}$ at location:

$$\Big((B_{1,r_2+1}(q) + B_{2,r_1+1}(q)) \mod u^{1/5}\Big)\, u^{1/5} + B_3(q). \tag{4.4.1}$$

4. If $r_3 = 1$ then we return `YES`, otherwise we return `NO`.

Next we describe how to construct the data structure, i.e., set the bits. Consider the two elements $x, y \in [0, u-1]$ that we wish to store, and assume without loss of generality that $x < y$.

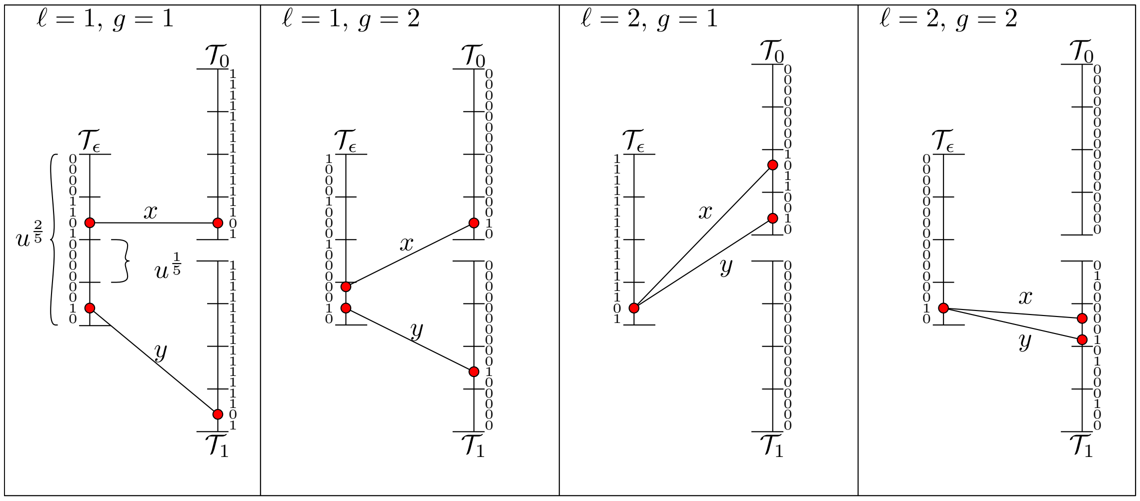


Figure 4.3: Four cases illustrating how to set the bits in tables $\mathcal{T}_\epsilon$, $\mathcal{T}_0$, and $\mathcal{T}_1$. In this example the universe is the range $[0, 1023]$.

Let ℓ be the smallest integer such that $B_\ell(x)$ differs from $B_\ell(y)$, and g be the smallest integer such that $B_{\ell,g}(x)$ differs from $B_{\ell,g}(y)$. See Figure 4.2 for an example. Let $g' = g - 1$: it is one bit representing in which subblock x and y differ. We next argue that we can assume $\ell < 3$, since a trivial assignment exists in the alternate case.

If $\ell = 3$ or $x = y$, then we are free to store x and y in any of the 7 tables $\{\mathcal{T}_\mathcal{S}\}$, where $|\mathcal{S}| = 2$. Without loss of generality, assume we choose $\mathcal{T}_{11}$. Thus, we assign the characteristic bit vector of $B_1(x)$ to table $\mathcal{T}_\epsilon$, and the characteristic bit vector of $B_2(x)$ to table $\mathcal{T}_1$. In the final table, $\mathcal{T}_{11}$, we store (at most) two ones in the locations that can be computed by plugging in x and y into Equation 4.4.1. All other entries are set to zero. It is not difficult to see that the search for q will function correctly for this assignment, because of the fact that x and y are identical in all blocks except possibly B_3.

If $\ell = 1$, then let $\mathcal{S}_x = 0g'$, and $\mathcal{S}_y = 1g'$. Otherwise, if $\ell = 2$, then let $\mathcal{S}_x = g'0$, and $\mathcal{S}_y = g'1$. Our aim is to store x in $\mathcal{T}_{\mathcal{S}_x}$ and y in $\mathcal{T}_{\mathcal{S}_y}$, and ensure that the search algorithm always returns the correct result. We use the notation $\mathcal{S}_{z,\beta}$ to denote the β-bit prefix of $\mathcal{S}_z$. For example, if $\mathcal{S}_z = 01$, then $\mathcal{S}_{z,0} = \epsilon$, $\mathcal{S}_{z,1} = 0$, $\mathcal{S}_{z,2} = 01$.

We now describe the assignment of bits to the tables:

1. For the value $\beta \in [0, 1]$, where $\beta \neq \ell - 1$, we describe how to set the values in $\mathcal{T}_{\mathcal{S}_{z,\beta}}$, for

$z \in \{x, y\}$. Let v be the $(\beta+1)$-th bit of $\mathcal{S}_z$. If v is a 0, then we set the bit in location $B_{\beta+1}(z)$ to 0, and all other bits to 1 in $\mathcal{T}_{\mathcal{S}_{z,\beta}}$. Otherwise, if v is 1, then we set the bit in location $B_{\beta+1}(z)$ to 1, and all other bits to 0.

2. Next, we explain how to set the bits in $\hat{\mathcal{T}} = \mathcal{T}_{\mathcal{S}_{x,\ell-1}} = \mathcal{T}_{\mathcal{S}_{y,\ell-1}}$. We store a 0 in location $B_\ell(x)$, and a 1 in location $B_\ell(y)$ in $\hat{\mathcal{T}}$. All locations $\gamma_x \neq B_\ell(x)$, such that $\lfloor \gamma_x / u^{(1-g')/5} \rfloor \equiv B_{\ell,g}(x) \mod u^{1/5}$ are assigned a 1. All locations $\gamma_y \neq B_\ell(y)$, such that $\lfloor \gamma_y / u^{(1-g')/5} \rfloor \equiv B_{\ell,g}(y) \mod u^{1/5}$ are assigned a 0. After setting the tables in the way described above, we perform a search for x and y using the search algorithm, and set the two bits corresponding to x and y to 1 in tables $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$, respectively. Finally, all table locations that remain unspecified are set to 0. We give an example of how to set tables $\mathcal{T}_\epsilon$, $\mathcal{T}_0$, and $\mathcal{T}_1$ in each of the four cases in Figure 4.3.

All that remains is to prove that the search algorithm will always return the correct result, provided we set the bits as described above. In the first case, if the query element q is equal to either x or y, then we will return `YES`, which is correct. In the second case, suppose $q \neq x$ *and* $q \neq y$, and $\mathcal{S}_q$ is the sequence of first 2 bits returned by the search algorithm for q. If $\mathcal{S}_q \neq \mathcal{S}_x$ *and* $\mathcal{S}_q \neq \mathcal{S}_y$, then we will return `NO`—since all tables are zeroed out except those containing x and y—which is correct. Otherwise, suppose $\mathcal{S}_q = \mathcal{S}_z$ where $z \in \{x, y\}$; note that we can assume $\mathcal{S}_x \neq \mathcal{S}_y$ by the assumption that $\ell < 3$. If q differs from z in block B_3, then the search algorithm will return `NO`, which is correct. Thus, we can assume q differs from z somewhere other than block B_3. Consider the table $\mathcal{T}_{\mathcal{S}_{q,\beta}} = \mathcal{T}_{\mathcal{S}_{z,\beta}}$, where $\beta \in [0,1]$ *and* $\beta \neq \ell - 1$, and let v denote the value of the $(\beta+1)$-th bit in $\mathcal{S}_q$. Since the table $\mathcal{T}_{\mathcal{S}_{z,\beta}}$ contains only one location that stores bit v, we can infer that q and z can differ *only* in block ℓ. However, $B_{\ell,g}(q) \neq B_{\ell,g}(z)$, according to the way we set the bits in table $\hat{\mathcal{T}}$, since $q \neq z$. Based on the discussion above, we have that either:

1. $B_{1,r_2+1}(q) \neq B_{1,r_2+1}(z)$ *and* $B_{2,r_1+1}(q) = B_{2,r_1+1}(z)$; or
2. $B_{1,r_2+1}(q) = B_{1,r_2+1}(z)$ *and* $B_{2,r_1+1}(q) \neq B_{2,r_1+1}(z)$.

This implies the following:

$$\left((B_{1,r_2+1}(q) + B_{2,r_1+1}(q)) \mod u^{1/5}\right) \neq \left((B_{1,r_2+1}(z) + B_{2,r_1+1}(z)) \mod u^{1/5}\right), \quad (4.4.2)$$

and we will return the correct answer of `NO`. This completes the proof of correctness. □

Now we give the generalization without as much intuition:

Theorem 4.4.2. *For $t \geq 3$, there is a fully explicit adaptive $(2, u, (2^t - 1)u^{1/(t-2^{2-t})}, t)$-scheme for the membership problem. The scheme is $(t-1)$-adaptive, and blocking.*

Proof. We define a *subblock* to be $(\lceil \lg u \rceil / (t2^{t-2} - 1))$ consecutive bits. A *block* is 2^{t-2} consecutive subblocks. Let $z \in [0, u-1]$ be an integer, and z_i be the i-th bit in the binary representation of z, for $1 \leq i \leq \lceil \lg u \rceil$, where z_1 is the most significant bit. We divide the binary representation of z into blocks, $B_1(z), ..., B_t(z)$, where $B_j(z)$ are bits $z_{(j-1)\chi+1}, z_{(j-1)\chi+2}, ..., z_{\min(j\chi, \lceil \lg u \rceil)}$, where $1 \leq j \leq t$ and $\chi = ((2^{t-2})\lceil \lg u \rceil)/(t2^{t-2} - 1)$. The block $B_t(z)$ is special in that it is not a complete block: i.e., it will only consist of $2^{t-2} - 1$ consecutive subblocks, rather than 2^{t-2}. Finally, we use $B_{j,k}(z)$ to denote the k-th subblock of the j-th block of z, for $1 \leq k \leq 2^{t-2}$. Note that in the following description when we refer to a particular block or subblock, we are referring to the binary number represented by the bits contained in the block, following the convention that the leftmost bit is the most significant bit.

Our scheme stores $2^t - 1$ tables. Each table is denoted $\mathcal{T}_{\mathcal{S}}$, where $\mathcal{S}$ is a binary string of length between 0 (an empty string), and $t-1$ bits. Each table $\mathcal{T}_{\mathcal{S}}$, where $|\mathcal{S}| \leq t$ will store $u^{(2^{t-2})/(t2^{t-2}-1)}$ bits. The sum of the sizes of these tables is no more than the space bound claimed in the statement of the theorem.

We begin by describing the algorithm for searching the data structure:

1. Let q be the query element, $\mathcal{S}$ be an empty binary string, and $i = 1$.

2. We probe table $\mathcal{T}_{\mathcal{S}}$ at location $B_i(q)$, and are given bit r_i. We append r_i to $\mathcal{S}$ (i.e., add r_i to the end of $\mathcal{S}$) and increment i. If $i \leq t-1$, then we repeat this step.

3. At this point $\mathcal{S}$ consists of $t-1$ bits. Let $\mathcal{S}[j]$ be the binary number that results from deleting the j-th digit (counting left to right) from $\mathcal{S}$. We read the bit r_t by probing table $\mathcal{T}_{\mathcal{S}}$ at location:

$$\left(\left(\sum_{j=1}^{t-1} B_{j,\mathcal{S}[j]+1}(q) \right) \mod u^{1/(t2^{t-2}-1)} \right) u^{(2^{t-2}-1)/(t2^{t-2}-1)} + B_t(q). \tag{4.4.3}$$

4. If $r_t = 1$ then we return `YES`, otherwise we return `NO`.

Next we describe how to construct the data structure, i.e., set the bits. Consider the two elements $x, y \in [0, u-1]$ that we wish to store, and assume without loss of generality that $x < y$. Let ℓ be the smallest integer such that $B_\ell(x)$ differs from $B_\ell(y)$, and g be the smallest integer such that $B_{\ell,g}(x)$ differs from $B_{\ell,g}(y)$. We next argue that we can assume $\ell < t$, since a trivial assignment exists in the alternate case.

If $\ell = t$ or $x = y$, then we are free to store x and y in any of the 2^{t-1} tables $\{\mathcal{T}_\mathcal{S}\}$, where $|\mathcal{S}| = t-1$. Without loss of generality, assume we choose $\mathcal{T}_{\mathcal{S}'}$, where $\mathcal{S}'$ is $t-1$ ones. Thus, we assign the characteristic bit vector of $B_1(x)$ to table $\mathcal{T}_\epsilon$, $B_2(x)$ to table $\mathcal{T}_1$, $B_3(x)$ to table $\mathcal{T}_{11}$, and so on. In the final table, $\mathcal{T}_{\mathcal{S}'}$, we store (at most) two ones in the locations that can be computed by plugging in x and y into Equation 4.4.3. All other entries are set to zero. It is not difficult to see that the search algorithm will function correctly for this assignment, because of the fact that x and y are the identical in all blocks except possibly B_t.

Let $g_1, ..., g_{t-2}$ be the digits of the binary representation of $g-1$, $\mathcal{S}_x = g_1, g_2, ..., g_{\ell-1}, 0, g_\ell, ..., g_{t-2}$, and $\mathcal{S}_y = g_1, g_2, ..., g_{\ell-1}, 1, g_\ell, ..., g_{t-2}$. We use the notation $\mathcal{S}_{z,\beta}$ to denote the β-bit prefix of $\mathcal{S}_z$. For each $\beta \in [0, t-2]$, where $\beta \neq \ell - 1$, we describe how to set the values in $\mathcal{T}_{\mathcal{S}_{z,\beta}}$, for $z \in \{x, y\}$. Let v be the $(\beta+1)$-th bit of $\mathcal{S}_z$. If v is a 0, then we set the bit in location $B_{\beta+1}(z)$ to 0, and all other bits to 1 in $\mathcal{T}_{\mathcal{S}_{z,\beta}}$. Otherwise, if v is 1, then we set the bit in location $B_{\beta+1}(z)$ to 1, and all other bits to 0. We now explain how to set the bits in $\hat{\mathcal{T}} = \mathcal{T}_{\mathcal{S}_{x,\ell-1}} = \mathcal{T}_{\mathcal{S}_{y,\ell-1}}$. We store a 0 in location $B_\ell(x)$, and a 1 in location $B_\ell(y)$ in $\hat{\mathcal{T}}$. All locations $\gamma_x \neq B_\ell(x)$, such that $\lfloor \gamma_x / u^{(2^{t-2}-1-g)/(t2^{t-2}-1)} \rfloor \equiv B_{\ell,g}(x) \mod u^{1/(t2^{t-2}-1)}$ are assigned a 1. All locations $\gamma_y \neq B_\ell(y)$, such that $\lfloor \gamma_y / u^{(2^{t-2}-1-g)/(t2^{t-2}-1)} \rfloor \equiv B_{\ell,g}(y) \mod u^{1/(t2^{t-2}-1)}$ are assigned a 0. After setting the tables in the way described above, we perform a search for x and y using the search algorithm, and set the two bits corresponding to x and y to 1 in tables $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$, respectively. Finally, all table locations that remain unspecified are set to 0.

All that remains is to prove that the search algorithm will always return the correct result, provided we set the bits as described above. In the first case, if the query element q is equal to either x or y, then we will return `YES`, which is correct. In the second case, suppose $q \neq x$ *and* $q \neq y$, and $\mathcal{S}_q$ is the sequence of first $t-1$ bits returned by the search algorithm for q. If $\mathcal{S}_q \neq \mathcal{S}_x$ *and* $\mathcal{S}_q \neq \mathcal{S}_y$, then we will return `NO`—since all tables are zeroed out except those containing x and y—which is correct. Otherwise, suppose $\mathcal{S}_q = \mathcal{S}_z$ where $z \in \{x, y\}$; note that we can assume

$\mathcal{S}_x \neq \mathcal{S}_y$ by the assumption that $\ell < t$. If q differs from z in block B_t, then the search algorithm will return `NO`, which is correct. Thus, we can assume q differs from z somewhere other than block B_t. Consider any table $\mathcal{T}_{\mathcal{S}_{q,\beta}} = \mathcal{T}_{\mathcal{S}_{z,\beta}}$, where $\beta \in [0, t-2]$ *and* $\beta \neq \ell - 1$, and let v denote the value of the β-th bit in $\mathcal{S}_q$. Since the table contains only one location that stores v, we can infer that q and z can differ *only* in block ℓ. However, $B_{\ell,g}(q) \neq B_{\ell,g}(z)$, according to the way we set the bits in table $\hat{\mathcal{T}}$, since $q \neq z$. Thus, the following equation holds:

$$\left(\sum_{j=1}^{t-1} B_{j,\mathcal{S}_q[j]+1}(q)\right) \not\equiv \left(\sum_{j=1}^{t-1} B_{j,\mathcal{S}_z[j]+1}(z)\right) \mod u^{1/(t2^{t-2}-1)}, \tag{4.4.4}$$

and we will return the correct answer of `NO`. This completes the proof of correctness. □

4.4.2 Limitations of the Blocking Scheme Approach for $n \geq 3$

In this section we discuss ρ-adaptive schemes for $\rho < t-1$, and, based on this discussion, explain why the techniques of the Theorem 4.4.2 appear to be limited to the case of $n = 2$.

We begin by noting that, based on our earlier discussion, there is a fully explicit $(2, u, 7u^{2/5}, 4)$-scheme that is blocking and 1-adaptive; we simulate the 2-adaptive scheme of Theorem 4.4.1 with 3 non-adaptive probes, followed by a single adaptive probe. This raises the question of whether there is a trade-off between adaptivity and space for fully explicit schemes.

In an attempt to answer this question, we describe a t-probe blocking scheme that is $(t-2)$-adaptive, but requires more space than the t-probe $(t-1)$-adaptive blocking scheme of Theorem 4.4.2. We note that a special case of the following theorem is the first fully explicit 2-adaptive $(2, u, \Theta(u^{1/3}), 4)$-scheme: the previous explicit 4-probe schemes that achieve $\Theta(u^{1/3})$ space is 3-adaptive [134].

Theorem 4.4.3. *For any $t \geq 4$, there is a*

$$(2, u, (2^t - 2)u^{2^{t-3}/((t-1/2)2^{t-3}-1)}, t)\text{-}scheme$$

that is $(t-2)$-adaptive, and blocking.

Proof. The idea is similar to the proof of Theorem 4.4.2, with some minor changes; to help with understanding we have included a running commentary on the values of various parameters for the case when $t = 4$. A *microblock* is $(\lg u)1/((t-1/2)2^{t-3}-1)$ consecutive bits, whereas a *miniblock* is $(\lg u)2^{t-4}/((t-1/2)2^{t-3}-1)$ consecutive bits. Thus, for the case of $t = 4$ a microblock has the same size as a miniblock: $\lg u/6$ bits. A block is 2^{t-3} consecutive microblocks, or, equivalently, 2 consecutive miniblocks. In the case when $t = 4$, a block is the size of either two miniblocks or two microblocks. We use the same notation as in Theorem 4.4.2, except $B_{j,k}(z)$ is used to denote the k-th microblock of the j-th block of z, and $\hat{B}_{j,k}(z)$ is used to denote the k-th miniblock of the j-th block of z; for the $t = 4$ case this notational change makes no difference. Since the scheme is non-adaptive on the first two probes, we use $\mathcal{T}_{\epsilon_1}$ to denote the first table probed and $\mathcal{T}_{\epsilon_2}$ to denote the second table probed.

We now describe the algorithm for searching the data structure:

1. Let q be the query element. Probe tables $\mathcal{T}_{\epsilon_1}$ and $\mathcal{T}_{\epsilon_2}$ at locations $B_1(q)$ and $B_2(q)$ to get bits r_1 and r_2, respectively.

2. Let $\mathcal{S} = r_1 r_2$, and $i = 3$.

3. We probe table $\mathcal{T}_{\mathcal{S}}$ at location $B_i(q)$, and are given bit r_i. We append r_i to $\mathcal{S}$ (i.e., add r_i to the end of $\mathcal{S}$) and increment i. If $i \le t-1$, then we repeat this step.

4. At this point $\mathcal{S}$ consists of $t-1$ bits. Let $\mathcal{S}[j]$ be the binary number that results from deleting the 1-st *and* j-th digit (counting left to right) from $\mathcal{S}$. We read the bit r_t by probing table $\mathcal{T}_{\mathcal{S}}$ at location:

$$\begin{aligned} &\left(\hat{B}_{1,r_1}(q)\right) u^{(2^{t-4})/((t-1/2)2^{t-3}-1)} + \\ \left(\left(\sum_{j=2}^{t-1} B_{j,\mathcal{S}[j]+1}(q)\right) \mod u^{1/((t-1/2)2^{t-3}-1)}\right) &u^{(2^{t-4}-1)/((t-1/2)2^{t-3}-1)} + B_t(q). \end{aligned} \tag{4.4.5}$$

For the case when $t = 4$ this simplifies to:

$$\left(B_{1,r_1}(q)\right) u^{1/6} + \left(B_{2,r_3}(q) + B_{3,r_2} \mod u^{1/6}\right). \tag{4.4.6}$$

5. If $r_t = 1$ then we return `YES`, otherwise we return `NO`.

To give some intuition, the bits returned during the first $t-1$ probes can be thought of as a code, where the first bit will specify whether x differs from y in miniblock $\hat{B}_{1,1}$ or $\hat{B}_{1,2}$. If the elements do not differ in these blocks, then we assign the bit arbitrarily. The remaining $t-2$ bits specify the first microblock where x differs from y in blocks $B_2, ..., B_{t-1}$, using the same interpretation as Theorem 4.4.2. Again, as in Theorem 4.4.2, if the elements do not differ in these blocks we are free to assign the bits arbitrarily. Thus, if x differs from y in $\hat{B}_{1,1}$, we set the bits in locations $B_1(x)$ and $B_1(y)$ in $\mathcal{T}_{\epsilon_1}$ to 0, and all other bits to 1. Otherwise, if x differs from y in $\hat{B}_{1,2}$ we set the bits in locations $B_1(x)$ and $B_1(y)$ in $\mathcal{T}_{\epsilon_1}$ to 1, and all other bits to 0.

Let ℓ be the smallest integer such that $B_\ell(x)$ differs from $B_\ell(y)$, for $\ell \in [2, t-1]$, and g be the smallest integer such that the microblock $B_{\ell,g}(x)$ differs from $B_{\ell,g}(y)$ for $g \in [0, 2^{t-3}]$. We encode ℓ and g into the remaining $t-2$ bits that specify the table where x and y are stored, and set the bits in the tables $\mathcal{T}_{\epsilon_2}$ and $\mathcal{T}_{\mathcal{S}}$, where $|\mathcal{S}| \leq t-2$, using exactly the same strategy as in Theorem 4.4.2, with respect to the remaining $t-2$ bits. Note that since we have $t-2$ bits, we can specify 2^{t-2} values, and this explains our microblock size. Finally, we use the search algorithm described above to set the bits in the final tables $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$. Any unspecified table entries are set to 0.

If $B_1(x) = B_1(y)$, then the scheme is correct. The only case to analyze is when $q \notin \{x, y\}$, but $\mathcal{S}_q = \mathcal{S}_x$ or $\mathcal{S}_y$; without loss of generality, assume $\mathcal{S}_q = \mathcal{S}_x$. In this case, it is clear that the bits of q must differ from x in miniblock $\hat{B}_{1,\mathcal{S}_{q,1}}$ or microblock $B_{\ell,g}(q)$ (both options are simultaneously possible). The way the bits are assigned guarantees that q will not collide with x (when plugged into Equation 4.4.5) in this case. □

We make the following conjecture regarding the limitations of our general approach. Note that the non-explicit scheme of Equation 4.2.6 achieves space $\Theta(u^{4/13})$ for $t = 4$, $n = 3$.

Conjecture 4.4.1. *There is no fully explicit $(2, u, \Theta(u^{1/3-\varepsilon}), 4)$-scheme that is both 2-adaptive and blocking for any $\varepsilon > 0$.*

Next we consider fully explicit $(3, u, s, 4)$-schemes that are blocking, and have the following property:

Split Property: Consider an arbitrary adaptive blocking scheme $\mathcal{A}$ for storing sets of size three from the universe $[1, u]$. $\mathcal{A}$ has the *split property* if for all elements $x, y \in [1, u]$ there exists an $z \in [1, u]$ such that:

1. $B_1(z) \neq B_1(x)$ and $B_1(z) \neq B_1(y)$; and
2. storing the set $\{x, y, z\}$ using $\mathcal{A}$ causes the bit stored in location $B_1(x)$ (in the first table) to have the same value as the bit stored in $B_1(y)$; and
3. the t-th probe ($t \geq 2$), when searching for element q, probes a different final table than when searching for $q' \neq q$, if the bit stored in location $B_1(q)$ is not equal to the bit stored in location $B_1(q')$.

Remark 4.4.1. *In our attempts to generalize our 2-element scheme, our $n = 3$ schemes all had the split property, and failed to match the space bounds of the non-explicit scheme of Equation 4.2.6.*

The next theorem gives some intuition as to why the fully explicit blocking techniques of Section 4.4.1 do not extend to match the bound of Equation 4.2.6 in the case when $n \geq 3$.

Theorem 4.4.4. *Assuming Conjecture 4.4.1, there is no explicit $(3, u, \Theta(u^{1/3-\varepsilon}), 4)$-scheme that both has the split property and is blocking, for any $\varepsilon > 0$.*

Proof. We will show by contradiction that if such a scheme $\mathcal{A}$ exists, then it violates Conjecture 4.4.1. The idea is to use the 3-element scheme $\mathcal{A}$ to store a set of size 2. Suppose we wish to store the set $\{x, y\}$. We find an element z that exists by the split property, and store $\{x, y, z\}$ in scheme $\mathcal{A}$.

We now explain the steps to modify $\mathcal{A}$ to get a scheme $\mathcal{A}'$ that violates Conjecture 4.4.1. We can assume, without loss of generality, that $\mathcal{A}$ is 3-adaptive: if $\mathcal{A}$ is 2 or 1-adaptive, then it immediately violates the conjecture. Additionally, since $\mathcal{A}$ is blocking, and has the split property, the table $\mathcal{T}_{\mathcal{S}_z}$ is different than both $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$. The first modification is to set all the bits in scheme $\mathcal{A}'$ table $\mathcal{T}_{\mathcal{S}_z}$ to 0.

Suppose the bit in location $B_1(x)$ of $\mathcal{T}_\epsilon$ (which is equal to the bit in location $B_1(y)$ by the split property), is equal to r. The second modification is to delete the table $\mathcal{T}_{|r-1|}$, and set all locations in $\mathcal{T}_\epsilon$, other than $B_1(x)$ and $B_1(y)$, to store bit $|r - 1|$. The query algorithm for scheme $\mathcal{A}'$, given query element z', will probe table $\mathcal{T}_\epsilon$ at location $B_1(z')$, followed by location $B_2(z')$ in $\mathcal{T}_r$. Note that, regardless of the value of r, the storage scheme is organized such that a single table in memory exists that is to be probed non-adaptively by the query algorithm on the second probe. All remaining probes are made exactly as in scheme $\mathcal{A}$. Thus, $\mathcal{A}'$ is 2-adaptive.

Clearly, $\mathcal{A}'$ returns `YES` for $\{x, y\}$, and since $\mathcal{T}_{\mathcal{S}_z}$ is zeroed out, a search for z will return `NO`. We now consider any element $z' \notin \{x, y\}$, and show that the answer must be `NO`. Note that if the bit in location $B_1(z')$ of $\mathcal{T}_\epsilon$ is not equal to r, then we will return `NO`, since $\mathcal{S}_{z'}$ will be different than both $\mathcal{S}_x$ and $\mathcal{S}_y$. Thus, if z' returns `YES`, it *must* match the bits of x or y in B_1; these are the only locations that store bit r. However, there is no difference between the search algorithm for scheme $\mathcal{A}$ and scheme $\mathcal{A}'$ for these elements. Thus, if $\mathcal{A}$ is correct, $\mathcal{A}'$ will return the correct result of `NO` for these elements. □

Remark 4.4.2. *Theorem 4.4.4 shows that, in order to match the bounds of Equation 4.2.6 using a blocking scheme, either (a) any scheme we devise for $n \geq 3$ must not have the split property, or (b) show that Conjecture 4.4.1 is false.*

4.4.3 Fully Explicit Adaptive Schemes for $n \geq 3$

In this section we describe a different general scheme for the case when $n \geq 3$; this bounds achieved by this fully explicit scheme are inferior to the non-explicit scheme of Equation 4.2.6, but the exponent-in-u is superior to previous fully explicit schemes. We begin by providing some definitions for decomposition techniques that are used in the recursive schemes described later. We then devise a scheme for small n that uses Theorem 4.4.2 recursively to outperform the previous explicit scheme of Radhakrishnan, Shah, and Shannigrahi (Equation 4.2.5). Finally, using a slightly more sophisticated decomposition technique, we develop the scheme of Inequality 4.3.2 that improves upon previous fully explicit results for the general case of $n \geq 3$, for many ranges of t.

A (j, k)-*decomposition* of the universe, $[1, u]$, divides the universe into buckets of size u^{1-k}, and assigns j bits to each bucket. Thus, a (j, k)-decomposition occupies ju^k bits of space. The *heaviest bucket* in a (j, k)-decomposition is the bucket which contains the most elements of the given set, breaking ties arbitrarily.

We now discuss some special kinds of (j, k)-decompositions. A $(1, k)$-*pivot decomposition* is a $(1, k)$-decomposition in which the heaviest bucket, is assigned a 1, all buckets to the left are assigned a 0, and all buckets to the right are assigned a 1 (see Figure 4.4 (a)). A $(2, k)$-*pivot decomposition* is a $(2, k)$-decomposition in which the heaviest bucket is assigned either a 10 or a 11 (the second bit is irrelevant for all of our forthcoming applications), the buckets to the left of the heaviest bucket are assigned 00, and the buckets to the right are assigned 01 (see Figure 4.4

(b)). A $(1,k)$-*balanced decomposition* is a $(1,k)$-decomposition in which the heaviest bucket, is assigned a 1, and all remaining buckets are assigned a 0 (see Figure 4.4 (c)).

Finally, a $(2,k)$-*balanced decomposition* is a $(2,k)$-decomposition in which the heaviest bucket is assigned a 10 or 11 (again, the second bit is irrelevant). We use the following lemma:

Lemma 4.4.1. *Suppose there is a set of buckets, where each bucket contains between* 0 *and* n *elements, and the total number of elements in all the buckets is* n*. Suppose the heaviest bucket is removed. It is possible to partition the remaining buckets into two groups* $\mathcal{G}_0$ *and* $\mathcal{G}_1$*, such that there are no more than* $\lfloor n/2 \rfloor$ *elements in total contained in the buckets of either group.*

Proof. We give a brief explanation by describing an algorithm to compute this partition. Initially, the groups $\mathcal{G}_0$ and $\mathcal{G}_1$ are empty. Consider any bucket that has not been assigned to a group, and call such a bucket *unassigned*. Clearly, any unassigned bucket contains no more than $j' = \min(n-j, j)$ elements, where j is the number of elements in the heaviest bucket. Suppose we add the leftmost unassigned bucket to $\mathcal{G}_0$ until the total number of elements contained in buckets in $\mathcal{G}_0$ is at least $\lfloor n/2 \rfloor - j' + 1$. Since $j' \geq 1$ in the non-trivial case, this is no more than $\lfloor n/2 \rfloor$. Furthermore, if we assign all remaining buckets to $\mathcal{G}_1$, it will contain no more than $n - j - (\lfloor n/2 \rfloor - j' + 1)$ elements. Regardless of whether $j' = j$ or $j' = n - j$ this value is no more than $\lfloor n/2 \rfloor$. □

We apply Lemma 4.4.1 to our $(2,k)$-balanced decomposition. The buckets in groups $\mathcal{G}_0$ and $\mathcal{G}_1$ are assigned bits 00 and 01, respectively (see Figure 4.4 (d)).

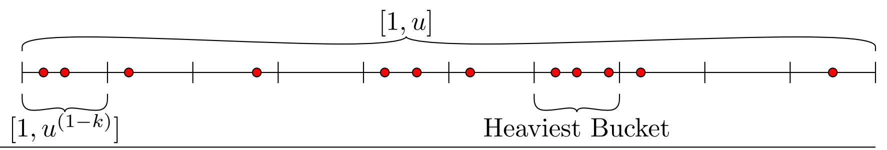
Basic Scheme for $n \geq 3$

We describe a recursive scheme for the case when $n \geq 3$. This scheme is improved upon in the next section, when $n \geq 6$. We begin by stating a useful preliminary result of Radhakrishnan, Raman, and Rao:

Lemma 4.4.2 (Theorem 1 from [133]). *For* $n \geq 2$*, there is a fully explicit adaptive*

$$(n, u, 2n\sqrt{u}, \lceil \lg(n+1) \rceil + 1)\text{-}\mathit{scheme}$$

for the membership problem.



a	0	0	0	0	0	0	1	1	1	1
b	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	X	1	1	1
c	0	0	0	0	0	0	1	0	0	0
d	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	1	1	X	1	1	1

Figure 4.4: Illustration of the various decomposition strategies on the universe represented by the horizontal line, containing elements marked with red dots. For each decomposition, each of the bits are drawn below the bucket with which it is associated. Bits labelled 'X' can either be set to 0 or 1. (a) is a $(1,k)$-pivot decomposition, (b) is a $(2,k)$-pivot decomposition, (c) is a $(1,k)$-balanced decomposition, and (d) is a $(2,k)$-balanced decomposition.

We next describe a scheme that allows us to extend the result of Theorem 4.4.2 to obtain better space bounds for the case when $n \geq 3$ and the number of probes allowed is $n+1$.

Theorem 4.4.5. *Suppose that for $n_0 \geq 2$, $f > 0$, and $c \in [1/3, 1]$, there is a fully explicit adaptive (n_0, u, fu^c, n_0+1)-scheme for the membership problem. Then, for any $n > n_0 \geq 2$, there is a fully explicit adaptive $(n, u, (f + \sum_{j=n_0+1}^{n}(2j+1))u^c, n+1)$-scheme for the membership problem.*

Proof. Proof by induction on n. The supposition in the statement of the theorem serves as the base case when $n = n_0$. For the induction hypothesis, we assume there is a $(n-1, u, (f + \sum_{j=n_0+1}^{n-1}(2j+1))u^c, n)$-scheme. We compute and store a $(1,c)$-balanced decomposition. The query proceeds by probing the bucket specified by the first c-th of the bits of the query element q. Since at least one element will be in the heaviest bucket, if the probe returns 0 it is sufficient to defer to an $(n-1, u, (f + \sum_{j=n_0+1}^{n-1}(2j+1))u^c, n)$-scheme, which is guaranteed to exist by the induction hypothesis. In this case, we have reduced the number of elements by at least one, but have not reduced the size of the universe. In the alternative case, if the probe returns 1, then we know that at most n elements are stored in a smaller universe, of size u^{1-c}. Thus, we can defer this case to an $(n, u^{1-c}, 2nu^{(1-c)/2}, n)$-scheme, which exists by Lemma 4.4.2 (since $n \geq 3$ and $\lceil \lg(n+1) \rceil + 1$ probes are sufficient for this scheme). Overall the total number of bits used by the scheme is:

$$\left(f+1+\sum_{j=n_0+1}^{n-1}(2j+1)\right)u^c+2nu^{(1-c)/2}\leq\left(f+1+\sum_{j=n_0+1}^{n}(2j+1)\right)u^c, \tag{4.4.7}$$

since $c \geq 1/3$. □

Combining Theorems 4.4.2 and 4.4.5 we get the following corollary, setting $n_0 = 2$ in Theorem 4.4.5, and by applying the scheme of Theorem 4.4.2 with $t = 3$.

Corollary 4.4.1. *For $n \geq 3$, there is a fully explicit adaptive $(n, u, (n^2+2n+6)u^{2/5}, n+1)$-scheme for the membership problem.*

Next, using roughly the same decomposition technique as Theorem 4.4.5, we extend the result to the more general case when $t > n \geq 2$.

Theorem 4.4.6. *Suppose for the membership problem there is a fully explicit adaptive*

$$(n_0, u, (2^{n_0+1}-1)u^c, n_0+1)\text{-}\textit{scheme}$$

for some $c \in (0, 1]$, and a fully explicit adaptive

$$(n_0-1, u, (2^{t_1}-1)u^{1/(t_1-n_0+1/c-1)}, t_1)\text{-}\textit{scheme}$$

for $n_0 \geq 3$, and $t_1 > n_0$. For any $n \geq n_0$, $t > n$, there is a fully explicit adaptive

$$(n, u, (2^t-1)u^{1/(t-n+1/c-1)}, t)\text{-}\textit{scheme}$$

for the membership problem.

Proof. Proof by induction, on n and t. The suppositions in the statement of the theorem serve as our base cases when $n = n_0$ and when $t - n = 1$. Thus, the induction hypothesis is that we have both an $(n-1, u, (2^{t-1}-1)u^{1/(t-n+1/c-1)}, t-1)$ scheme and an $(n, u, (2^{t-1}-1)u^{1/(t-n+1/c-2)}, t-1)$-scheme. The strategy is roughly the same as in Theorem 4.4.5: we compute a $(1, 1/(t-n+1/c-1))$-balanced decomposition. The query proceeds by probing the bucket specified by the first $(1/(t-n+1/c-1))$-th fraction of the bits of the query element q. Since at least one element will be in the heaviest bucket, if the probe returns 0 it is sufficient to defer to an $(n-1, u, (2^{t-1} - 1)u^{1/(t-n+1/c-1)}, t-1)$-scheme, which is guaranteed to exist by the induction hypothesis. On the other hand, if the probe returns 1, then we know that at most n elements are stored in a universe of size $u^{(t-n+1/c-2)/(t-n+1/c-1)}$. Thus we can defer to the $(n, u^{(t-n+1/c-2)/(t-n+1/c-1)}, (2^{t-1} - 1)u^{1/(t-n+1/c-1)}, t-1)$-scheme, which is also guaranteed to exist by induction. Overall, the total number of bits required is $(2(2^{t-1}-1)+1)u^{1/(t-n+1/c-1)} = (2^t - 1)u^{1/(t-n+1/c-1)}$. □

We can combine Theorem 4.4.6 (with $c = 2/5$) with Corollary 4.4.1, and the observation that

$$n^2 + 2n + 6 < 2^t - 1,$$

for $n \geq 3$ and $t > n$. This simultaneously improves the exponent-of-u in the best previous explicit scheme (Equation 4.2.5), and (arguably) simplifies the construction.

Corollary 4.4.2. *For $n \geq 3$ and $t > n$, there is a fully explicit adaptive $(n, u, (2^t - 1)u^{1/(t-n+3/2)}, t)$-scheme for the membership problem.*

Improved Scheme for $n \geq 3$

In this section we present a better fully explicit adaptive scheme that achieves significantly better bounds than the one from the previous section. We start by stating the bound of the trivial folklore scheme for storing one element:

Lemma 4.4.3. *There is an explicit non-adaptive $(1, u, tu^{1/t}, t)$-scheme for the membership problem.*

Proof. Divide the bits of the sole element, x, to be stored into blocks of size $\lg u/t$. For each block $B_i(x)$, a table stores the characteristic bit string representing $B_i(x)$. Given a query element, q, we probe the t locations $B_1(q), ..., B_t(q)$ and return `YES` iff all these bits are 1. □

Using Lemma 4.4.3 we prove the following:

Theorem 4.4.7. *Let $\mathfrak{R}(n)$ be the recurrence defined by $\mathfrak{R}(0) = \mathfrak{R}(1) = 0$ and $\mathfrak{R}(n) = \mathfrak{R}(\lfloor n/2 \rfloor) + 1$. For $n \geq 2$ and $t \geq 2\mathfrak{R}(n) + 1$, there is an explicit adaptive*

$$(n, u, (2^t - 1)u^{1/(t-2\mathfrak{R}(n))}, t)\text{-scheme},$$

for the membership problem.

Proof. The proof is by strong induction on both t and n. In the base case we have $t = 2\mathfrak{R}(n) + 1$, and we store the trivial $(n, u, u, 1)$-scheme. In the inductive case, we assume $t > 2\mathfrak{R}(n) + 1$. We compute and store a $(2, 1/(t - 2\mathfrak{R}(n)))$-balanced decomposition. The search algorithm proceeds as follows: if we probe the bucket associated with the query element and read a 1 bit, we immediately recurse to the

$$(n, u, (2^{t-1} - 1)u^{1/(t-2\mathfrak{R}(n)-1)}, t - 1)\text{-scheme},$$

that is guaranteed to exist by the induction hypothesis. In this case we have reduced the size of the universe, but not the number of elements. Otherwise, we read both bits associated with the bucket. After reading either 00 or 01, it is sufficient to recurse to a scheme that represents a set of $\lfloor n/2 \rfloor$ elements. Thus, in this case we have reduced the number of elements, but not the size of the universe.

If $\lfloor n/2 \rfloor = 1$, then we can recurse to two copies of the trivial $(1, u, (t-2)u^{1/(t-2)}, t-2)$-scheme of Lemma 4.4.3. Otherwise, we assume the existence of a $(n, u, (2^{t-2} - 1)u^{1/(t-2-2(\mathfrak{R}(n)-1))}, t-2)$-scheme, and recurse to *two* separate copies of this scheme (one for 00 and one for 01).

We now analyze the space bound. If $n \in \{2, 3\}$ then $\lfloor n/2 \rfloor = 1$ and $\mathfrak{R}(n) = 1$. In this case, we have stored no more than: $(2 + 2^{t-1} - 1 + 2(t-2))u^{1/(t-2)}$ bits, which is no more than the claimed space bound since $t > 3$. If $n > 3$ then overall space is no more than:

$$(2 + 2^{t-1} - 1 + 2(2^{t-2} - 1))u^{1/(t-2\mathfrak{R}(n))} \text{ bits}, \tag{4.4.8}$$

which is exactly $(2^t - 1)u^{1/(t-2\mathfrak{R}(n))}$ bits, completing the proof. □

Since $\mathfrak{R}(n) = \lfloor \lg n \rfloor$, we get the following corollary by combining Corollary 4.4.2 and Theorem 4.4.7:

Corollary 4.4.3. *For $n \geq 2$ and $t \geq 2\lfloor \lg n \rfloor + 1$, there is a fully explicit adaptive*

$$(n, u, (2^t - 1)u^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})}, t)\text{-}scheme,$$

for the membership problem.

Remark 4.4.3. *Note that the case where $n = t = 5$ is not covered by the combination of Corollary 4.4.2 and Theorem 4.4.7, but we can achieve the bound claimed in Corollary 4.4.3 for this case rather trivially by combining a $(2, 2/5)$-balanced decomposition, Theorem 4.4.2, and Lemma 4.4.2.*

4.5 Rank Queries

In this section we show how to adapt our two bit recursive scheme for membership to the problem of *rank*: i.e., return the cardinality of $[1, x] \cap \mathcal{E}$ for any $x \in [1, u]$. We begin by stating a lemma that describes a scheme that uses an optimal number of probes, by explicitly storing the rank of each element in the universe.

Lemma 4.5.1. *For $n \geq 1$, there is a fully explicit adaptive $(n, u, \lceil \lg(n+1) \rceil u, \lceil \lg(n+1) \rceil)$-scheme for the rank problem.*

Next, we use the two bit decomposition technique to do significantly better for the one element case.

Lemma 4.5.2. *For $t \geq 1$, there is a fully explicit adaptive $(1, u, (2t-1)u^{1/t}, t)$-scheme for the rank problem.*

Proof. The proof is by induction on t. In the base case, $t = 1$, we can solve the problem trivially by storing a $(1, 1)$-pivot decomposition, and probing and returning the bit stored by the bucket associated with the query element. (Note that the definition of a pivot decomposition explains how the bits are set in the storage scheme.) In the inductive case, assume that we have a solution matching the bound for $t-1$ probes: i.e., $(1, u, (2t-3)u^{1/(t-1)}, t-1)$-scheme. We store a $(2, 1/t)$-pivot decomposition. The query proceeds by examining the first $1/t$ fraction of the bits of q, then probing the first bit associated with the appropriate bucket. If we read a 1, we immediately defer to the scheme guaranteed to exist by induction, reducing the universe to the size of the bucket: $u^{(t-1)/t}$. Otherwise, we probe and return the second bit. □

Before proving our main theorem for the rank problem, we digress to note an important property of the previous lemma. In particular, we are free to associate two pieces of satellite data, e.g., values $k_1, k_2 \in [0, n']$, with the two possible outcomes of a query to the data structure of Lemma 4.5.2. Returning this satellite data instead of the 0-1 rank value can be done by storing an additional $2\lceil \lg(n'+1)\rceil$ bits, and making an additional $\lceil \lg(n'+1)\rceil$ probes, *provided* we know n' in advance.

We make use of the following lemma in our general scheme for rank, when $n \geq 2$:

Lemma 4.5.3. *Suppose we are given a query element q, and a $(2,k)$-balanced decomposition $\mathcal{D}$ on a universe $[1,u]$ that contains n elements. Also suppose that $\mathcal{D}$ is computed using the algorithm described in the proof of Lemma 4.4.1. There are $n+1$ possible outcomes for a rank query on q a priori, but this is reduced to at most $\lfloor n/2\rfloor + 2$ if we probe the two bits associated with q in $\mathcal{D}$, and they are* 00 *or* 01.

Proof. Recall that, by construction, the buckets assigned 00 and 01 span contiguous subranges of $[1,u]$, with the exception of at most one hole (or gap), i.e., the subrange spanned by the heaviest bucket. Without loss of generality, consider the subrange spanned by the union of buckets assigned bits 00. Since there are at most $\lfloor n/2\rfloor$ elements contained in the subrange, the answer to a rank query can take on at most $\lfloor n/2\rfloor + 1$ possible values. If the subrange contains the hole, then this is increased to at most $\lfloor n/2\rfloor + 2$; regardless of how many elements are in the heaviest bucket. □

This observation is crucial for proving the following theorem:

Theorem 4.5.1. *Let $\mathfrak{R}(n)$ be the recurrence defined by $\mathfrak{R}(2) = 1$, $\mathfrak{R}(3) = 2$, and $\mathfrak{R}(n) = \mathfrak{R}(\lfloor n/2\rfloor + 1) + 1$. For $n \geq 2$ and $t \geq \lceil \lg(n+1)\rceil + 2\mathfrak{R}(n)$, there is an explicit adaptive*

$$(n, u, (2^t - 1)\lceil \lg(n+1)\rceil u^{1/(t-\lceil \lg(n+1)\rceil - 2\mathfrak{R}(n)+1)}, t)\text{-}\textit{scheme}, \tag{4.5.1}$$

for the rank problem.

Proof. The idea is similar to the proof of Theorem 4.4.7, with some small modifications. The proof is by strong induction on t and n. In the base case we have $t = \lceil \lg(n+1)\rceil + 2\mathfrak{R}(n)$, and store the data structure of Lemma 4.5.1. In the inductive case, we assume $t > \lceil \lg(n+1)\rceil + 2\mathfrak{R}(n)$, and that we have the scheme guaranteed to exist by the induction hypothesis: a

$$(n, u, (2^{t-1} - 1)\lceil \lg(n+1)\rceil u^{1/(t-\lceil \lg(n+1)\rceil - 2\mathfrak{R}(n))}, t-1)\text{-scheme}. \tag{4.5.2}$$

To prove the induction hypothesis we use a data structure called a $(n, u, [0, n], t)$-node: the tuple denotes the number of elements, the size of the universe, and the range from which the answer comes, and the number of remaining probes, respectively. In the node, we store a $(2, 1/(t - \lceil \lg(n+1) \rceil - 2\mathfrak{R}(n) + 1))$-balanced decomposition. The search algorithm proceeds in the usual way: if we probe the bucket associated with the query element and read a 1 bit, we immediately recurse to the scheme guaranteed to exist by the induction hypothesis. Otherwise, we read both bits.

After reading either 00 or 01, it is sufficient to recurse to a scheme that represents a set of $\lfloor n/2 \rfloor + 1$ elements, each storing satellite data from the range $[0, n]$. This follows from Lemma 4.5.3, since we can simulate the $\lfloor n/2 \rfloor + 2$ possible answers to a rank query on the subrange spanned by either 00 or 01 using $\lfloor n/2 \rfloor + 1$ elements. If $n > 2$, we recurse to two $(\lfloor n/2 \rfloor + 1, u, [0, n], t - 2)$-nodes. Otherwise, we need to represent a set of two elements, and we handle this with a slight modification to our decomposition scheme: when $n = 2$, we represent the heaviest bucket with a 1 bit *only if* it contains both elements. If the heaviest bucket does not contain both elements, we instead decompose the universe into two contiguous ranges (that have buckets marked with either 00 or 01), such that each range need only represent one element. At this point we can recurse to two copies of the data structure from Lemma 4.5.2.

We now analyze the space bound by induction on n. If $n = 2$, then we are in the base case. Overall, we have stored no more than:

$$(2 + 2^{t-1} - 1 + 2(2(t-2) - 1)) \lceil \lg(n+1) \rceil u^{1/(t - \lceil \lg(n+1) \rceil - 2\mathfrak{R}(n) + 1)} \text{bits},$$

which is less than the claimed space bound. In the inductive case, we assume we have a $(n, u, (2^{t-2} - 1) \lceil \lg(n+1) \rceil u^{1/(t - 2\lceil \lg(n+1) \rceil - 2(\mathfrak{R}(n) - 1) + 1}, t - 2)$-scheme. This scheme is used to represent the $(\lfloor n/2 \rfloor + 1, u, [0, n], t - 2)$-nodes, and thus the overall space is no more than: $(2 + 2^{t-1} - 1 + 2(2^{t-2} - 1)) \lceil \lg(n+1) \rceil u^{1/(t - \lceil \lg(n+1) \rceil - 2\mathfrak{R}(n) + 1)}$ bits, completing the proof. □

4.6 Range Counting and Emptiness Queries

Recall that *range counting queries* ask us to return the cardinality of $[x_1, x_2] \cap \mathcal{E}$ for $1 \leq x_1 \leq x_2 \leq u$. Range emptiness queries ask whether the range $[x_1, x_2]$ is empty. Range emptiness and counting queries are clearly not much harder than rank, since we can answer them using two rank queries. We formalize this in the following theorem.

Theorem 4.6.1. *Suppose there is a (n, u, s, t)-scheme for the rank problem. Then, there is a $(n, u, s, 2t)$-scheme for the range counting problem.*

Thus we get the following corollary:

Corollary 4.6.1. *Let $\mathfrak{R}(n)$ be the recurrence defined by $\mathfrak{R}(2) = 1$, and $\mathfrak{R}(n) = \mathfrak{R}(\lfloor n/2 \rfloor + 1) + 1$. For $n \geq 2$ and even $t \geq 2\lceil \lg(n + 1)\rceil + 4\mathfrak{R}(n)$, there is a fully explicit adaptive*

$$(n, u, (2^t - 1)\lceil \lg(n + 1)\rceil u^{1/(t/2 - \lceil \lg(n+1)\rceil - 2\mathfrak{R}(n) + 1)}, t)\text{-scheme},$$

for the range counting problem.

For the problem of emptiness, we can explicitly record one bit representing the yes or no answer to each of the $\binom{u}{2} + u$ possible query ranges. It is not difficult to see, by an adversarial argument, that this is the best possible space bound for 1 probe:

Theorem 4.6.2. *There is a fully explicit $(n, u, \binom{u}{2} + u, 1)$-scheme for the range emptiness problem. This space bound is the best possible for 1 probe.*

Proof. Suppose there is a $(n, u, s, 1)$-scheme for the emptiness problem that uses less than $\binom{u}{2} + u$ bits. Thus, there are two query ranges that must read the same bit. Since it is always possible to force the query result to be different for these ranges, we have a contradiction. $\square$

These previous theorems raise the question of what trade-offs exist for range emptiness. We have the following result:

Theorem 4.6.3. *For any constant $t \geq 1$, there is a fully explicit adaptive*

$$(n, u, \Theta\left(\sum_{i=0}^{\lg u} 2^i (u/2^i)^{1/t}\right), 2t)\text{-scheme}$$

for the range emptiness problem.

Proof. We conceptually build a canonical balanced binary tree over the universe $[1, u]$ such that each element appears in $\lg(u) + 1$ nodes: each node is in a different *level* in the tree, numbered

$0, ..., \lg(u)$ from the root to leaf. Each node u represents the range $[x_1(u), x_2(u)]$, where $x_1(u)$ and $x_2(u)$ are the minimum and maximum elements represented by leaves in the subtree rooted at u. Each node stores two one-sided range emptiness structure (Lemma 4.5.2). If u is a node representing the range $[x_1(u), x_2(u)]$, then these structure can answer emptiness queries on the ranges $[x_1(u), x']$ and $[x', x_2(u)]$ for $x \in [x_1(u), x_2(u)]$. Overall, the one-sided emptiness structures require space:

$$\Theta\left(\sum_{i=0}^{\lg u} 2^i (u/2^i)^{1/t}\right) \text{ bits,} \tag{4.6.1}$$

where t is the number of probes made to the one-sided structures.

To answer a query we find the level ℓ in this binary tree that exists by Observation 3.5.1. Each query can be decomposed into two one-sided emptiness queries to structures at this level. Thus, we have shown that the total number of probes required to answer a query is $2t$. $\square$

By plugging $t = 1, 2$ into the previous theorem we get:

Corollary 4.6.2. *There is a fully explicit adaptive* $(n, u, \Theta(u \lg u), 2)$*-scheme and a fully explicit adaptive* $(n, u, \Theta(u), 4)$*-scheme for the range emptiness problem.*

4.7 Summary and Concluding Remarks

We have presented several new fully explicit data structures for the membership problem in the bitprobe model. In particular, we have presented new fully explicit schemes for the case when $n = 2$ that match (or improve upon) the performance of previous non-explicit schemes. For the case where $n \geq 3$, we have described a fully explicit scheme for the membership problem that can be used to solve the related problems of rank and range counting, initiating the study of these problems in the bitprobe model.

Chapter 5

Conclusions and Open Problems

In this thesis we have examined three data structure problems, each involving some aspect of space efficiency. In particular, we have focused on designing succinct data structures (i.e., data structures that occupy space matching the information theory lower bounds), and also using succinct data structures to improve the efficiency of certain query operations. We conclude with a list of open problems from each chapter.

5.1 Succinct Posets

In Chapter 2 we presented a succinct data structure for representing arbitrary partial orders (posets) that supports efficient query operations. The main technique we presented was the application of a theorem from the area of extremal graph theory to compressing posets. Unlike previous enumeration proofs (which yield succinct representations of posets), our technique has the additional feature that queries about the structure of the poset can be answered efficiently. Our proposal is the first to compress an arbitrary poset optimally while still supporting queries.

1. Is there a way to lower the lower order term in our representation of posets? The enumeration argument of Kleitman and Rothschild [105] has a linear lower order term, whereas ours is $O(n^2 \lg\lg n / \lg n)$. Is it possible to achieve succinctness by using higher order empirical entropy compression on the adjacency matrix of a transitive closure graph? Achieving the

zeroth order empirical entropy does not suffice to make the representation succinct, but we have said nothing about higher order compression.

2. The trivial adjacency bit matrix representation of a poset occupies $\binom{n}{2}$ bits, and can be used to answer a precedence query in 1 bit probe. The succinct representation we have described occupies $n^2/4 + o(n^2)$ bits, but requires $\Theta(\lg n)$ bit probes. Is there a representation that is succinct and uses $o(\lg n)$ bit probes? Is there a trade-off between succinctness and the number of probes required for this problem?

3. We have introduced the problem of supporting extended precedence queries, in which we would like to determine for two vertices whether they are incomparable, have an edge between them in the transitive reduction graph, or have an edge between them in the transitive closure graph. Is is possible to support such queries using $n^2/4 + o(n^2)$ bits? Our best solution requires about $0.39624n^2 + o(n^2)$ bits. Is it possible to generalize this to answer queries—on an arbitrary DAG—of the form, "Is there a shortest path of length no more than k between vertices s_1 and s_2?"

4. We have discussed evidence showing that meet and join operations must be costly in our data structure. However, we have not given any upper bounds for these queries. Is it possible to improve upon the trivial $O(n^2)$ bound to compute the meet and join of two vertices in our data structure? The chain based data structure of Farzan and Fischer [55] is currently much better suited to these kinds of queries.

5.2 Range Majority Queries

In Chapter 3 we described techniques that show how succinct data structures could be used to speed up range searching queries for certain frequency-based queries on strings (or arrays). Though the proposed data structures were not succinct, they were more space efficient than previous proposals. We also discussed numerous trade-offs, including compression, dynamization, as well as geometric problems.

1. Many of the open problems for the static case have been answered in the affirmative by Belazzougui, Gagie, and Navarro [14]. However, it is still not known whether there is a

linear space data structure that can answer (parameterized) β-majority queries in $O(1/\beta)$ time. There are also still issues open relating to succinctness of these data structures.

2. Is it possible to answer queries faster if we only desire an arbitrary α-majority character instead of *all* of them? What about output-sensitive results?

3. In the dynamic case is it possible to improve the update time of our data structure, if we assume the colours of the points are drawn from a bounded universe $[1, \sigma]$? It seems difficult to increase the fan-out of our weight-balanced B-tree solution while maintaining the linear space bound and optimal query time.

4. Recent results for approximate range counting can be used to improve both the static and dynamic results for higher dimensional geometric versions of the α-majority query. What is the optimal query time for two-dimensional range α-majority queries using linear space?

5.3 Explicit Bitprobe Data Structures

Finally, in Chapter 4 we designed data structures in the bitprobe model for the membership problem, as well as other similar problems. For the problems we studied, where the number of elements in the set and number of probes used is small, there is a distinct lack of meaningful lower bounds. We presented several new upper bounds for these cases, though many open problems remain.

1. Is there a fully explicit $(2, u, O(u^{2/5}), 3)$-scheme for membership that is 1-adaptive? Our explicit scheme is adaptive on the final two probes (2-adaptive), whereas the non-explicit scheme of Radhakrishnan, Shah, and Shannigrahi [134] is 1-adaptive. Is it possible to close this gap?

2. Is there a fully explicit scheme that matches the non-explicit membership scheme of Equation 4.2.6 for $n \geq 3$? We have shown that it is unlikely to be a blocking scheme with the split property, but perhaps another approach could work.

3. Even for the $n = 2$ and $t = 2$ case for the membership problem there is no tight lower bound. Either improve the lower bound of $\Omega(u^{4/7})$ or provide a new scheme beating $O(u^{2/3})$. For larger values of t, is the scheme of Theorem 4.4.2 the best possible?

4. We initiated the study of the rank and range counting problems in the bitprobe model, and there are lots of open problems. For example, for the rank problem on sets of 2 elements, what can be said for lower and upper bounds? Is it possible to achieve $o(u)$ bits with 2 adaptive probes?

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 253–262. ACM Press, 1989.

[2] M. Aigner. Variants of the majority problem. *Discrete Applied Mathematics*, 137:3–25, 2004.

[3] N. Alon and U. Feige. On the power of two, three and four probes. In *Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 346–354. SIAM, 2009.

[4] L. Alonso and E. M. Reingold. Determining plurality. *ACM Transactions on Algorithms*, 4(3):26:1–26:19, 2008.

[5] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 198–207. IEEE, 2000.

[6] S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. of the 33rd annual ACM Symposium on Theory of Computing (STOC)*, pages 476–482. ACM, 2001.

[7] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.

[8] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007.

[9] L. Arge and J. S. Vitter. Optimal External Memory Interval Management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[10] J. Barbay, L. Castelli Aleardi, M. He, and J. I. Munro. Succinct Representation of Labeled Graphs. *Algorithmica*, 62(1-2):224–257, 2012.

[11] J. Barbay, F. Claude, and G. Navarro. Compact Binary Relation Representations with Rich Functionality. *CoRR*, abs/1201.3602, 2012.

[12] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet Partitioning for Compressed Rank/Select and Applications. In *Proc. of the 21st International Symposium on Algorithms and Computation (ISAAC) (2)*, volume 6507 of *LNCS*, pages 315–326. Springer, 2010.

[13] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms (TALG)*, 7(4):52, 2011.

[14] D. Belazzougui, T. Gagie, and G. Navarro. Better space bounds for parameterized range majority and minority. *CoRR*, abs/1210.1765, 2012.

[15] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. *Proc. of the 20th European Symposium on Algorithms (ESA)*, pages 181–192, 2012.

[16] A. M. Ben-Amram. What is a “pointer machine”? *SIGACT News*, 26(2):88–95, June 1995.

[17] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. of the 15th ACM Symposium on Theory of Computing (STOC)*, STOC ’83, pages 80–86, New York, NY, USA, 1983. ACM.

[18] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[19] J.L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[20] R. Blue. The Bit Probe Model for Membership Queries: Non-Adaptive Bit Queries. Master’s thesis, University of Maryland, 2009.

[21] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[22] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *11th International Symposium on Algorithms and Data Structures (WADS)*, volume 5664 of *LNCS*, pages 98–109. Springer, 2009.

[23] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate Range Mode and Range Median Queries. In *Proc. of the 22nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3404 of *LNCS*, pages 377–388. Springer, 2005.

[24] R. S. Boyer and J. S. Moore. MJRTY - A fast majority vote algorithm. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–117. Kluwer, Dordrecht, The Netherlands, 1991.

[25] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proc. of the 22nd International Colloquium Automata, Languages and Programming (ICALP)*, volume 944 of *LNCS*, pages 464–474. Springer, 1995.

[26] G. Brightwell, H. Jurgen Promel, and A. Steger. The average number of linear extensions of a partial order. *Journal of Combinatorial Theory, Series A*, 73(2):193–206, 1996.

[27] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011.

[28] G. S. Brodal and A. Jørgensen. Data Structures for Range Median Queries. In *Proc. of the 20th International Symposium on Algorithms and Computation (ISAAC)*, volume 5878 of *LNCS*, pages 822–831. Springer, 2009.

[29] A. Brodnik and J. Munro. Membership in Constant Time and Almost-Minimum Space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[30] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723–1744, 2002.

[31] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. Linear-space data structures for range mode query in arrays. In *Proc. of the 29th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 14, pages 290–301, 2012.

[32] T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson. Linear-space data structures for range minority query in arrays. *Proc. of the 13th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 295–306, 2012.

[33] T. M. Chan and B. T. Wilkinson. Adaptive and Approximate Orthogonal Range Counting. In *Proc. of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 241–251. SIAM, 2013.

[34] B. Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[35] Y. Choi and W. Szpankowski. Compression of Graphical Structures: Fundamental Limits, Algorithms, and Experiments. *IEEE Transactions on Information Theory*, 58(2):620–638, 2012.

[36] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391. Society for Industrial and Applied Mathematics, 1996.

[37] F. Claude. *Space Efficient Indexes for Information Retrieval.* PhD thesis, University of Waterloo, 2013.

[38] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

[40] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[41] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and Selection in Posets. *SIAM Journal on Computing*, 40(3):597–622, 2011.

[42] M. de Berg and H. Haverkort. Significant-presence range queries in categorical data. In *8th International Workshop on Algorithms and Data Structures (WADS)*, volume 2748 of *LNCS*, pages 462–473. Springer, 2003.

[43] K. De Loof, H. De Meyer, and B. De Baets. Exploiting the Lattice of Ideals Representation of a Poset. *Fundam. Inf.*, 71(2,3):309–321, February 2006.

[44] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 348–360. Springer, 2002.

[45] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12(3):255–263, 1980.

[46] Y. Dodis, M. Pătraşcu, and M. Thorup. Changing base without losing space. In *Proc. of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 593–602. ACM, 2010.

[47] S. Durocher. A simple linear-space data structure for constant-time range minimum query. In *Space Efficient Data Structures, Streams and Algorithms*, volume 8066 of *LNCS*, pages 48–60. Springer, 2013.

[48] S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range Majority in Constant Time and Linear Space. In *Proc. of the 38th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 6755 of *LNCS*, pages 244–255. Springer, 2011.

[49] S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. *Information and Computation*, 222(0):169–179, 2013.

[50] S. Durocher and J. Morrison. Linear-space data structures for range mode query in arrays. arXiv:1101.4068v1, 2011.

[51] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. on Inf. Theory*, 21(2):194–203, 1975.

[52] A. Elmasry, M. He, J. I. Munro, and P. K. Nicholson. Dynamic Range Majority Data Structures. In *Proc. of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *LNCS*, pages 150–159. Springer Berlin / Heidelberg, 2011. Yokohama, Japan, December 5-8, 2011.

[53] A. Elmasry, M. He, J. I. Munro, and P. K. Nicholson. Dynamic Range Majority Data Structures. *CoRR*, abs/1104.5517, 2013.

[54] A. Farzan. *Succinct representation of trees and graphs.* PhD thesis, University of Waterloo, 2009.

[55] A. Farzan and J. Fischer. Compact Representation of Posets. In *Proc. of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *LNCS*, pages 302–311. Springer, 2011.

[56] A. Farzan and J. I. Munro. Succinct representations of arbitrary graphs. In *Proc. of the 16th Annual European Symposium on Algorithms (ESA)*, LNCS, pages 393–404. Springer, 2008.

[57] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[58] S. Felsner, V. Raghavan, and J. Spinrad. Recognition algorithms for orders of small width and graphs of small Dilworth number. *Order*, 20(4):351–364, 2003.

[59] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13:12, 2009.

[60] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[61] P. Ferragina, I. Nitto, and R. Venturini. Succinct Oracles for Exact Distances in Undirected Unweighted Graphs. Technical Report TR-07-11, Università di Pisa, 2007.

[62] J. Fischer. Optimal Succinctness for Range Minimum Queries. In *Proc. of the 9th Latin American Theoretical Informatics Symposium (LATIN)*, volume 6034 of *LNCS*, pages 158–169. Springer, 2010.

[63] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, 2007.

[64] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[65] T. Gagie. *New algorithms and lower bounds for sequential-access data compression.* PhD thesis, Bielefeld University, 2009.

[66] T. Gagie, M. He, J. I. Munro, and P. K. Nicholson. Finding Frequent Elements in Compressed 2D Arrays and Strings. In *18th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7024 of *LNCS*, pages 295–300. Springer, 2011.

[67] T. Gagie and J. Kärkkäinen. Counting Colours in Compressed Strings. In *Proc. of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 6661 of *LNCS*, pages 197–207. Springer, 2011.

[68] T. Gagie, S. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *String Processing and Information Retrieval*, pages 1–6. Springer, 2009.

[69] A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theoretical computer science*, 379(3):405–417, 2007.

[70] G. Gambosi, J. Nešetřil, and M. Talamo. Efficient representation of taxonomies. In *TAPSOFT'87: Proc. of the International Joint Conference on Theory and Practice of Software Development*, volume 249 of *LNCS*, pages 232–240. Springer, 1987.

[71] G. Gambosi, J. Nešetřil, and M. Talamo. On locally presented posets. *Theoretical computer science*, 70(2):251–260, 1990.

[72] G. Gambosi, J. Nešetřil, and M. Talamo. Posets, boolean representations and quick path searching. In Thomas Ottmann, editor, *Proc. of the 14th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 267 of *LNCS*, pages 404–424. Springer Berlin Heidelberg, 1987.

[73] G. Gambosi, M. Protasi, and M. Talamo. An efficient implicit data structure for relation testing and searching in partially ordered sets. *BIT Numerical Mathematics*, 33(1):29–45, 1993.

[74] V. K. Garg and C. Skawratananond. String realizers of posets with applications to distributed computing. In *Proc. of the 20th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 72–80. ACM, 2001.

[75] B. Gfeller and P. Sanders. Towards Optimal Range Medians. In *Proc. of the 36th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5555 of *LNCS*, pages 475–486. Springer, 2009.

[76] O. Giménez and M. Noy. Asymptotic enumeration and limit laws of planar graphs. *J. Amer. Math. Soc*, 22(2):309–329, 2009.

[77] A. Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[78] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. *Algorithms–ESA 2007*, pages 371–382, 2007.

[79] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373. ACM, 2006.

[80] A. Golynski, R. Raman, and S. S. Rao. On the redundancy of succinct data structures. *Algorithm Theory–SWAT 2008*, pages 148–159, 2008.

[81] M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen. Cell Probe Lower Bounds and Approximations for Range Mode. In *Proc. of the 37th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6198 of *LNCS*, pages 605–616. Springer, 2010.

[82] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[83] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More Haste, Less Waste: Lowering the Redundancy in Fully Indexable Dictionaries. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 517–528, 2009.

[84] P. Gupta, R. Janardan, and M. Smid. Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *Journal of Algorithms*, 19(2):282–317, 1995.

[85] M. Habib, M. Huchard, , and L. Nourine. Embedding partially ordered sets into chain-products. In *Proc. of KRUSE'95*, pages 147–161, 1995.

[86] M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In *International Workshop on Orders, Algorithms, and Applications (ORDAL)*, volume 831 of *LNCS*, pages 1–12. Springer, 1994.

[87] M. Habib and L. Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165(2):391 – 405, 1996.

[88] T. Hagerup. Sorting and searching on the word RAM. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *Proc. of the 15th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1373 of *LNCS*, pages 366–398. Springer Berlin Heidelberg, 1998.

[89] S. Har-Peled and S. Muthukrishnan. Range Medians. In *Proc. of the 16th European Symposium on Algorithms (ESA)*, volume 5193 of *LNCS*, pages 503–514. Springer, 2008.

[90] M. He. *Succinct Indexes.* PhD thesis, University of Waterloo, 2007.

[91] M. He, J. I. Munro, and P. K. Nicholson. Dynamic Range Selection in Linear Space. In *22nd International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *LNCS*, pages 160–169. Springer, 2011.

[92] M. He, J. I. Munro, and P. K. Nicholson. Dynamic Range Selection in Linear Space. *CoRR*, abs/1106.5076, 2011.

[93] M. He, J. I. Munro, and G. Zhou. A Framework for Succinct Labeled Ordinal Trees over Large Alphabets. In *Proc. of the 23rd International Symposium on Algorithms and Computation (ISAAC)*, pages 537–547. Springer, 2012.

[94] R. Hegde and K. Jain. The hardness of approximating poset dimension. *Electronic Notes in Discrete Mathematics*, 29:435–443, 2007.

[95] C. Hernández and G. Navarro. Compressed representation of web and social networks via dense subgraphs. In *19th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *LNCS*, pages 264–276. Springer, 2012.

[96] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101, 1952.

[97] T. Husfeldt and T. Rauhe. New lower bound techniques for dynamic partial sums and related problems. *SIAM Journal on Computing*, 32(3):736–753, 2003.

[98] G. Jacobson. Space-efficient static trees and graphs. *30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[99] J. JaJa, C. Mortensen, and Q. Shi. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In *Proc. of the 16th International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *LNCS*, pages 1755–1756, 2005.

[100] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: a high-compression indexing scheme for reachability query. In *Proc. of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 813–826. ACM, 2009.

[101] A.G. Jørgensen and K.G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. of the 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011.

[102] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28:51–55, 2003.

[103] M. Karpinski and Y. Nekrich. Searching for Frequent Colors in Rectangles. In *Proc. of the 20th Canadian Conference on Computational Geometry (CCCG)*, pages 11–14, 2008.

[104] D. J. Kleitman and B. L. Rothschild. The Number of Finite Topologies. *Proc. of the American Mathematical Society*, 25:276, 1970.

[105] D. J. Kleitman and B. L. Rothschild. Asymptotic enumeration of partial orders on a finite set. *Transactions of the American Mathematical Society*, 205:205–220, 1975.

[106] D. E. Knuth. *Sorting and Searching: The Art of Computer Programming III.* Addison-Wesley, Reading, Mass., 1973.

[107] T. Kővári, V. T. Sós, and P. Turán. On a problem of Zarankiewicz. *Coll. Math*, 3(1954):50–57, 1954.

[108] D. Krizanc, P. Morin, and M. Smid. Range Mode and Range Median Queries on Lists and Trees. *Nordic Journal of Computing*, 12:1–17, 2005.

[109] Y. K. Lai, C. K. Poon, and B. Shi. Approximate colored range and point enclosure queries. *Journal of Discrete Algorithms*, 6(3):420–432, 2008.

[110] M. L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, 1969.

[111] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[112] N. Mittal and V. K. Garg. Rectangles are Better than Chains for Encoding Partially Ordered Sets, 2005.

[113] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[114] D. Mubayi and G. Turán. Finding bipartite subgraphs efficiently. *Information Processing Letters*, 110(5):174–177, 2010.

[115] J. I. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42. Springer, 1996.

[116] J. I. Munro and P. K. Nicholson. Succinct Posets. In *Proc. of the 20th Annual European Symposium on Algorithms (ESA)*, volume 7501 of *LNCS*, pages 743–754. Springer, 2012.

[117] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 2012.

[118] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[119] J. I. Munro and M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, 1976.

[120] M. Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(3):303–307, 1990.

[121] G. Navarro. Wavelet trees for all. In *Proc. of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 7354 of *LNCS*, pages 2–26. Springer, 2012.

[122] G. Navarro, Y. Nekrich, and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[123] G. Navarro and L. M. S. Russo. Space-Efficient Data-Analysis Queries on Grids. In *Proc. of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *LNCS*, pages 323–332. Springer, 2011.

[124] Gonzalo Navarro and Yakov Nekrich. Optimal Dynamic Sequence Representations. In *Proc. of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 865–876. SIAM, 2013.

[125] P. K. Nicholson, V. Raman, and S. S. Rao. Data Structures in the Bitprobe Model. In *Space Efficient Data Structures, Streams and Algorithms*, volume 8066 of *LNCS*, pages 303–318. Springer, 2013.

[126] R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[127] R. Pagh. On the cell probe complexity of membership and perfect hashing. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 425–432. ACM, 2001.

[128] M. Pătraşcu. Succincter. In *Proc. of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313. IEEE Computer Society, 2008.

[129] M. Pătraşcu and C. E. Tarniţă. On dynamic bit-probe complexity. *Theoretical Computer Science*, 380(1):127–142, 2007.

[130] H. Petersen. Improved Bounds for Range Mode and Range Median Queries. In *Proc. of the Conference on Current Trends in Theory and Practice of Computer Science*, volume 4910 of *LNCS*, pages 418–423. Springer, 2008.

[131] H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109:225–228, 2009.

[132] M. Pătraşcu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '10, pages 117–122, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[133] J. Radhakrishnan, V. Raman, and S. S. Rao. Explicit deterministic constructions for membership in the bitprobe model. In *Proc. of the 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 290–299. Springer, 2001.

[134] J. Radhakrishnan, S. Shah, and S. Shannigrahi. Data Structures for Storing Small Sets in the Bitprobe Model. In *European Symposium on Algorithms (ESA) (2)*, volume 6347 of *LNCS*, pages 159–170. Springer, 2010.

[135] M. Z. Rahman. Data Structuring Problems in the Bit Probe Model. Master's thesis, University of Waterloo, 2007.

[136] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[137] S. S. Rao. *Succinct Data Structures.* PhD thesis, Institute of Mathematical Sciences, 2001.

[138] D. R. Raymond. *Partial-order databases.* PhD thesis, University of Waterloo, 1996.

[139] O. Raynaud and E. Thierry. The complexity of embedding orders into small products of chains. *Order*, 27(3):365–381, 2010.

[140] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239. ACM, 2006.

[141] M. Skala. Array Range Queries. In *Space Efficient Data Structures, Streams and Algorithms*, volume 8066 of *LNCS*, pages 333–350. Springer, 2013.

[142] Maurizio T. and Paola V. Representing graphs implicitly using almost optimal space. *Discrete Applied Mathematics*, 108(1âĂŞ2):193 – 210, 2001.

[143] M. Talamo and P. Vocca. Fast lattice browsing on sparse representation. In *International Workshop on Orders, Algorithms, and Applications (ORDAL)*, volume 831 of *LNCS*, pages 186–204. Springer, 1994.

[144] M. Talamo and P. Vocca. An Efficient Data Structure for Lattice Operations. *SIAM Journal on Computing*, 28(5):1783–1805, 1999.

[145] T. Tao, E. Croot III, and H. Helfgott. Deterministic methods to find primes. *Mathematics of Computation*, 81(278):1233–1246, 2012.

[146] A. R. Taraz. *Phase transitions in the evolution of partially ordered sets.* PhD thesis, Humboldt-Universität zu Berlin, 1999.

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

[148] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

[149] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[150] W. T. Trotter. *Combinatorics and partially ordered sets: dimension theory.* Johns Hopkins Series in the Mathematical Sciences, 1992.

[151] A. K. Tsakalidis. The nearest common ancestor in a dynamic tree. *Acta Informatica*, 25(1):37–54, 1988.

[152] G. Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[153] E. Viola. Bit-probe lower bounds for succinct data structures. *SIAM Journal on Computing*, 41(6):1593–1604, 2012.

[154] Z. Wei and K. Yi. Beyond simple aggregates: indexing for summary queries. In *Proc. Symposium on the Principles of Database Systems (PODS)*, pages 117–128, 2011.

[155] B. T. Wilkinson. Adaptive range counting and other frequency-based range query problems. Master's thesis, University of Waterloo, 2012.

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

[157] V. V. Williams. Breaking the Coppersmith-Winograd barrier. *Unpublished manuscript*, 2011.

[158] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic Discrete Methods*, 3(3):351–358, 1982.

[159] A. C. C. Yao. Should Tables Be Sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

[160] D. Zeilberger. Enumerative and algebraic combinatorics. *The Princeton companion to mathematics, Princeton University Press, USA*, pages 550–561, 2008.