EXACT AND APPROXIMATE RANGE MODE QUERY DATA STRUCTURES IN PRACTICE

by

Zhen Liu

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
June 2022

Table of Contents

List of Tables

List of Figures

Abstract

We conduct an experimental study on the range mode problem. In the exact version of the problem, we preprocess an array A, such that given a query range $[a, b]$, the most frequent element in $A[a, b]$ can be found efficiently. For this problem, our most important finding is that the strategy of using succinct data structures to encode more precomputed information not only helped Chan et al. (Linear-space data structures for range mode query in arrays, Theory of Computing Systems, 2013) improve previous results in theory but also helps us achieve the best time/space tradeoff in practice; we even go a step further to replace more components in the solution of Chan et al. with succinct data structures. In the approximate version of this problem, a $(1 + \epsilon)$-approximate range mode query looks for an element whose occurrences in $A[a, b]$ is at least $F_{a,b}/(1 + \epsilon)$, where $F_{a,b}$ is the frequency of the mode in $A[a, b]$. We implement all previous solutions to this problems and find that, even when $\epsilon = \frac{1}{2}$, the average approximation ratio of these solutions is close to 1 in practice, while providing much faster query time than the best exact solution. Among these solutions, El-Zein et al. (On Approximate Range Mode and Range Selection, 30th International Symposium on Algorithms and Computation, 2019) provide us with one solution that takes only $35.6\% \sim 93.8\%$ space cost of the input array of 32-bit integers (in most cases, the space cost is closer to the lower end). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space in practice.

Acknowledgements

I would like to thank my supervisor Dr. Meng He. During my master's period, Dr. He provides me with professional advice and guidance, and I am impressed by all of these. Without his help, I would not have been able to complete my master's thesis smoothly. I am grateful to be mentored by such a kind and professional supervisor.

I am also grateful to my committee members, Dr. Chris Whidden and Dr. Nauzer Kalyaniwalla, for participating in my thesis defence and providing professional comments, as well as Dr. Travis Gagie for serving as my thesis defence chair.

I appreciate and acknowledge the support of my family, especially my parents and grandparents, for their continuing support and encouragement during my graduate studies.

Chapter 1

Introduction

The *mode*, or the most frequent element, in a dataset is a widely used descriptive statistic. In the *range mode query problem*, we preprocess an array A of length n, such that, given a query range $[a, b]$, the *mode* in $A[a..b]$ can be computed efficiently. This problem has many applications in data analytics and retrieval. For example, an online shopping platform may be interested in finding out the most popular item purchased by customers over a certain period, which can be modeled as a range mode query over the sales records in its database.

The range mode problem is also connected to the matrix multiplication problem. It has been shown that the problem of multiplying two $\sqrt{n} \times \sqrt{n}$ Boolean matrices can be reduced to the problem of answering n range mode queries in an array of length $\mathcal{O}(n)$ [5]. This reduction provides a conditional lower bound showing that, with current knowledge, the time required to preprocess an array and answer n range mode queries over it must be at least $\mathcal{O}(n^{\omega/2})$, save for polylogarithmic speedups, where $\omega < 2.3727$ is the best exponent in the matrix multiplication [27]. To further speed up queries, researchers further define the $(1+\varepsilon)$-*approximate range mode query problem*, where $\epsilon \in (0, 1)$. Given a query range $[a, b]$, let $F_{a,b}$ denote the frequency of the mode in $A[a, b]$. A $(1 + \varepsilon)$-approximate range mode query then asks for an element whose occurrences in $A[a, b]$ is at least $F_{a,b}/(1 + \epsilon)$.

Due to the importance in both theory and practice, exact and approximate range mode problems have been studied extensively [19, 24, 5, 4, 15, 10, 11, 28, 26, 16], and many efficient solutions have been designed. Despite these efforts, there have not been any published experimental studies on them. Hence, to connect theory to practice, we conduct an empirical study of both exact and approximate range mode structures using large practical datasets.

1.1 Related Work

Krizanc et al. [19] first considered the exact range mode problem and introduced an $\mathcal{O}(n+s^2)$-word solution with $\mathcal{O}((n/s)\lg n)$ query time for any $s \in [1,n]$, and setting $s=\sqrt{n}$ yields a linear space solution with $\mathcal{O}(\sqrt{n}\lg n)$ query time. They also presented another solution with constant query time and $\mathcal{O}(n^2 \lg\lg n/\lg n)$ words of space cost. Later Petersen et al. [24] proposed an $\mathcal{O}(n^2\lg\lg n/\lg^2 n)$-word structure with constant query time. Chan et al. [5] further improved the time-space tradeoff of Krizanc et al. by designing an $O(n+s^2/w)$-word data structure with $O(n/s)$ query time, where w is the number of bits in a word. This result implies a linear space solution in words with $\mathcal{O}(\sqrt{n/w})$ query time. They also proved a conditional lower bound by showing that the multiplication of two $\sqrt{n}\times\sqrt{n}$ Boolean matrices can be performed by answering n range mode queries in an array of length $\mathcal{O}(n)$. Therefore, any range mode structure requires either $\mathcal{O}(n^{\omega/2})$ preprocessing time or $\mathcal{O}(n^{\omega/2-1})$ query time in the worst case, where ω is the matrix multiplication exponent. Since the current best matrix multiplication algorithm requires $O(n^{2.3727})$ time [27], this implies that, with current knowledge, any data structure for range mode uses either $\Omega(n^{1.18635})$ preprocessing time or $\Omega(n^{0.18635})$ query time. Furthermore, since the running time of the best combinatorial algorithm for Boolean matrix multiplication is only a polylogarithmic factor best than cubic [2], we cannot use pure combinatorial approaches to design range mode structures that have preprocessing time lower than $\Omega(n^{3/2})$ and query time less than $\Omega(\sqrt{n})$ simultaneously, save for polylogarithmic speedups.

Regarding the $(1+\varepsilon)$-approximate range mode problem, Bose et al. [4] first used persistent search trees to design a solution with $\mathcal{O}(\lg\lg n+\lg\frac{1}{\varepsilon})$ query time and $O(\frac{n}{\varepsilon})$ words of space. Greve et al. [15] provided another structure with $\mathcal{O}(\lg\frac{1}{\varepsilon})$ query time and $\mathcal{O}(\frac{n}{\varepsilon})$ words of space, and they used succinct data structures in their solution. More recently, El-Zein et al. [10] designed an encoding data structure occupying only $\mathcal{O}(\frac{n}{\varepsilon})$ bits of space, and without requiring access to the original array, it can also report the position of a $(1+\varepsilon)$-approximate range mode in the array in $\mathcal{O}(\lg\frac{1}{\varepsilon})$ time.

Chan et al. [5] also considered the dynamic range mode query problem, in which updates to the arrays are supported. They designed a linear-word structure that answers queries in $\mathcal{O}(n^{3/4}\lg n/\lg\lg n)$ time and supports updates in $\mathcal{O}(n^{3/4}\lg\lg n)$ time. When more space is allowed, they have another tradeoff that has $\mathcal{O}(n^{2/3}\lg n/\lg\lg n)$

query time, $\mathcal{O}(n^{2/3} \lg n / \lg \lg n)$ update time and $\mathcal{O}(n^{4/3})$ words of space cost. Later, El-Zein et al. [11] improved these results by designing an $\mathcal{O}(n)$-word structure with $\mathcal{O}(n^{2/3})$ query and update times. More recently, researchers considered the problem of answering a batch of range mode queries given offline, and used algebraic approaches such as min-plus product of matrices to speed up query processing [28, 16]. With global rebuilding, this idea has been further developed to improve dynamic range mode [26, 16], achieving $O(n^{0.6524})$ query and update times, albeit at a space cost of $O(n^{1.3262})$. There has also been research on the dynamic approximate range mode query problem [10], as well as range mode in higher dimensions [5, 10].

1.2 Our Work

We first study the performance of those exact range mode structures that use linear space; the solutions that are left out use near quadratic space which are often not affordable in practice. Much of our study on exact range mode focuses on the following two data structures of Chan et al. [5]: a simple linear word structure with $\mathcal{O}(\sqrt{n})$ query time, and a linear word structure with $\mathcal{O}(\sqrt{n/w})$ query Time, where w is the number of bits in a word. They both outperform other solutions, and the latter, which is their final structure, essentially combines the former with succinct data structures to encode more procomputed information. However, in practice, constant-time operations over succinct data structures are usually slower than those operations over their non-succinct counterparts, so we compare the performance of different tradeoffs of both structures (they both take parameters to achieve time/space tradeoffs) to see whether the use of succinct data structures improves performance in practice. Our experimental results show that, when the same amount of space is used, the latter approach indeed provides much faster query support than the former. This is because, in the query algorithm, only a constant number of succinct data structure operations are performed, and their execution time is dominated by other steps of the algorithm which require about $\mathcal{O}(\sqrt{n})$ time.

Encouraged by this observation, we further use succinct data structures to swap out more structures in the final solution of Chan et al. Experimental studies show that our variant of their structure achieves even better time/space tradeoffs, and the improvement is significant when the space cost needs to be reasonable for large

datasets, e.g., limited to under ten times the storage cost of the input array.

Regarding $(1+\epsilon)$-approximate range mode, we mainly focus on solutions by Bose et al. [4], Greve et al. [15] and El-Zein et al. [11], as well as a non-succinct version of the $\mathcal{O}(\frac{n}{\varepsilon})$-bit encoding structure of El-Zein et al. [11] which stores the sequences they encode succinctly in plain arrays instead. When setting $\epsilon = 1/2$, all these data structures provide much faster query time than the best exact solution (which already answers a query in microseconds), and the average approximation ratio of the answers is between 1.00001 and 1.02630. They also typically use less than $5n$ words of space, and thus they are excellent solutions to applications for which high average qualify of answers is sufficient. When encoded using compressed bit vectors, the space cost of the succinct encoding structure of El-Zein et al. [11] is only $35.6\% \sim 93.8\%$ of the input array of 32-bit integers (in most cases, the space cost is closer to the lower end, and the average space cost is 20.2 bits per symbol among all datasets). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space in practice. When decreasing the value of ϵ, we find the query times of these solutions increase at a logarithmic rate, but the space costs tend to be proportional to $1/\epsilon$. Hence, when a very small value of ϵ is desired, the best exact solution may be more viable instead.

1.3 Organization

The rest of the thesis is organised as follows. In Chapter 2, we introduce notation and machine model, as well as some key structures used in the design of the data structures for range mode. Chapter 3 describes the exact range mode data structures and approximate range mode data structures that we study in this thesis. An experimental study of these data structures is presented in Chapter 4. Chapter 5 presents our conclusion and possible future work.

Chapter 2

Preliminaries

This chapter includes the notation, the machine model and the succinct representations of bit vectors which is used as building blocks of some range mode structures.

2.1 Notation

Input array. We assume that the input is an array $A[1..n]$ of integers from $\{1, 2, \ldots, \Delta\}$, where $\Delta \leq n$.

Zeroth-order empirical entropy. We use the zeroth-order empirical entropy to evaluate the compactness of data structures. Assume we have one string or sequence $S[1..n]$, and we use f_x to represent the frequency of each symbol $x \in \{1, 2, \ldots, \Delta\}$ in S. The zeroth-order empirical entropy of S is then defined as $H_0(S) = \sum_{x=1}^{\Delta} \frac{f_x}{n} \log \frac{n}{f_x}$.

2.2 Machine Model

The model used in this thesis is the word RAM model. The key character of this model is that it allows random access to one single word and can perform operations (such as bitwise operations) on it. Under the word RAM model, if the input array is stored in one continuous memory block, we could access any element in this array by its index in constant time. If each element in the array uses w bits, then we can perform read, write, arithmetic and bitwise operations on each element in $\mathcal{O}(1)$ time. The solution that we implement use these operations.

2.3 Bit Vectors

A bit vector supporting `rank` and `select` operations is used in the design of many succinct data structures. The following operations are defined over a bit vector $B[1..n]$:

- `access`(i): return the bit stored in $B[i]$.
- `rank`$_b(i)$: return the frequency of bit $b \in \{0,1\}$ in the range $B[1..i]$.
- `select`$_b(i)$: return the index of the i-th occurrence $b \in \{0,1\}$ in B.

Pătraşcu [23] proposed the following solution:

Lemma 1 *A bit vector of length n in which t bits are 1s can be represented in $\lg \binom{n}{t} + \mathcal{O}(\frac{n}{\lg^c n}) \leq n + \mathcal{O}(\frac{n}{\lg^c n})$ bits for any positive constant c to support* `access`, `rank`, *and* `select` *in $\mathcal{O}(1)$ time.*

Chapter 3

Data Structure for Range Mode Queries

We review the range mode structures that we will implement. For the solutions to approximate range mode, the original authors did not present the algorithms for preprocessing, so we also discuss how to build these data structures efficiently.

3.1 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n}\lg n)$ Time

Krizanc et al. [19] introduced an $\mathcal{O}(n+s^2)$ word solution to support exact range mode in $\mathcal{O}((n/s)\lg n)$ time for any $s \in [1,n]$. This data structure can be constructed in $\mathcal{O}(ns)$ time. If we set $s = \sqrt{\lfloor n \rfloor}$, then the query time is $\mathcal{O}(\sqrt{n}\lg n)$, the space is $\mathcal{O}(n)$ words and the preprocessing time is $\mathcal{O}(n^{3/2})$.

In this data structure, we construct, for each integer $a \in \{1,2,\ldots,\Delta\}$, a sorted array Q_a of the positions of the occurrences of a in the input array A. Furthermore, array A is divided into s blocks each of size $\lceil \frac{n}{s} \rceil$, and we precompute an $s \times s$ tables S. For any integers $i,j \in [1,s]$, $S[i,j]$ stores the mode of the subarray consisting of blocks $i, i+1, \ldots, j$. Each row of S can be constructed by scanning A once. Thus, all these data structures occupy $\mathcal{O}(n+s^2)$ words and can be built in $\mathcal{O}(ns)$ time.

Having these structures, we can compute the mode in $A[a..b]$ by decomposing the query range $[a..b]$ into up to three subranges: the *span* of the range consists of all the blocks that are entirely contained in $[a..b]$, while the *prefix* and the *suffix* are the two subranges of $[a..b]$ before and after the span, respectively. Using S, we can find the mode, c, of the span in $\mathcal{O}(1)$ time. The answer to the query is either c, or an element in the prefix or the suffix. We call each of these elements a candidate, and hence there are at most $2\lceil \frac{n}{s} \rceil - 1$ candidates. For each candidate x, we use a binary search in $Q_A[x]$ to get the frequency of this candidate in the query range in $\mathcal{O}(\lg n)$ time. Since we have at most $2\lceil \frac{n}{s} \rceil - 1$ possible candidates, the total query time is $\mathcal{O}((n/s)\lg n)$.

3.2 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n})$ Time

One major component of the solution of Chan et al. [5] to the exact range mode problem is a data structure of $O(n + s^2)$ words which answers range mode queries in $\mathcal{O}(n/s)$ time, for any $s \in [1, n]$, and this data structure can be constructed in $\mathcal{O}(ns)$ time. Thus when $s = \sqrt{\lfloor n \rfloor}$, this data structure uses linear space to answer a range mode query in $\mathcal{O}(\sqrt{n})$ time, and the preprocessing time is $\mathcal{O}(n^{3/2})$. Even though this result is subsumed by their final solution which essentially combines it with succinct data structures to encode more procomputed information, it is simple. Thus, in our experimental studies, we also tested its performance to see whether the use of succinct data structures also improves the performance in practice.

In this solution, we construct an array Q_a for each $a \in \{1, 2, \ldots, \Delta\}$ as in Section 3.1. Then, we define a rank array A', in which $A'[i]$ stores the index of i in $Q_{A[i]}$. These structures allow us to determine, in constant time, whether $A[i]$ occurs at least q times in $A[i..j]$ for any given i, j and q, by checking if $Q_{A[i]}[A'[i] + q - 1] \leq j$. Furthermore, we partition A into s blocks and construct the $s \times s$ table S as in Section 3.1. An additional $s \times s$ table S' is also precomputed, in which $S'[i, j]$ stores the frequency of the mode in blocks $i, i + 1, \ldots, j$. All these data structures occupy $\mathcal{O}(n + s^2)$ words and can be built in $\mathcal{O}(ns)$ time.

To answer a query with these structures, we decompose the query range $[a..b]$ into up to three subranges: the *span* of the range consists of all the blocks that are entirely contained in $[a..b]$, while the *prefix* and the *suffix* are the two subranges of $[a..b]$ before and after the span, respectively. Using S and S', we can find the mode, c, of the span and its frequency, f_c, in the span in $\mathcal{O}(1)$ time. This is one possible candidate of the mode in $A[i..j]$. We then look for the elements whose frequencies in $A[i..j]$ are greater than f_c; these elements must occur in either the prefix or the suffix. We scan the prefix, and for each element $A[x]$ in it, we check whether we have seen it before in the scan by checking whether $Q_{A[x]}(A'[x] - 1)$ is at least i. If not, we determine whether $A[x]$ occurs more than f_c times in $A[x..j]$; this can be done in $O(1)$ time as discussed before. If it does, then $A[x]$ is a possible candidate, and we can compute its frequency in $A[i..j]$ by skipping the next $f_c - 1$ occurrences in $Q_A[x]$ and then continuing the scan of $Q_A[x]$ to find its remaining occurrences in $A[i..j]$. Since the number of times that $A[x]$ occurs in the span is at most f_c, the number of entries

that we scan in $Q_{A[x]}$ is thus upper bounded by the number of occurrences of $A[x]$ in the prefix and the suffix. Therefore, the time needed to compute the frequencies of all these candidates is linear in the sum of the lengths of the prefix and the suffix, which is $O(n/s)$. We scan the suffix in a similar manner, and the candidate with the highest frequency in $A[i..j]$ is the answer.

3.3 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n/w})$ Time

The final solution of Chan et al. [5] is a data structures of $O(n + s^2/w)$ words of space which answers range mode queries in $\mathcal{O}(n/s)$ time, for any $s \in [1, n]$, and the construction time is $\mathcal{O}(ns + n \lg(n/s))$. Therefore, when $s = \lceil\sqrt{nw}\rceil$, this data structure occupies $\mathcal{O}(n)$ words and has $\mathcal{O}(\sqrt{n/w})$ query time, and the preprocessing time is $\mathcal{O}(n^{3/2}\sqrt{w})$.

In this data structure, the input array A is partitioned into two subsequences B_1 and B_2 as follows: We scan A. If the current element appears at most s times in A, we append it to B_1. Otherwise, it is appended to B_2. Additionally, we define two $2 \times n$ tables $I_a[i]$ and $J_a[i]$, in which, for every $a \in [1, 2]$ and each $i \in [1, n]$, $I_a[i]$ (or $J_a[i]$) stores the index in B_a of the closest element in A to the left (or right) of $A[i]$ that lies in B_a. With these structures, a range mode query in A can be answered by answering one range mode query in B_1 and another in B_2.

The frequencies of the elements in array B_1 are upper bounded by s, and this allows Chan et al. to design a compact version of the solution in Section 3.2 to support range mode over B_1. More precisely, we still construct Q_a for each integer a, but we encode the entries of the table S' using succinct data structures, and we do not store S explicitly. To encode S', observe that, for each $i \in \{1, 2, \ldots, s\}$, the i-th row of S' essentially encodes the following sequence of at most s numbers: the frequency of the mode in block i, the frequency of the mode in blocks i and $i+1$, the frequency of the mode in blocks i, $i+1$ and $i+2$, etc. Hence, these numbers form a monotonically increasing sequence, and the largest number is upper bounded by s. Thus, to encode the ith row of S', we can start by encoding the first number in this sequence in unary, i.e., if this number is v, we write down v 0-bits followed by a 1-bit. Then we encode each subsequent number by expressing the difference between this number and the previous number in unary. The i-th row of S' is then represented as

a concatenation of these unary codes, forming a bit vector of at most s 1-bits and at most s 0-bits. It can be represented in $O(s)$ bits by Lemma 1 to support `rank` and `select` operations in constant time, using which we can retrieve any entry of the i-th row of S' in constant time given its column number. If we use this encoding scheme for every row of S', then S' can be presented in $O(s^2)$ bits, or $O(s^2/w)$ words, and we can retrieve any entry of S' in constant time. Furthermore, we can also use this structure to infer any entry of S without storing S explicitly. Recall that $S[i, j]$ is the actual mode element of blocks $i, i + 1, \ldots j$. To compute it, we first retrieve $S'[i, j]$, which gives us the frequency, f, of $S[i, j]$ in blocks $i, i+1, \ldots, j$. Then, among these blocks, we can find the last block containing at least one occurrence of $S[i, j]$ by performing `rank` and `select` operations over S'. Finally, we use a process similar to the one described in Section 3.2 to scan this block in $O(n/s)$ time and find out which element occurs f times in these blocks. Hence, the encoding scheme here decreases the space cost to $O(n + s^2/w)$ words, while allowing us to use the algorithm in Section 3.2 to answer a range mode query in $O(n/s)$ time. The preprocessing time is still $O(ns)$, as each bit vector can be built in linear time.

As for those elements in array B_2, observe that, since each element occurs more than s times in A, the number of distinct elements, Δ', is at most n/s. Hence, we convert B_2 into an array of at most n elements, each from universe $\{1, 2, \ldots, \Delta'\}$. This rank reduction can be done in $O(n \lg \Delta')$ time. Then the array B_2 is divided into $\lfloor n/\Delta' \rfloor$ blocks each of size Δ'. Moreover, we precompute a $\lfloor n/\Delta' \rfloor \times \Delta'$ table C, in which, for each $i \in \{1, ..., \lfloor n/\Delta' \rfloor\}$ and every $x \in \{1, ..., \Delta'\}$, $C_i[x]$ stores the frequency of x in $A[1, i\Delta']$. C can be built by scanning A once in $O(n)$ time, and it uses $O(n)$ words of space. To answer a query, we divide the query range into the prefix, the span and the suffix as in Section 3.2. We use the table C to find the frequencies of all Δ' distinct elements in the span, and scan the prefix and the suffix to count the number of additional occurrences of each element in them. Thus, this process can compute the frequency of each element in the query range in $O(\Delta')$ time, and the element with the highest frequency is the answer.

By adding up the time and space costs of the structures for B_1 and B_2 and applying the inequality that $\Delta' \leq n/s$, the bounds claimed at the beginning of this section thus follow.

Remarks. We can further decrease the space overhead by replacing I_a and J_a, where $a \in \{1, 2\}$, with a bit vector F, in which $F[i] = 0$ if $A[i]$ is stored in B_1 and $F[i] = 1$ if $A[i]$ is stored in B_2. Then, given a query range $[i, j]$, we have that the elements in $A[i..j]$ are stored in two subsequences $B_1[\texttt{rank}_0(i-1)+1, \texttt{rank}_0(j)]$ and $B_2[\texttt{rank}_1(i-1)+1, \texttt{rank}_1(j)]$. With it, we can use $n + o(n) + \mathcal{O}(s^2/w)$ words of space to answer a query in $\mathcal{O}(n/s)$ time. We will study both the original approach and this variant experimentally.

3.4 $(1+\varepsilon)$-Approximation in $O(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ Time

A simple solution. To design approximate solutions, Bose et al. [4] first presented a simple approach: For each $i \in \{1, 2, \ldots, n\}$, build a table T_i in which $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs $\lceil (1+\epsilon)^j \rceil$ times in $A[i..j]$. Thus T_i has $\mathcal{O}(\log_{1+\varepsilon} n)$ entries, and by Taylor series expansion, $1/\lg(1+\varepsilon) = \mathcal{O}(1/\epsilon)$, so the length of T_i is $\mathcal{O}((\lg n)/\varepsilon)$. Given a query range $[a, b]$, they perform an binary search in T_a to find the entry $T_i[k]$ with $T_i[k] \leq b < T_i[k+1]$, and $A[T_i[k]]$ is a $(1+\varepsilon)$-approximate answer. The query algorithm hence uses $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time, and these tables occupy $O(\frac{n \lg n}{\varepsilon})$ words in total.

An improved solution. To improve space efficiency, Bose et al. [4] define these tables differently so that they share many common entries and can thus be stored in persistent data structures to save space. In their approach, they first define two number series, f_{low} and f_{high}. The first entries of these series, f_{low_1} and f_{high_1}, are both defined to be 1. Then, the $(r+1)$st entries, $f_{low_{r+1}}$ and $f_{high_{r+1}}$, can be computed recursively using $f_{low_{r+1}} = f_{high_r} + 1$ and $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$. They then construct a table T_i for each $i = 1, 2, \ldots, n$ as follows. In T_1, an entry $T_1[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[1..j]$. To compute an entry $T_i[r]$ for any $i \geq 2$, we first determine whether $T_{i-1}[r]$ occurs at least f_{low_r} times in $A[i..T_{i-1}[r]]$. If it does, then we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[i..j]$. To answer a query, observe that, the frequency of the mode of any query range $[a, b]$ with $T_a[r] \leq b < T_a[r+1]$ is at most $f_{high_{r+1}} - 1$. Since $A[T_a[r]]$ occurs at least f_{low_r} times in $A[a..T_a[r]] \subseteq A[a..b]$, the ratio of its frequency in $A[a..b]$ to $F_{a,b}$ is at least $f_{low_r}/(f_{high_{r+1}} - 1) = f_{low_r}/\lfloor (1+$

$\varepsilon) f_{low_r} \rfloor \leq 1/(1+\varepsilon)$. [1] Therefore, $A[T_a[r]]$ is a $(1+\varepsilon)$-approximate answer. With this observation, it suffices to perform a binary search in a table to answer a query approximately.

Each table has at most $2\lceil \lg_{1+\varepsilon} n \rceil$ entries. To reduce storage costs, Bose et al. view $T_1, T_2, \ldots, T_n$ as n different versions of the same table T, and, to obtain T_i (version i) from T_{i-1} (version $i-1$), an update is needed for each r with $T_i[r] \neq T_{i-1}[r]$. A crucial observation is that, if $T_i[r] \neq T_{i-1}[r]$, then $T_i[r] = T_{i+j}[r]$ for any $j \in [0, f_{high_r} - f_{high_{r-1}} - 1]$, because the frequency of $A[T_i[r]]$ in $A[i+j..T_i[r]]$ is at least $f_{high_r} - j \leq f_{high_r} - (f_{high_r} - f_{high_{r-1}} - 1) = f_{high_{r-1}} + 1 = f_{low_r}$. They then used this observation to prove that the total number of updates needed to obtain T_{i+1} from T_i over all $i = 1, 2, \ldots, n-1$ is $\mathcal{O}(n/\varepsilon)$. Hence they can use a persistent binary search tree [9] to store all these tables in $\mathcal{O}(n/\varepsilon)$ words of space while supporting binary search in any table in $\mathcal{O}(\lg(2\lceil \lg_{1+\varepsilon} n \rceil)) = \mathcal{O}(\lg \lg n + \lg \frac{1}{\varepsilon})$ time. This results in an $\mathcal{O}(\frac{n}{\varepsilon})$-word structure supporting $(1+\varepsilon)$-approximate range mode in $\mathcal{O}(\lg \lg n + \lg \frac{1}{\varepsilon})$ time. The preprocessing time is dominated by the time needed to compute the content of all n tables, and Bose et al. used the idea of maintaining frequency counters as in [8] to compute them in $O(\frac{n \lg n}{\varepsilon})$ time.

3.5 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg \frac{1}{\varepsilon})$ Time

Greve et al. [15] further improves query times to $\mathcal{O}(\lg \frac{1}{\varepsilon})$ while still using $\mathcal{O}(\frac{n}{\varepsilon})$ words of space. In their solution, they define $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$, and the data structure consists of two parts. Given a query range $[a, b]$, the first part determines whether $F_{a,b} \leq \lceil \frac{1}{\varepsilon'} \rceil$, and if so (the low frequency case), answer the query. Otherwise, the algorithm further queries the second part (the high frequency case).

Low Frequency: For each $i = 1, 2, \ldots, n$, we precompute a table Q_i of length $\lceil \frac{1}{\varepsilon'} \rceil$, in which $Q_i[r]$ stores the rightmost index j such that $F_{i,j} = r$. Given a query range $[a, b]$, we can perform a binary search in Q_a to look for the successor of b. If b does not have a successor, then $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$, and we use the structures for high frequencies

[1] Bose et al. [4] originally defined $f_{high_{r+1}} = \lceil (1+\varepsilon) f_{low_r} \rceil + 1$. However, with their definition, the ratio of the frequency of $A[T_a[r]]$ in $A[a..b]$ to $F_{a,b}$ is at least $f_{low_r} / \lceil (1+\varepsilon) f_{low_r} \rceil$ which is not guaranteed to be at least $1/(1+\varepsilon)$. Our implementation also confirms that the original definition does not guarantee the approximation ratio to be $1+\epsilon$. Therefore, we fix this issue by defining $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$ instead.

to compute an answer. Otherwise, let s be the index of the successor of b in Q_a, and we have $F_{i,j} = s$. In this case, as observed by El-Zein et al. [10], $A[Q_a[s-1]+1]$ is the mode in $A[a,b]$. [2]

High Frequency: For each $i = 1, 2, \ldots, n$, we precompute a table T_i of length at most $\lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil$: For each $r \in [1, \lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil]$, if $i > 1$ and $F_{i,T_{i-1}[r]} \geq \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$, then we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the rightmost index j with $F_{i,j} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1} \rceil - 1$. We also build another table L_i for each i. To define $L_i[r]$, let j be the smallest positive integer such that $T_{i+j}[r] \neq T_i[r]$. Then $L_i[r] = A[i+j-1]$. By these definitions, $L_i[r]$ occurs at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$ times in $A[i..T_i[r]]$.

With these tables, the high frequency case can be handled as follows. Let $[a,b]$ be a query range with $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$. We find the successor, $T_a[s]$, of b in T_a. Then, based on the above definitions, we have that $F_{a,T_a[s]} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \rceil - 1$ and that the frequency of $L_a[s-1]$ in $A[a..T_a[s-1]]$ is at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s-1} \rceil + 1$. Since $[a, T_a[s-1]] \subseteq [a,b] \subseteq [a, T_a[s]]$, we have that $F_{a,b} \leq F_{a,T_a[s]} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \rceil - 1$, and that the frequency of $L_a[s-1]$ in $A[a..b]$ is at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s-1} \rceil + 1$. Since $(\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s-1} \rceil + 1)/(\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \rceil - 1) \geq 1/(1+\varepsilon')^2 = 1/(1+\varepsilon)$, $L_a[s-1]$ is a $(1+\varepsilon)$-approximate mode of $A[a..b]$ and can be returned as the answer.

Hence, the total query time over both the low frequency and the high frequency structures is $\mathcal{O}(\lg \frac{1}{\varepsilon'} + \lg\lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil) = \mathcal{O}(\lg\lg n + \lg \frac{1}{\varepsilon'}) = \mathcal{O}(\lg\lg n + \lg \frac{1}{\varepsilon})$; these identities hold as $1/\lg(1+\varepsilon') = \mathcal{O}(1/\epsilon')$ and $\varepsilon' = \frac{\sqrt{(1+\varepsilon)+1}}{\varepsilon}$. To further speed up the query, Greve et al. designed a data structure that can compute a 3-approximation of the answer in constant time. Then, in the high frequency case, the query algorithm starts by computing this 3-approximation and finding its predecessor in T_a. Afterwards, a binary search in the next $\mathcal{O}(\log_{1+\varepsilon'} 3) = \mathcal{O}(\frac{1}{\varepsilon})$ entries of T_a gives an $(1+\varepsilon')$-approximation in $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time.

This 3-approximate structure relies on constant-time support for lowest common ancestor (LCA) queries over trees with exponential degrees whose heights are only $O(\lg\lg n)$. Unfortunately, an experimental study by Bender et al. [3] on LCA queries

[2] To return the mode, the original work of Greve et al. [15] augments the low frequency structure by storing the mode in $A[i..Q_i[k]]$ with each $Q_i[k]$. This approach is easy to understand and does not break the asymptotic space bound. However, to implement this data structure, we avoid storing these mode elements and use the observation by El-Zein et al. [10] to save storage costs in practice.

suggests that, in practice, for trees with small heights, structures with constant query time in theory are outperformed by naive approaches whose worst-case query time is linear in the tree height. Indeed, the value of $\lg \lg n$ is small for practical data, and theoretical approaches that remove this additive term are often impractical. Hence, we do not implement this 3-approximate structure and performs a binary search over the high frequency structure instead.

To bound the space costs, first observing that the low frequency structure uses $\mathcal{O}(\frac{n}{\varepsilon'}) = \mathcal{O}(\frac{n}{\varepsilon})$ words. In the high frequency structure, however, the total number of entries of T_i's and L_i's is $\mathcal{O}(n\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil)$. To store them in a compact manner, we again view these T_i tables as n version of the same table T as was done by Bose et al. [4] for their data structures (see Section 3.4). Then, each time we update $T[r]$ in some version, we need not update it again in the next $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil - 1 - \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k}\rceil - 1 \geq \lfloor (1+\varepsilon')^k - 3\rceil$ versions. This can be further used to bound the total number updates to T by $\mathcal{O}(\frac{1}{\varepsilon})$. A similar argument apples to L_i's. It is possible to store T_i's and L_i's in a persistent search tree, but this scheme cannot be combined with the 3-approximate structure to achieve $\mathcal{O}(\lg \frac{1}{\varepsilon})$ query time.

Instead, Greve et al. designed a scheme based on sampling to store these tables in $\mathcal{O}(n/\varepsilon)$ words while supporting the retrieval of an arbitrary entry in constant time. Here we only sketch the scheme of storing T_i's; the entries of L_i's can be paired with those of T_i's and stored as additional fields in the same structures. In this scheme, we explicitly store T_l in an array S_l if $l \bmod t = 1$, i.e., we sample and store one out of every t versions of T. Let r be an arbitrary integer in $[1, \lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil$. Between two consecutive sampled versions, $T_l[r]$ and $T_{l+t}[r]$, of $T[r]$, there may be updates to $T[r]$. Observe that, if $r \geq 1 + \lceil \log_{1+\varepsilon'} t\rceil$, then there can only be at most one update to $T[r]$ between versions l and $l+t$. In this case, we store with each sampled entry $T_l[r]$ the next update to $T[r]$. If $r \leq \lceil \log_{1+\varepsilon'} t\rceil$, then, for each sampled entry $T_l[r]$, construct a bit vector of length t with constant-time support for `rank` which uses one bit for each of the next t versions to encode whether an update to $T[r]$ is performed. We also store the (distinct) values used to update $T[r]$ in an array.

Preprocessing: As Greve et al. did not provide information on preprocessing, we discuss it here. The low frequency structure can be constructed in $\mathcal{O}(n/\varepsilon)$ time using frequency counters [8] as was done by Bose et al. [4] to compute similar tables. For

the high frequency structure, if we have already computed the content of T_i's and L_i's, we can encode them in time linear in the total number of entries in T_i's and L_i's, and there are $\mathcal{O}(n\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n \lg n)/\varepsilon)$ entries.

What remains is to compute the entries of T_i's and L_i's, and for this we scan A $\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil$ times. In the r-th scan, we compute $T_i[r]$ and $L_i[r]$ for all $i \in [1, n]$ in increasing order of i as follows. We maintain an array $C[1..\Delta]$ of counters; initially all entries of C are $0s$. We use an integer m to keep track of the number of entries of C that are greater than or equal to $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$; m can be updated each time an entry of C is updated. During the scan, we maintain the following invariant: immediately after computing $T_i[r]$, each counter $C[j]$, stores the number of occurrences of j in $A[i..T_i[r]]$. To compute $T_1[r]$, we retrieve $A[k]$ for $k = 1, 2, ...$, and for each k, we increment $C[A[k]]$. We repeat until $C[A[k]]$ is the first counter in C that reaches $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil$. This means $A[1..k-1]$ is the longest prefix of A whose mode has frequency $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil - 1$ in it. Therefore, we set $T_1[r] = k - 1$. Then we put the entry $A[k]$ back to the portion of A that we have not scanned by decrementing $C[A[k]]$ and then k. To compute $T_i[r]$ for any $i > 1$, we first decrement $C[A[i-1]]$ and then check whether m is still greater than 0. If it is, then there is at least one element whose frequency in $A[i..T_{i-1}[r]]$ is $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$, and we set $T_i[r] = T_{i-1}[r]$. Otherwise, we resume the scanning of A to compute $T_i[r]$ using the approach used to compute $T_1[r]$. We also store $A[i-1]$ in $L_r[u], L_r[u+1], \ldots, L_r[r-1]$, where u is the smallest integer such that $T_u[r] = T_{r-1}[r]$. With this implementation, we need to scan the input array A $\mathcal{O}(\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil)$ times, and hence the total preprocessing time is $\mathcal{O}(n\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n \lg n)/\varepsilon)$.

3.6 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Bits and $\mathcal{O}(\lg \frac{1}{\varepsilon})$ Time

El-Zein et al. [10] designed an encoding data structure of $\mathcal{O}(\frac{n}{\varepsilon})$ bits which can compute the index of a $(1+\varepsilon)$-approximate mode of a query range in $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time. It also consists of two parts: The low frequency structure can check whether $F_{a,b} \leq \lceil \frac{1}{\varepsilon}\rceil$ and answer the query if so. Otherwise, the high frequency structure is used.

Low Frequency: For each integer $k \in [1, \lceil \frac{1}{\varepsilon}\rceil]$, we build a table Q_k of length n, in which $Q_k[i]$ stores the rightmost index j such that $F_{i,j} = k$. Thus these tables store essentially the same data as the low frequency structure of Greve et al. [15] does,

but the data are organized differently. A key observation is that each Q_k is an array of n increasing numbers, and the largest number is n. This allows us to store Q_k succinctly: Encode $Q_k[1]$ and $Q_k[i] - Q_k[i-1]$ for each i in unary, i.e., as $Q_k[1]$ or $Q_k[i] - Q_k[i-1]$ 0-bits followed by a 1. The concatenation is a $2n$-bit bit vector, and to compute the $Q_k[i]$, we can locate the i-th 0 and count how many 1-bits are before it. Thus, by constructing $o(n)$-bit structures supporting `rank` and `select` operations in constant time [6] with $o(n)$ extra bits, each entry of Q_k can be computed in $\mathcal{O}(1)$ time. Hence, we can store all these tables in $\mathcal{O}(\frac{n}{\varepsilon})$ bits. Then, for a given range $[a, b]$, we perform a binary search in $Q_1[a], Q_2[a], \ldots, Q_{\lceil \frac{1}{\varepsilon} \rceil}[a]$ to handle the low frequency case.

High Frequency: The high frequency structure is based on the following trichotomy:

Lemma 2 ([10]) *Let k be an arbitrary integer in $[1, \lfloor \log_{1+\varepsilon}(\varepsilon n) \rfloor]$. There is a data structure of $\mathcal{O}(k\varepsilon n/(1+\varepsilon)^k + n/\lg^2 n)$ bits that can find in constant time one of the following inequalities that holds for any query range $[a, b]$: 1) $F_{a,b} < (1+\varepsilon)^k/\varepsilon$, 2) $F_{a,b} > (1+\varepsilon)^k/\varepsilon$, or 3) $(1+\varepsilon)^{k-1/2}/\varepsilon < F_{a,b} < (1+\varepsilon)^{k+1/2}/\varepsilon$.*

In case 2, the structure finds an element that occurs more than $(1+\varepsilon)^k/\varepsilon$ times in $A[a, b]$. In case 3, an element that occurs more than $(1+\varepsilon)^{k-1/2}/\varepsilon$ times in $A[a, b]$ is found.

To prove this lemma, let $\varepsilon' = \sqrt{1+\varepsilon} - 1$ and $f_j = (\varepsilon'/\varepsilon) \times (1+\varepsilon')^j$. We then define four integer sequences s, s', r and r' as follows. For each integer $i \in [0, n/\lceil f_{2k-1} \rceil]$, define s_i, the i-the element in s, as $i\lceil f_{2k-1} \rceil + 1$, and r_i is then defined as the smallest index such that $F_{s_i,r_i} \geqslant (1+\varepsilon')^{2k}/\varepsilon$. similarly, for each integer $i \in [0, n/\lceil f_{2k} \rceil]$, define $s'_j = i\lceil f_{2k} \rceil + 1$, and r'_j is the smallest index such that $F_{s'_j,r'_j} \geqslant (1+\varepsilon')^{2k+1}/\varepsilon$.

Given a query range $[a, b]$, we compute the largest elements, s_i and s'_j, of s and s' that are less than or equal to a. Then, El-Zein et al. proved that, if $b < r_i$, then case 1 of the lemma applies. If $b \geq r'_j$, then case 2 applies, and r'_j occurs more than $(1+\varepsilon)^k/\varepsilon$ times in $A[a, b]$. Finally, if $r_i \leq b < r'_j$, case 3 applies, and r_i occurs more than $(1+\varepsilon)^{k-1/2}/\varepsilon$ times in $A[a, b]$. Therefore, if we precompute sequences r and r', all these can be determined in constant time. To store r and r', observe that they are increasing sequences containing respectively $\lceil n/f_{2k-1} \rceil$ and $\lceil n/f_{2k} \rceil$ elements upper bounded by n. Hence we can again encode them using Lemma 1 to achieve the space bound in Lemma 2.

To use this trichotomy to answer queries, for each integer $k \in [1, \lfloor \log_{1+\varepsilon}(\varepsilon n) \rfloor]$, build the data structures for Lemma 2; El-Zein et al. showed that they occupy $\mathcal{O}(\frac{n}{\varepsilon})$ bits in total. Then, in the high frequency case, perform a binary search to compute a k such that either $(1+\varepsilon)^{k-1/2}/\varepsilon < F_{i,j} < (1+\varepsilon)^{k+1/2}/\varepsilon$ (i.e., case 3 applies for this k) holds, or $(1+\varepsilon)^k/\varepsilon < F_{i,j} < (1+\varepsilon)^{k+1}/\varepsilon$ (i.e., case 2 applies for k and case 1 applies for $k+1$). The element found by either case 3 or case 2 of the encoding is a $(1+\varepsilon)$-approximate mode. This requires $\mathcal{O}(\lg\lfloor \log_{1+\varepsilon}(\varepsilon n)\rfloor) = \mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time. Finally, to speed up the query time to $\mathcal{O}(\lg\frac{1}{\varepsilon})$, El-Zein et al. designed an $\mathcal{O}(n)$-bit structure that answers 4-approximate range mode queries in constant time, and used it to narrow down the initial range of binary search.

Remarks: The $\mathcal{O}(n)$-bit 4-approximate structure of El-Zein et al. contains a network of fusion trees [13]. It is not practical, and hence our implementation does not include this speedup. El-Zein et al. did not discuss preprocessing, but we can build their data structures by maintaining frequency counters [8] in $\mathcal{O}(n \lg n/\varepsilon)$ time. Finally, if we do not implement the low frequency or high frequency structures as succinct bit vectors but store them in integer arrays instead, we would achieve a simple $\mathcal{O}(n)$-word solution. Thus, we conduct experimental studies on both this plain version and the original succinct version.

Chapter 4

Experimental Results

This chapter presents our experimental studies on data structures on range mode queries.

4.1 Implementation

We implemented the data structures discussed in Chapter 3, and Table 4.1 gives an outline. Among them, the first naive approach, $\texttt{nv}_1$, sorts the elements in the given range to answer a query, while the second one, $\texttt{nv}_2$, scans the elements in the query range and uses an array of length Δ to keep track of the number of times each element seen so far. Four data structures, $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` and `succ`, use succinct bit vectors, for which we use the implementation in the succinct data structures library, `sdsl-lite`, of Gog et al. [14]. Two types of bit vectors from `sdsl-lite` are used: a plain bit vector, `sdsl::bit_vector` and a compressed bit vector [25], `sdsl::rrr_vector`. To distinguish them, we combine $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` or `succ` with superscripts `p` or `c`, e.g., $\texttt{succ}^{\texttt{p}}$ and $\texttt{succ}^{\texttt{c}}$, to respectively indicate whether plain bit vectors or compressed bit vectors are used. Note that, even though $\texttt{subsr}_2^{\texttt{c}}$ uses compressed bit vectors to encode the table S', a plain bit vector is still used to represent F. This is because, in preliminary studies, we found that, due to the small space cost of F (n bits), compressing it would achieve negligible space savings at the cost of increasing query times by $4.5\% \sim 25\%$. Finally, for a fair comparison, we modified the implementation of persistent search trees by Jansens [18] slightly to decrease the space overhead for generic programming, and used it to implement `pst`.

4.2 Experimental Setup

Five publicly available datasets are used in our experiments; see Table 4.2. This table also shows the zeroth-order empirical entropy, H_0, of each dataset. To convert

Table 4.1: The data structures we implemented, with their abbreviations. The first half of the table present exact solutions, while the second half are for $(1+\varepsilon)$-approximate range mode, for which, as discussed in Chapter 3, we implement practical solutions with $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ query time.

abbr.	description
$\texttt{nv}_1$,$\texttt{nv}_2$	two naive solutions in Section 4.1
`supsr`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n}\lg n)$ query time structure for exact range mode in Section 3.1
`sqrt`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n})$ query time structure for exact range mode in Section 3.2
$\texttt{subsr}_1$	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n/w})$ query time structure for exact range mode in Section 3.3
$\texttt{subsr}_2$	modifying $\texttt{subsr}_1$ with more succinct data structures; see the remarks in Section 3.3
`simple`	simple $\mathcal{O}(\frac{n\lg n}{\varepsilon})$-word approximate solution in Section 3.4
`pst`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with persistent search trees in Section 3.4
`sample`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with sampling in Section 3.5
`tri`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with the trichotomy in Section 3.6
`succ`	$\mathcal{O}(\frac{n}{\varepsilon})$-bit approximate solution with the trichotomy in Section 3.6

Table 4.2: The data sets used in our experiments, each stored as an array of n integers in $[1, \Delta]$.

data	n	Δ	$\lg \Delta$	H_0	description
`reviews`	10,000,000	1,367,909	20.38	18.46	the books of the first 10^8 book reviews by Amazon customers in 2018 [21]
`IPs`	8,571,089	135,542	17.04	7.96	the source IP addresses of the DDoS attacks recorded in [12]
`words`	6,715,122	127,886	16.96	12.74	the words in a text string containing the 100 most frequently downloaded Project Gutenberg [1] e-books in July 2021, with stop words removed
`library`	10,000,000	314,358	18.26	15.75	the first 10^8 call numbers in the Seattle Public Library checkout records of 2016 and 2017 [20]
`tickets`	10,000,000	79,027	16.27	11.10	the street names of the first 10^8 parking tickets issued in New York in 2017 [22]

raw data into an integer array over which we perform queries, we extract elements from their respective records (e.g., extract call numbers from the checkout records in `library`) and compute the number, Δ, of distinct elements in the dataset. Then we encode each element as an integer in $[1, \Delta]$ and store the dataset as an array of these integers. In our experiment, we adopt the query generation method in [7, 17]. To generate a query range $[a, b]$, we pick an integer from $[1, n]$ uniformly at random (u.a.r.) and assign it to a, and b is chosen u.a.r. in the interval of $[a, a + \lceil \frac{n-a}{K} \rceil]$ for a parameter K. We generate three categories of queries, `large`, `medium` and `small`, by setting $K = 1$, 10 and 100, respectively.

Our platform is a sever with an Intel(R) Xeon(R) Gold 6234 CPU at 3.30GHz and 128GB of RAM, running 4.15.0-54-generic 58-Ubuntu SMP x86_64 kernel. We complied the programs using `g++` 7.4.0 with `-O2` flags.

4.3 An Initial Performance Study for Exact Range Mode

We first perform experimental studies on exact range mode queries. Initially, we set $s = \sqrt{n}$ for `supsr` and `sqrt`, then set $s = \sqrt{nw}$ for $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve linear space as in the original work [19, 5]. Tables 4.3 and 4.4 present the query time, space usage and construction time of the exact query structures. We measure space costs by bits per symbol, which is the total space usage in bits divided by the length, n, of the input array. When calculating space, the cost of the input array A (32 bits per symbol) is included in the space usage of `supsr` and `sqrt` but excluded for $\texttt{subsr}_1$ and $\texttt{subsr}_2$. This is because `supsr` and `sqrt` scan A when answering a query but $\texttt{subsr}_1$ and $\texttt{subsr}_2$ do not. Nevertheless, the space cost of A is not significant enough to affect our conclusions. These tables show that most data structure solutions implemented have much faster query time than naive approaches, and `supsr` is the only exception: it has better query performance than naive approaches over the `reviews` dataset, while $\texttt{nv}_2$ performs better in `small` and `medium` queries over other four datasets than `supsr`. Between two naive approaches, $\texttt{nv}_2$ is faster because the number of distinct elements is relatively small compared to input array length.

Before comparing the performance of data structure solutions, we discuss how the distributions of the datasets affect $\texttt{subsr}_1$ and $\texttt{subsr}_2$. In these two solutions, the array entries are distributed into two subsequences B_1 and B_2: the entries of the input array A storing elements with frequency higher than s are stored in B_2, while the rest are stored in B_1. Different structures are then constructed to answer queries over B_1 and B_2. Since B_2 stores elements of higher frequency, the lower the entropy of a dataset is, the ratio of the length of B_2 to n tends to be larger. Indeed, we found that, for `reviews`, `words` and `library`, the ratios are 0, 0.037 and 0.010, respectively, while for `IPs` and `tickets`, the ratios are 0.58 and 0.14, respectively, which are higher. These ratios are consistent with the values of H_0 shown in Table 4.2. This immediately explains why, for the dataset `reviews`, there is no difference in time/space costs between $\texttt{subsr}_1$ and $\texttt{subsr}_2$: These two solutions differ in the data structure components used to map the query range to ranges in B_1 and B_2. Since the length of B_2 is 0 for `reviews`, no mapping is needed. Our implementation detects this special case without constructing these components and uses the data structures for B_1 to answer queries.

Table 4.3: Average time to answer an exact range mode query, measured in micro seconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^6 queries.

	Query	$\texttt{nv}_1$	$\texttt{nv}_2$	supsr	sqrt	$\texttt{subsr}_1^{\texttt{p}}$	$\texttt{subsr}_1^{\texttt{c}}$	$\texttt{subsr}_2^{\texttt{p}}$	$\texttt{subsr}_2^{\texttt{c}}$
reviews	small	1134	442	338.93	51.70	10.74	11.56	10.74	11.56
	medium	13262	870	366.90	51.00	9.94	10.82	9.94	10.82
	large	144642	5686	363.42	50.85	9.39	10.20	9.39	10.20
IPs	small	532	51	218.75	15.58	3.93	4.40	3.94	4.48
	medium	5938	186	240.03	15.19	3.86	4.37	3.91	4.46
	large	66121	1531	239.35	14.48	3.53	4.01	3.60	4.07
words	small	678	45	298.83	31.49	8.01	8.75	8.14	9.06
	medium	7094	149	334.53	31.27	7.60	8.41	7.82	8.63
	large	73401	1235	349.22	28.28	6.53	7.24	6.67	7.38
library	small	1160	125	384.07	49.54	11.90	13.14	12.13	13.37
	medium	12960	408	422.32	47.11	10.66	11.87	10.71	11.98
	large	132605	3407	444.30	43.68	9.32	10.42	9.40	10.53
tickets	small	990	37	362.47	43.16	9.99	10.67	10.19	10.99
	medium	9931	187	414.76	42.39	9.92	10.65	10.15	10.97
	large	101281	1756	436.93	37.44	8.56	9.35	8.80	9.60

Table 4.4: Space (bits per symbol) and construction time (minutes) of exact range mode structures.

	Dataset	supsr	sqrt	$\texttt{subsr}_1^{\texttt{p}}$	$\texttt{subsr}_1^{\texttt{c}}$	$\texttt{subsr}_2^{\texttt{p}}$	$\texttt{subsr}_2^{\texttt{c}}$
space	reviews	109.1	173.2	174.3	144.1	174.3	144.1
	IPs	97.5	161.5	332.6	255.9	205.8	129.0
	words	97.8	161.9	329.1	284.1	202.2	157.2
	library	99.0	163.0	315.2	294.5	188.3	167.6
	tickets	96.7	160.8	311.0	289.9	184.1	163.0
construct time	reviews	0.911	0.911	7.205	7.460	7.205	7.460
	IPs	0.695	0.695	1.865	1.867	1.890	1.892
	words	0.438	0.438	2.755	2.760	2.762	2.765
	library	0.806	0.806	5.923	5.933	5.971	5.974
	tickets	0.720	0.720	4.251	4.275	4.756	4.809

With this in mind, we now compare the data structure solutions. We first find that the query time of `supsr` is $6.6 \sim 16.5$ times as much as that of `sqrt`. This is because that the query time of `supsr` is $\mathcal{O}(\sqrt{n} \lg n)$ whereas the query time of `sqrt` is $\mathcal{O}(\sqrt{n})$. The space used by the `supsr` is less since we do not use A' and S' structures in `supsr`. Then we observe that, by using a succinct bit vector to replace multiple arrays, $\texttt{subsr}_2$ saves much space compared to $\texttt{subsr}_1$ over all datasets except `reviews` (which does not require these components as discussed before). At the same time, there is almost no sacrifice in query performance. This is because we only use `rank` operations over this bit vector to map the query ranges to ranges in two subsequences of the input array, and as the support for `rank` is efficient, this time is dominated by subsequent operations which use $O(\sqrt{n/w})$ time. The use of compressed bit vectors in $\texttt{subsr}_1^{\texttt{c}}$ and $\texttt{subsr}_2^{\texttt{c}}$ also achieves some space savings, albeit at the cost of a small increase in query time. The theoretical analysis indicates that, when we double s, the query time halves but the space cost of tables S and S' will become four times as large. Hence, based on this initial study, we predict that, $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ achieve the best query-space tradeoffs among these data structure solutions, and more experiments will be run in Section 4.4 to confirm this.

The sizes of query ranges affect query times greatly for the naive approaches since they either sort or scan the elements in the range. On the other hand, these sizes only affect the query times of `supsr`, `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ slightly. For `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$, larger queries even tend to take less time to answer. This is because the query algorithm of `sqrt` (which is also performed over B_2 in $\texttt{subsr}_1$ and $\texttt{subsr}_2$) keeps updating a candidate by a new candidate with higher frequency in the query range, until the mode of the range is found. The initial candidate is the mode of the span of the query. When the query range is larger, the span is also longer, and hence its mode tends to a better candidate, thus decreasing the query time.

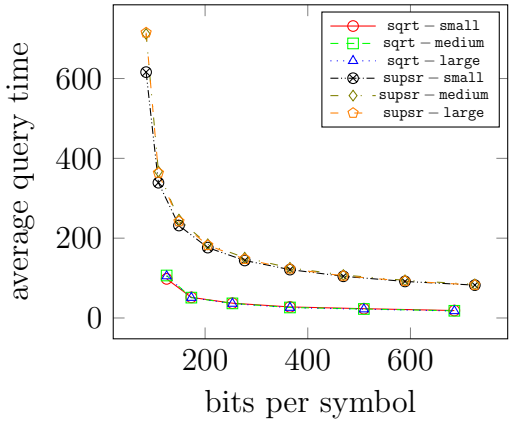
Regarding construction times, observer that, for `supsr` and `sqrt`, their processing times are about same. For `reviews`, `words`, `library` and `tickets`, the preprocessing time of $\texttt{subsr}_1$ and $\texttt{subsr}_2$ is $5.9 \sim 8.2$ times as much as that of `supsr` and `sqrt`. This is because, with the choices of parameters, it takes $\mathcal{O}(n^{3/2}\sqrt{w})$ time to build $\texttt{subsr}_1$ and $\texttt{subsr}_2$, but the preprocessing time of `supsr` and `sqrt` is $\mathcal{O}(n^{3/2})$. However, the difference is much smaller for `IPs`. This is because, when constructing $\texttt{subsr}_1$ and

$\texttt{subsr}_2$ for this dataset, 58% of array entries are in B_2, whose query structure can be built in linear time.

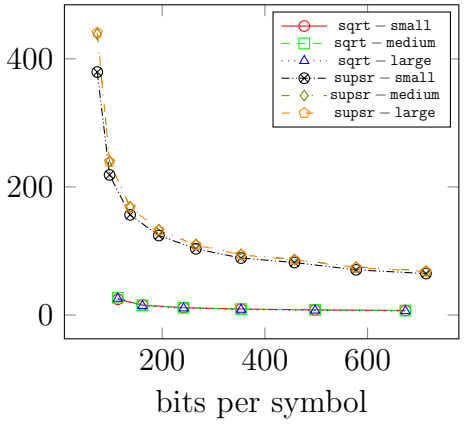
4.4 Different Parameter Values for Exact Range Mode Queries

Previously, we set $s = \sqrt{n}$ for `supsr` and `sqrt`, and then set $s = \sqrt{nw}$ for $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve linear space. Since different values of s yield different time-space tradeoffs, we conduct a series of experiments to compare these data structures more thoroughly. First, we compare the time-space tradeoffs between `supsr` and `sqrt`. The experimental result is showed in Figure 4.1, in which a subfigure is used for each dataset. To draw each subfigure, we construct `supsr` (and similarly `sqrt`) over each dataset for different values of s. The initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage of the data structure exceeds 640 bits per symbol. Each point in the figure represents a tradeoff achieved between the space cost and the average query time of a category of queries. We then connect the points for the same category (small, medium or large) of queries into a polyline. Hence, over each dataset, we show how the average query time changes when more space is used for either data structure using three plotted polylines, one for each category of queries. In Figure 4.1, our experimental study shows that `sqrt` use less query time than `supsr` when these data structures use the same space among all datasets. Therefore, `sqrt` outperforms `supsr` over all datasets.

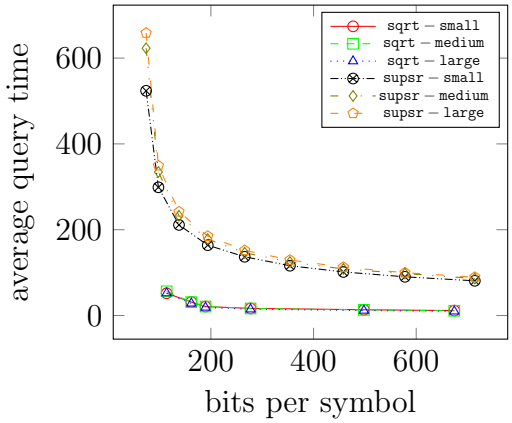
We then compare the time-space tradeoffs that can be achieved by $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ with different parameters. Figure 4.2 shows our experimental results between $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$. To draw each subfigure in Figure 4.2, we construct $\texttt{subsr}_2^{\texttt{p}}$ (and similarly $\texttt{subsr}_2^{\texttt{c}}$) over the corresponding dataset for different values of s. The initial value of s is $0.5\sqrt{nw}$, and each time we increase s by $0.5\sqrt{nw}$ until the space usage of the data structure exceeds 640 bits per symbol. In Figure 4.2 (a), for the same category of queries, the polyline plotted for $\texttt{subsr}_2^{\texttt{p}}$ is always above that for $\texttt{subsr}_2^{\texttt{c}}$. This suggests that, with the same space cost, $\texttt{subsr}_2^{\texttt{c}}$ uses less time to answer a query on average. Hence, $\texttt{subsr}_2^{\texttt{c}}$ outperforms $\texttt{subsr}_2^{\texttt{p}}$ over `reviews`. It is however the opposite for `IPs`, and there is no discernible differences between the performance of these two data structures over the three other datasets.



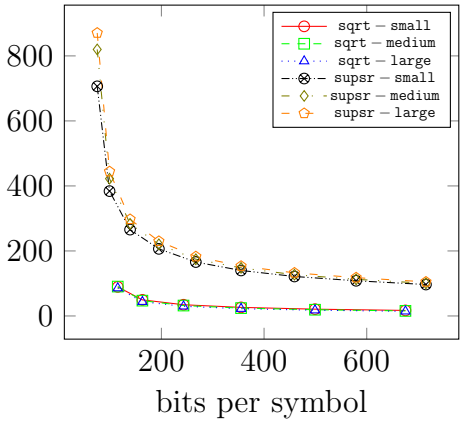
(a) `reviews`



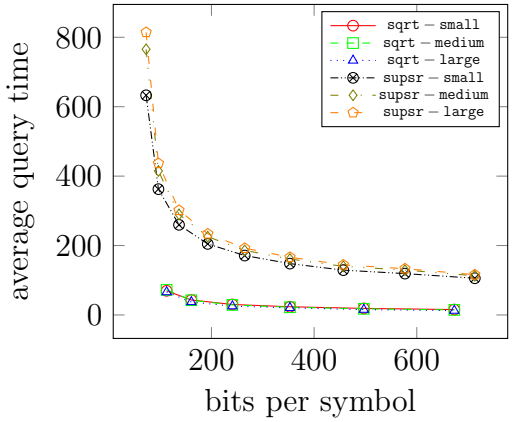
(b) `IPs`



(c) `words`

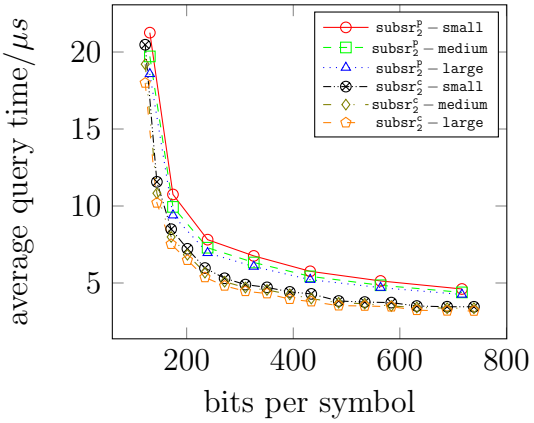


(d) `library`

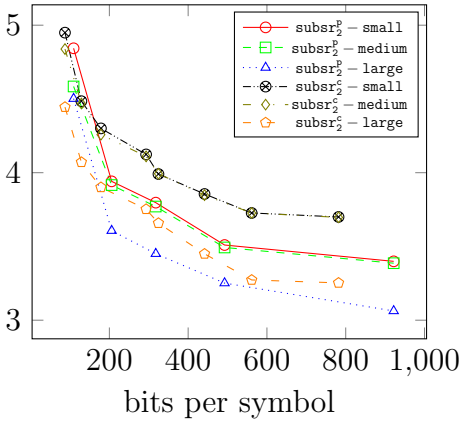


(e) `tickets`

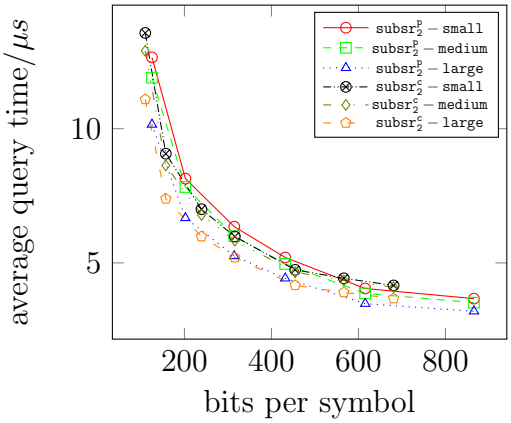
Figure 4.1: Different time-space tradeoffs achieved by `supsr` and `sqrt`.



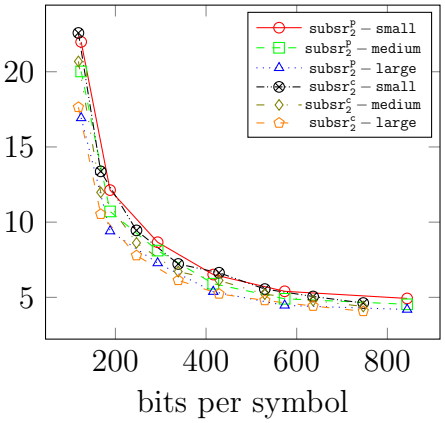
(a) `reviews`



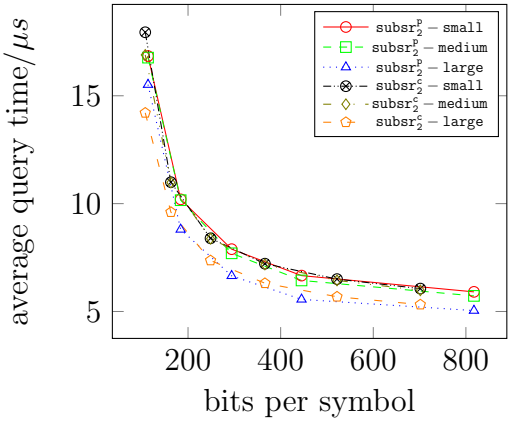
(b) `IPs`



(c) `words`



(d) `library`



(e) `tickets`

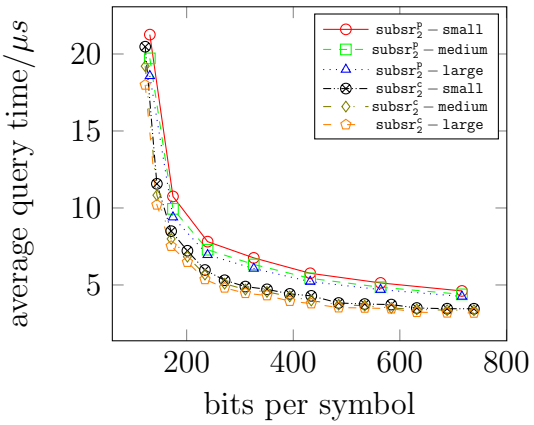
Figure 4.2: Different time-space tradeoffs achieved by $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$.

To discuss why $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ compare differently for different datasets, observe that whether to use plain or compressed bit vectors to encode S' in $\texttt{subsr}_2$ affects the structures built over B_1 only. Furthermore, when s increases, the block size decreases, and more adjacent entries of S' tend to store the same values, making S' more compressible. The dataset `reviews` has the largest entropy, which means the table S' constructed over it is less compressible than that over any other dataset for small s, so the increase of s makes it more compressible rapidly. All arrays entries of `reviews` are also stored in B_1 for all the values of s that we have used, making the compression more sensitive to the choice of the value of s. Hence, for `reviews`, the increase of s improves the compression ratio of $\texttt{subsr}_2^{\texttt{c}}$ at a faster rate than what it does for any other dataset. When the same storage cost is allowed, the parameter s for $\texttt{subsr}_2^{\texttt{c}}$ can be much larger than that for $\texttt{subsr}_2^{\texttt{p}}$, as shown in Figure 4.2 (a). This means S' stores much more precomputed information for $\texttt{subsr}_2^{\texttt{c}}$, speeding up the queries despite the increased operation time for `rank` or `select` over compressed bit vectors. The other datasets perform differently when s changes, due to their smaller entropy which also affects the number of array entries distributed into B_2.

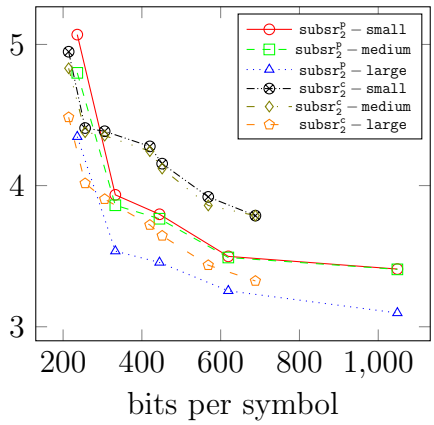
In the extreme case of `IPs`, $\texttt{subsr}_2^{\texttt{p}}$ performs better, while for the rest, the choice of whether to use compression does not make a significant or consistent difference to justify using one approach over the other. Since it takes less time to construct plain bit vectors, we say that $\texttt{subsr}_2^{\texttt{p}}$ is also a better solution for `words`, `library` and `tickets`.

We further conducted similar experiments to compare $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$ and arrive at the same conclusion: $\texttt{subsr}_1^{\texttt{c}}$ achieves better time/space tradeoffs for `reviews`, while $\texttt{subsr}_1^{\texttt{p}}$ works better for other datasets. See Figure 4.3 for details. Hence, in the rest of this thesis, when the context is clear, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ respectively represent $\texttt{subsr}_1^{\texttt{c}}$ and $\texttt{subsr}_2^{\texttt{c}}$ for `reviews`, while they represent $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{p}}$ for all other datasets.

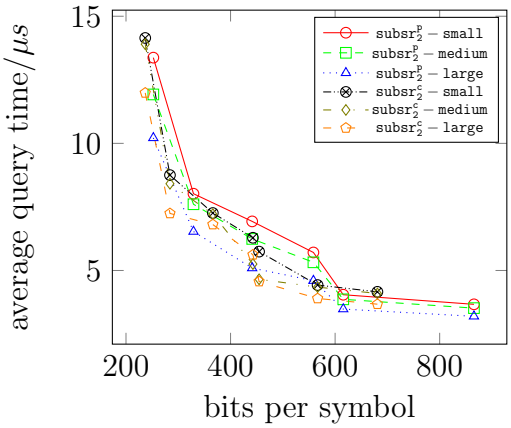
After deciding on the bit vector implementations, we conduct experiments to compare `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$. We use the same parameters for $\texttt{subsr}_1$ and $\texttt{subsr}_2$, while for `sqrt`, the initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage exceeds 640 bits per symbol. Figure 4.4 shows the results, in which, when s increases, all data structures have faster query times but use



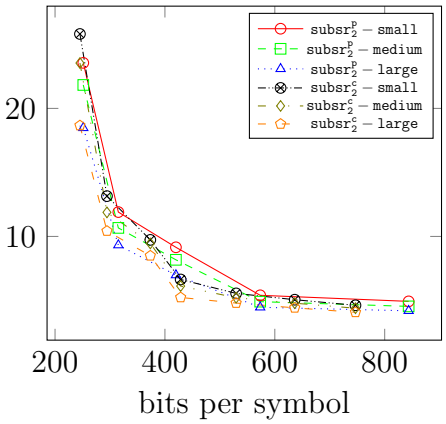
(a) `reviews`



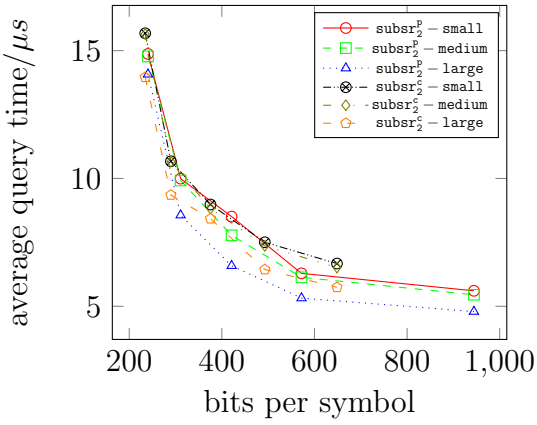
(b) `IPs`



(c) `words`

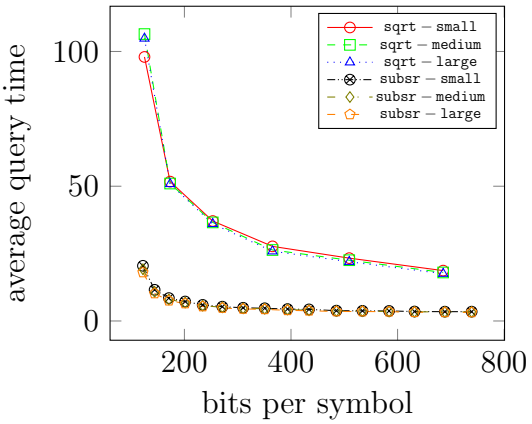


(d) `library`

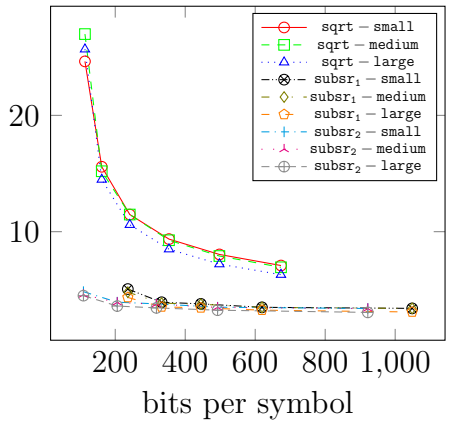


(e) `tickets`

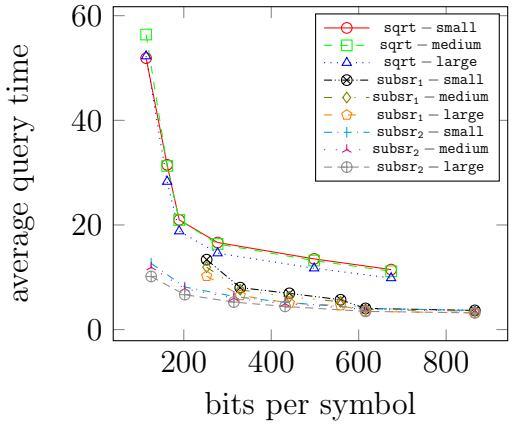
Figure 4.3: Different time-space tradeoffs achieved by $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$



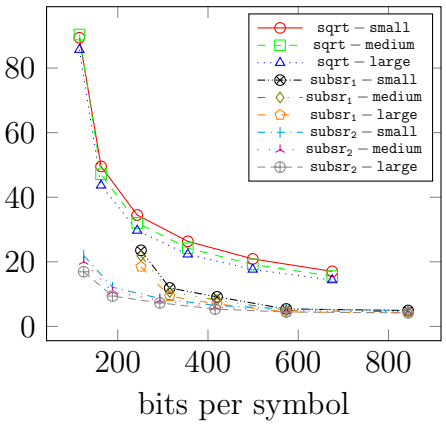
(a) `reviews`



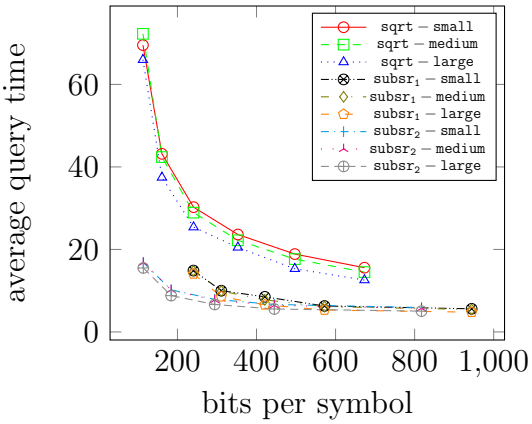
(b) `IPs`



(c) `words`



(d) `library`



(e) `tickets`

Figure 4.4: Different time-space tradeoffs achieved by `sqrt`, `subsr`$_1$ and `subsr`$_2$.

more space. Note that for the `reviews` dataset, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have the same performance because no structures are used to map query ranges to subranges in B_1 and B_2. Hence Figure 4.4 (a) only compares `sqrt` and `subsr` (which represents either `subsr1` or `subsr2`). Our results show that $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have much better query performance than `sqrt` when the data structures have the same storage costs. This is because the succinct data structures used in $\texttt{subsr}_1$ and $\texttt{subsr}_2$ allow us to store more precomputed information to speed up query; at the same time, since only a few operations are performed over these succinct structures, the time spent on operating over them is still dominated by the other steps in the algorithm. Hence the slower operation time over succinct data structures in practice matters little in this case. Between $\texttt{subsr}_1$ and $\texttt{subsr}_2$, for `IPs`, `words`, `library` and `tickets`, our results show that $\texttt{subsr}_2$ achieves better time-space tradeoffs than $\texttt{subsr}_1$ does. The difference is significant for smaller values of s, but as the value of s grows much larger, the plotted lines start to converge. This is because the space savings by replacing four integer arrays of length n by a bit vector of length n becomes insignificant when it is dominated by the space cost of S' for large s. Nevertheless, when we require a reasonable space costs for data structures in practice (e.g., when the space of the data structures must be at most several times more than that of the input; recall that the input array is encoded using 32 bits per symbol), $\texttt{subsr}_2$ still improves $\texttt{subsr}_1$ significantly. Therefore, we conclude that $\texttt{subsr}_2$ preforms best among all data structure solutions.

4.5 Performance of Data Structures for Approximate Range Mode

We now perform experimental studies on approximate range mode by choosing $\epsilon = 1/2$. Tables 4.5 and 4.6 present the query time, space usage and construction time of approximate range mode data structures, while Tables 4.7 and 4.8 show the average and maximum approximation ratios of the answers; these ratios are computed as the frequency of the reported approximate mode in the query range divided by the frequency of the actual mode in the range. When calculating space usage, we do not include the cost of the input array A, since all these data structures can compute the indexes of approximate range modes without accessing A.

From Table 4.7, we can see that the average approximation ratios range between

Table 4.5: Average time to answer an approximate query for $\epsilon = 1/2$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

	Query	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
reviews	small	0.098	0.861	0.869	1.016	0.191	1.122	2.970
	medium	0.095	0.714	0.598	0.610	0.135	1.009	3.178
	large	0.089	0.556	0.440	0.453	0.116	0.864	3.703
IPs	small	0.110	1.561	0.545	0.796	0.138	1.003	4.003
	medium	0.113	1.343	0.358	0.430	0.105	0.696	3.198
	large	0.120	1.120	0.285	0.304	0.091	0.581	3.030
words	small	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	medium	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	large	0.105	0.546	0.281	0.309	0.095	0.595	2.547
library	small	0.099	0.760	1.017	1.164	0.200	1.230	3.508
	medium	0.099	0.581	0.603	0.629	0.144	1.152	3.809
	large	0.106	0.434	0.360	0.370	0.112	0.766	3.023
tickets	small	0.112	1.072	0.773	1.108	0.172	1.281	3.861
	medium	0.109	0.817	0.460	0.585	0.129	0.997	3.371
	large	0.119	0.580	0.300	0.327	0.105	0.634	2.669

Table 4.6: Space (bits per symbol) and construction time (minutes) when $\epsilon = 1/2$.

	Dataset	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
space	reviews	680.0	100.6	225.4	204.7	291.2	56.9	11.4
	IPs	1038.6	1051.5	327.9	311.3	291.5	82.9	30.0
	words	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	library	769.6	37.6	231.6	210.8	291.3	65.6	13.9
	tickets	896.6	115.8	248.2	228.1	291.5	74.2	24.2
construct time	reviews	0.084	0.142	0.655	0.668	0.050	0.082	0.085
	IPs	0.075	0.172	0.564	0.568	0.031	0.063	0.067
	words	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	library	0.084	0.136	0.663	0.673	0.042	0.065	0.067
	tickets	0.081	0.122	0.648	0.649	0.027	0.068	0.070

Table 4.7: Average approximation ratio when answering queries for $\epsilon = 1/2$.

	Query	simple	pst	sample	tri/succ
reviews	small	1.01644	1.01320	1.01263	1.00493
	medium	1.01026	1.00772	1.00913	1.00324
	large	1.01378	1.00792	1.02630	1.00389
IPs	small	1.00773	1.00453	1.00639	1.00248
	medium	1.00155	1.00096	1.00121	1.00047
	large	1.00070	1.00029	1.00047	1.00016
words	small	1.01437	1.01025	1.01088	1.00394
	medium	1.01250	1.00680	1.00984	1.00335
	large	1.00453	1.00333	1.00818	1.00190
library	small	1.00916	1.00922	1.00679	1.00255
	medium	1.00337	1.00274	1.00279	1.00113
	large	1.00079	1.00066	1.00078	1.00029
tickets	small	1.00123	1.00179	1.00106	1.00038
	medium	1.00020	1.00026	1.00017	1.00006
	large	1.00003	1.00003	1.00003	1.00001

Table 4.8: Maximum approximation ratio when answering queries for $\epsilon = 1/2$.

	Query	simple	pst	sample	tri/succ
reviews	small	1.49231	1.5	1.47183	1.47826
	medium	1.49231	1.5	1.47183	1.47826
	large	1.49231	1.5	1.47518	1.42105
IPs	small	1.49831	1.5	1.48454	1.41667
	medium	1.49977	1.5	1.47468	1.36749
	large	1.49494	1.5	1.47589	1.37500
words	small	1.49658	1.5	1.48227	1.36364
	medium	1.49924	1.5	1.48837	1.36364
	large	1.49790	1.5	1.48879	1.33333
library	small	1.48276	1.5	1.43396	1.36364
	medium	1.48276	1.5	1.43396	1.33333
	large	1.46154	1.5	1.42857	1.33333
tickets	small	1.46154	1.5	1.37931	1.36364
	medium	1.46154	1.5	1.37931	1.33333
	large	1.41667	1.5	1.37500	1.33333

1.00001 and 1.02630 across all data structures, datasets and query categories, and in most cases, the ratio is below 1.01. This means the average quality of the answers to the queries is excellent. The maximum approximation ratios are closer to 1.5. Since these structures have slower query support and higher space usage for smaller ϵ, this means setting $\epsilon = 1/2$ is attractive to applications for which a high average approximation ratio is sufficient. Another phenomenon is that larger queries tend to be faster. This is because all query algorithms are essentially based on binary searches in lists of possible candidates, and in each list, the farther it is away from the list head, the larger the gaps between the indexes (in A) of two consecutive candidates are, benefiting larger query ranges.

We also observe that the space cost of `pst` can vary greatly between different datasets, with the space cost of `IPs` being about 28 times of that of `library`. Recall that in this solution, we view n different tables as versions of the same table T to store them in a persistent search tree, and each tree node corresponds to an update to the table (the initial version of the table is not stored explicitly since $T[i] = i$ for all $i \in [1, n]$). Thus, we recorded the number of updates to T for each dataset, and it is $1,380,391$ for `reviews`, $10,773,911$ for `IPs`, $1,232,046$ for `words`, $485,498$ for `library` and $1,386,886$ for `tickets`. The difference in the numbers of updates is consistent with the difference in space costs. To see why there is such a difference in updates, recall that an update to T happens when the frequency of a candidate within a certain range $A[i, j]$ drops below a threshold when we increment i. This happens more often when the entropy of the dataset is lower or when the locality of reference of the dataset is higher, since a lower entropy or higher locality of references means we are more likely to decrease the frequency of this candidate by one each time we increment i. Indeed, `IPs` has the lowest entropy by Table 4.2, and since the same subset of IPs occur frequently in a DDoS attack event, it has high locality of reference. This explains the high space cost of `pst` over IPs. On the other hand, `library` has the second highest entropy, and compared to `reviews` which has even higher entropy, due to the limited number of copies that a library has for each different book, the borrowing records tend to be less affected by trends such as "best sellers of the month" than the Amazon reviews are. This explains the low space usage for `library`. Note that the space cost of `sample` also fluctuates among different datasets

for similar reasons, but due to the sampling technique used, the difference in space costs between datasets is small.

With these discussed, we are now ready to compare the performance of different approximate range mode structures. Among them, `simple` has the fastest query time due to its simplicity, but its space cost is high. Among more sophisticated solutions which use $O(n/\epsilon)$ words but are not succinct, `tri` stands out as its query time is comparable to that of `simple` (it even beats `simple` in some cases), but its space cost is only $28.1\% \sim 42.8\%$ of that of `simple`. Compared to `pst` and `sample`, it has the smallest worst-case space cost. This is because it is not based on persistence and is thus not sensitive to entropy or locality of reference. The average approximation ratio of the answers computed by `tri`/`succ` are also much smaller than that of other solutions, due to the finer-grained approximation based on the trichotomy it uses. On the other hand, for most datasets, `pst` and `sample` also provide useful tradeoffs with lower space usage but slower query time, with `pst` especially attractive for datasets of high entropy but low locality of reference. For example, the space cost of `library` with `pst` is only slightly more than the cost of encoding the input array as 32-bit integers. Finally, both $\texttt{succ}^{\texttt{p}}$ and $\texttt{succ}^{\texttt{c}}$ provide compact solutions. Between them, $\texttt{succ}^{\texttt{p}}$ uses $0.89n \sim 1.30n$ words, with query time only slightly slower than `pst` and `sample` in most cases, while $\texttt{succ}^{\texttt{c}}$ is highly compact, with space costs $35.6\% \sim 93.8\%$ of the array of 32-bit integers (in most cases, the space cost is closer to the lower end), while the query time is $2.6 \sim 5.2$ times of the query time of $\texttt{succ}^{\texttt{p}}$.

4.6 Different Values of ϵ Among All Approximate Range Mode Data Structures

We further conduct experiments with more values of ϵ by setting ϵ to 1/4, 1/8 and 1/16. For each different ϵ and each solution, we compute the average and maximum approximation ratios achieved across all datasets, and they are recorded in Table 4.9. From it we can tell that the average ratios decrease when ϵ decreases, though it is already close to 1 for $\epsilon = 1/2$. The maximum ratios are close to $1 + \epsilon$; `pst` is the solution whose maximum approximation ratios are always $1 + \epsilon$. This is due to the integer sequences that they defined as lower and upper bounds on the frequency of possible candidates.

Table 4.9: Average and Maximum approximation ratios for different values of ϵ.

	ε	simple	pst	sample	tri/succ
Average	1/2	1.00644	1.00464	1.00644	1.00192
	1/4	1.00218	1.00164	1.00188	1.00085
	1/8	1.00075	1.00068	1.00055	1.00019
	1/16	1.00020	1.00017	1.00016	1.00006
Maximum	1/2	1.49977	1.5	1.48879	1.47826
	1/4	1.24952	1.25	1.24701	1.25
	1/8	1.12474	1.125	1.12148	1.11765
	1/16	1.06240	1.0625	1.06107	1.05882

For each dataset, we also measure the query time, space cost, and construction time of each solution for different values of ϵ, and these results are recorded in Table 4.10 - Table 4.24. These tables show that the query times of these data structures increase slowly as ϵ decreases, and this indeed fits the growth of the function of $\lg \frac{1}{\epsilon} + \lg \lg n$. The space costs, however, grows at a much faster rate, as they are more or less proportional to $1/\epsilon$.

For different values of ϵ, how the performance of different solutions compares to each other is similar to the case where $\epsilon = 1/2$ which we discussed before. The main notable difference is that, due to persistence or compression, the ratio at which the space costs of `pst`, `sample`, and `succ`$^{\mathtt{c}}$ grow is slower than other data structures. Therefore, when other data structures become less attractive due to high space usage caused by smaller value of ϵ, these solutions may still remain attractive, but this may depend on the dataset (especially for `pst`).

Table 4.10: Average time to answer an approximate query over the `reviews` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
1/2	small	0.098	0.861	0.869	1.016	0.191	1.122	2.970
	medium	0.095	0.714	0.598	0.610	0.135	1.009	3.178
	large	0.089	0.556	0.440	0.453	0.116	0.864	3.703
1/4	small	0.125	0.986	1.054	1.217	0.279	1.582	3.646
	medium	0.119	0.785	0.918	0.927	0.199	1.403	3.748
	large	0.119	0.614	0.537	0.543	0.163	1.173	4.158
1/8	small	0.150	1.290	1.204	1.334	0.385	2.221	4.483
	medium	0.142	1.068	1.056	1.086	0.298	2.091	4.563
	large	0.137	0.755	0.717	0.747	0.222	1.635	4.691
1/16	small	0.181	1.343	1.343	1.361	0.476	2.850	5.070
	medium	0.177	1.042	1.308	1.332	0.420	2.721	5.263
	large	0.161	0.716	1.002	1.052	0.308	2.234	5.399

Table 4.11: Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `reviews` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
space	1/2	680.0	100.6	225.4	204.7	291.2	56.9	11.4
	1/4	1220.3	163.0	373.9	332.5	547.3	98.0	16.2
	1/8	2307.3	302.7	679.0	592.4	1062.5	175.9	25.8
	1/16	4469.6	345.7	1276.2	1121.8	2068.9	319.8	43.1
construc -tion	1/2	0.084	0.142	0.655	0.668	0.050	0.082	0.085
	1/4	0.148	0.277	1.265	1.282	0.088	0.148	0.150
	1/8	0.276	0.503	2.456	2.516	0.160	0.240	0.243
	1/16	0.542	0.968	4.953	5.000	0.300	0.481	0.488

Table 4.12: Average approximate ratio and maximum approximate ratio when answering approximate queries over the `reviews` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

Ratio	ε	Query	`simple`	`pst`	`sample`	`tri/succ`
Average Ratio	1/2	`small`	1.01644	1.01320	1.01263	1.00493
		`medium`	1.01026	1.00772	1.00913	1.00324
		`large`	1.01378	1.00792	1.02630	1.00389
	1/4	`small`	1.00592	1.00481	1.00429	1.00218
		`medium`	1.00324	1.00279	1.00359	1.00118
		`large`	1.00416	1.00297	1.00412	1.00124
	1/8	`small`	1.00170	1.00195	1.00120	1.00048
		`medium`	1.00116	1.00124	1.00114	1.00041
		`large`	1.00095	1.00110	1.00131	1.00035
	1/16	`small`	1.00051	1.00053	1.00033	1.00015
		`medium`	1.00043	1.00038	1.00040	1.00016
		`large`	1.00038	1.00034	1.00035	1.00010
Maximum Ratio	1/2	`small`	1.49231	1.5	1.47183	1.47826
		`medium`	1.49231	1.5	1.47183	1.47826
		`large`	1.49231	1.5	1.47518	1.42105
	1/4	`small`	1.24396	1.25	1.23102	1.25
		`medium`	1.24710	1.25	1.24392	1.25
		`large`	1.24471	1.25	1.23985	1.25
	1/8	`small`	1.12205	1.125	1.12127	1.11538
		`medium`	1.12461	1.125	1.11794	1.11550
		`large`	1.12261	1.125	1.11840	1.11212
	1/16	`small`	1.06175	1.0625	1.05667	1.05882
		`medium`	1.06231	1.0625	1.05955	1.05763
		`large`	1.06186	1.0625	1.06006	1.05708

Table 4.13: Average time to answer an approximate query over the `IPs` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
1/2	small	0.110	1.561	0.545	0.796	0.138	1.003	4.003
	medium	0.113	1.343	0.358	0.430	0.105	0.696	3.198
	large	0.120	1.120	0.285	0.304	0.091	0.581	3.030
1/4	small	0.138	1.845	0.852	1.258	0.203	1.506	5.153
	medium	0.134	1.550	0.450	0.546	0.154	1.073	4.347
	large	0.135	1.246	0.319	0.333	0.137	0.834	3.927
1/8	small	0.160	2.355	1.473	2.032	0.286	2.305	6.677
	medium	0.159	2.315	0.682	0.861	0.219	1.694	5.820
	large	0.154	1.961	0.366	0.393	0.177	1.261	5.194
1/16	small	0.187	2.628	1.935	2.830	0.394	3.393	8.442
	medium	0.190	2.461	1.200	1.537	0.285	2.578	7.550
	large	0.185	1.850	0.490	0.558	0.225	1.891	6.550

Table 4.14: Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `IPs` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
space	1/2	1038.6	1051.5	327.9	311.3	291.5	82.9	30.0
	1/4	1873.1	2044.5	535.6	501.4	548.2	146.9	53.4
	1/8	3535.7	6206.2	946.4	873.0	1065.4	268.9	97.6
	1/16	6854.3	6915.0	1769.3	1605.9	2079.4	501.1	180.2
construc -tion	1/2	0.075	0.172	0.564	0.568	0.031	0.063	0.067
	1/4	0.137	0.314	0.965	0.978	0.054	0.141	0.147
	1/8	0.260	0.907	2.117	2.139	0.098	0.216	0.232
	1/16	0.519	1.242	4.319	4.337	0.183	0.356	0.376

Table 4.15: Average approximate ratio and maximum approximate ratio when answering approximate queries over the `IPs` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

Ratio	ε	Query	`simple`	`pst`	`sample`	`tri/succ`
Average Ratio	1/2	`small`	1.00773	1.00453	1.00639	1.00248
		`medium`	1.00155	1.00096	1.00121	1.00047
		`large`	1.00070	1.00029	1.00047	1.00016
	1/4	`small`	1.00290	1.00174	1.00236	1.00088
		`medium`	1.00059	1.00034	1.00046	1.00018
		`large`	1.00021	1.00011	1.00020	1.00006
	1/8	`small`	1.00094	1.00070	1.00074	1.00027
		`medium`	1.00019	1.00015	1.00014	1.00006
		`large`	1.00005	1.00005	1.00007	1.00002
	1/16	`small`	1.00027	1.00018	1.00021	1.00008
		`medium`	1.00006	1.00004	1.00004	1.00002
		`large`	1.00001	1.00001	1.00002	1.00001
Maximum Ratio	1/2	`small`	1.49831	1.5	1.48454	1.41667
		`medium`	1.49977	1.5	1.47468	1.36749
		`large`	1.49494	1.5	1.47589	1.37500
	1/4	`small`	1.24952	1.25	1.24701	1.25
		`medium`	1.24773	1.25	1.24402	1.25
		`large`	1.24851	1.25	1.24203	1.25
	1/8	`small`	1.12461	1.125	1.12148	1.11765
		`medium`	1.12461	1.125	1.12108	1.11429
		`large`	1.12423	1.125	1.12033	1.09524
	1/16	`small`	1.06231	1.0625	1.06027	1.05882
		`medium`	1.06231	1.0625	1.06034	1.05634
		`large`	1.06231	1.0625	1.05699	1.04545

Table 4.16: Average time to answer an approximate query over the `words` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
1/2	small	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	medium	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	large	0.105	0.546	0.281	0.309	0.095	0.595	2.547
1/4	small	0.122	1.178	1.069	1.486	0.248	1.568	4.537
	medium	0.119	0.924	0.738	0.841	0.184	1.371	4.597
	large	0.119	0.639	0.357	0.386	0.142	0.906	3.512
1/8	small	0.148	1.637	1.277	1.809	0.349	2.231	5.778
	medium	0.138	1.586	1.253	1.280	0.269	2.128	5.940
	large	0.133	1.090	0.543	0.563	0.194	1.402	4.598
1/16	small	0.178	1.704	1.439	2.061	0.468	2.993	6.849
	medium	0.170	1.479	1.459	1.690	0.383	3.104	7.230
	large	0.161	0.935	1.025	1.077	0.261	2.095	5.894

Table 4.17: Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `words` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
space	1/2	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	1/4	1418.9	264.7	393.0	352.4	547.8	117.4	35.9
	1/8	2677.9	657.9	704.3	619.5	1063.9	212.2	62.6
	1/16	5185.2	753.6	1337.7	1157.9	2074.5	389.5	111.7
construc -tion	1/2	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	1/4	0.091	0.166	0.744	0.746	0.032	0.066	0.077
	1/8	0.171	0.318	1.610	1.636	0.058	0.148	0.150
	1/16	0.335	0.554	3.224	3.247	0.108	0.229	0.238

Table 4.18: Average approximate ratio and maximum approximate ratio when answering approximate queries over the `words` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

Ratio	ε	Query	`simple`	`pst`	`sample`	`tri/succ`
Average Ratio	1/2	`small`	1.01437	1.01025	1.01088	1.00394
		`medium`	1.01250	1.00680	1.00984	1.00335
		`large`	1.00453	1.00333	1.00818	1.00190
	1/4	`small`	1.00474	1.00359	1.00351	1.00163
		`medium`	1.00406	1.00217	1.00302	1.00120
		`large`	1.00155	1.00090	1.00261	1.00163
	1/8	`small`	1.00148	1.00138	1.00103	1.00039
		`medium`	1.00137	1.00104	1.00088	1.00033
		`large`	1.00171	1.00073	1.00061	1.00011
	1/16	`small`	1.00041	1.00035	1.00026	1.00010
		`medium`	1.00035	1.00021	1.00025	1.00009
		`large`	1.00016	1.00009	1.00021	1.00003
Maximum Ratio	1/2	`small`	1.49658	1.5	1.48227	1.36364
		`medium`	1.49924	1.5	1.48837	1.36364
		`large`	1.49790	1.5	1.48879	1.33333
	1/4	`small`	1.24773	1.25	1.24503	1.25
		`medium`	1.24952	1.25	1.24559	1.25
		`large`	1.24950	1.25	1.24637	1.25
	1/8	`small`	1.12461	1.125	1.11848	1.09524
		`medium`	1.12461	1.125	1.12148	1.09524
		`large`	1.12474	1.125	1.11997	1.09524
	1/16	`small`	1.06231	1.0625	1.05828	1.05
		`medium`	1.06240	1.0625	1.06057	1.05128
		`large`	1.06231	1.0625	1.06107	1.05

Table 4.19: Average time to answer an approximate query over the `library` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
1/2	small	0.099	0.760	1.017	1.164	0.200	1.230	3.508
	medium	0.099	0.581	0.603	0.629	0.144	1.152	3.809
	large	0.106	0.434	0.360	0.370	0.112	0.766	3.023
1/4	small	0.121	0.937	1.110	1.409	0.303	1.746	4.305
	medium	0.121	0.731	0.902	0.912	0.223	1.688	4.667
	large	0.126	0.516	0.474	0.479	0.164	1.197	4.015
1/8	small	0.154	1.448	1.306	1.668	0.387	2.458	5.369
	medium	0.146	1.376	1.330	1.357	0.320	2.546	5.936
	large	0.145	0.967	0.752	0.791	0.229	1.842	5.195
1/16	small	0.178	1.440	1.651	1.799	0.462	3.114	6.198
	medium	0.174	1.260	1.717	1.721	0.427	3.573	7.258
	large	0.172	0.864	1.350	1.402	0.308	2.768	6.764

Table 4.20: Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `library` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
space	1/2	769.6	37.6	231.6	210.8	291.3	65.6	13.9
	1/4	1385.8	66.2	379.6	338.3	547.6	115.5	23.3
	1/8	2610.7	181.6	684.4	598.1	1063.4	209.0	41.5
	1/16	5057.1	194.4	1301.5	1123.8	2072.3	383.8	76.3
construc -tion	1/2	0.084	0.136	0.663	0.673	0.042	0.065	0.067
	1/4	0.146	0.225	1.259	1.260	0.069	0.128	0.133
	1/8	0.274	0.469	2.477	2.525	0.122	0.245	0.247
	1/16	0.547	0.887	5.118	5.170	0.225	0.410	0.418

Table 4.21: Average approximate ratio and maximum approximate ratio when answering approximate queries over the `library` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

Ratio	ε	Query	`simple`	`pst`	`sample`	`tri/succ`
Average Ratio	1/2	`small`	1.00916	1.00922	1.00679	1.00255
		`medium`	1.00337	1.00274	1.00279	1.00113
		`large`	1.00079	1.00066	1.00078	1.00029
	1/4	`small`	1.00315	1.00316	1.00224	1.00161
		`medium`	1.00131	1.00112	1.00106	1.00056
		`large`	1.00034	1.00032	1.00033	1.00015
	1/8	`small`	1.00092	1.00104	1.00060	1.00022
		`medium`	1.00048	1.00045	1.00037	1.00015
		`large`	1.00015	1.00016	1.00014	1.00006
	1/16	`small`	1.00023	1.00027	1.00014	1.00005
		`medium`	1.00016	1.00013	1.00012	1.00005
		`large`	1.00006	1.00006	1.00006	1.00002
Maximum Ratio	1/2	`small`	1.48276	1.5	1.43396	1.36364
		`medium`	1.48276	1.5	1.43396	1.33333
		`large`	1.46154	1.5	1.42857	1.33333
	1/4	`small`	1.24138	1.25	1.225	1.25
		`medium`	1.24138	1.25	1.225	1.25
		`large`	1.24138	1.25	1.225	1.25
	1/8	`small`	1.12121	1.125	1.10909	1.11429
		`medium`	1.12121	1.125	1.10870	1.09524
		`large`	1.12121	1.125	1.10952	1.09524
	1/16	`small`	1.05882	1.0625	1.05028	1.05634
		`medium`	1.06186	1.0625	1.05694	1.05
		`large`	1.06231	1.0625	1.05607	1.05

Table 4.22: Average time to answer an approximate query over the `tickets` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
1/2	small	0.112	1.072	0.773	1.108	0.172	1.281	3.861
	medium	0.109	0.817	0.460	0.585	0.129	0.997	3.371
	large	0.119	0.580	0.300	0.327	0.105	0.634	2.669
1/4	small	0.137	1.311	1.233	1.611	0.258	1.864	4.828
	medium	0.134	0.990	0.634	0.806	0.192	1.539	4.497
	large	0.134	0.683	0.349	0.390	0.152	1.011	3.579
1/8	small	0.165	1.944	1.547	2.194	0.359	2.686	6.307
	medium	0.157	1.894	1.068	1.316	0.276	2.278	5.778
	large	0.155	1.407	0.475	0.530	0.200	1.602	4.673
1/16	small	0.196	2.031	1.857	2.643	0.471	3.600	8.061
	medium	0.188	1.802	1.791	1.980	0.372	3.383	7.716
	large	0.179	1.186	0.784	0.803	0.261	2.401	6.210

Table 4.23: Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `tickets` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample$^{\text{p}}$	sample$^{\text{c}}$	tri	succ$^{\text{p}}$	succ$^{\text{c}}$
space	1/2	896.6	115.8	248.2	228.1	291.5	74.2	24.2
	1/4	1618.2	225.9	402.5	362.1	548.0	130.7	42.6
	1/8	3051.1	766.1	719.8	634.3	1064.9	238.5	78.6
	1/16	5912.5	815.6	1367.3	1184.3	2077.9	441.2	148.7
construc -tion	1/2	0.081	0.122	0.648	0.649	0.027	0.068	0.070
	1/4	0.145	0.238	1.220	1.223	0.049	0.130	0.137
	1/8	0.282	0.487	2.446	2.473	0.090	0.248	0.254
	1/16	0.556	0.883	5.060	5.152	0.170	0.354	0.362

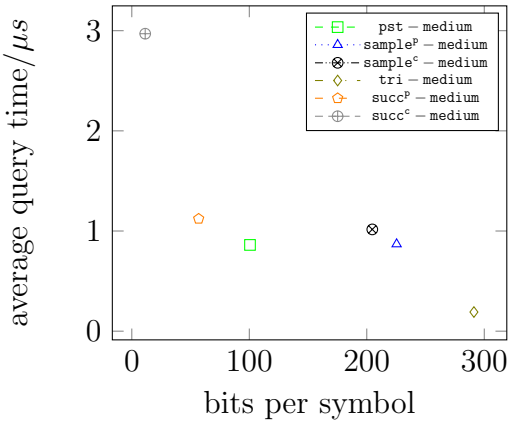
Table 4.24: Average approximate ratio and maximum approximate ratio when answering approximate queries over the `tickets` datasets for $\epsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

Ratio	ε	Query	`simple`	`pst`	`sample`	`tri/succ`
Average Ratio	1/2	`small`	1.00123	1.00179	1.00106	1.00038
		`medium`	1.00020	1.00026	1.00017	1.00006
		`large`	1.00003	1.00003	1.00003	1.00001
	1/4	`small`	1.00046	1.00053	1.00034	1.00030
		`medium`	1.00007	1.00008	1.00006	1.00004
		`large`	1.00007	1.00001	1.00001	1.00001
	1/8	`small`	1.00014	1.00017	1.00010	1.00004
		`medium`	1.00002	1.00003	1.00002	1.00001
		`large`	1.00001	1.00001	1.00001	1.00001
	1/16	`small`	1.00004	1.00005	1.00002	1.00001
		`medium`	1.00001	1.00001	1.00001	1.00001
		`large`	1.00001	1.00001	1.00001	1.00001
Maximum Ratio	1/2	`small`	1.46154	1.5	1.37931	1.36364
		`medium`	1.46154	1.5	1.37931	1.33333
		`large`	1.41667	1.5	1.37500	1.33333
	1/4	`small`	1.22222	1.25	1.21154	1.25
		`medium`	1.23214	1.25	1.2	1.25
		`large`	1.22222	1.25	1.2	1.25
	1/8	`small`	1.11765	1.125	1.10870	1.09524
		`medium`	1.11765	1.125	1.09756	1.09524
		`large`	1.11765	1.125	1.08696	1.08333
	1/16	`small`	1.05882	1.0625	1.05369	1.05
		`medium`	1.05882	1.0625	1.04959	1.05
		`large`	1.05714	1.0625	1.04959	1.05

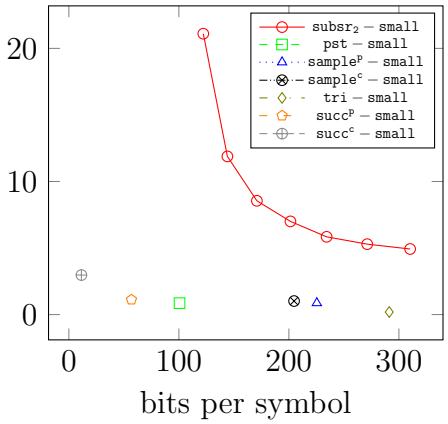
4.7 Comparisons between Approximate Queries Structures and Exact Queries Structures

For $\epsilon = 1/2$, we plotted figures to compare approximate structures to the different tradeoffs achieved by the best exact range mode structure we implemented which is `subsr2`. Due to its high space costs, the figures do not show `simple`, and for `IPs`, `pst` is not shown for the same reason. In these figures, we also omit some tradeoffs with low space cost that can be achieved using `subsr2`, because their query times are so

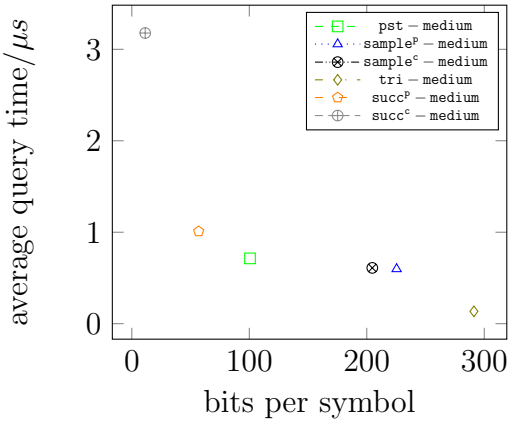
large that, with them, it would not be possible to tell how other tradeoffs compare to each other in the same figure. Furthermore, to better compare the tradeoffs achieved by approximate solutions, for each dataset and each category of queries, we plot a subfigure without `subsr2`, before plotting another one with `subsr2`. From Figure 4.5 to Figure 4.9, we can tell these approximate structures outperform exact structures greatly (except for the high space cost of `pst` over `IPs` or that of `simple`), making them suitable for applications that require high average approximations. They still achieve better time/space tradeoffs over `subsr2` for $\epsilon = 1/4$, but may lose the appeals when we keep decreasing ϵ due to the increase in space costs.



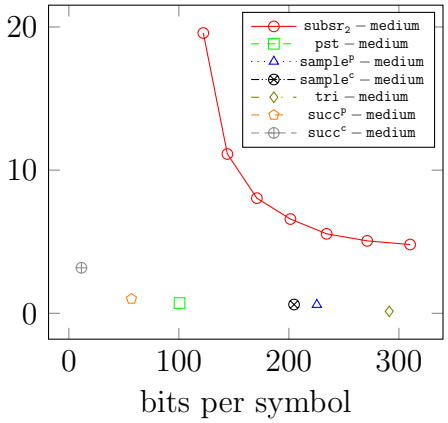
(a) `reviews` – `small` without $\texttt{subsr}_2$



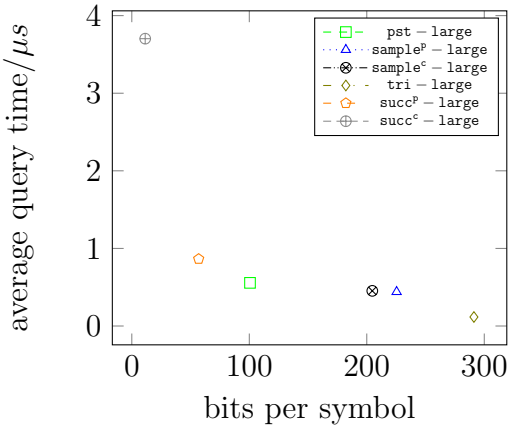
(b) `reviews` – `small` with $\texttt{subsr}_2$



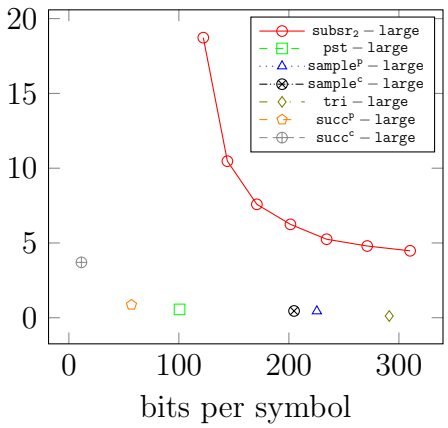
(c) `reviews` – `medium` without $\texttt{subsr}_2$



(d) `reviews` – `medium` with $\texttt{subsr}_2$

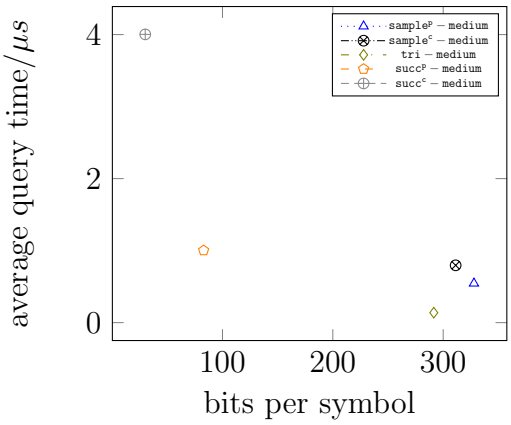


(e) `reviews` – `large` without $\texttt{subsr}_2$

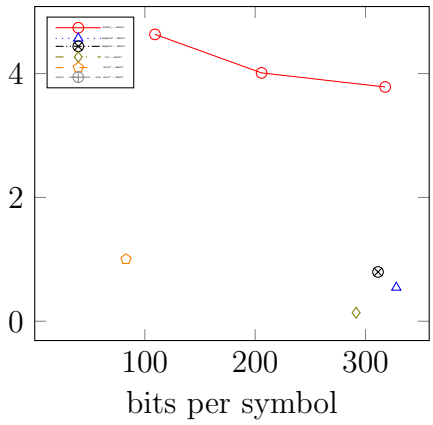


(f) `reviews` – `large` with $\texttt{subsr}_2$

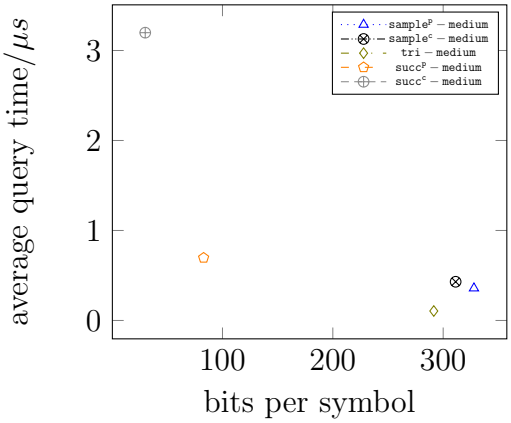
Figure 4.5: Different time-space tradeoffs achieved by $\texttt{subsr}_2$, `pst`, $\texttt{sample}^{\texttt{p}}$, $\texttt{sample}^{\texttt{c}}$, `tri`, $\texttt{succ}^{\texttt{p}}$, and $\texttt{succ}^{\texttt{c}}$ on `reviews`.



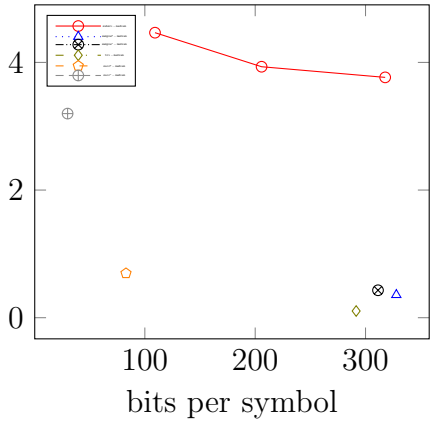
(a) `IPs` – `small` without $\texttt{subsr}_2$



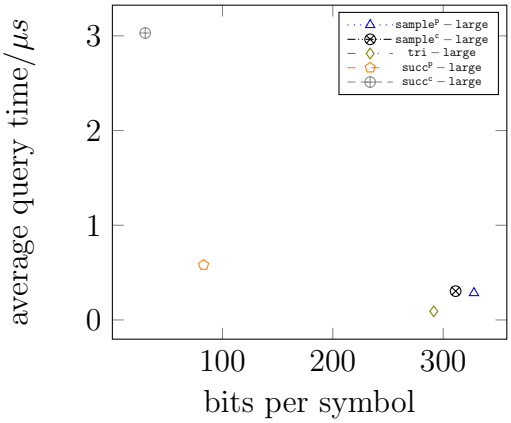
(b) `IPs` – `small` with $\texttt{subsr}_2$



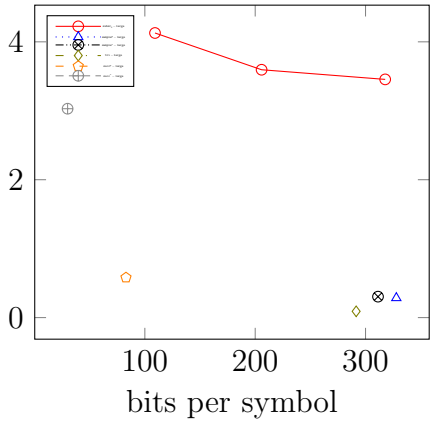
(c) `IPs` – `medium` without $\texttt{subsr}_2$



(d) `IPs` – `medium` with $\texttt{subsr}_2$

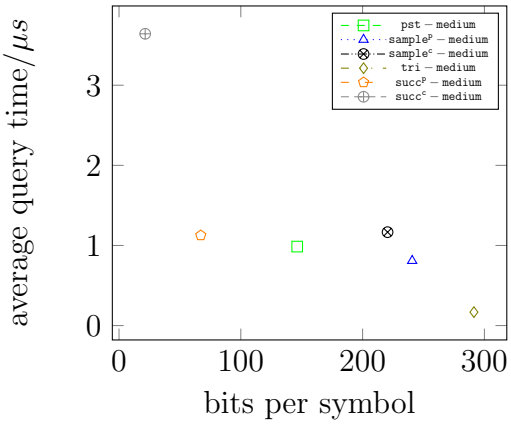


(e) `IPs` – `large` without $\texttt{subsr}_2$

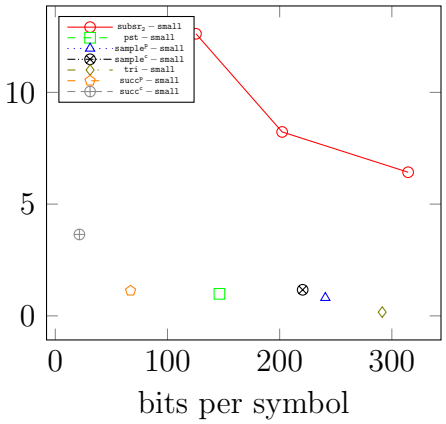


(f) `IPs` – `large` with $\texttt{subsr}_2$

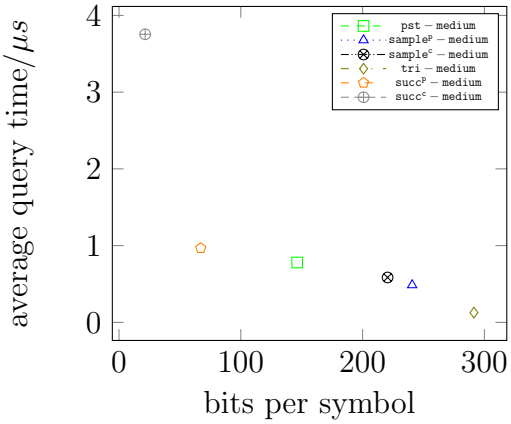
Figure 4.6: Different time-space tradeoffs achieved by $\texttt{subsr}_2$, $\texttt{sample}^{p}$, $\texttt{sample}^{c}$, `tri`, $\texttt{succ}^{p}$, and $\texttt{succ}^{c}$ on `IPs`.



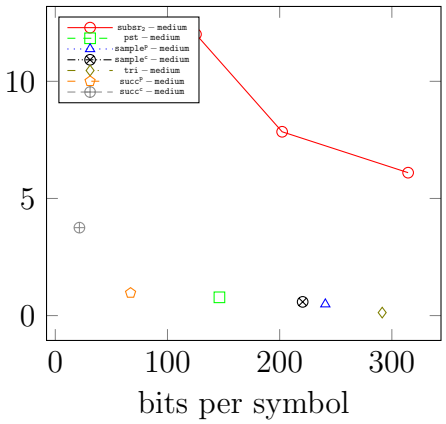
(a) `words` – `small` without $\texttt{subsr}_2$



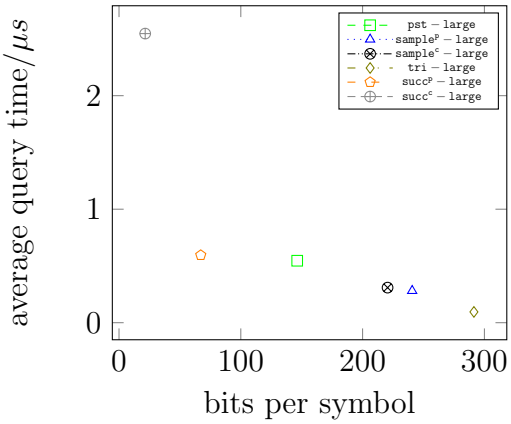
(b) `words` – `small` with $\texttt{subsr}_2$



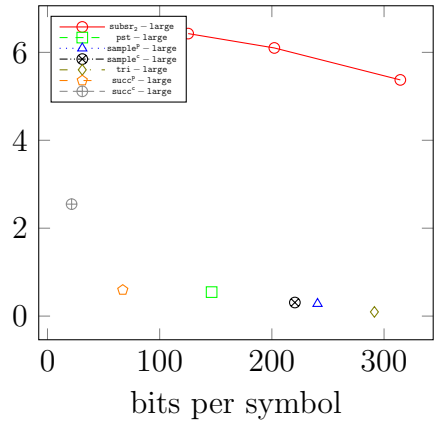
(c) `words` – `medium` without $\texttt{subsr}_2$



(d) `words` – `medium` with $\texttt{subsr}_2$

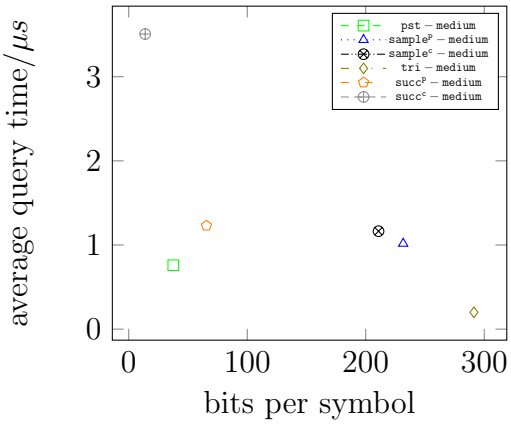


(e) `words` – `large` without $\texttt{subsr}_2$

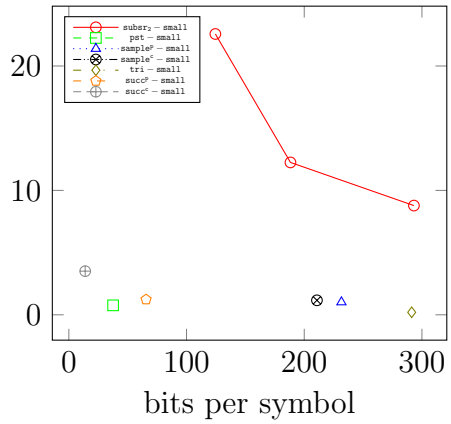


(f) `words` – `large` with $\texttt{subsr}_2$

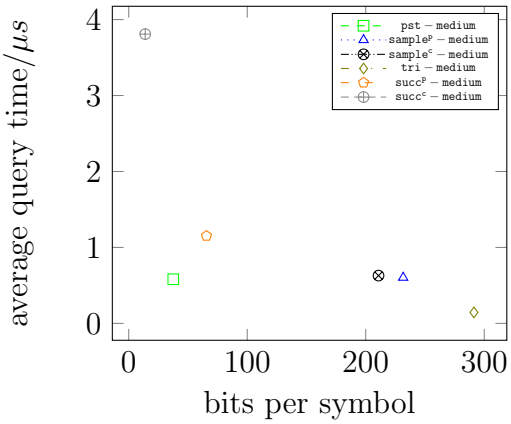
Figure 4.7: Different time-space tradeoffs achieved by $\texttt{subsr}_2$, `pst`, $\texttt{sample}^{\texttt{p}}$, $\texttt{sample}^{\texttt{c}}$, `tri`, $\texttt{succ}^{\texttt{p}}$, and $\texttt{succ}^{\texttt{c}}$ on `words`.



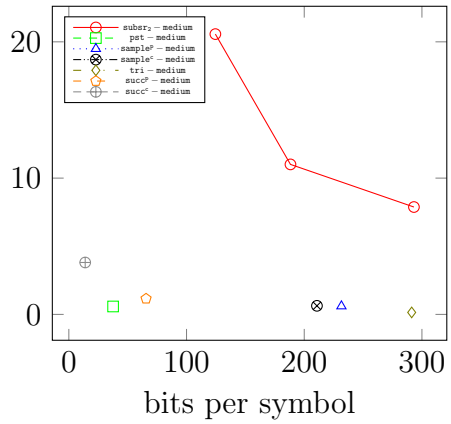
(a) `library` – `small` without $\texttt{subsr}_2$



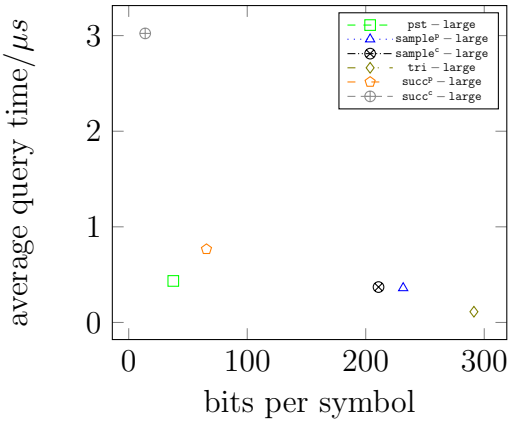
(b) `library` – `small` with $\texttt{subsr}_2$



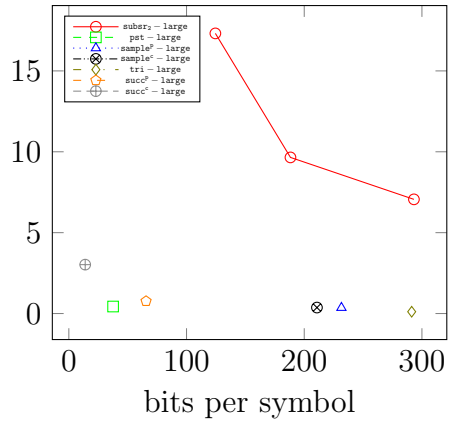
(c) `library` – `medium` without $\texttt{subsr}_2$



(d) `library` – `medium` with $\texttt{subsr}_2$

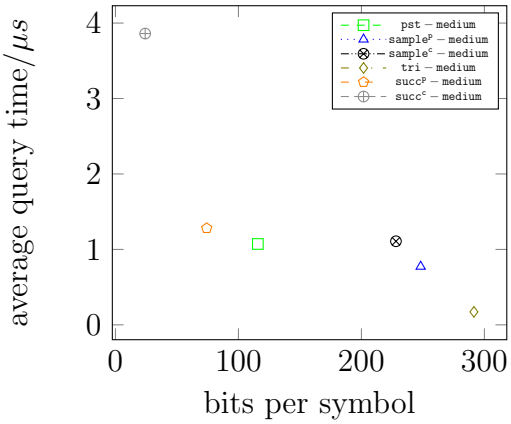


(e) `library` – `large` without $\texttt{subsr}_2$

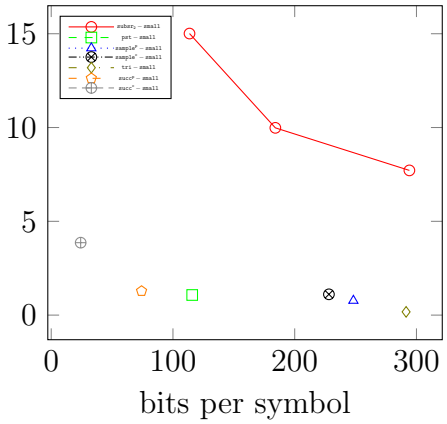


(f) `library` – `large` with $\texttt{subsr}_2$

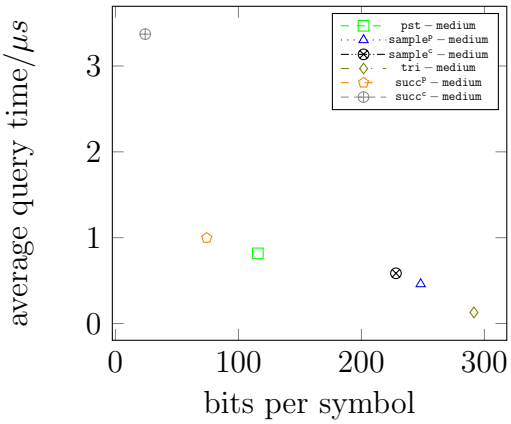
Figure 4.8: Different time-space tradeoffs achieved by $\texttt{subsr}_2$, `pst`, $\texttt{sample}^{\texttt{p}}$, $\texttt{sample}^{\texttt{c}}$, `tri`, $\texttt{succ}^{\texttt{p}}$, and $\texttt{succ}^{\texttt{c}}$ on `library`.



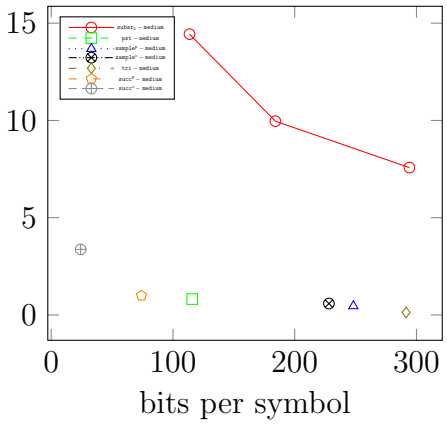
(a) `tickets` – `small` without $\texttt{subsr}_2$



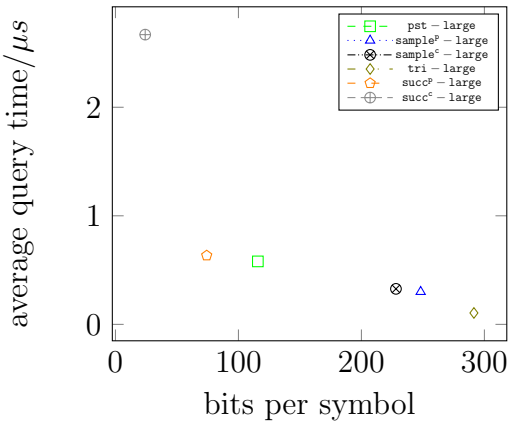
(b) `tickets` – `small` with $\texttt{subsr}_2$



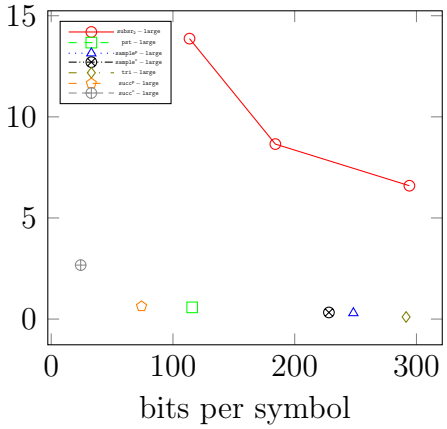
(c) `tickets` – `medium` without $\texttt{subsr}_2$



(d) `tickets` – `medium` with $\texttt{subsr}_2$



(e) `tickets` – `large` without $\texttt{subsr}_2$



(f) `tickets` – `large` with $\texttt{subsr}_2$

Figure 4.9: Different time-space tradeoffs achieved by $\texttt{subsr}_2$, `pst`, $\texttt{sample}^{p}$, $\texttt{sample}^{c}$, `tri`, $\texttt{succ}^{p}$, and $\texttt{succ}^{c}$ on `tickets`.

Chapter 5

Conclusions

In this thesis, we study various exact and approximate range mode query structures. To examine the performance of different data structures, we utilize five publicly available datasets and measure the query time, construction time, space consumption and approximate mode ratios of these solutions.

As for exact range mode query solutions, most of our experiments are on the two data structures of Chan et al. [5]: `sqrt` and `subsr`. They outperform two naive approaches $\texttt{nv}_1$ and $\texttt{nv}_2$, as well as an earlier data structure solution `supsr` [19]. Two variants of `subsr` are implemented: $\texttt{subsr}_1$ is an implementation of the final method of Chan et al. [5], and we use succinct data structure to further compress some components in $\texttt{subsr}_1$ to design $\texttt{subsr}_2$ as described in section 3.3. There are two different implementations of bit vectors in the `sdsl-lite` [14], a plain bit vector which we refer to as `p` and a compressed bit vector which we refer to as `c`. Then we compare $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$ and find that $\texttt{subsr}_1^{\texttt{p}}$ achieves better time-space tradeoffs on the `IPs`, `words`, `library` and `tickets` datasets while $\texttt{subsr}_1^{\texttt{c}}$ achieves better time-space tradeoffs on the `reviews` dataset. We observe the same result when comparing $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$. By performing experimental study using different values of s, we find that $\texttt{subsr}_2$ outperforms other solutions when the same amount of space is used.

Regarding approximate range mode data structures, we study `simple`, `pst`, `sample`, `tri` and `succ`. All these data structures outperform the best exact range mode solutions in query time when we choose $\epsilon = \frac{1}{2}$, and the average approximate ratio is excellent. We also find that the space cost of $\texttt{succ}^{\texttt{c}}$ is only $35.6\% \sim 93.8\%$ of that of the input array of 32-bit integers. Its non-succinct version, `tri`, uses slightly more space than `pst`, but achieves better query performance. Furthermore, the query time of these solutions increases at a logarithmic rate when we decrement the value of ϵ, while the space costs is proportional to $1/\epsilon$.

One interesting open problem is how to design a practical $(1+\epsilon)$-approximate range

mode structure with $\mathcal{O}(\frac{n}{\varepsilon})$ space and $\mathcal{O}(\lg \frac{1}{\varepsilon})$ query time. We have not implemented the 3-approximate structure of Greve et al. [15] or the 4-approximate structure of El-Zein et al. [10], because they are not practical, but these solutions are needed to achieve $\mathcal{O}(\lg \frac{1}{\varepsilon})$ query time in their solutions.

Bibliography

[1] Project Gutenberg. (n.d.)., retrieved in July 2021. Available from https://www.gutenberg.org/.

[2] Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. *Theory OF Computing*, 8:69–94, 2012.

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

[4] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388. Springer, 2005.

[5] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems*, 55(4):719–741, Mar 2013.

[6] David R Clark and J Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, 1996.

[7] Francisco Claude, J Ian Munro, and Patrick K Nicholson. Range queries over untangled chains. In *International Symposium on String Processing and Information Retrieval*, pages 82–93. Springer, 2010.

[8] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[9] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[10] Hicham El-Zein, Meng He, J Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149, page 57. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[11] Hicham El-Zein, Meng He, J Ian Munro, and Bryce Sandlund. Improved time and space bounds for dynamic range mode. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, page 25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[12] Derya Erhan. Boğaziçi university DDoS dataset, 2019. Available from https://dx.doi.org/10.21227/45m9-9p82.

[13] Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7, 1990.

[14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[15] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *International Colloquium on Automata, Languages, and Programming*, pages 605–616. Springer, 2010.

[16] Yuzhou Gu, Adam Polak, Virginia Vassilevska Williams, and Yinzhan Xu. Faster monotone min-plus product, range mode, and single source replacement paths. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 75:1–75:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[17] Meng He and Serikzhan Kazi. Path query data structures in practice. In *18th International Symposium on Experimental Algorithms*, volume 160, pages 27:1–27:16, 2020.

[18] D. Jansens. *Persistent Binary Search Trees*. https://cglab.ca/ dana/pbst/.

[19] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005.

[20] Seattle Public Library. Seattle library checkout records, 2017. Available from https://www.kaggle.com/seattle-public-library/seattle-library-checkout-records.

[21] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, 2019.

[22] City of New York. NYC parking tickets, 2017. Available from https://www.kaggle.com/datasets/new-york-city/nyc-parking-tickets.

[23] Mihai Patrascu. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008.

[24] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

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

[26] Bryce Sandlund and Yinzhan Xu. Faster dynamic range mode. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[27] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898, 2012.

[28] Virginia Vassilevska Williams and Yinzhan Xu. Truly subcubic min-plus product for less structured matrices, with applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–29. SIAM, 2020.