저작자표시-비영리-변경금지 2.0 대한민국

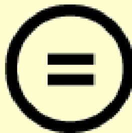
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Practical Encodings for Range Top-2 Queries and Compressed RAM

구간 top-2 질의 및 압축 RAM 문제 해결을 위한 실용적 인코딩

FEBRUARY 2023

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Park Wooyoung

Ph.D. DISSERTATION

Practical Encodings for Range Top-2 Queries and Compressed RAM

구간 top-2 질의 및 압축 RAM 문제 해결을 위한 실용적 인코딩

FEBRUARY 2023

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Park Wooyoung

Practical Encodings for Range Top-2 Queries and Compressed RAM

구간 top-2 질의 및 압축 RAM 문제 해결을 위한 실용적 인코딩

지도교수 하순회

이 논문을 공학박사학위논문으로 제출함

2022 년 11 월

서울대학교 대학원

컴퓨터 공학부

박우영

박우영의 박사학위논문을 인준함

2022 년 12 월

위 원 장	박근수	(인)
부위원장	하순회	(인)
위 원	Srinivasa Rao Satti	(인)
위 원	이인복	(인)
위 원	조승범	(인)

Abstract

Practical Encodings for Range Top-2 Queries and Compressed RAM

Wooyoung Park
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

In this thesis, we design various compressed/succinct data structures. Also, we implemented our data structures from the practical view and conducted experiments to evaluate data structures. Experimental results show that our data structures are time- and space-efficient. Also, our data structures show simpler substructures that enable software engineers to utilize and maintain easily. In this thesis, we consider the following two problems: (1) range Top-2 encodings, and (2) dynamic compressed strings supporting access and update operations. Given an array of elements from a total ordering, a Top-2 query returns the first and second largest elements within a given query range. The Top-2 encoding problem encodes a given input array to support Top-2 queries efficiently. In the dynamic compressed string problem, we would like to maintain a string in compressed form while supporting access and update operations efficiently.

For the Top-2 encoding problem, we designed two approaches. The first implementation is based on an alternative representation of Davoodi et al.'s [1] data structure, which supports queries efficiently. Our data structure not only

gives improved practical space and time efficiency, but also gives simpler substructures compared to Davoodi et al.'s [1] data structure, which uses tree-covering [2]. The other implementation is based on an RT2Q encoding on a $2 \times n$ array [3]. Our data structure uses less construction time while being competitive in terms of space.

For the second problem, we designed two implementations based on the compressed RAM of Jansson et al. [4], and Grossi et al. [5], respectively. For this problem, we designed a data structure that is simpler than the data structures [4], [5]. Also, since our substructures are simpler to implement, we have room for further optimization. Experimental results show that our data structure supports operations efficiently while keeping the space proportional to the entropy of the input.

Keywords: Range top-2 query, range minimum query, Cartesian tree, encoding model, succinct encoding, succinct data structure, dynamic data structure, dynamic string, access query, replace query, insert query, delete query
Student Number: 2016-21203

Contents

Abstract **i**

Contents **iii**

List of Figures **v**

List of Tables **viii**

Chapter 1 Introduction **1**

1.1 Contributions of the thesis 3

1.2 Organization of the thesis 4

Chapter 2 Preliminaries **5**

2.1 RAM model. 5

2.2 Range maximum query. 6

2.3 Cartesian tree. 6

2.4 Encoding data structures. 7

2.5 Dynamic data structures. 8

Chapter 3 Practical Implementation of Encoding Range Top-2 Queries **9**

3.1 Introduction. 9

3.2 A practical implementation of encoding RT2Q with efficient queries 13

3.2.1 DFUDS and 2d-max heap 13
3.2.2 Practical implementation of Davoodi et al.'s data structure 15
3.3 Space-efficient encoding of RT2Q with fast construction 22
3.4 Experimental results 26
3.4.1 Experimental results on data structures for answering RT2Q efficiently 27
3.4.2 Experimental results on space-efficient encodings for RT2Q 33

Chapter 4 Practical Implementations of Compressed RAM 53
4.1 Introduction 53
4.2 Practical implementation 56
4.3 Experimental results 59

Chapter 5 Conclusions and Open Problems 70

요약 82

Acknowledgements 84

List of Figures

Figure 3.1 Example of (a) $C(A)$ and (b) $2dmax(A)$ of the array $A[1, 12]$. Red and blue colored nodes are the nodes in $\mathsf{linspine}(5)$ and $\mathsf{rinspine}(5)$ of $C(A)$ respectively. 20

Figure 3.2 $r2dmax(A)$ of the array in Figure 3.1. Red and blue colored nodes correspond to the nodes in $C(A)$ in Figure 3.1 with the same colors. 21

Figure 3.3 Example of D_A and its encoding S_A of the array $A[1, 8]$. 23

Figure 3.4 The distribution of the depth of nodes in $2dmax(A)$. . . 28

Figure 3.5 The distribution of the depth of the nodes in $2dmax(A)$ which correspond to the RMQ of A. 28

Figure 3.6 Query time based on the allotted space for E and E'. . . 30

Figure 3.7 Space usage on random arrays of size from 10^6 to 10^9. . 36

Figure 3.8 Query time on the random array of size 10^9. 37

Figure 3.9 Space usage on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$. 37

Figure 3.10 Query time on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$. 38

Figure 3.11 Space usage on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$. 38

Figure 3.12 Query time on on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$. 39
Figure 3.13 Query time on the pseudo-increasing array with $\delta = 10^3$. 40
Figure 3.14 Query time on the pseudo-increasing array with $\delta = 10^6$. 40
Figure 3.15 Query time on the pseudo-decreasing array with $\delta = 10^3$. 41
Figure 3.16 Query time on the pseudo-decreasing array with $\delta = 10^6$. 41
Figure 3.17 Space usage on the inputs from real data sets. 42
Figure 3.18 Query times on the inputs from real data sets (weather). 42
Figure 3.19 Query times on the inputs from real data sets (dna). . . 43
Figure 3.20 Query times on the inputs from real data sets (google). . 43
Figure 3.21 Query times on the inputs from real data sets (youtube). 44
Figure 3.22 Construction time on random array of size from 10 to 10^6. 44
Figure 3.23 Construction time on pseudo-increasing array of size $n = 10^6$. 45
Figure 3.24 Construction time on pseudo-decreasing array of size $n = 10^6$. 45
Figure 3.25 Size of the encodings (without compression) on random array of size from 10 to 10^6. 46
Figure 3.26 Size of the encodings (without compression) on pseudo-increasing array of size $n = 10^6$. 46
Figure 3.27 Size of the encodings (without compression) on pseudo-decreasing array of size $n = 10^6$. 47
Figure 3.28 Size of the encodings (with Huffman encoding) on random array of size from 10 to 10^6. 47
Figure 3.29 Size of the encodings (with Huffman encoding) on pseudo-increasing array of size $n = 10^6$. 48
Figure 3.30 Size of the encodings (with Huffman encoding) on pseudo-decreasing array of size $n = 10^6$. 48

Figure 3.31 Size of the encodings (with LZW compression) on random array of size from 10 to 10^6. 49
Figure 3.32 Size of the encodings (with LZW compression) on pseudo-increasing array of size $n = 10^6$. 49
Figure 3.33 Size of the encodings (with LZW compression) on pseudo-decreasing array of size $n = 10^6$. 50
Figure 3.34 Construction time of the encodings on the inputs from real data sets. 50
Figure 3.35 Space usage (original) of the encodings on the inputs from real data sets. 51
Figure 3.36 Space usage (huffman) of the encodings on the inputs from real data sets. 51
Figure 3.37 Space usage (lzw) of the encodings on the inputs from real data sets. 52

Figure 4.1 replace-seq test (a) from ENGLISH to DNA 64
Figure 4.2 replace-seq test (a) from XML to PROTEINS 65
Figure 4.3 replace-random1 test on ENGLISH. 65
Figure 4.4 replace-random2 test on ENGLISH. 66
Figure 4.5 insert-seq test on ENGLISH. 66
Figure 4.6 delete-seq test on ENGLISH. 67
Figure 4.7 indel-random test on ENGLISH. 67

List of Tables

Table 4.1 File statistics of test files. H_k denotes the k-th order entropy. 68

Table 4.2 Operation time tests on replace. Time unit is nanoseconds. 68

Table 4.3 Operation time tests on insert. Time unit is nanoseconds. 68

Table 4.4 Operation time tests on delete. Time unit is nanoseconds. 69

Chapter 1

Introduction

Over the decades, a wide range of research has been done on compressed/succinct data structures, where the aim is to design data structures whose space usage is close to the space needed to store the data in compressed form while supporting the operations of the data efficiently. For example, on the FM index [6], wavelet tree [7], and Elias-Fano encoding [8], compressed/succinct data structures played a great role. The main difference between compressed/succinct data structures and raw compressed data is that the former can support predefined queries without decompressing itself, but the latter cannot support any useful query without decompressing itself. Due to their supporting query, compressed data structures are used in other areas of computer science, e.g., database [9, 10, 11, 12, 13] and bioinformatics [14, 15, 16, 17]. But from the point of practical usage, compressed/succinct data structures have some limitations even though previous works give impressive results. The following points are limitations for practical usage of compressed/succinct data structures.

- As compressed/succinct data structures are highly optimized from the theoretical view (RAM model [18]), compressed/succinct data structures

have more complex substructures and query algorithms in theory. Mathematical optimization and complex substructure make the data structure work poorly in practice. For example, consider space and time-efficient data structures for answering the RMQ [19]. And those complex substructures need various word sizes to satisfy given theoretical bound. But since computers only support 8,16,32, and 64 bits primitive word size, and 128,256, and 512 bits SSE or AVX word size, whoever wants to build their own customized word size must sacrifice time performance, or sacrifice space usage from ceiling up word size.

- Another limitation from complex substructures is software aging. Software aging is an inevitable phenomenon even though software has correct mathematical proof [20]. Although many data structure research exist as a library/framework, not application software, library/framework are also not free from aging. Because hardware and operating systems evolve, and this makes the environment of the library/framework change. However, complex substructures make it hard for software engineers to maintain implementations. And aging from lack of maintenance makes libraries from compressed/succinct data structures less adaptive.

- Every random access memory requires two steps. One is address translation from virtual memory address to physical memory address, and the other is accessing physical memory access. To optimize these procedures, modern CPUs contain translation lookaside buffer (TLB) and cache memory. So, programmers strive to utilize the property of cache memory to write performance-strong programs. However, succinct data structures such as FM-indexes show random memory access patterns [21], which is not a cache-friendly memory access pattern.

Due to those limitations from compressed/succinct data structures, several researchers propose its practical implementation that overcomes some limita-

tions. Researchers change or re-invent complex substructures to overcome limitations [22, 23, 24]. This thesis also contains practical implementations that overcome those limitations.

1.1 Contributions of the thesis

In this thesis, we propose the following practical implementations of compressed/succinct data structures. Overall, our proposed data structures and its implementation give simpler substructures and better experimental results than previous work.

- **Practical Implementation of Encoding Range Top-2 Queries:** Given an array $A[1, n]$ of n elements from a total order, the range Top-2 encoding problem is to construct a data structure that answers RT2Q, which returns the positions of the first and second largest elements within a given range of A, without accessing the array A at query time. In this thesis, we design a practical variant of an encoding for *range Top-2 query (RT2Q)*, and evaluate its performance. We design the following two implementations: (i) An implementation based on an alternative representation of Davoodi et al.'s [1] data structure, which supports queries efficiently. Experimental results show that our implementation is efficient in practice, and gives improved time-space tradeoffs compared to the indexing data structures (which keep the original array A as part of the data structure) for range maximum queries. (ii) Another implementation based on Jo et al.'s RT2Q encoding on $2 \times n$ array [3], which can be constructed in $O(n)$ time. We compare our encoding with Gawrychowski and Nicholson's optimal encoding [25], and show that in most cases, our encoding shows faster construction time while using a competitive space in practice. This work is accepted in The Computer Journal [26].

- **Practical Implementations of Compressed RAM:** Given a string S over an alphabet of size σ, we consider practical implementations of *extended compressed RAM* on S, which supports access, replace, insert, and delete operations on S while maintaining S in compressed form. In this thesis, we proposed two implementations where each of them is based on the compressed RAM of Jansson et al. [4], and Grossi et al. [5], respectively. Also, we proposed optimization schemes for our proposed data structures. Experimental results show that our implementations support the operations efficiently while keeping the space proportional to the entropy of the input during the updates. This work will appear at the Data Compression Conference (DCC) 2023 [27].

1.2 Organization of the thesis

The rest of this thesis is organized as follows. In chapter 2, we introduce some preliminaries of our contributions. In chapter 3, we introduce our alternative representations of data structures of Encoding Range Top-2 Queries. In chapter 4, we introduce our modified practical implementation of dynamic string. This thesis ends at chapter 5 with a summarization of our results and some open problems.

Chapter 2

Preliminaries

In this chapter, we introduce preliminaries to show our contributions for two problems.

2.1 RAM model.

In this thesis, our data structures assume standard word-RAM model [28], which is the variant of the classic RAM model [18]. Word-RAM model has some assumptions for realistic modeling of a computer. First, word-RAM model assumes that the word size as $w = \Theta(logn)$ bits. Here, n denotes the number of bits for given raw data. Second, the word-RAM model supports constant time access, arithmetic, and bitwise operations on w-bit word. The assumption of $w = \Theta(logn)$ bits word size is reasonable assumption, because many modern computer systems use 64-bit words, and there is no known practical computer system using more than 2^{64} bits.

2.2 Range maximum query.

Given an array $A[1, n]$ of n elements from a total order, the *range maximum query* on $A[i, j]$ (denoted by $\mathsf{RMQ}(i, j)$) returns the position of the largest element in $A[i, j]$. We assume that all elements in A are distinct (if there are equal elements, we can break the ties according to their positions, by considering the leftmost one as the largest value among them). The problem of constructing space and/or time-efficient data structures for answering RMQ is one of the fundamental problems in data structures. There were many theoretical solutions for answering RMQ before solutions using succinct data structures were introduced. The idea of those solutions is that they use problem conversion. One can convert the RMQ problem on A into the LCA (lowest common ancestor) problem on Cartesian tree [29] (The formal definition of Cartesian tree is explained in the next section). Those solutions [30, 31] use $O(n)$ words ($O(nlogn)$ bits) space usage and constant query time. Also, based on those solutions, simplified solutions [32, 33, 34, 35] were proposed. Those simplified solutions show ($O(nlogn)$ bits) space usage and constant query time. But $O(nlogn)$ bits space usage is far from the theoretical lower bound [36, 37]. RMQ solutions using succinct data structures were proposed in this background [19, 38, 39]. Recently, variants of RMQ solutions using succinct data structures supporting such as offline queries [40, 41], sublinear size data structures [42], and average cases [43] were proposed.

2.3 Cartesian tree.

Given an array $A[1, n]$ of size n, the *Cartesian tree* [29] of A, denoted by $C(A)$, is a binary tree where (i) the root node of $C(A)$ corresponds to the position $i = \mathsf{RMQ}(1, n)$, and (ii) the left and right subtrees of $C(A)$ are the Cartesian trees of $A[1, i-1]$ and $A[i+1, n]$ respectively. From the definition, the i-th node in the inorder traversal of $C(A)$ corresponds to the i-th position of A

(see Figure 3.1 (a) for an example). In the rest of this chapter, we refer to the nodes in the Cartesian tree by their inorder numbers (i.e., their corresponding positions in the array A). Also, one can convert the RMQ problem on A into the LCA (lowest common ancestor) problem on $C(A)$ [30]. More precisely, for any $i, j \in [1, n]$, $\mathsf{RMQ}(i, j)$ is the same as $\mathsf{LCA}(i, j)$, which is the LCA of the nodes i and j in $C(A)$. This implies that one can support RMQ on A by storing $C(A)$ instead of A (thus, $C(A)$ is an encoding for answering RMQ on A).

2.4 Encoding data structures.

In general, the data structures for answering specific queries can be categorized into two types: (i) *indexing data structures*, and (ii) *encoding data structures.* In indexing data structures, one can access the input data at query time, while it is not allowed in encoding data structures. Given a set of objects S and a set of queries Q, one can partition S into a set of equivalence classes $\mathcal{C}$ such that two objects belong to the same equivalence class if and only if both the objects have the same set of answers to all the queries in Q. In this case, the information-theoretic lower bound for encoding any arbitrary object is $\log|\mathcal{C}|$ bits[1], which is also referred to as the *effective entropy* of S with respect to Q [44].

For many problems, including RMQ problem, the effective entropy is much less than the input size – for example, the effective entropy for answering RMQ on A is $2n - o(n)$ bits [19], whereas storing A requires at least $n \log n$ bits, if all the elements in A are distinct. Thus, encoding data structures can be highly space-efficient in some cases compared to their indexing counterparts. Recent results [45, 46] show that encoding data structures for RMQ perform well both in theory and in practice.

[1]Throughout this chapter, we use log to denote the logarithm to the base 2.

2.5 Dynamic data structures.

In general, the data structures can be categorized into two types: (i) *static data structures*, and (ii) *dynamic data structures*. In static data structures, input cannot be changed after the data structure is constructed. Due to this limitation, static data structures have relatively simple substructures and query algorithms in theory. And maintaining static data structures in compressed form is relatively work well in practice. Thus, there are many practical implementations on static data structures [21, 47, 48]. In dynamic data structures, input can be changed after the data structure is constructed. In general, compared to static structures, dynamic compressed data structures have more complex substructures and query algorithms in theory, which makes the data structure work poorly in practice. Thus, only a few practical implementations of dynamic compressed data structures were proposed, mainly on the compressed bitvector (a data structure to represent a bitstring, supporting rank and select operations) [22, 23, 49]. Recently, dynamic succinct graph representation works have been proposed [24, 50]. However, these are only a few compared to static succinct data structures.

Chapter 3

Practical Implementation of Encoding Range Top-2 Queries

3.1 Introduction.

In this chapter, we consider the problem of answering *range top-2 queries*, which is an extension of RMQ.

Encoding range top-2 queries Range top-k queries are extension of the RMQ. The range top-k query on a given input array returns the positions of k largest elements in a given query range [51]. Range top-k queries have various applications in data and log mining to find the highest or lowest values in a range of a stream; in web search engines, to find the k most highly ranked pages restricted to a range of page identifiers etc. [52]. There has been a significant amount of research done in supporting top-k queries efficiently, mainly by the database community. See [53] for a detailed survey of the techniques and applications. In this chapter, we consider the problem of encoding *range top-2 queries*, which is a special case of the range top-k queries. The range top-2 query on $A[i, j]$ (denoted by RT2Q) returns the positions of the

largest and the second largest elements in $A[i, j]$. If $p = \mathsf{RMQ}(i, j)$, one can easily observe that the position of the second largest element in $A[i, j]$ is one of $p_1 = \mathsf{RMQ}(i, p-1)$ or $p_2 = \mathsf{RMQ}(p+1, j)$. Thus, any indexing data structure for answering RMQ also can answer RT2Q by comparing $A[p_1]$ and $A[p_2]$. Davoodi et al. [54] proposed the first encoding data structure for answering RT2Q in $O(1)$ time using $3.272n + o(n)$ bits, which is close to the effective entropy of $2.755n - \Theta(polylog(n))$ bits [25] for RT2Q. However, their encoding is not very practical since it represents the *Cartesian tree* [29] of A succinctly using the *tree-covering* approach of Farzan and Munro [55], which is hard to implement (compared to other succinct tree representations [56]). Gawrychowski and Nicholson [25] proposed an optimal $(2.755n + o(n))$-bit encoding for answering RT2Q. This encoding does not support the queries efficiently, and it can be constructed in $O(n^2)$ time in the worst case.

Previous work In this section, We introduce the $(3.272n + o(n))$-bit data structure of Davoodi et al. [54], which answers RT2Q in $O(1)$ time on an array $A[1, n]$ of size n. Their data structure answers the $\mathsf{RT2Q}(i, j)$ query by performing the following three steps:

1. Compute and return the position $k = \mathsf{RMQ}(i, j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compare $A[k_1]$ and $A[k_2]$, and return k_1 if $A[k_1] > A[k_2]$, or k_2 otherwise.

For answering $k = \mathsf{RMQ}(i, j)$, they maintain the *tree-covering* [55] representation of $C(A)$ to support LCA queries in $O(1)$ time, using $2n + o(n)$ bits. Next, to compare $A[k_1]$ and $A[k_2]$ without accessing the array A, they store the *spine sequence* S of A, defined as follows. For any node i which has left child i_l and right child i_r, let *left spine* (resp., *right spine*) of i, denoted by $\mathsf{lspine}(i)$ (resp.,

$\mathsf{rspine}(i)$), be the path from the node i to the leftmost (resp., rightmost) descendant of i. Also, let *left inner spine* (resp., *right inner spine*) of i, denoted by $\mathsf{linspine}(i)$ (resp., $\mathsf{rinspine}(i)$), be the $\mathsf{rspine}(i_l)$ (resp., $\mathsf{lspine}(i_r)$). Also, let L_i, R_i, l_i, and r_i be the number of nodes in $\mathsf{lspine}(i)$, $\mathsf{rspine}(i)$, $\mathsf{linspine}(i)$, and $\mathsf{rinspine}(i)$ respectively. Then by the property of $C(A)$, the nodes k_1 and k_2 in $C(A)$ are always on $\mathsf{linspine}(k)$ and $\mathsf{rinspine}(k)$, respectively. Now we define an array $S_k[1, m_k]$ to be a bit array of size $m_k = \max(l_k + r_k - 1, 0)$ where $S_k[j] = 0$ if the j-th largest element of A among the positions corresponding to $\mathsf{linspine}(k) \cup \mathsf{rinspine}(k)$ is in $\mathsf{linspine}(k)$, and 1 otherwise. Let $\mathsf{depth}(k)$ be the depth of node k, and for any given pattern b and sequence B, let $\mathsf{rank}_b(B, i)$ be the number of occurrences of b in the first i positions of B, and $\mathsf{select}_b(B, i)$ be the position of i-th occurrence of b in B. Then one can compare $A[k_1]$ and $A[k_2]$ by comparing $\mathsf{select}_0(S_k, \mathsf{depth}(k_1) - \mathsf{depth}(k))$ and $\mathsf{select}_1(S_k, \mathsf{depth}(k_2) - \mathsf{depth}(k))$ (i.e., by checking which of the two bits corresponding to the nodes k_1 and k_2 comes first in S_k). The sequence S is simply defined by concatenating all S_k's for all nodes $k \in C(A)$ in the increasing order of their inorder numbers. Finally, to locate the starting position of S_k in S efficiently, they introduce the following lemma.

Lemma 3.1 ([54]). *For any $u \in C(A)$, $\sum_{j<u} m_j =$*

$$2u - L_\tau - l_u + \mathit{Ldepth}(u) - \mathit{Rdepth}(u) + 1 - (u - \mathit{Lleaves}(u))$$

In the above lemma, τ denotes the root of $C(A)$. Also, for any node $u \in C(A)$, $\mathsf{Ldepth}(u)$ (resp., $\mathsf{Rdepth}(u)$) denotes the number of nodes which have their left (resp., right) child, in the path from τ to u; and, $\mathsf{Lleaves}(u)$ denotes the number of leaves $v \in C(A)$ which satisfies $v < u$.

Davoodi et al. [54] showed that all the operations used in the lemma can be computed in $O(1)$ time using the tree covering representation of $C(A)$ along with some auxiliary data structures. Furthermore, they showed that the size of S is at most $1.5n$, which implies that there exists the data structure for

answering RT2Q in $O(1)$ time using at most $3.5n + o(n)$ bits. With further optimization, they improved the space usage to $3.272n + o(n)$ bits while still supporting RT2Q in $O(1)$ time.

Example 3.1. *To answer* $\mathsf{RT2Q}(3,9)$ *on the array* $A[1,12]$ *in Figure 3.1 using* $C(A)$ *with spine sequence* S *of* A*, we first compute and return* $\mathsf{RMQ}(3,9) = \mathsf{LCA}(3,9) = 5$*. Next, to compare* $A[3]$ *and* $A[7]$ *(note that* $\mathsf{RMQ}(3,4) = 3$ *and* $\mathsf{RMQ}(6,9) = 7$*), we first locate the starting position of* S_5 *in* S *by* $\sum_{j<5} m_j = 2{\cdot}5-3-3+0-0+1-(5-2) = 2$*. Since* $\mathsf{depth}(5) = 0$*,* $\mathsf{depth}(3) = \mathsf{depth}(7) = 2$ *and* $\mathsf{select}_0(S_5, 2) > \mathsf{select}_1(S_5, 2)$*, We return* 7 *as the position of the second largest element in* $A[3,9]$*.*

Our contributions In this chapter, we give the first practical implementation of an encoding for RT2Q. We propose the following two implementations[1]:

- We first design an implementation which supports RT2Q efficiently, based on the data structure of Davoodi et al. [54]. However, instead of using the tree-covering approach, we use the DFUDS representation [57] of *2d-max heap* [19] which is easier to implement, and works well in practice. Our implementation supports RT2Q in $\log n \cdot g(n)$ time, for any increasing function $g(n) = \omega(1)$, using at most $3.5n + o(n)$ bits. The experimental results show that our data structure gives a better space-time tradeoffs, compared to the indexing data structures for RT2Q (that have access to the input array A, along with an auxiliary data structures for answering the RMQ queries).

- We also design another encoding based on Jo et al.'s encoding for answering RT2Q on $2 \times n$ array [3]. We first show that our encoding can be constructed in $O(n)$ time while using at most $11n/3 \sim 3.67n$ bits,

[1] All the implementations are available at `https://github.com/wyptcs/R2MQ`.

which is more than the space used by the encoding of Davoodi et al. [54] in the worst case. However, experimental results show that our encoding takes less space than the encoding of Davoodi et al. [54] for all the test cases, and even better than Gawrychowski and Nicholson's optimal encoding [25] in some cases.

The rest of this chapter is organized a follows. In Section 3.2, we describe how to implement the data structure of Davoodi et al. [54] using DFUDS [57] representation. In Section 3.3, we introduce an encoding for answering RT2Q on one dimensional array based on the encoding of Jo et al. [3]. We provide the empirical evaluations of our encodings in Section 3.4.

3.2 A practical implementation of encoding RT2Q with efficient queries

Davoodi et al.'s data structure [54], described in the previous section, uses the tree-covering method for encoding $C(A)$, which is not practical compared to other succinct tree representations such as BP (balanced parenthesis) [36] and DFUDS (depth-first unary degree sequence) [57] representations. In this section, we describe a practical implementation of Davoodi et al.'s data structure for answering RT2Q on $A[1, n]$, which uses the DFUDS representation of the *2d-max heap* of A [19]. We first describe the general definition of DFUDS and 2d-max heap, and show how to convert Davoodi et al.'s data structure using these tools.

3.2.1 DFUDS and 2d-max heap

Given an ordinal tree T with n nodes, DFUDS of T (denoted by $D(T)$) is a balanced parenthesis sequence of size $2n$ defined as follows: (i) if $n = 1$, $D(T)$ is (). (ii) Otherwise, if T has k subtrees $T_1, T_2, \ldots, T_k$, $D(T)$ is $(^{k+1})$ followed by $d(T_1), d(T_2), \ldots, d(T_k)$, where $d(T_i)$ is $D(T_i)$ with the first open parenthesis removed (see Figure 3.1 for an example). Since $D(T)[1, 2n]$ is a balanced paren-

thesis sequence, one can define two operations $\mathsf{findopen}(i)$ / $\mathsf{findclose}(i)$ which find the matching open / closed parenthesis of the closed / open parenthesis in $D(T)[i]$. It is known that by storing a $o(n)$-bit auxiliary structure along with $D(T)$, one can support rank, select, findopen and findclose operations in $O(1)$ time. This in turn enables us to represent T to support a comprehensive list of navigation queries on T in $O(1)$ time using $2n + o(n)$ bits [39] (see Table 2 in [56] for the list of operations).

One of the main reasons for using the tree-covering based approach for representing $C(A)$ in Davoodi's et al.'s structure is to support the $\mathsf{inorder}(i)$ operation, which returns the i-th node in the inorder traversal of $C(A)$. To our best knowledge, one cannot support this operation on the BP or DFUDS of $C(A)$ (note that LCA can be supported in $O(1)$ time on both BP and DFUDS [56]). Sadakane [38] showed that (i) if the difference between any two consecutive values in A is ± 1, then one can answer RMQ (also referred to as ± 1RMQ in this special case) on A in $O(1)$ time using $2n + o(n)$ bits, and (ii) for general A, one can support both inorder and LCA operations on $D(C(A))$ (thus, RMQ on A) in $O(1)$ time using $4n + o(n)$ bits, by converting $C(A)$ into a ternary tree by adding a dummy leaf to each node in $C(A)$.

Fischer and Heun [19] proposed the *2d-max heap* to support RMQ. The 2d-max heap on A (denoted by $2dmax(A)$) is an alternative representation of $C(A)$, defined as follows: $2dmax(A)$ is an ordered tree with $n + 1$ nodes, where for $1 \leq i \leq n$,

1. The i-th node in the preorder traversal of $2dmax(A)$ corresponds to $A[i - 1]$ (we assume that $A[0] = \infty$). In the rest of this chapter, we refer to this node as node $(i - 1) \in 2dmax(A)$. Therefore, the root of $2dmax(A)$ is 0.

2. For any non-root node $i \in 2dmax(A)$, the parent of i is the node j where j is the rightmost position in $A[0, i - 1]$ such that $A[j] > A[i]$.

The above definition implies that for $1 \leq i \leq n$, the node $i \in C(A)$ and the node

$i \in 2dmax(A)$ both correspond to the position i in A. The example in Figure 3.1 (a) and (b) shows the $C(A)$ and $2dmax(A)$ of the input array A, respectively. Fischer and Heun also showed that $\mathsf{RMQ}(i, j)$ operation can be supported in $O(1)$ time by using $D(2dmax(A))$ along with $o(n)$-bit auxiliary structures for supporting rank, select, findopen, and ± 1RMQ queries on $D(2dmax(A))$ – using $2n + o(n)$ bits in total.

3.2.2 Practical implementation of Davoodi et al.'s data structure

In this section, we propose an alternative implementation of the data structure of [54] on A using $D(2dmax(A))$. Since one can support RMQ using $D(2dmax(A))$ [19], it is enough to show how to find the position of the second largest element in $A[i, j]$. We first introduce the following lemma to apply Lemma 3.1 to $2dmax(A)$. Note that the left/right spine of a node $i \in 2dmax(A)$ is defined as the path from node i to the leftmost/rightmost descendant of i.

Lemma 3.2. *Given an array $A[1, n]$ of size n where all elements in A are distinct, the following properties hold for any node k in the Cartesian tree, $C(A)$, of A.*

(a) l_k (number of nodes in linspine(k) in $C(A)$) is equal to the number of nodes in rspine(k_l) in $2dmax(A)$, where k_l = presibling(k) denotes the previous sibling of k.

(b) r_k (number of nodes in rinspine(k) in $C(A)$) is equal to the number of children of $k \in 2dmax(A)$.

(c) Ldepth(k) is equal to the number of right siblings of all the nodes on the path from node k to the root in $2dmax(A)$.

(d) Rdepth$(k) = d_k - 1$, where d_k is the depth of $k \in 2dmax(A)$.

(e) $\mathsf{Lleaves}(k)$ *is equal the number of leftmost children* $u < k$ *which are also leaves in* $2dmax(A)$*.*

Proof. (a) Let $i_0 < k$ be the rightmost position of A which satisfies $\mathsf{RMQ}(i_0, k) \neq k$. Then by the definition of $C(A)$, $\mathsf{linspine}(k)$ is composed of the nodes $\{i_1, i_2, \dots, i_{l_k}\}$ of $C(A)$ where $i_j = \mathsf{RMQ}(i_{j-1}+1, k-1)$. Thus, if $l_k > 0$, the node k in $2dmax(A)$ always has the previous sibling k_l (otherwise, k has no left child in $C(A)$, which implies $l_k = 0$). Furthermore, since $k-1$ is the rightmost leaf of the subtree of $2dmax(A)$ rooted at k_l, all the nodes $i_1, i_2, \dots, i_{l_k}$ are on the $\mathsf{rspine}(k_l)$ in $2dmax(A)$.

(b) For any node i in $C(A)$, i is the node in $\mathsf{rinspine}(k)$ if and only if for any $k < j < i$, $\mathsf{RMQ}(j, i) = i$ and $\mathsf{RMQ}(k, i) = k$, which implies k is the rightmost position in A which satisfies $A[k] > A[i]$. Thus, by the definition of $2dmax(A)$, the set of all nodes in $\mathsf{rinspine}(k)$ in $C(A)$ is same as the set of all children of the node k in $2dmax(A)$.

(c) Let $Lpath(k)$ be the set of nodes in $C(A)$ which have their left child in the path from k to the root (hence, $|Lpath(k)| = \mathsf{Ldepth}(k)$). Now suppose $Lpath(k) = \{i_1, i_2, \dots, i_{\mathsf{Ldepth}(k)}\}$ where $i_1 < i_2 < \dots < i_{\mathsf{Ldepth}(k)}$. Then for any $j \in \{1, 2, \dots, \mathsf{Ldepth}(k)\}$, (i) $i_j > k$, and (ii) $\mathsf{RMQ}(k, i_j) = i_j$. Therefore, for the node $k \in 2dmax(A)$, $Lpath(k)$ is the same as the set of nodes in $2dmax(A)$ which are the right siblings of the nodes on the path from the node k to the root.

(d) Similar to the case of $\mathsf{Ldepth}(k)$, let $Rpath(k)$ be the set of nodes in $C(A)$ which have their right child in the path from k to the root (hence, $|Rpath(k)| = \mathsf{Rdepth}(k)$). Then $Rpath(k)$ consists all the nodes i_j in $C(A)$

which satisfy: (i) $i_j < k$, and (ii) $\mathsf{RMQ}(i_j, k) = i_j$. Thus by the definition of $2dmax(A)$, $Rpath(k)$ is the same as the set of proper ancestors of k in $2dmax(A)$.

(e) Note that a node in $C(A)$ is a leaf if and only if its corresponding node in $2dmax(A)$ is a leftmost child which is also a leaf. Thus, set of all leaf nodes in $C(A)$ are the same as the set of all leftmost children $u < k$ which are also leaves in $2dmax(A)$.

$\square$

We define the spine sequence S of A, analogous to the spine sequence in [54] (that is, concatenating all the S_k's for each non-root node $k \in 2dmax(A)$ according to their preorder value in $2dmax(A)$). Then by Lemma 3.2 (a) and (b), we can answer $\mathsf{RT2Q}(i, j)$ using the following procedure:

1. Compute and return the position $k = \mathsf{RMQ}(i, j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compute the node $k_l = \mathsf{presibling}(k)$ in $2dmax(A)$, and the values $l = \mathsf{depth}(k_1) - \mathsf{depth}(k_l) + 1$ and $r = \mathsf{childrank}(k_2)$, where $\mathsf{childrank}(k_2)$ denotes the number of siblings of k_2 which are to its left.
4. Locate the starting position of S_k in S, and return k_1 if $\mathsf{select}_0(S_k, l) < \mathsf{select}_1(S_k, r)$, or k_2 otherwise.

Note that the operations used in the above procedure (RMQ, $\mathsf{presibling}$, $\mathsf{childrank}$, and depth) can be supported in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [58]. Also, to locate the position of S_k in S, we need to compute $\sum_{1 \le j < k} m_j$ (recall that $m_j = |S_j|$). Now we describe how to compute each value in Lemma 3.1 (therefore, $\sum_{1 \le j < k} m_j$) using $D(2dmax(A))$ with Lemma 3.2.

1. L_τ: The node $\tau = \mathsf{RMQ}(1, n)$ is the rightmost child of the node 0 (the root of $2dmax(A)$). Also all the nodes of $\mathsf{lspine}(\tau)$ in $C(A)$ are all the left siblings of τ (including τ itself) in $2dmax(A)$. Thus this value can be computed in $O(1)$ time by $\mathsf{degree}(0)$ (note that degree can be computed in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [56]).

2. l_k: By Lemma 3.2, $\mathsf{linspine}(k)$ of $C(A)$ is the same as the $\mathsf{rspine}(k_l)$ of $2dmax(A)$. Since the rightmost leaf of k_l is $k-1$, this can be computed in $O(1)$ time by $\mathsf{depth}(k-1) - \mathsf{depth}(k_l) + 1$.

3. $\mathsf{Ldepth}(k)$: For $k \in 2dmax(A)$, let $L(k)$ be the number right siblings of the nodes on the path from the node k to the root in $2dmax(A)$. Now we describe how to compute $L(k)$ using $D(2dmax(A))$. Let d be the depth of $2dmax(A)$, and suppose $f(n) = \log n \cdot g(n)$ where $g(n)$ is any increasing function which satisfies $g(n) = \omega(1)$. Then we fix a value $0 \le \ell < f(n)$, and define the array E which stores all the values of $L(k)$ for every node $k \in 2dmax(A)$ whose depth is $\ell + j \cdot f(n)$, for all $0 \le j \le \lfloor (d-\ell)/f(n) \rfloor$. By a simple counting argument, we can choose ℓ to satisfy $|E| \le n/f(n)$. Thus, E can be stored using at most $n/f(n) \cdot \log n = n/g(n) = o(n)$ bits. The values in E are stored according to the preorder number of corresponding nodes in $2dmax(A)$. In addition, we maintain the bit array $B[1, n]$ of size n where for $1 \le i \le n$, $B[i] = 1$ if and only if the $L(i)$ is stored in E. Using the data structure of Raman et al. [59], we can store B using $\log \binom{n}{f(n)} + o(n) = o(n)$ bits while supporting rank queries in $O(1)$ time (we can also access any position of B in $O(1)$ time by two rank queries). To answer $L(k)$, we initialize the counter $c = 0$, and start the scanning nodes on $Lpath(k)$ starting from the node k. During this scan, when we are at node j, we first check $B[j]$. If $B[j] = 0$, we increase c to be $c + r$ where $r = \mathsf{degree}(\mathsf{parent}(j)) - \mathsf{childrank}(j)$ (note that parent can be computed in $O(1)$ time using $D(2dmax(A))$ [56]), and move to the

parent of j. If $B[j] = 1$, we return $L(k) = c + \mathsf{rank}_1(B, j)$. Thus, using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures, we can answer $L(k)$ in $O(f(n))$ time.

4. $\mathsf{Rdepth}(k)$: By Lemma 3.2, this is the same as the number of proper ancestors of k in $2dmax(A)$, which can be computed $O(1)$ time by $\mathsf{depth}(k) - 1$.

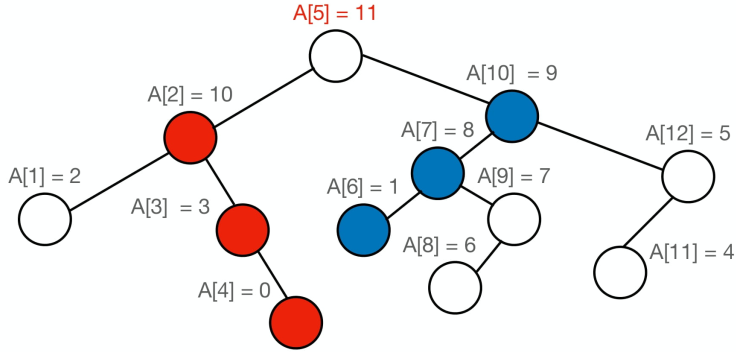
5. $\mathsf{Lleaves}(k)$: By Lemma 3.2, this is the same as the number of leftmost children $u < k$ which are also leaves in $2dmax(A)$. This value can be computed by counting the number of occurrences of the pattern '()))' before the closing parenthesis corresponding to node k, in $O(1)$ time, using $D(2dmax(A))$ with $o(n)$-bit auxiliary data structures [60].

Example 3.2. *We show how to locate the starting position of S_5 in S using $2dmax(A)$ in Figure 3.1 (b). From node $5 \in 2dmax(A)$ in the figure, one can observe that $\mathsf{presibling}(5) = 2$, $L(5) = 0$, and $\mathsf{select}_{)}(D(2dmax(A)), 5) + 1 = 12$. Also $\mathsf{degree}(0) = 3$, $\mathsf{depth}(4) - \mathsf{depth}(2) + 1 = 3$, $\mathsf{depth}(5) - 1 = 0$, and $\mathsf{rank}_{()))}(D(2dmax(A)), 12) = 2$. Thus, the starting position of S_5 in S is $\sum_{j<5} m_j = 2 \cdot 5 - 3 - 3 + 0 - 0 + 1 - (5 - 2) = 2$.*

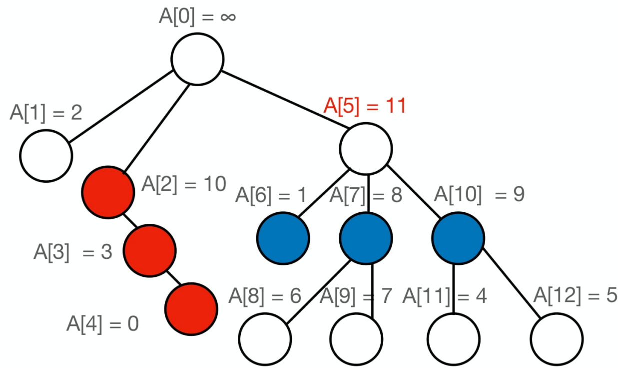
We summarize the result in the following theorem.

Theorem 3.1. *Given an array $A[1, n]$ of size n, RT2Q on A can be computed in $O(\log n \cdot g(n))$ time, for any increasing function $g(n) = \omega(1)$. The data structure uses at most $1.5n + o(n)$ additional bits, along with the DFUDS sequence of the 2d-max heap of A, $D(2dmax(A))$.*

Alternative representation of $2dmax(A)$. In practice, the performance of the data structure of Theorem 3.1 highly depends on the depth of $2dmax(A)$. To



(a) $C(A)$



(b) $2dmax(A)$

$$A[1,12] = 2 \quad 10 \quad 3 \quad 0 \quad 11 \quad 1 \quad 8 \quad 6 \quad 7 \quad 9 \quad 4 \quad 5$$

$$D(2dmax(A)) = (\; (\; (\; (\;) \;) \; (\;) \; (\;) \;) \; (\; (\; (\;) \;) \; (\; (\;) \;) \;) \; (\; (\;) \;) \;)$$

$$S_2 = 1 \quad S_5 = 0 \;\; 1 \;\; 1 \;\; 0 \;\; 1 \quad S_7 = 1 \;\; 1 \quad S_{10} = 0 \;\; 0 \;\; 1 \quad S_1, S_3, S_4, S_6, S_8, S_9, S_{11}, S_{12} = \epsilon$$

Figure 3.1: Example of (a) $C(A)$ and (b) $2dmax(A)$ of the array $A[1,12]$. Red and blue colored nodes are the nodes in $\mathsf{linspine}(5)$ and $\mathsf{rinspine}(5)$ of $C(A)$ respectively.

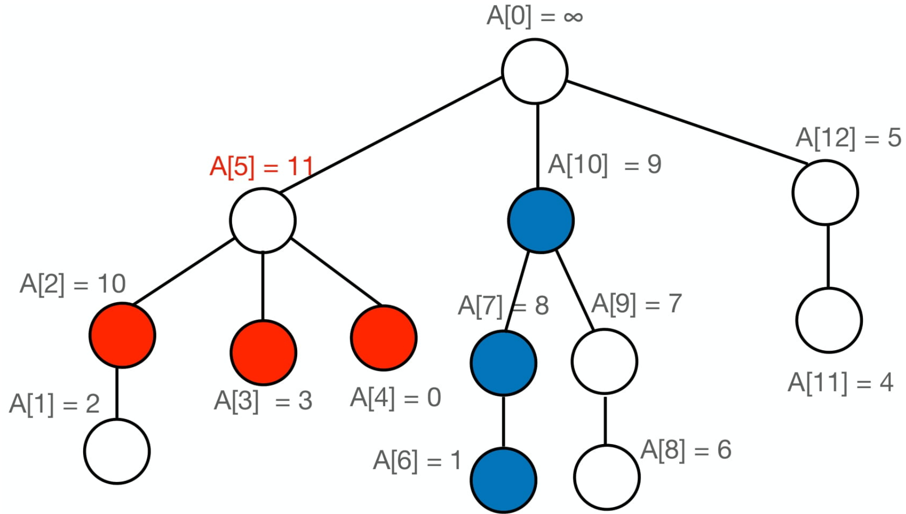


Figure 3.2: $r2dmax(A)$ of the array in Figure 3.1. Red and blue colored nodes correspond to the nodes in $C(A)$ in Figure 3.1 with the same colors.

reduce the depth of $2dmax(A)$, Ferrada and Navarro [46] considered *rightmost-path* $2dmax(A)$ (denoted as $r2dmax(A)$), which can be obtained from $C(A)$ by applying τ_1 (first-child, next-sibling) transformation [1]. Note that the original $2dmax(A)$ can be obtained from $C(A)$ by applying τ_4 (previous-sibling, last-child) transformation [1]. Davoodi et al. [1] also noted that the i-th position of A corresponds to the node in $r2dmax(A)$ whose postorder number is i (see Figure 3.2 for an example). For example, if A is a strictly decreasing array from 1 to n, the depths of $2dmax(A)$ and $r2dmax(A)$ are n and 1, respectively. Ferrada and Navarro [46] showed that one can answer RMQ queries on A as using $r2dmax(A)$ with $o(n)$-bit auxiliary structures, which are different from the structures used for answering the same query using $2dmax(A)$. Baumstark et al. [45] showed that $r2dmax(A)$ is isomorphic to $2dmax(\overleftarrow{A})$, where $\overleftarrow{A}$ is an array of size n constructed by reversing the all elements of A. Thus, one can simulate the $\mathsf{RMQ}(i, j)$ on A using $r2dmax(A)$ by answering $\mathsf{RMQ}(n+1-j, n+1-i)$ on $\overleftarrow{A}$ using $2dmax(\overleftarrow{A})$ (note that in this case, one breaks the ties with rightmost policy when constructing $2dmax(\overleftarrow{A})$, i.e., among all the equal

elements in a range, the rightmost element is considered as the largest).

To implement the data structure of Theorem 3.1, we first check the depth of $2dmax(A)$ and $2dmax(\overleftarrow{A})$ at pre-processing step, and maintain the one with smaller depth (along with the auxiliary structures).

3.3 Space-efficient encoding of **RT2Q** with fast construction

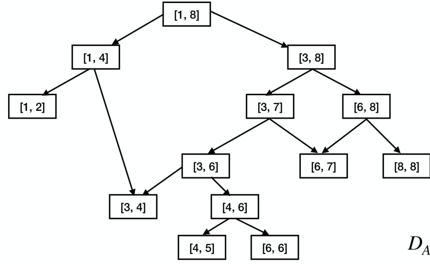
In this section, we consider an encoding whose space usage is close to Gawrychowski and Nicholson's optimal encoding [25] in practice while supporting $O(n)$ construction time. Note that in this encoding, efficient query time is not of concern (to answer RT2Q, we need to reconstruct the entire encoding in the worst case, which takes $O(n)$ time).

Given an array A of size n, as in Section 3.2, we first construct $D(T)$ (DFUDS of $2dmax(A)$) using $2n$ bits, to support RMQ on A. Next, to answer the position of the second largest element in the query range, we define a DAG D_A. A similar DAG structure is first proposed by Jo et al. [3] to support RT2Q queries on $2 \times n$ array, but we can easily adapt it to the 1-dimensional array A as follows (see Figure 3.3 for an example).

- Each node p of D_A is labeled with a closed interval I_p.
- The root of D_A is labeled with $[1, n]$.
- A node $p \in D_A$ is a leaf node if and only if the length of the interval I_p is less than three.
- For any node $p \in D_A$ with $I_p = [p_l, p_r]$, suppose $\mathsf{RT2Q}(p_l, p_r)$ is $\{a, b\}$ with $a < b$. Then p has a left child with label $[p_l, b-1]$, and a right child with label $[a+1, p_r]$.

In the rest of the chapter, for any node $p \in D_A$ with label $I_p = [p_l, p_r]$, we use $\mathsf{RT2Q}(I_p)$ (resp. $\mathsf{RMQ}(I_p)$) and $\mathsf{RT2Q}(p_l, p_r)$ (resp. $\mathsf{RMQ}(p_l, p_r)$) inter-

$A[1,8] = 2 \;\; 10 \;\; 3 \;\; 0 \;\; 11 \;\; 1 \;\; 8 \;\; 9$



Label	[1, 8]	[1, 4]	[3, 8]	[1, 2]	[3, 7]	[6, 8]	[3, 6]	[6, 7]	[8, 8]	[3, 4]	[4, 6]	[4, 5]	[6,6]
s_i	0	1	1	-1	1	-1	0	-1	-1	-1	1	-1	-1

$S_A = 0 \;\; 1 \;\; 1 \;\; 1 \;\; 0 \;\; 1$

Figure 3.3: Example of D_A and its encoding S_A of the array $A[1,8]$.

changeably. The following lemma shows that we can answer RT2Q queries by storing $D(T)$ and D_A.

Lemma 3.3 ([3]). *Given an array $A[1,n]$ of size n, the following properties hold:*

1. *For any two distinct nodes p and q, RT2Q$(I_p) \neq$ RT2Q(I_q).*

2. *For any range $[i,j]$ with $1 \leq i \leq j \leq n$, there exists a node $p^{[i,j]} \in D_A$ such that RT2Q$(i,j) =$ RT2Q$(I_{p^{[i,j]}})$. Specifically, $p^{[i,j]}$ is the last node in D_A according to the level-order traversal which satisfies $[i,j] \subset I_p$.*

3. *The node $p^{[i,j]}$ can be found in $O(|D_A|)$ time by traversing the nodes in D_A according to the level order [2].*

Now we describe how to encode D_A. The basic idea is that for any node $p \in D_A$, if I_p and RT2Q(I_p) are given, one can decode the labels of p's children

[2]For any node $p \in D_A$ we define a level of p as the number of edges in the longest path from the root of D_A to p.

by the properties of D_A. Also, if the answer of the $\mathsf{RMQ}(i,j)$ is given, then at most one additional bit (which indicates whether the position of the second largest element is left or right to the of $\mathsf{RMQ}(i,j)$) is necessary to decode the position of the second largest element in $A[i,j]$. Now for each node $p \in D_A$, we define $s_p \in \{-1,0,1\}$ as follows. We set $s_p = -1$ if (i) p is a leaf node, or (ii) $\mathsf{RMQ}(I_p)$ is the leftmost or the rightmost point of I_p, i.e., $s_p = -1$ if and only if the p is a leaf node or one can answer $\mathsf{RT2Q}(I_p)$ by using only RMQ. Otherwise, we set $s_p = 0$ (resp. 1) if the position of the second largest element of $A[i,j]$ is to the left (resp. right) of $\mathsf{RMQ}(I_p)$. Let S_A be the sequence obtained by concatenating all s_i with $s_i \neq -1$ according to the level order of D_A (see Figure 3.3 for an example). One can decode all the labels of the nodes in D_A using S_A and $D(T)$. The following lemma shows that the size of S_A is at most $5n/3$; and thus the encoding uses at most $2n + 5n/3 = 11n/3 \sim 3.67n$ bits in total.

Lemma 3.4. *For any array $A[1,n]$ of size n, $|S_A| \leq 5n/3$.*

Proof. To construct S_A, suppose the encoder visits the node p (according to the level-order traversal) where $s_p \neq -1$. Then the position $i \in \{1,2,\dots,n\}$ is *picked* at the node p if and only if i is the position of the second largest element in $\mathsf{RT2Q}(I_p)$, where $I_p = [p_l, p_r]$ (thus, each node in D_A picks at most one position). Furthermore, Jo et al. [3] showed that after all nodes in D_A are traversed, any position in $\{1,2,\dots,n\}$ is picked at most twice. Now we claim that if any two consecutive positions i and $i+1$ (for some $i > 1$) are picked twice, then the position $i-1$ is picked at most once (in other words, any three consecutive positions are picked at most 5 times). If the claim is true, the theorem statement holds since $|S_A|$ is the total number of times any position is picked.

To prove the claim, without loss of generality, assume that both i and $(i+1)$ are picked twice, and $A[i] > A[i+1] > A[i-1]$ (which means the position i

is picked earlier than the position $(i+1)$). Also, suppose i is picked at the nodes p and q where $I_p = [p_l, p_r]$ and $I_q = [q_l, q_r]$. Then $I_p \not\subset I_q$ and $I_q \not\subset I_p$ by Lemma 3.3 (otherwise, $\mathsf{RT2Q}(I_p) = \mathsf{RT2Q}(I_q)$). Now consider the case $p_l < q_l < i < p_r < q_r$ (other case can be handled similarly). Then, by the construction of D_A, both $A[q_l - 1]$ and $A[p_r + 1]$ are larger than $A[i]$. Thus, $\mathsf{RT2Q}(I_p) = \{(q_l - 1), i\}$ and $\mathsf{RT2Q}(I_q) = \{i, (p_r + 1)\}$. Furthermore, $(i+1)$ can only be picked at (i) a descendant of the right child of p (which is same as the left child of q) and (ii) a descendant of the right child of q. Also, any node whose label contains both i and $(i+1)$ cannot pick the position $(i-1)$ after picking $(i+1)$ since $A[i] > A[i+1] > A[i-1]$, and any node in Case (ii) does not contain $(i-1)$. Thus, the position $(i-1)$ can only be an answer of $\mathsf{RT2Q}$ at either (a) a descendant of the left child of p, or (b) a descendant of q which contains i but not $i+1$. However, since i is always the rightmost position of any node in Case (b) which is not covered by Case (a). Thus, $(i-1)$ can be picked at most once, which proves the claim.

□

To construct the encoding, we first construct $D(T)$ and the auxiliary structures, which can be constructed in $O(n)$ time and supports RMQ on A in $O(1)$ time [19]. Next, we construct S_A by encoding the label of nodes of D_A in the level-order. This takes $O(1)$ time per node (by making use of the RMQ in $O(1)$ time). Since D_A has $O(n)$ nodes [3], S_A can be constructed in $O(n)$ time. Note that auxiliary structures for supporting RMQ can be deleted after constructing D_A.

Example 3.3. *To answer the query $\mathsf{RMQ}(4,7)$ on the array $A[1,8]$ in Figure 3.3, we first decode the left and right children of the root node $[1,8]$ in D_A by referring to the first element in S_A, which is 0. This indicates $\mathsf{RMQ}(1,4)$ is the position of the second largest element in $A[1,8]$. After that, decode the rest*

of the nodes according to the level-order traversal of D_A. Then after decoding the children of the node $[3,7]$, we can conclude $p^{[4,7]} = [3,7]$ (note that $[3,7]$ is the last node in D_A according to the level-order traversal which contains $[4,7]$). Since $s_{[3,7]} = 1$, we can answer the position of the second largest element in $A[4,7]$ is $\mathsf{RMQ}(5,7) = 7$.

3.4 Experimental results

Our data structures were implemented in C++ (compiled by g++ 9.3.0 with O3 optimization), and all the experiments were done on the Desktop PC (Intel i7-9900KS CPU with 128GB of RAM). We use the input array $A[1,n]$ which stores 32-bit unsigned integers. We consider three different types of input arrays: (a) *random*, (b) *pseudo-increasing*, and (c) *pseudo-decreasing*, where each $A[i]$ is randomly generated from the range (a) $[1,n]$, (b) $[i-\delta, i+\delta]$, and (c) $[n-i-\delta, n-i+\delta]$, respectively for a given parameter $\delta > 0$.

We also consider the following the real-world datasets:

- Weather: The data of temperature in Seoul for the year 2021. We collected the data from Open MET Data Portal by Korea Meteorological Administration (KMA)[3] (array size $n = 524,236$).
- DNA: LCP array of the text DNA from Pizza&Chilli Corpus[4] – we constructed the LCP array for a prefix of the DNA sequence of length n ($n = 2 \times 10^6$).
- Google: Degree sequence of the web graph of Google from Stanford Large Network Dataset Collection[5] ($n = 875,713$).

[3] `https://data.kma.go.kr/resources/html/en/aowdp.html`
[4] `http://pizzachili.dcc.uchile.cl/texts.html`
[5] https://snap.stanford.edu

- Youtube: Degree sequence of the social network graph of Google from Stanford Large Network Dataset Collection ($n = 1,134,890$).

3.4.1 Experimental results on data structures for answering RT2Q efficiently

In this section, we compare the space usage (bits per element) and query time (μs) of our encoding structure of Section 3.2 (referred to as R2MQ-ENCODING) with the following four indexing data structures for answering RT2Q: (i) A + RMQ encoding of Fisher and Heun [19] (FH-DFUDS), (ii) A + RMQ encoding of Ferrada and Navarro [46] (FN-BP), (iii) A + RMQ encoding of Baumstark et al. [46] (BGHL-BP), and (iv) A + Fischer and Huen's indexing data structure for RMQ queries [19] (FH-INDEXING). Note that the encoding of (i) uses $D(2dmax(A))$, whereas the encodings of (ii) and (iii) use the BP of $2dmax(A)$. For (i) and (ii), we use the implementation of Ferrada and Navarro [46][6], and for (iii), we use the implementation of Baumstark et al. [46][7]. Finally for (iv), we use our own implementation.

To support RMQ on A, and navigation queries on $2dmax(A)$ except depth, we use sdsl-lite [61] to support rank, select, findopen, findclose (for presibling operation), and ± 1RMQ on $D(2dmax(A))$. Note that for findopen, findclose and ± 1RMQ, we use a simplified RMM-tree [39] which maintains only the *min* field for these queries. For computing depth(k) queries, we use the same data structure for computing $L(k)$. More precisely, if $L(k)$ is stored in E, we also store depth(k) in a separate array E' at the same position (thus, the same bit array B can be used for $L(k)$ and depth(k)). For computing depth(k), we perform the parent query iteratively until we find the node whose depth is stored in E'. Note that we do not keep any additional data structures for both depth and $L(K)$ queries if the depth of the tree is less than $\lceil \log n \rceil$.

Since the overhead for depth and $L(k)$ is the main drawback of our imple-

[6]Code is available at https://github.com/hferrada/rmqFischerDFUDS and https://github.com/hferrada/rmq.

[7]Code is available at https://github.com/kittobi1992/rmq-experiments.

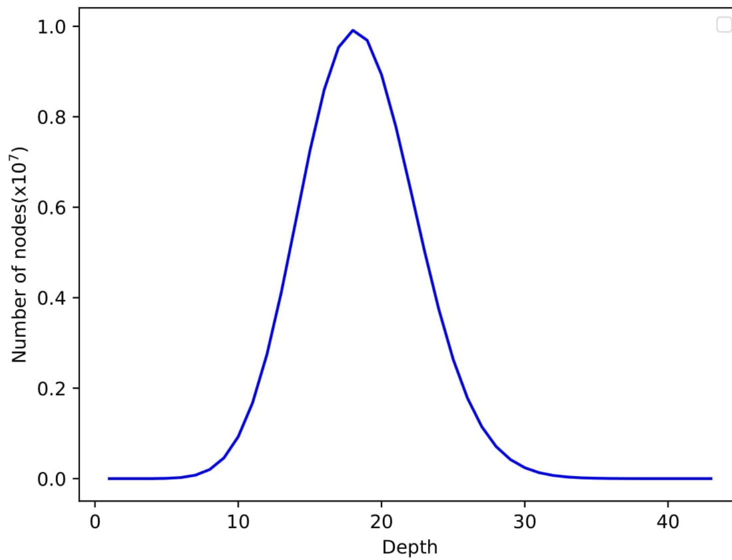


Figure 3.4: The distribution of the depth of nodes in $2dmax(A)$.

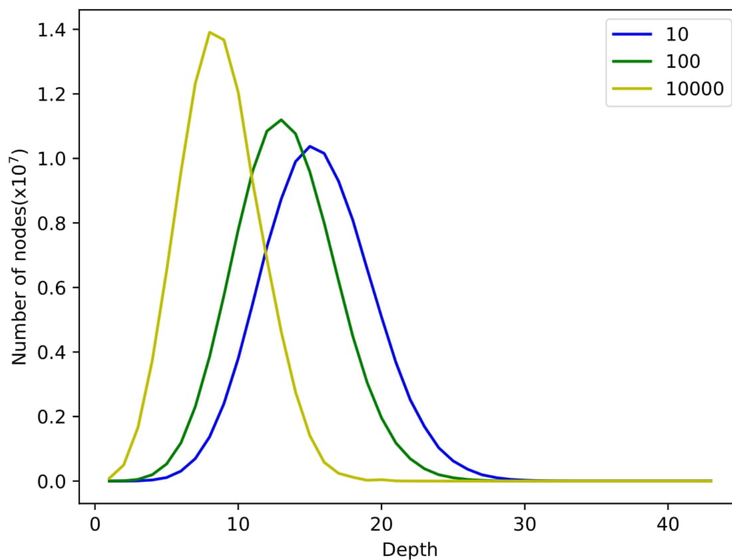


Figure 3.5: The distribution of the depth of the nodes in $2dmax(A)$ which correspond to the RMQ of A.

mentation, we do an empirical evaluation to decide the sizes of E and E'. When A is a randomly generated array of size 10^8, the depth of $2dmax(A)$ is less than 50 in most cases (in theory, the expected depth of $C(A)$ for a random array A is about $\Theta(\log n)$, and the depth of $2dmax(A)$ is at most the depth of $C(A)$ [62]), and the depth of nodes has close to the normal distribution (see Figure 3.4). Next, we evaluate the distribution of the depth of the nodes $2dmax(A)$ which correspond to the RMQ of A (note that we only need the value of $\mathsf{depth}(k)$ and $L(k)$ when $k = \mathsf{RMQ}(i,j)$ for some $1 \le i \le j \le n$). As shown in Figure 3.5, when the query range is 10^4, the depth of all the nodes corresponding to RMQ is less than half of the depth of $2dmax(A)$. Furthermore, even for the small query ranges (10), the depth of 95.5% of the nodes is less than half of the depth of $2dmax(A)$. From the distribution of the nodes corresponding to RMQ of A, we consider two greedy algorithms for selecting the nodes to be stored in E and E'. Suppose we want to allot at most N nodes to be stored in E and E', and let D_N be the smallest depth where the number of the nodes with depth D_N is more than N (if there is no such depth, D_N is the depth of $2dmax(A)$); and let $d = \min(\lfloor D/2 \rfloor, D_N - 1)$, where D is the depth of $2dmax(A)$. Then greedy algorithm 1 (GA1) repeats the following procedure from $i = 0$ to d:

1. Choose all the nodes with depth i, if the total number of chosen nodes is at most N.

2. Increase i by 1.

Similarly, greedy algorithm 2 (GA2) repeats the first step of the above procedure by decreasing the value i from d to 0.

We evaluate the time for answering RT2Q with different amounts of space allotted for E and E'. As shown in Figure 3.6, increasing the allotted space does not significantly improve the query time when the size of the query range is 10^6 since most of the nodes corresponding to the answer of RMQ are close to the root node. The same tendency is shown for other sizes of query ranges (10, 10^2,

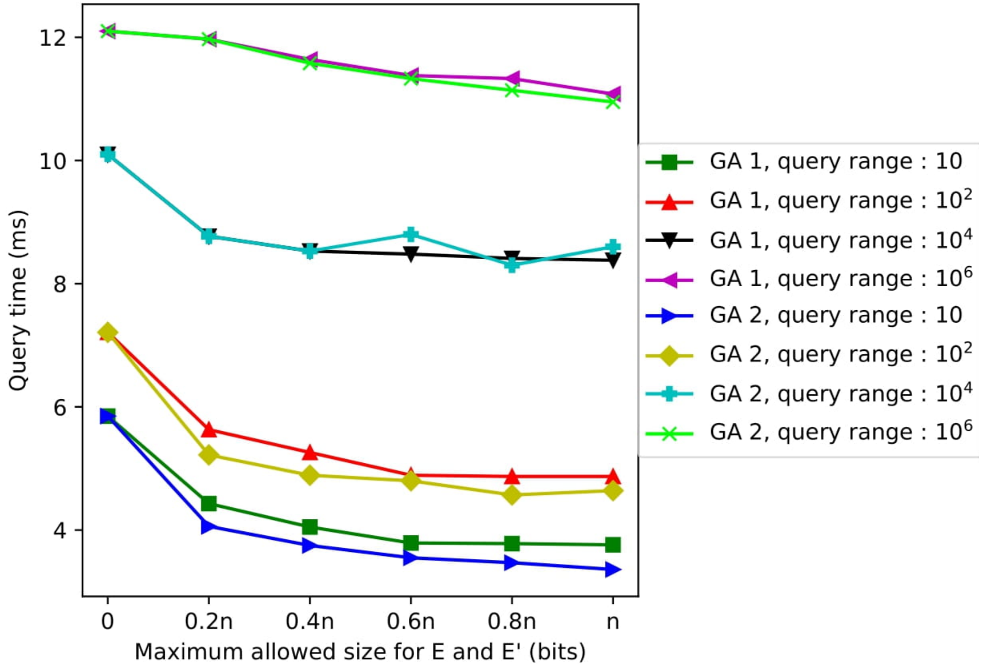


Figure 3.6: Query time based on the allotted space for E and E'.

and 10^4) when allotting more than $0.4n$ bits for E and E', since both GA1 and GA2 cannot significantly increase the number of nodes to be stored (note that the number of nodes increases roughly exponentially with the depth, from 1 to d). In our implementation, we choose GA2 which shows better query time for small query ranges. Also, the space allotted for storing E and E' is determined based on the maximum values (either 8, 16 or 32-bit values) stored in those arrays, as follows. We allot $n/\lfloor \log\log n \rfloor$ bits if the maximum values of E and E' are both at most 2^8 (in this case, we use 8-bit integer arrays for storing these). In general, if the maximum values of E and E' are at most 2^{8c} and $2^{8c'}$, respectively, for some $c, c' \in \{1, 2, 4\}$, then we allot $(\frac{c+c'}{2})n$ bits for storing these arrays. For example, if 32 and 8-bit integer arrays are necessary to store E and E' respectively for an array A of size 10^8, we use $(\frac{4+1}{2}) \cdot n / \lfloor \log\log 10^8 \rfloor = 0.625n$

bits for storing E and E'.

Next, we evaluate the space usage on randomly generated arrays of size $n = 10^6$ to $n = 10^9$ (see Figure 3.7). Our structure uses up to 4.6 and 4.8 bpe (bits per element) for $n = 10^6$ and $n = 10^9$ respectively. This shows that our data structure's average space is not much changed by increasing the array size, like other indexing structures except FH-INDEXING. For FH-INDEXING, each precomputed value needs 32 bits even for an array of size 10^6 (note that $\lceil \log 10^6 \rceil$ is 19), which is wasteful in terms of space. Since the input array is necessary to answer $\mathsf{RT2Q}$ using indexing data structures, our data structure takes at least 7.1 times less space than the existing indexing data structures (each input array in Figure 3.7 takes $32n$ bits). Next, we fix the size of the (randomly-generated) input array to be 10^9, and evaluate query time for various query ranges (see Figure 3.8). Our data structure and FH-DFUDS are highly dependent on the query range, compared to BP-based indexing structures. This is because, in the implementation, the running time of $\mathsf{findopen}$ operation is an increasing function of the range (note that $\mathsf{findopen}$ operation is used for computing RMQ, depth, and $L(k)$ when 2d-max heap is represented by DFUDS). Interestingly, when the query range is changed from 10^6 to 10^8, the query time of FH-DFUDS increases much more rapidly than our data structures. This shows that the overhead for answering RMQ on FH-DFUDS is more than computing $L(k)$ and $\mathsf{depth}(k)$ for the nodes with small depths. Compared to the fastest indexing solutions (BGHL-BP and FH-INDEXING), our data structure shows up to 10 and 4.1 times slower query times when the query range is 10 and 10^8 respectively. Note that the space usage of our encoding is roughly 7 times smaller than the space usage of the indexing structures (including the space usage of the input array).

Next, we evaluate the space and query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ with various δ values, with the size of the query range fixed to $\sqrt{n}$ (see Figures 3.9, 3.10, 3.11, and 3.12.). Note that when A is pseudo-increasing (resp. pseudo-decreasing), $\overleftarrow{A}$ is pseudo-decreasing

(resp. pseudo-increasing). Thus, our data structure and BGHL-BP show similar space usage and query time on both pseudo-increasing and pseudo-decreasing arrays (note that for FH-DFUDS, the query time on pseudo-increasing arrays is up to 3 times slower than the query time on pseudo-decreasing arrays because of the larger depth of $2dmax(A)$. Note that the average distance between two matching parenthesis in DFUDS decreases proportional to the depth of $2dmax(A)$). The space usage of our data structure is not much affected by δ (up to 4.03 bpe to 4.39 bpe) since we do not maintain the arrays E and E' for all the cases (the depth of $2dmax(A)$ is still less than $\log 10^9 \sim 30$ even for large $\delta = 10^8$). Also, the query time for our data structure increases with δ because the average depth of the nodes corresponding to RMQ is increases with δ. Overall, our data structure shows better time-space tradeoffs (takes up to 7.5 times less space while spending up to 4.2 times slower the query time) than all other indexing data structures in the evaluation.

Next, for $\delta = 10^3$ and 10^6, we evaluate the query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ for various query ranges (see Figures 3.13, 3.14, 3.15, and 3.16). Again, DFUDS-based implementations (R2MQ-ENCODING and FH-DFUDS) highly depend on the depth of $2dmax(A)$ and query ranges because of findopen operation, whereas BP-based implementations (FN-BP and BGHL-BP) have similar results compared to the random array case. Especially compared to the random array case, our data structure supports much (up to 2.2 times) faster queries on pseudo-increasing and decreasing arrays for most query ranges since there is no additional overhead for accessing E and E' in both cases.

Finally, we evaluate the space and query time for the inputs from the real dataset for various query ranges (see Figures 3.17, 3.18, 3.19, 3.20, and 3.21). For all the input datasets except for the dataset Weather with query range 10^3, our data structure shows a better time-space tradeoffs than the other indexing data structures. Since Weather has both long increasing and decreasing runs of

elements, its 2d-max heap has a large depth (more than 1900), in both cases when the input is stored in the forward and reversed order. In this case, the greedy algorithm fails to support fast depth queries for some query ranges.

3.4.2 Experimental results on space-efficient encodings for RT2Q

In this section, we compare the space usage (bits per element) and construction time (ms) of our encoding of Section 3.3 (referred to as DAG-ENCODING) with the following two encodings for answering RT2Q: (i) Davoodi et al. [54]'s encoding without auxiliary structures for efficient queries, that is, BP of $2dmax(A)$ along with the spine sequence of $2dmax(A)$ (SPINE), and (ii) Gawrychowski and Nicholson's optimal encoding [25] (GN). In theory, to construct the DAG-ENCODING in linear time, we need a data structure that answers RMQ queries in $O(1)$ time. However, empirical testing shows that when A is randomly generated, using a simple linear scan to answer RMQ shows faster construction time (up to 2 times) than using the RMQ data structure for constructing D_A. This is because, most of the labels in D_A are short intervals. For pseudo-increasing and decreasing inputs, we use the implementation of Baumstark et al. [45] to support RMQ queries. Since Baumstark et al. [45]'s implementation uses BP of $2dmax(A)$ for answering RMQ queries, in the DAG-ENCODING we use BP (instead DFUDS) of $2dmax(A)$.

Finally for all the encodings, we consider the space usage of the following three quantities: (i) encoding without any compression, (ii) encoding compressed using an entropy-based compression, and (iii) encoding compressed using grammar-based compression. For (ii), we first represent the encodings as the integer sequences from 0 to 255 and compress them with Huffman encoding. For (iii), we use LZW (Lempel–Ziv–Welch) compression algorithm [63] on each encoding. Note that DAG-ENCODING and SPINE are composed of two bit sequences. For these encodings, we compress each sequence separately and measure the total space. Also, note that the encoding GN of Gawrychowski and

Nicholson [25] actually takes at most $3n$ bits without compression, and if we compress this bit sequence using the fact that there are exactly n ones in it, we get the optimal space bound of $2.755n + o(n)$ bits.

We now summarize our experimental results. First, we evaluate the construction time of the encodings without any compression algorithms (see Figures 3.22, 3.23, and 3.24). Note that compression algorithms do not affect the overall construction time other than the randomly generated input with size $n = 10^2$ or 10^3 (in this case, SPINE is up to 5 times slower than the DAG-ENCODING for Huffman encoding). For random inputs, the construction time of GN increases much faster than the other two encodings by increasing the size of inputs, since only GN is an $O(n^2)$-time construction algorithm in the worst case. Also, SPINE shows a faster construction time than DAG-ENCODING since each input position is accessed at most once to construct the spine sequence of $2dmax(A)$ (in D_A, the labels of two nodes can have large overlaps).

When the input is a pseudo-increasing array, GN shows the fastest construction time for small δ, and faster construction time than DAG-ENCODING for all δ. To construct GN, one needs to construct n sequences $P_1, P_2, \ldots, P_n$ where $P_i[j]$ is the smaller value between 2 and the number of elements larger than $A[j]$ in $A[j+1, i]$. Thus, when δ is small, one can obtain P_{i+1} from P_i efficiently since most of the values in P_i are already 2. However, when the input is pseudo-decreasing, one needs to compute almost every value in P_{i+1}. Thus, the construction time of GN is much slower than the other two encodings (~ 30 times slower than DAG-ENCODING when $\delta = 10^5$) for pseudo-decreasing inputs. For SPINE, the construction time is only affected by the input size and not by the types of input which only affect the shape of $2dmax(A)$. The construction time of DAG-ENCODING increases for pseudo-increasing and decreasing inputs with increasing δ since the array resembles more closure to a random array.

Next, we evaluate the encoding sizes with Huffman encoding and LZW

compression. Without any compression (see Figures 3.25, 3.26, and 3.27), DAG-ENCODING shows the smallest size for all types of inputs (up to 2.78 bpe), compared to the SPINE (up to 3.33 bpe) and GN (up to 3 bpe). Surprisingly, although Lemma 3.4 says the size of DAG-ENCODING can be up to 3.67 bpe, DAG-ENCODING takes less than 3 bpe for all cases. We believe that the analysis of the Lemma 3.4 can be improved by counting the number of leaf nodes in D_A carefully.

When the encodings are compressed using Huffman encoding, DAG-ENCODING still shows the smallest size for most of the inputs (see Figures 3.28, 3.29, and 3.30). Also, GN is more compressible than other encodings since the bit sequence GN, without compression, is represented as a unary code of integers. For random inputs, DAG-ENCODING and SPINE show almost the same sizes compared to their uncompressed form. This is because the BP of $2dmax(A)$ has the same number of open and closed parentheses without long runs, leading to a low compression performance with entropy-based compression algorithms. For pseudo-increasing and decreasing inputs, BP of $2dmax(A)$ is more compressible for small δ since $2dmax(A)$ has long left or right spines, which lead to the long runs of open or closed parentheses, and hence, the spine sequence of $2dmax(A)$ is also more compressible in these cases.

Next, we evaluate the size when the encodings are compressed by LZW compression algorithm (see Figures 3.31, 3.32, and 3.33). In this case, GN achieves the smallest space for most of the inputs. For pseudo-increasing and decreasing inputs, the original sequence of GN has long runs of same integers, which are highly compressible with grammar-based compression algorithms. SPINE and DAG-ENCODING are also more compressible with LZW compression than Huffman encoding for pseudo-increasing and decreasing inputs due to the long runs of open and closed parentheses in the BP of $2dmax(A)$. For random inputs, GN is slightly better than the other two encodings, but all three encodings show similar compression ratios (better than using Huffman encoding when the

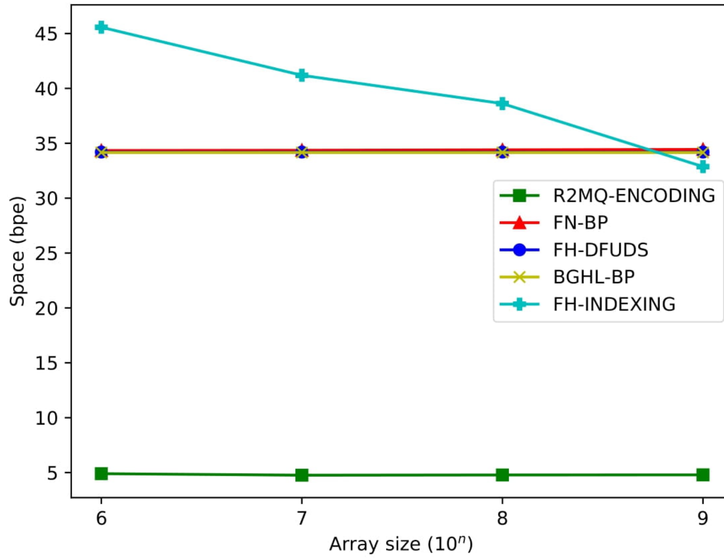


Figure 3.7: Space usage on random arrays of size from 10^6 to 10^9.

input size is more than 10^4). Also, all three encodings show better compression with increasing input size.

Finally, we evaluate the construction time of the encodings and their sizes for the inputs from the real datasets (see Figures 3.34, 3.35, 3.36, and 3.37). For all the inputs, DAG-ENCODING shows the smallest size for all types of inputs with compression algorithms, compared to SPINE and GN, while the construction time is at least 6.8 times faster than GN.

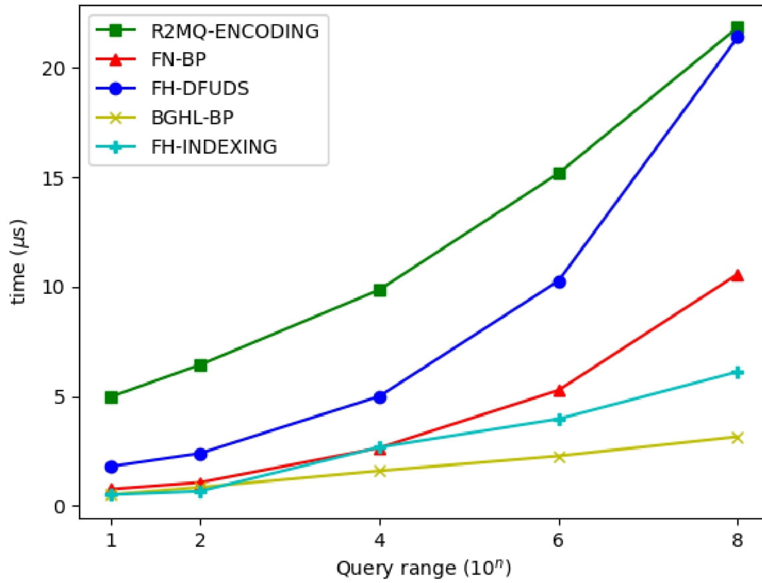


Figure 3.8: Query time on the random array of size 10^9.

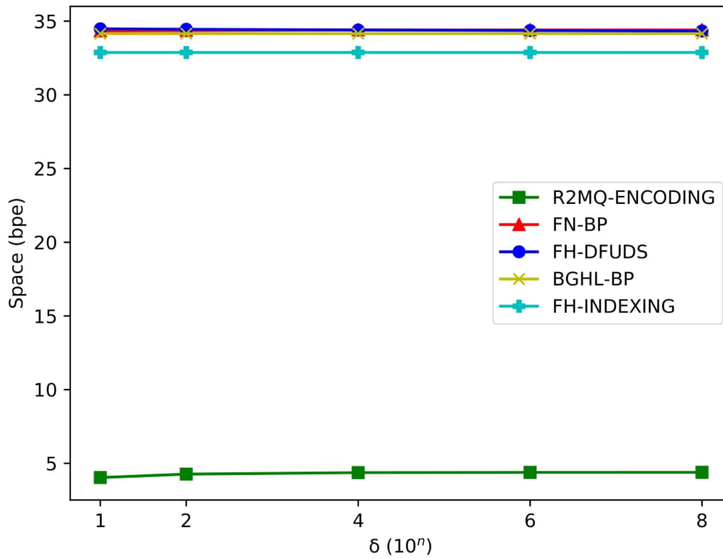


Figure 3.9: Space usage on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

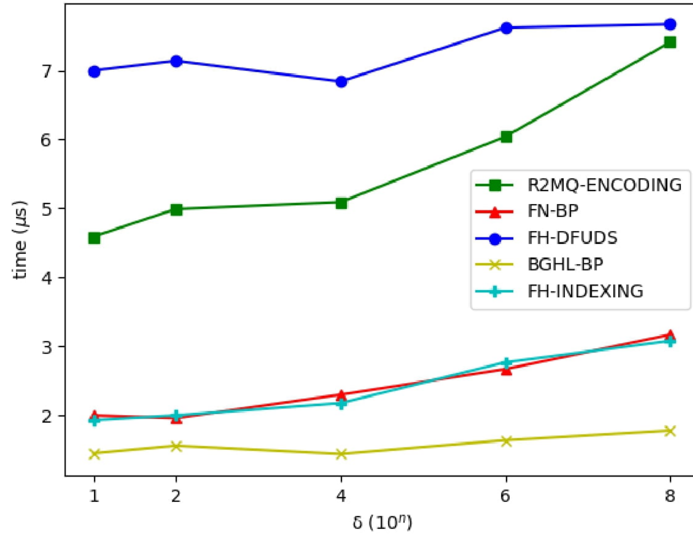


Figure 3.10: Query time on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

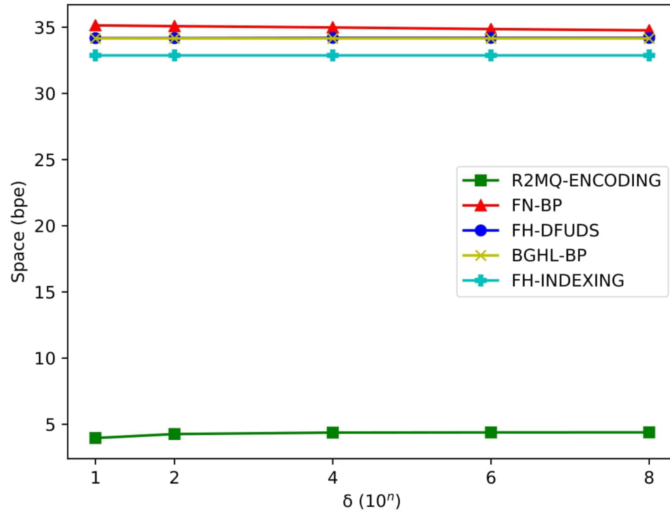


Figure 3.11: Space usage on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

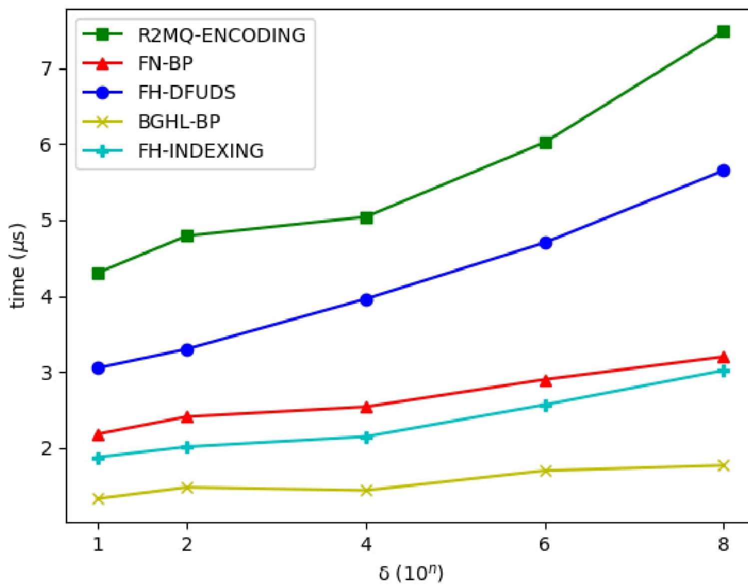


Figure 3.12: Query time on on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

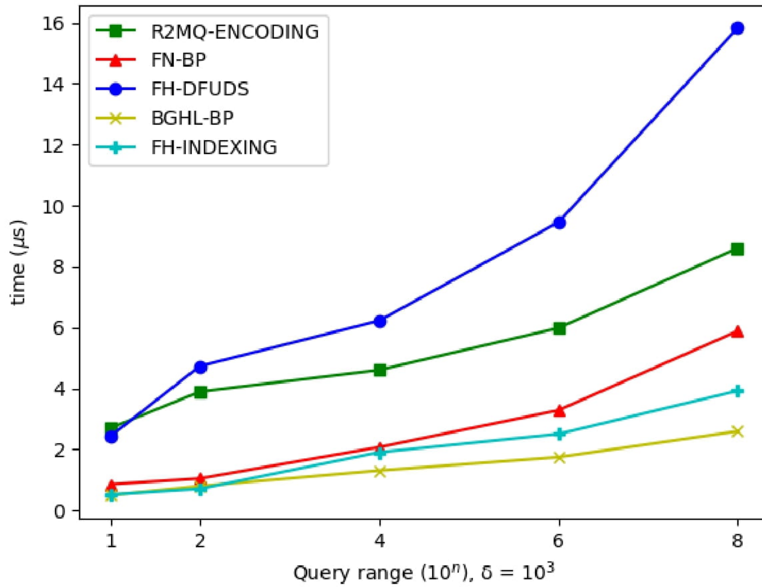


Figure 3.13: Query time on the pseudo-increasing array with $\delta = 10^3$.

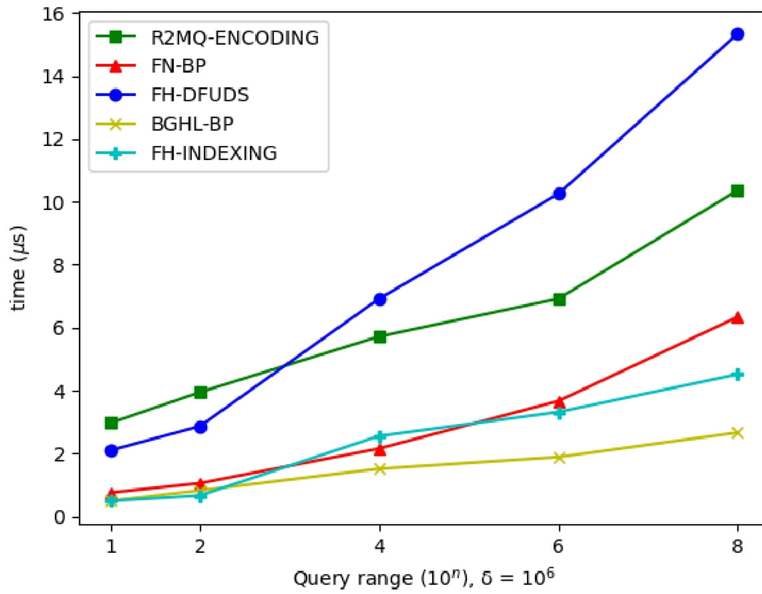


Figure 3.14: Query time on the pseudo-increasing array with $\delta = 10^6$.

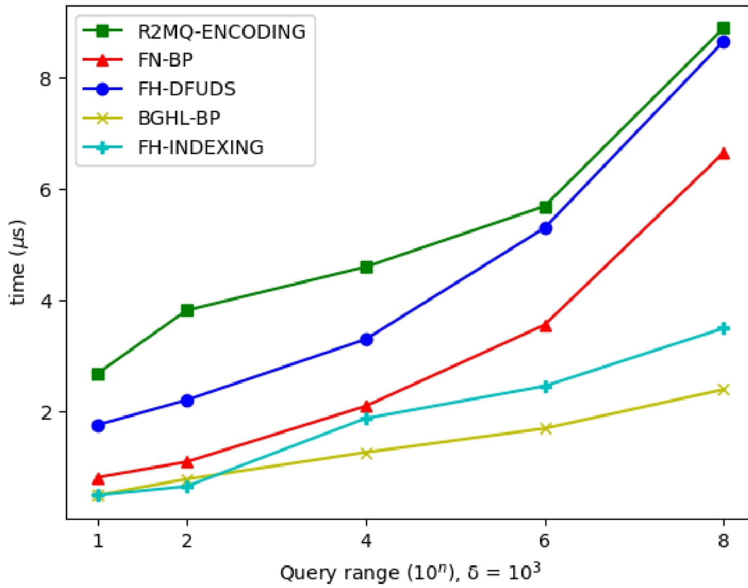


Figure 3.15: Query time on the pseudo-decreasing array with $\delta = 10^3$.

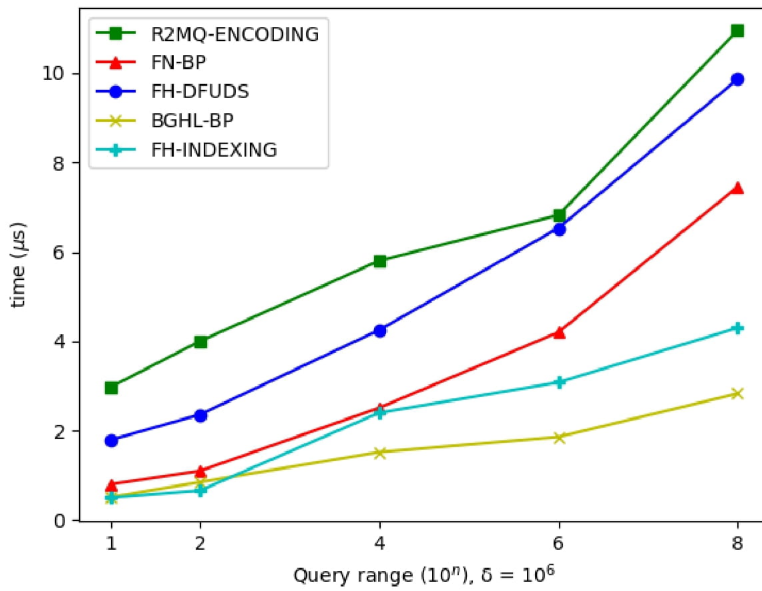


Figure 3.16: Query time on the pseudo-decreasing array with $\delta = 10^6$.

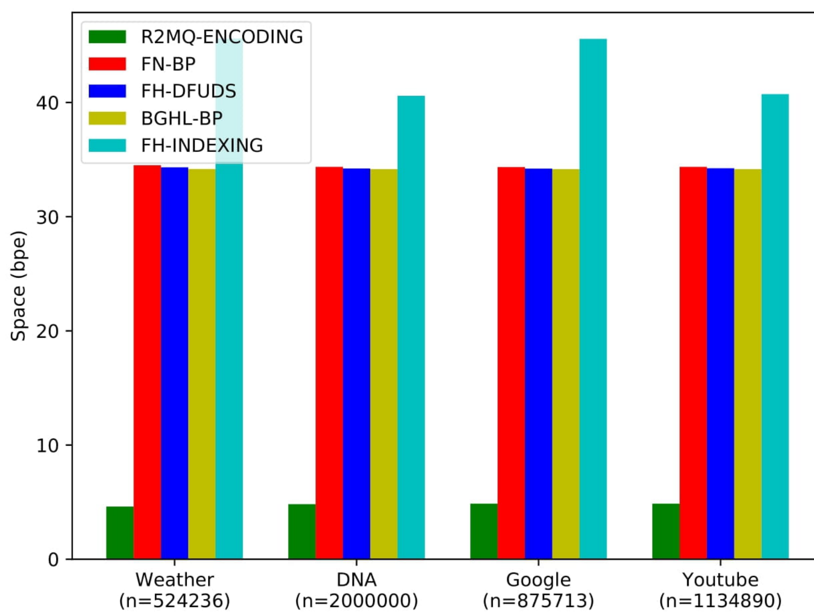


Figure 3.17: Space usage on the inputs from real data sets.

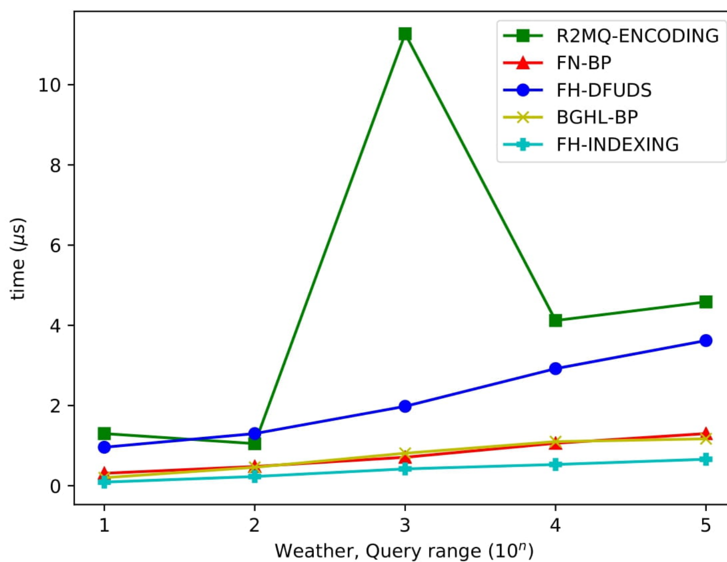


Figure 3.18: Query times on the inputs from real data sets (weather).

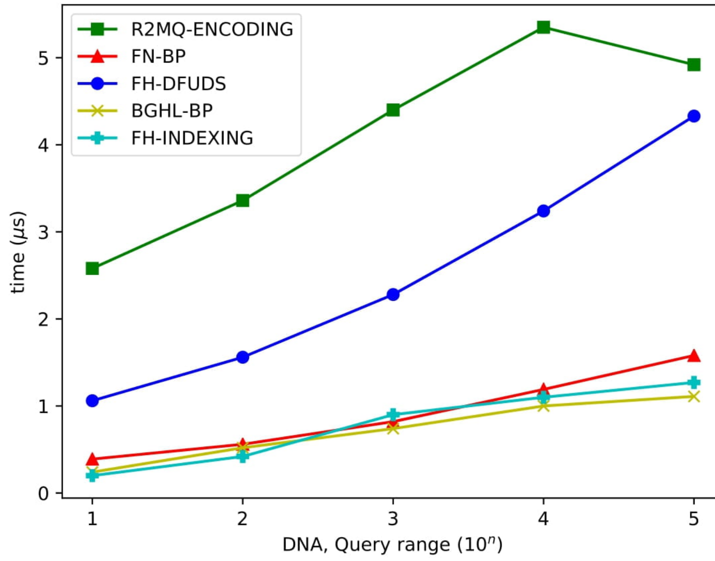


Figure 3.19: Query times on the inputs from real data sets (dna).

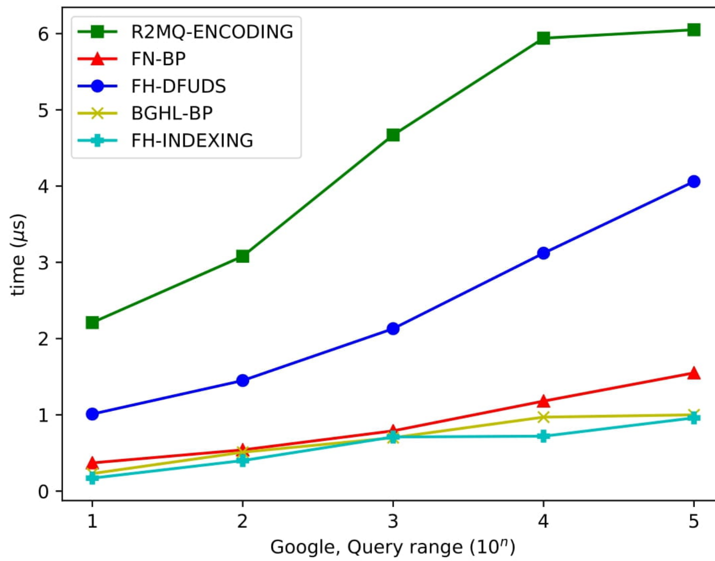


Figure 3.20: Query times on the inputs from real data sets (google).

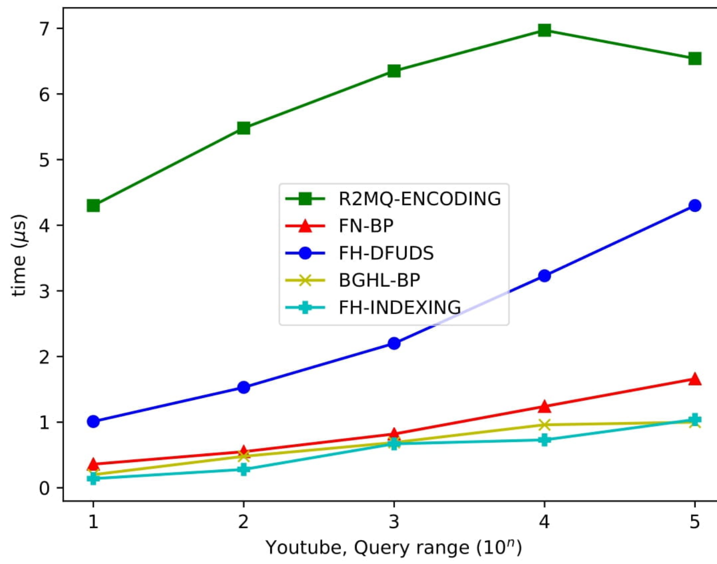


Figure 3.21: Query times on the inputs from real data sets (youtube).

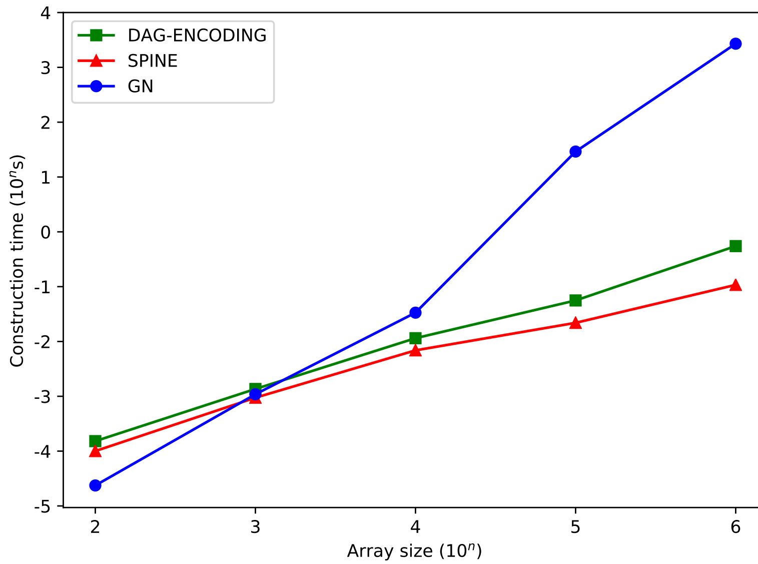


Figure 3.22: Construction time on random array of size from 10 to 10^6.

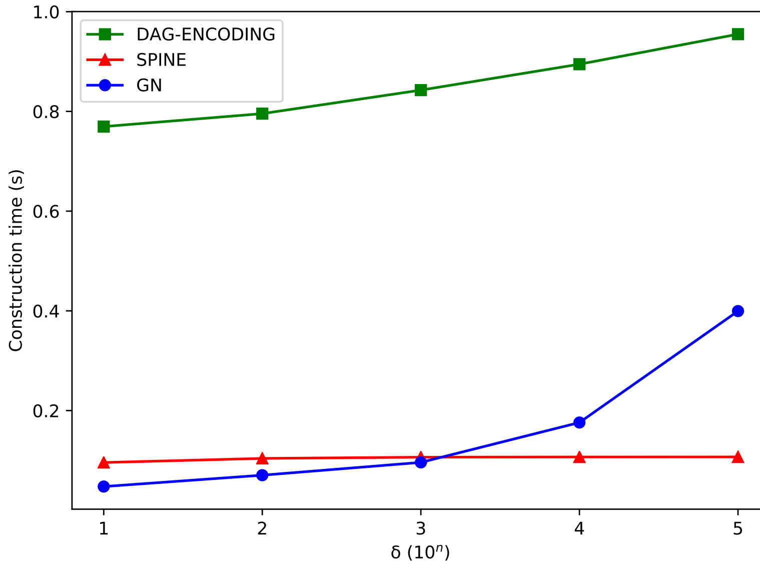


Figure 3.23: Construction time on pseudo-increasing array of size $n = 10^6$.

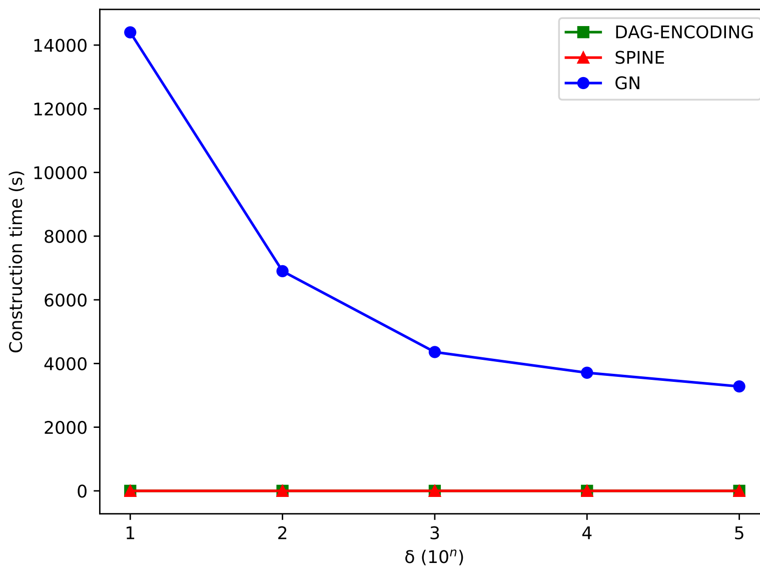


Figure 3.24: Construction time on pseudo-decreasing array of size $n = 10^6$.

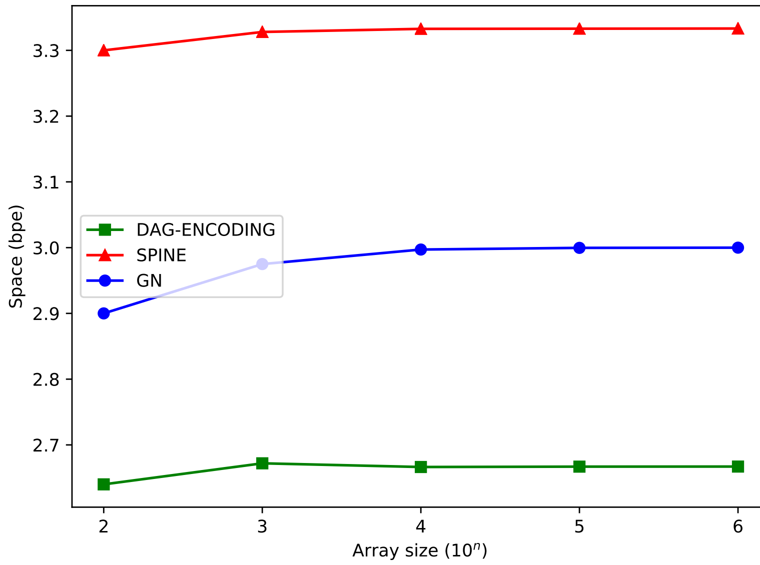


Figure 3.25: Size of the encodings (without compression) on random array of size from 10 to 10^6.

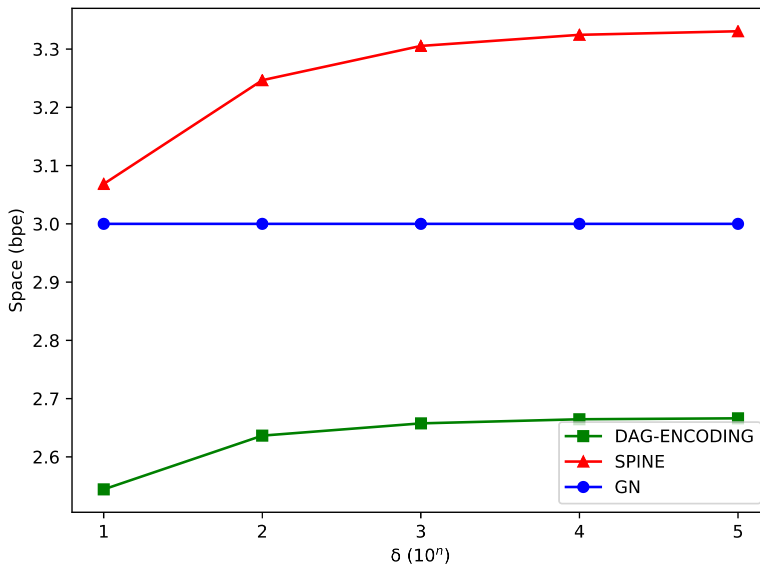


Figure 3.26: Size of the encodings (without compression) on pseudo-increasing array of size $n = 10^6$.

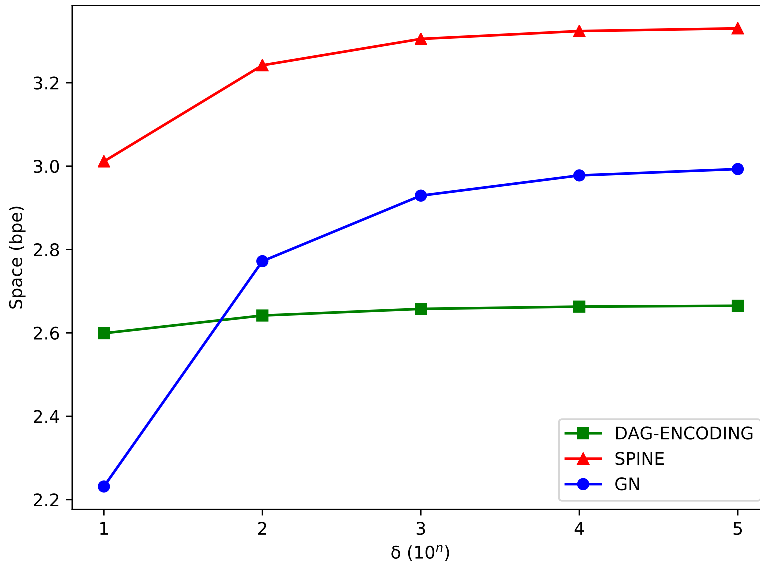


Figure 3.27: Size of the encodings (without compression) on pseudo-decreasing array of size $n = 10^6$.

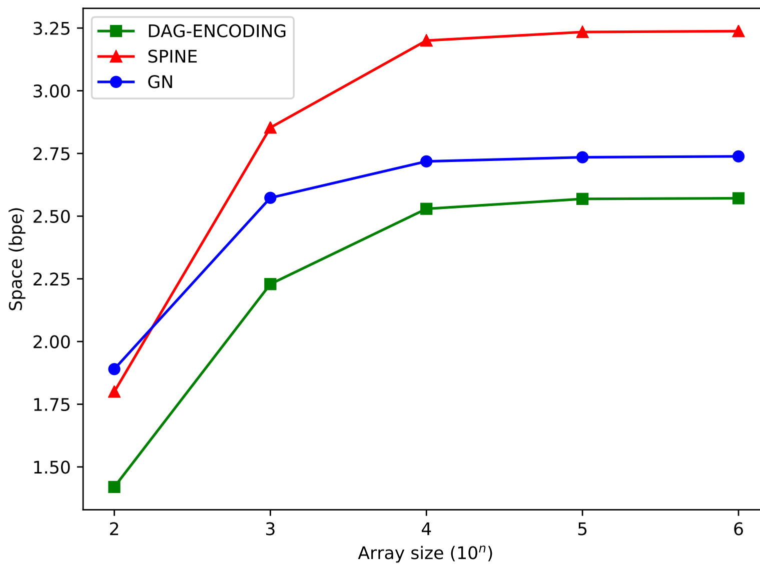


Figure 3.28: Size of the encodings (with Huffman encoding) on random array of size from 10 to 10^6.

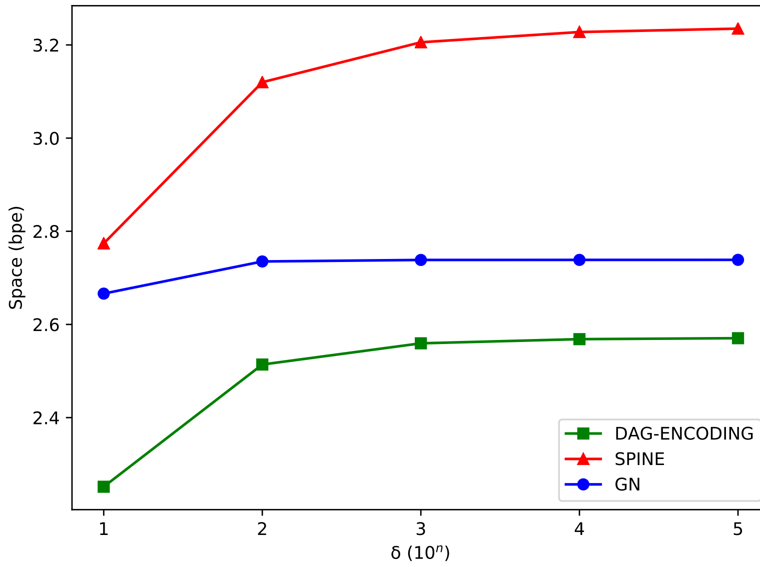


Figure 3.29: Size of the encodings (with Huffman encoding) on pseudo-increasing array of size $n = 10^6$.

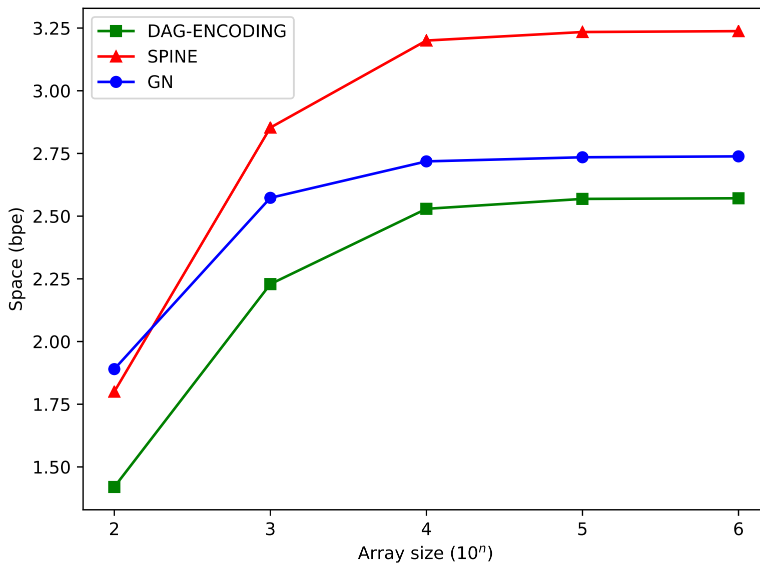


Figure 3.30: Size of the encodings (with Huffman encoding) on pseudo-decreasing array of size $n = 10^6$.

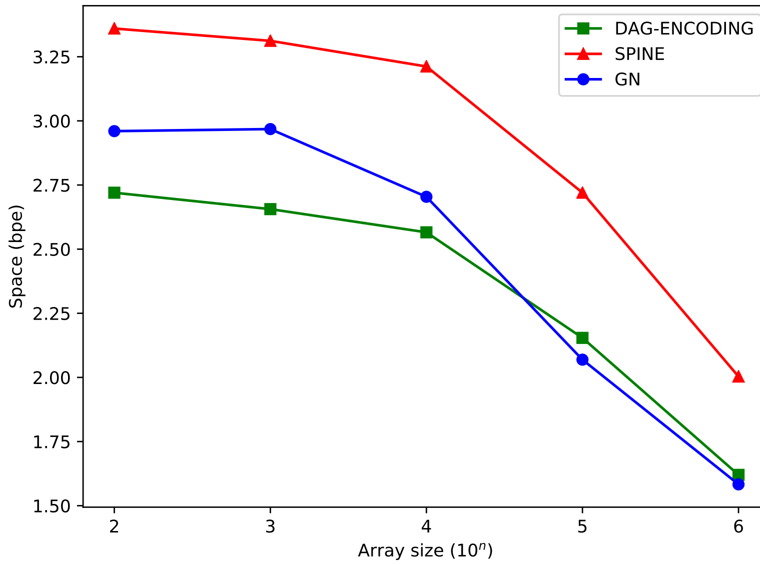


Figure 3.31: Size of the encodings (with LZW compression) on random array of size from 10 to 10^6.

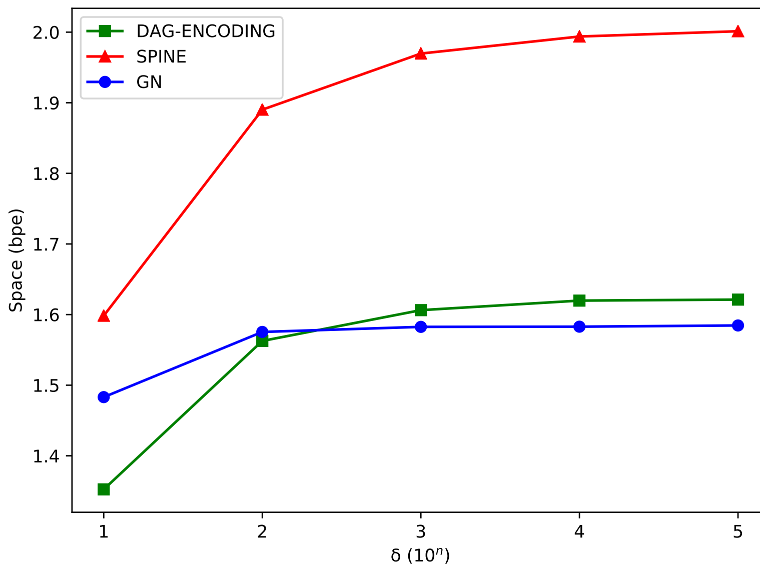


Figure 3.32: Size of the encodings (with LZW compression) on pseudo-increasing array of size $n = 10^6$.

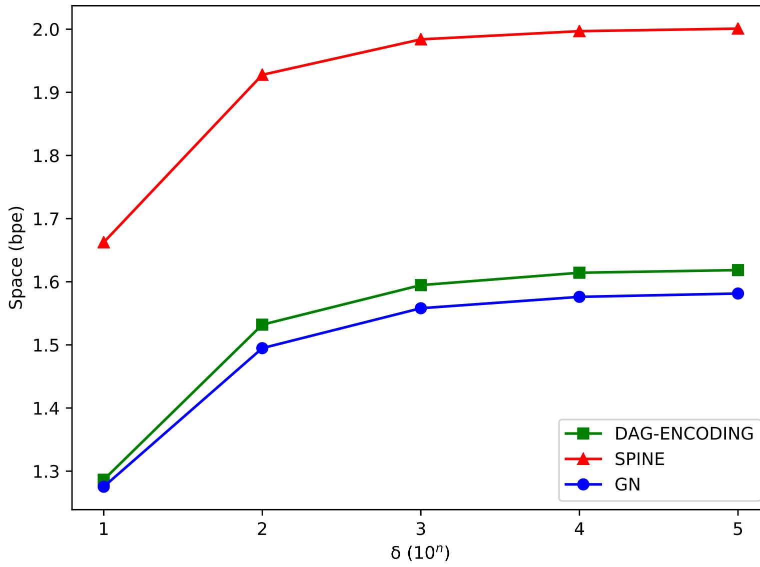


Figure 3.33: Size of the encodings (with LZW compression) on pseudo-decreasing array of size $n = 10^6$.

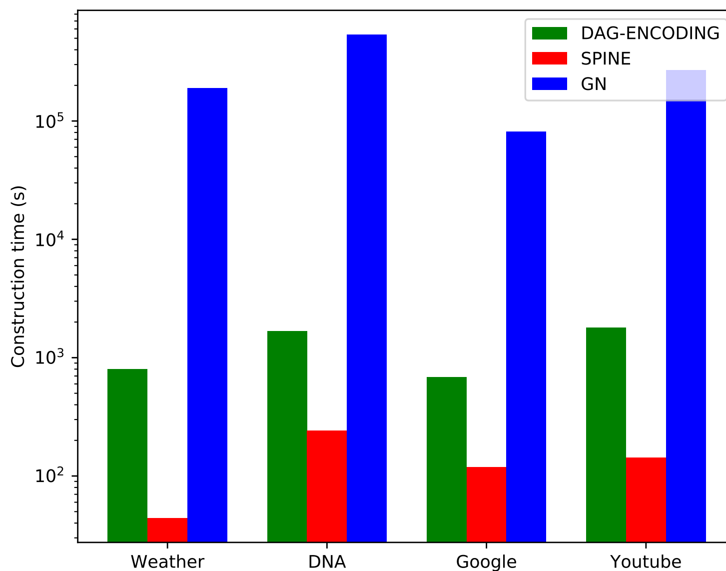


Figure 3.34: Construction time of the encodings on the inputs from real data sets.

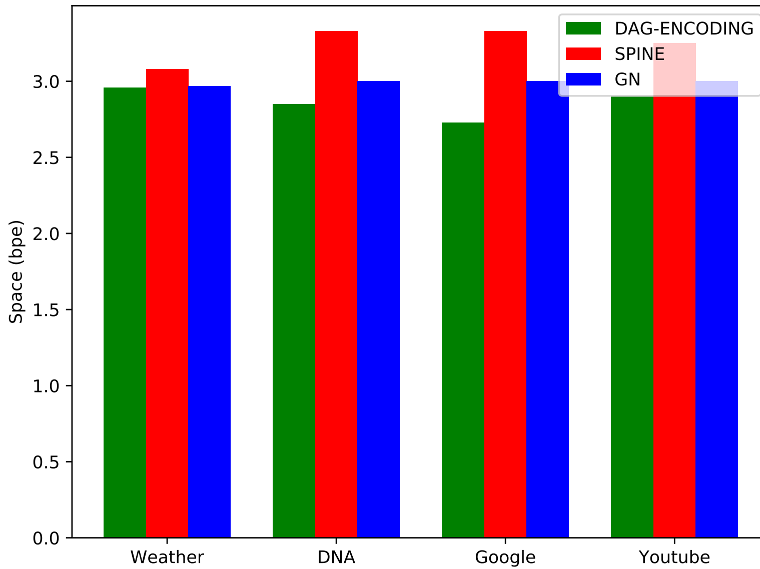


Figure 3.35: Space usage (original) of the encodings on the inputs from real data sets.

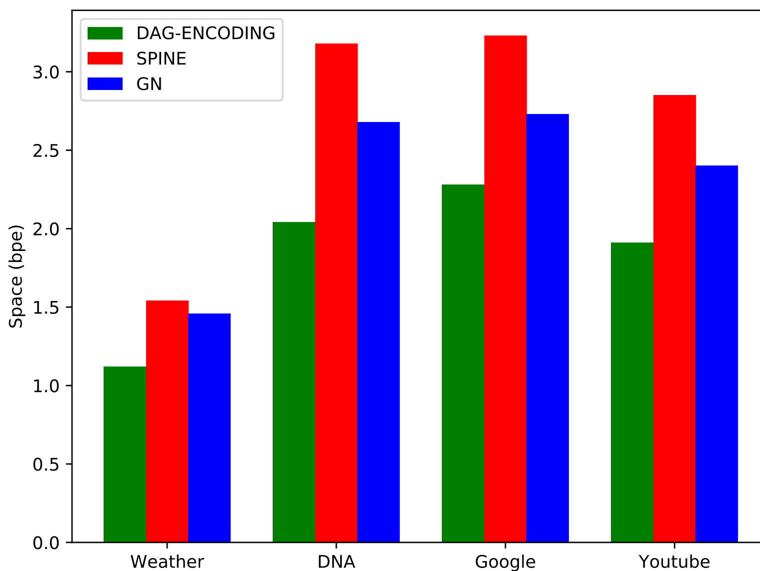


Figure 3.36: Space usage (huffman) of the encodings on the inputs from real data sets.

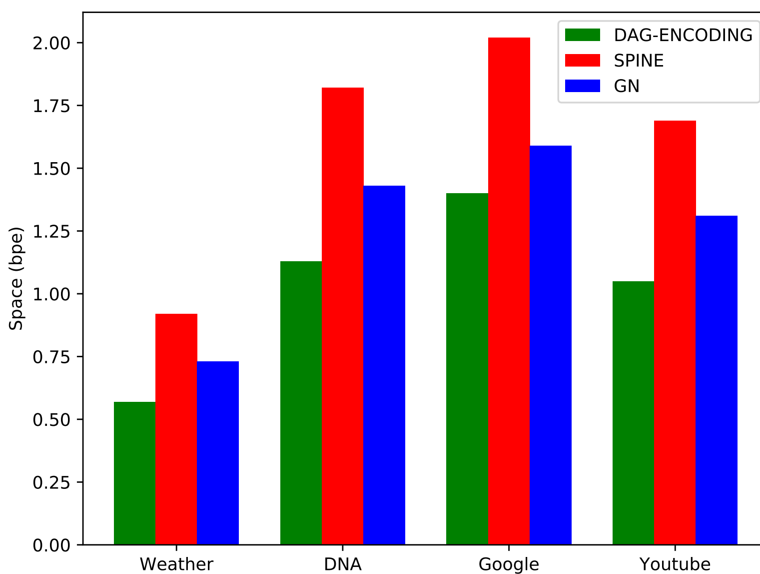


Figure 3.37: Space usage (lzw) of the encodings on the inputs from real data sets.

Chapter 4

Practical Implementations of Compressed RAM

4.1 Introduction

In this chapter, we consider the problem of compressed RAM data structures. Before we introduce our data structures, we briefly describe the two compressed RAM data structures of Jansson et al. [4], and Grossi et al. [5]. In the rest of this chapter, we denote these data structures as CRAM and DCRAM, respectively. We do not discuss how to support insert and delete operations on these structures in theory, as they do not affect the main features of our implementation.

CRAM: In CRAM, one first divides S into *blocks* of size $\ell = \frac{1}{2}\log_\sigma n$, and considers S as a string S' of length $n' = n\epsilon/\ell$ over an alphabet of size $2^\ell = \sqrt{n}$. We then compress the string S' by first assigning binary codewords of lengths from 1 to $O(\lg n)$ such that the length of a codeword for a given character in S' is inversely proportional to its frequency in S'. We then concatenate the codewords of every $1/\epsilon$ consecutive characters in S' (i.e., ℓ/ϵ consecutive characters in S) into a *superblock*, for some constant $0 < \epsilon < 1$. The lengths of the encoded

superblocks of S are managed by a darray data structure, which supports the following two operations efficiently: (i) address(i): access the position of the i-th superblock, and (ii) realloc(i, b): update the length of the encoded i-th superblock to b bits.

The execution of the algorithm is decomposed into *phases* where j-th phrase consists of all the operations between the $((j-1)n'+1)$-th and (jn')-th replace operations on S. For the 0-th phrase, one initializes the following substructures along with the compressed S and darray: (i) a frequency table F that maintains the frequencies of the blocks in S, (ii) two encoding tables, C_{old} and C_{new}, (iii) two decoding tables D_{old} and D_{new}, corresponding to C_{old} and C_{new}, respectively. We initialize $C_{old} = C_{new}$, and $D_{old} = D_{new}$. Finally, CRAM maintains a bit array R such that $R[k]$ indicates whether the k-th superblock is encoded using C_{new} or C_{old}. Then during the j-th phase, one can answer the access(S, i) by (i) finding the k-th superblock that contains $S[i]$, and (ii) decoding $S[i]$ using $D_{R[k]}$.

Next, we describe how to perform the replace(S, i, c), which is the x-th replace operation during the j-th phrase. We first find the k-th superblock that contains $S[i]$, and re-encode the superblock by referring to $C_{R[k]}$ and $D_{R[k]}$, and update F accordingly. In addition, we re-encode the superblocks from left to right (one at a time). If $x = n'$, all the superblocks are encoded with C_{new}. We then replace C_{old} and D_{old} as current C_{new} and D_{new}, respectively, and reconstruct C_{new} and D_{new} from F. In theory, these reconstructions can be deamortized and one can perform replace in $O(1/\epsilon)$ time in the worst case.

DCRAM: As in the description above, DCRAM also considers the input string S of length n over an alphabet of size σ as another string S' of length $n' = n/\ell$, where $\ell = \frac{1}{2}\log_\sigma n$ is the size of a block. Then each block can be considered as a character from the alphabet of size $2^\ell = \sqrt{n}$. Initially, for each character in S', we assign a codeword whose length is inversely proportional to its frequency

in S'. For this purpose, we first create a *codetable* containing all possible binary codes of length j for $1 \leq j \leq \frac{1}{2}\log_\sigma n$ and assign the shortest available code to each character in S' by considering the characters in the decreasing order of their frequencies. Now the initial codeword assigned to a character is simply the code assigned to that character from the codetable preceded by "000" (i.e., each codeword is extended by prepending on three zeroes). This ensures that plenty of codewords of each length are available in the codetable, which would be assigned to the characters whose frequency increases 'significantly'. The compressed string is obtained by concatenating the codewords of each of the n' (meta) characters in S'. More precisely, each block b is initially encoded as a codeword of length $j+3$ if its frequency in S' is between $(n'/2^j)+1$ and $n'/2^{j+1}$ (thus, $2^{j+3} - 2^j$ (check 2^{j+2}) codewords of length $j+3$ are unused initially). Note that for each j, at most one codeword of length j is used for encoding any block b. Also, DCRAM maintains a list of codewords corresponding to each block of S, a list of unused codewords, and darray to support address(i) and realloc operations on S'.

To answer access(S,i), one first finds the block that contains $S[i]$, using the address query, and then decodes that block to report $S[i]$ (note that the lengths of the compressed blocks are stored explicitly, and also that each block encodes a fixed number of characters in S). To answer replace(S,i,c), we first find the block b_i that contains $S[i]$, and update the frequencies of b_i and c appropriately (i.e., increase the frequency of c and decrease the frequency of b_i). Let l_i be the length of the current codeword of b_i. If the frequency of block b_i exceeds the upper bound (2^{l_i-3}) on the frequency of codewords of length l_i, we assign an unused codeword of length l_{i-1} for b_i, from the free codewords of length l_{i-1} if it exists. If all the codewords of length l_i are used for encoding other blocks of S, we reconstruct all the structures of DCRAM. This reconstruction process can be deamortized, and one can perform replace operation on DCRAM in $O(1)$ time.

4.2 Practical implementation

This section describes our two practical implementations of extended compressed RAM on S. The implementations are based on CRAM and DCRAM, respectively. As in the case of CRAM, we divide S into blocks of size ℓ, and define a superblock as $1/\epsilon$ consecutive blocks in S (we describe how to choose ℓ and ϵ in the next section). Note that we maintain the superblocks in both implementations, although DCRAM does not maintain any superblocks. For both implementations, we made the following modifications:

- We encode the blocks of S using Huffman codes [64]. Since Huffman code is fixed-to-variable, we do a linear scan on the superblock to decode the blocks contained in the superblock. Compared to using the binary code as in [4] and [5], the time for access operation gets slower, but the size of the compressed representation of S takes significantly less space in practice (since we don't need to encode the length of each codeword explicitly). After constructing a Huffman tree from the block frequencies, we construct a code table (and delete the Huffman tree) that is then used to convert between the codewords and the blocks. This optimization is a variant of previous efficient Huffman decoding works [65, 66].

- Instead of darray, we use a B^+-tree of *fixed-height* h (h is decided depending on the size of S) to manage the superblocks of S. See [67] for a detailed survey of the B-tree and its variants. In a fixed-height B^+-tree, all the operations are performed in the same manner as in a standard B^+-tree, except that the root node does not split regardless of the number of insert operations on S. Compared to using the standard B^+-tree, fixed-height B^+-tree could increase the time for scanning the superblocks, whereas the number of random access operations, which takes much more time than linear scanning in practice, is bounded by h. Using the fixed-height B^+-tree with height h, all the operations take $O(h \log(\ell/\epsilon) + \ell/\epsilon)$ for both

implementations.

For both implementations, we applied the following optimization schemes.

- To optimize memory access pattern with spatial locality, we save every block position in the compressed representation for recently used superblock. This scheme does not work on a series of random position operations, because when recently used superblocks differ every operation, previous saved block positions are invalidated and new new block positions need to be recalculated. In the access pattern with spatial locality, for example, access pattern with sequential positions, the target superblock is always recently used, so we know every block position. Using the saved block position, we can know the target block position. For access operation, all we have to do is decode one block and return decoded characters. For modifying operations (insert, delete, replace), we can shift codewords from the target block to the end of the superblock. And perform modification operations. So decoding procedures that take lots of running time can be skipped.

- For insert and delete operations on B^+-tree, there are linear modifications on B^+-tree nodes, which consist of a partial sum index. When modifying partial sum, we apply the SIMD technique for faster update of the partial sum index of each node. See [68] for a detailed survey of the SIMD technique. Applying multiple updates on a single CPU instruction enables our data structures to run fast on the fixed and low height B^+-tree. Also, SIMD technique can also be applied on the sequential optimization scheme when there is block position modification from the target block to the end of the superblock.

Now we describe the additional modifications to the implementation of DCRAM. In this case, we assign the new codeword to the block when its frequency is increased by a factor of $c = 2$. In the standard Huffman code, one has

to reconstruct the Huffman tree whenever the new codeword is assigned, which takes significantly more time than assigning the new codeword in theory. To resolve the issue, we use either a *Type-1* and *Type-2* extended Huffman tree, defined as follows. In a Type-1 tree, each codeword in the standard Huffman tree is extended by prepending on a single 0, so that we can assign the codewords starting with 1 (codewords on the right subtree of the root) to accommodate future insertions into the Huffman tree. Similarly, for a Type-2 tree, we first partition the blocks of S into B_1 and B_2 where the total frequency of B_2 is two times the total frequency of B_1 in S. After that, we build the standard Huffman tree on the blocks of B_1 and B_2. Finally, 0 and 10 are prepended on the codewords of B_1 and B_2, respectively. Thus, we can assign the codewords starting with 11 for the new codewords. We reconstruct the tree and re-encode S when there is no free code of the required length available in the tree. The following lemma shows that the $\Omega(n/\ell)$ replace operations are necessary before reconstructing the tree, as in the case of binary codes used in [5].

Lemma 4.1. *Given a Type-1 or 2 tree, at least $\Omega(n/\ell)$ replace operations are necessary before reconstructing the tree.*

Proof. For simplicity, we assume that we assign new codes in the lexicographically increasing order. We only consider the case of the Type-1 tree in the proof (the case of the Type-2 tree can be proved by a similar argument). In the Type-1 tree, any new codeword of length l is assigned only after at least $n/(\ell \cdot 2^{l-1})$ replace operations are performed. Thus, if at most $n/(4\ell)$ replace operations are performed on S, all the new codewords start with 10, and we can still assign codewords starting with 11.

□

Finally, in the implementation of DCRAM, we do *lazy update*, which com-

bines the update algorithms of CRAM and DCRAM as follows. We maintain two Type-1 or Type-2 trees and re-encode the superblocks as in the implementation of CRAM. We reconstruct the tree when (i) every superblock is re-encoded, and (ii) there is no available free code in the tree. Note that without lazy updates, CRAM (resp. DCRAM) reconstructs the codetable when only the condition (i) (resp. (ii)) is fulfilled.

4.3 Experimental results

We implemented our CRAM and DCRAM using C++ (complied by g++ 11.1). The experiments were done on Desktop PC (AMD Ryzen 5 1600 Six-Core Processor 3.2 GHz with 32GB RAM). The tests are composed of (i) the tests for access and replace operations and (ii) the tests for access and insert, and delete operations. Note that the former tests benchmark the performance as compressed RAM, whereas the later tests benchmark the performance as extended compressed RAM. The details of the tests are as follows:

- replace-seq: This test overwrites the source file to the destination file from left to right using replace operations.
- replace-random1: From the source file, we first choose $n\epsilon/\ell$ superblocks uniformly at random, and replace all the characters in these superblocks to the superblock where each of the characters in the superblock is chosen uniformly at random.
- replace-random2: From the source file, we first choose a character α. After that we choose $n\epsilon/\ell$ superblocks uniformly at random and replace all the characters in these superblocks to α.
- insert-seq: We first choose a character α. Then we pick a position i from the source file uniformly at random, and insert α $n/2$ times consecutively from the position i.

- delete-seq: We pick a position $i \leq n/2$ from the source file uniformly at random, and delete the characters at the positions from i to $i + n/2$.
- indel-random: We perform $n/8$ insert and delete operations on the source file, where all the arguments of the operations are chosen uniformly at random.

Also, designed experiments that consider operation time. We tested three modification operations, replace, insert, and delete. For the replace and insert operation, we choose a character from the destination file at the same position as the source file. The details of the tests are as follows:

- seq: We pick a position $i \leq n$ from the source file uniformly at random, and perform operation on the characters at the positions from i to $i+n/2$.
- random: We perform $n/8$ operations on the source file, where all the arguments of the operations are chosen uniformly at random.

We use the test files from Pizza&Chilli Corpus[1] with size 200MB (the detailed statistics of the test files are in Table 4.1). In our implementations, we choose the block size $\ell = 2$ (thus, we can fit each block within a single 16-bit data type), and $\epsilon = 1/512$ (i.e., each superblock contains 512 blocks). Also, compressed superblocks are stored using an 64-bit array (uint64_t array). For the implementation of B^+-tree of fixed-height h, we choose $h = 2$, so that each entry of the root node contains the information of about $2 \cdot 10^8/2^{20} \sim 200$ consecutive superblocks. Also, we use an additional parameter $u \in \{1, 2, 4\}$ for the implementations of CRAM and DCRAM with the lazy update. The parameter u decides how many superblocks are re-encoded for each update (replace, insert, or delete) operation. More precisely, for every update operation, we re-encode u consecutive superblocks from $((u-1)/u) \cdot n\epsilon/\ell)$-th superblock of the input. Thus, exactly u reconstructions occur during every $n\epsilon/\ell$ update operation (note that $n\epsilon/\ell$ is the number of superblocks in the input). We compared

[1]http://pizzachili.dcc.uchile.cl/texts.html

our implementations with the compressed CRAM implementation of Klitzke and Nicholson (KN for short) [69], and SPSI (Searchable Partial Sums with Inserts) of Prezza [49]. Note that the latter does not compress the input.

Our tests show the results of replace-seq test (i) from ENGLISH to DNA, and (ii) from XML to PROTEINS, respectively (see Figures 4.1 and 4.2). In the figures, each implementation is labeled by 'name / u / number of reconstructions (denoted as nr) / total time'. Type-1 and Type-2 denote Type-1 and Type-2 trees, respectively. For both tests, the total space of our implementations changes depends on the H_1 of the input and the time from the last reconstruction during the replacements. The total time of our implementations mainly depends on the number of total reconstructions. (for example, in (i), DCRAM with no lazy update do four reconstructions during the test, whereas DCRAM with $u = 1$ do one reconstruction for the same test). If the number of reconstructions is the same, DCRAM works faster than CRAM since the DCRAM contains less codewords than in CRAM. KN is 4 to 9 times faster than our implementations since it is highly optimized for sequential update operations. However, the total space usage of KN strictly increases in both (i) and (ii) since it uses the compressed table for the source input and does not update (and reconstruct) during the replace operations. Thus, our implementations take less space than KN when the entropy decreases during replace operations. Also, since SPSI does not compress the input, it shows the fastest running time on the replace operation, while using the most space.

Also, in DCRAM, the lazy update with $u = 1$ works up to 1.5 times faster than the implementation without the lazy update while taking up to 30% more space. If u is more than 1, the working time of DCRAM entirely depends on u, which shows worse than CRAM with the same u in both time and space.

Our next experiments show the results of replace-random1 and replace-random2 tests from ENGLISH, respectively (see Figures 4.3 and 4.4). In the replace-random1 test, the number of distinct codewords increases during the

test, which increases the time to reconstruct the Huffman tree. Thus, DCRAM with lazy update works slower than DCRAM without lazy update. On the other hand, in the replace-random2 test, the number of distinct codewords is strictly decreasing (after every reconstruction) during the test. Hence, DCRAM with lazy update works fastest among all our implementations. All the reconstructions depend on u since the probability of each character increasing its frequency is quite small, which implies DCRAM does not assign multiple codewords for a single block. Thus, CRAM outperforms DCRAM in both time and space. However, in replace-random2 test, DCRAM assigns multiple codewords to the same block, which delays the reconstruction compared to the CRAM. Hence DCRAM works up to two times faster than CRAM while keeping the same space overhead in other replace tests. KN works faster than our implementations for replace-random1 test since it is not highly compressible, which implies lots of sequential operations are necessary to re-encode the superblock. However, in the replace-random2 test, KN works worse than our implementations in both time and space since the superblocks are highly compressible.

Our next experiments show the results of insert-seq and delete-seq tests on ENGLISH, respectively (see Figures 4.5 and 4.6). Since SPSI currently does not support delete operations, we only do insert-seq test on SPSI. Since insert-seq test only inserts the same character, both the entropy and the space usage of our implementations decrease during the test as in replace tests. KN is up to 9 times faster than ours in insert-seq test since it is highly optimized for sequential updates. However, its space usage is more than all our implementations. Also, the compression ratio of KN does not change during the test. This can happen when the codeword of α compresses α with a compression ratio close to the compression ratio of KN. SPSI shows worse than other implementations in both time and space. For delete-seq, the entropy does not change during the delete operations since the frequency distribution of the characters in ENGLISH does not change significantly during the sequential deletions. Hence, the space

usage of our implementations does not change during the test. Interestingly, the space usage of KN increases slowly during the test. Again, KN is up to 5 times faster than ours in delete-seq test, while it uses more space.

Our next experiment shows the results of indel-random test on ENGLISH (see Figure 4.7) Since KN does not support character-wise insert and delete operations and SPSI does not support delete operations, we only tested with our implementations. Overall, each operation is up to 65 times slower than in insert-seq and delete-seq tests (per each operation). This is because the entire superblock that contains the character can be re-encoded when we insert or delete a single character, whereas only a single block that contains the character is enough to be re-encoded when we perform insert and delete operations at consecutive positions.

Our final experiment shows the results of seq and random tests on ENGLISH. (see Tables 4.2, 4.3, and 4.4) We compared ours with KN and SPSI. Also, we tested our CRAM and DCRAM with an additional parameter $u \in \{1, 2, 4\}$. As mentioned in the previous paragraph, since SPSI does not compress the input, it shows the fastest running time on the replace operation, while using the most space. And, KN shows highly optimized operation results on sequential operations. But, ours gives decent operation times in both sequential and random cases. Also, note that since SPSI currently does not support delete operations, we only do replace and insert tests on SPSI. Lastly, the reason why CRAM takes more time than DCRAM in the random case is the cold start, which means that the code size of CRAM is larger than that of DCRAM at the early stage, and this affects the mean time.

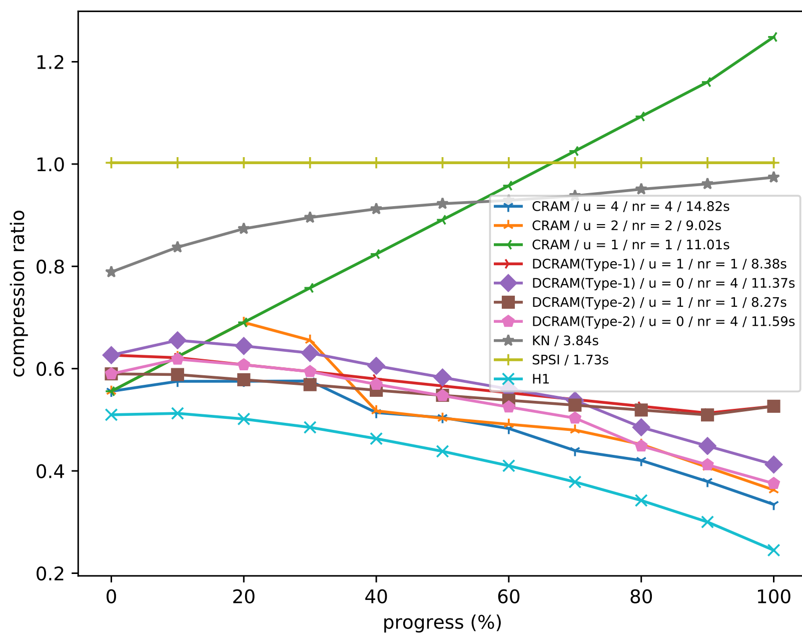


Figure 4.1: replace-seq test (a) from ENGLISH to DNA

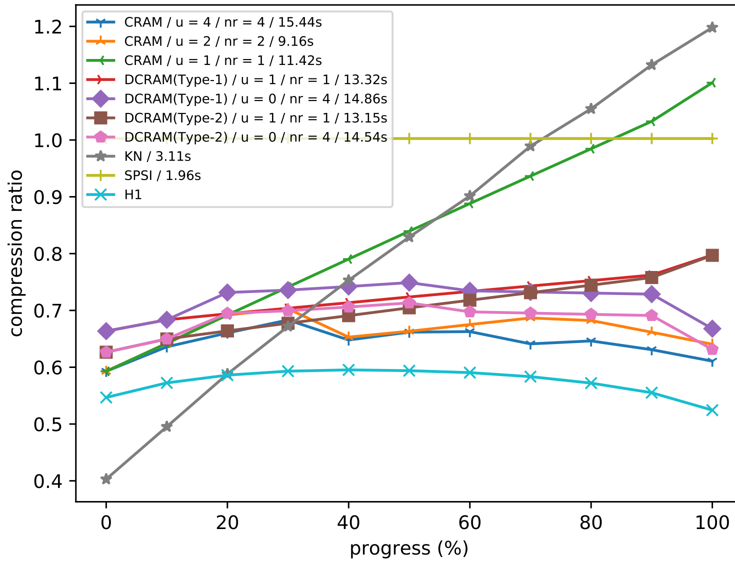


Figure 4.2: replace-seq test (a) from XML to PROTEINS

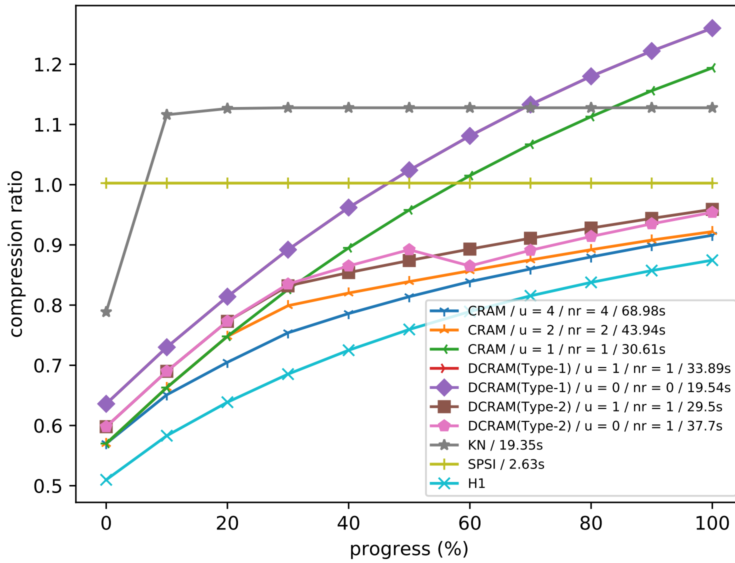


Figure 4.3: replace-random1 test on ENGLISH.

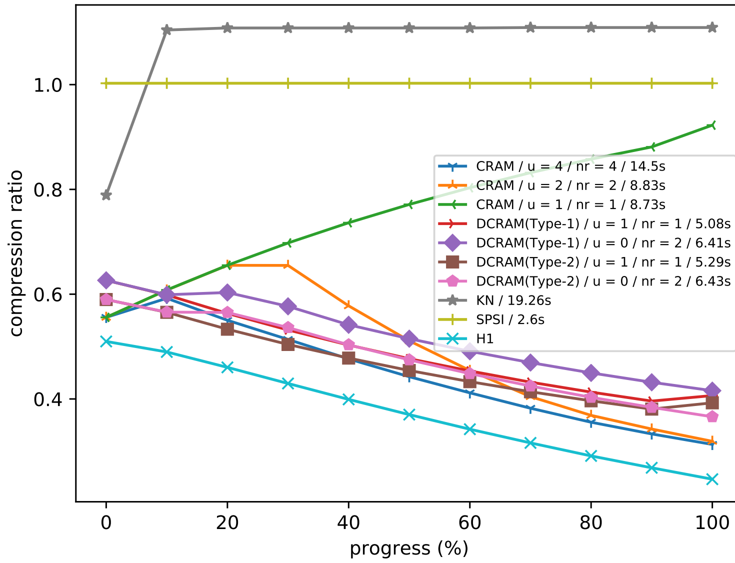


Figure 4.4: replace-random2 test on ENGLISH.

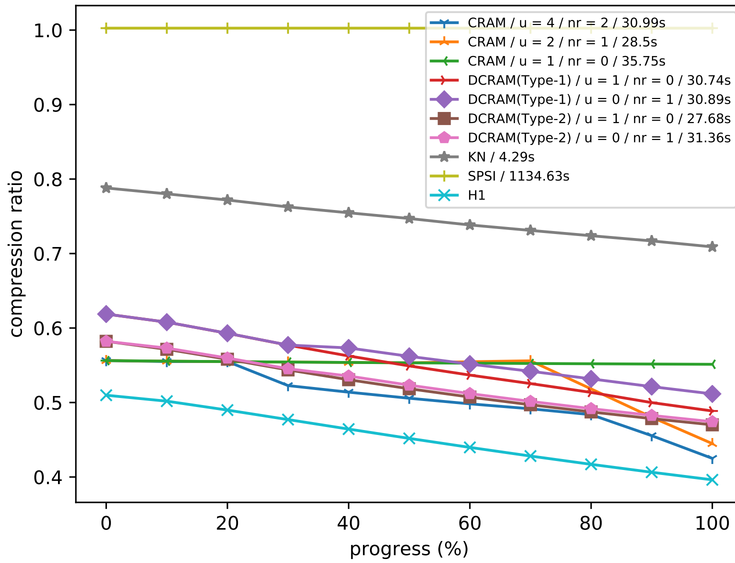


Figure 4.5: insert-seq test on ENGLISH.

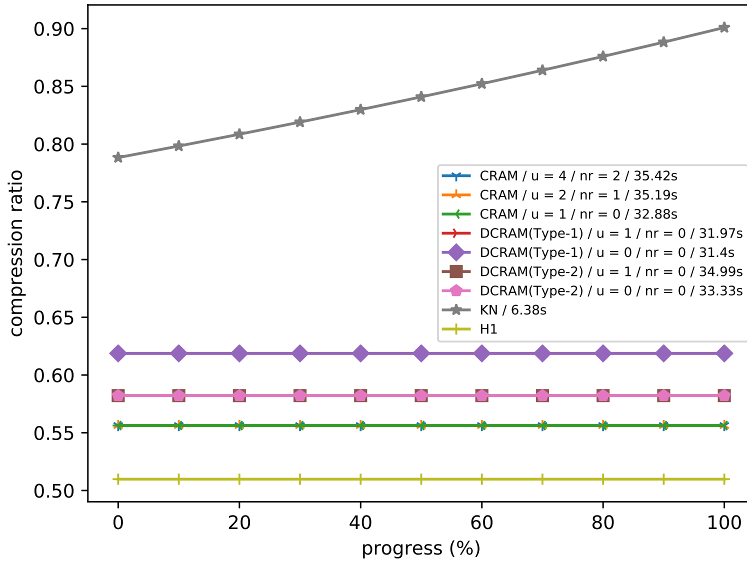


Figure 4.6: delete-seq test on ENGLISH.

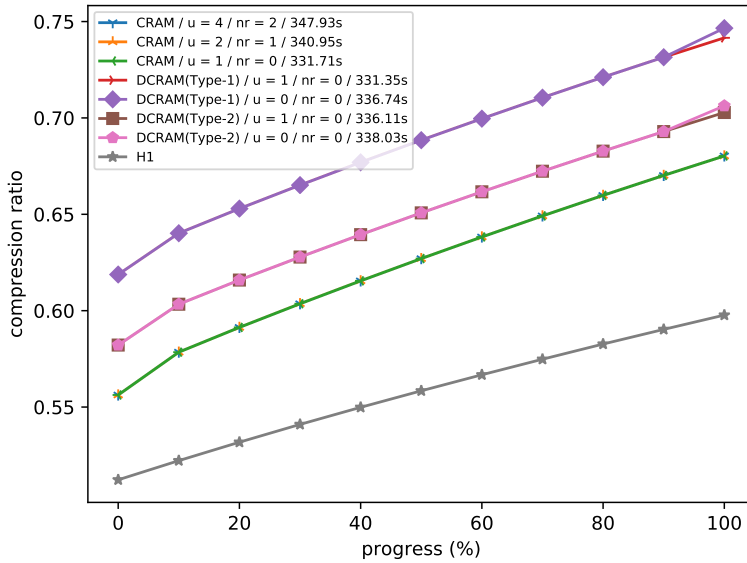


Figure 4.7: indel-random test on ENGLISH.

File name	Alphabet size	H_0	H_1
DNA	16	0.247	0.245
ENGLISH	225	0.565	0.509
PROTEINS	25	0.525	0.524
XML	96	0.657	0.547

Table 4.1: File statistics of test files. H_k denotes the k-th order entropy.

Replace	sequential	random
KN	69.19	52970.8
SPI	66.53	318.39
CRAM ($u = 1$)	80	11090
CRAM ($u = 2$)	91	11398
CRAM ($u = 4$)	131	11657
DCRAM ($u = 1$)	78	10292
DCRAM ($u = 2$)	93	10391
DCRAM ($u = 4$)	136	10530

Table 4.2: Operation time tests on replace. Time unit is nanoseconds.

Insert	sequential	random
KN	264.787	52843
SPI	80000	56525
CRAM ($u = 1$)	659	12616
CRAM ($u = 2$)	665	12615
CRAM ($u = 4$)	722	12304
DCRAM ($u = 1$)	717	11239
DCRAM ($u = 2$)	727	11514
DCRAM ($u = 4$)	774	11314

Table 4.3: Operation time tests on insert. Time unit is nanoseconds.

Delete	sequential	random
KN	253.22	52437
CRAM ($u = 1$)	663	10489
CRAM ($u = 2$)	662	10405
CRAM ($u = 4$)	698	10181
DCRAM ($u = 1$)	637	9063
DCRAM ($u = 2$)	708	9029
DCRAM ($u = 4$)	774	9319

Table 4.4: Operation time tests on delete. Time unit is nanoseconds.

Chapter 5

Conclusions and Open Problems

In this thesis, we proposed space-efficient data structures from a practical view. We proposed encoding data structures for range-top2 queries and dynamic compressed strings that support modification operations. Our data structures show better practical space/time usage compared to previous works.

In chapter 3, we first proposed a practical implementation for encoding RT2Q . Our data structure takes much less space than the current indexing data structures, while still giving better time-space tradeoffs for most cases in practice. Also, when query time is not of concern, we propose an encoding that supports fast construction time and shows comparable space usage compared to the optimal encoding in practice. In Chapter 4, we proposed practical implementations of extended compressed RAM. We proposed substructures with B^+-tree to achieve simpler substructures and practical performance. Our experimental results show our data structures give better practical performance in most cases.

We suggest the following open problems for chapter 3 and chapter 4.

- In chapter 3, implementing the data structure based on the BP of $2dmax(A)$

is an open problem. Here, an efficient and practical implementation of degree and childrank queries would be a challenging problem.

- In chapter 3, Another interesting open problem is improving the space analysis of our DAG-based encoding (Lemma 3.4).
- In chapter 4, highly optimizing our data structures for sequential and random use cases is an open problem.
- In chapter 4, making compressed RAM data structures with supporting concurrency is an interesting open problem.

Bibliography

[1] Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti, "On succinct representations of binary trees," *Math. Comput. Sci.*, vol. 11, no. 2, pp. 177–189, 2017.

[2] Richard F. Geary, Rajeev Raman, and Venkatesh Raman, "Succinct ordinal trees with level-ancestor queries," *ACM Trans. Algorithms*, vol. 2, no. 4, pp. 510–534, 2006.

[3] Seungbum Jo, Rahul Lingala, and Srinivasa Rao Satti, "Encoding two-dimensional range top-k queries," *Algorithmica*, vol. 83, no. 11, pp. 3379–3402, 2021.

[4] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung, "CRAM: compressed random access memory," in *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Proceedings, Part I.* 2012, vol. 7391 of *Lecture Notes in Computer Science*, pp. 510–521, Springer.

[5] Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini, "Dynamic compressed strings with random access," in *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Proceedings, Part I.* 2013, vol. 7965 of *Lecture Notes in Computer Science*, pp. 504–515, Springer.

[6] Paolo Ferragina and Giovanni Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[7] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. 2003, pp. 841–850, ACM/SIAM.

[8] Roberto Grossi and Jeffrey Scott Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)," in *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, F. Frances Yao and Eugene M. Luks, Eds. 2000, pp. 397–406, ACM.

[9] Rajeev Raman, "In-memory representations of databases via succinct data structures: Tutorial abstract," in *PODS, 2018*. 2018, pp. 323–324, ACM.

[10] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyen, Jouni Sirén, and Niko Välimäki, "Fast in-memory xpath search using compressed indexes," *Softw. Pract. Exp.*, vol. 45, no. 3, pp. 399–434, 2015.

[11] O'Neil Delpratt, Rajeev Raman, and Naila Rahman, "Engineering succinct DOM," in *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, Eds. 2008, vol. 261 of *ACM International Conference Proceeding Series*, pp. 49–60, ACM.

[12] Satoshi Koide, Yukihiro Tadokoro, Chuan Xiao, and Yoshiharu Ishikawa, "Cinct: Compression and retrieval for massive vehicular trajectories via relative movement labeling," in *34th IEEE International Conference on*

Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. 2018, pp. 1097–1108, IEEE Computer Society.

[13] Julian Shun and Guy E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, Eds. 2013, pp. 135–146, ACM.

[14] Bahar Alipanahi, Martin D. Muggli, Musa Jundi, Noelle R. Noyes, and Christina Boucher, "Metagenome SNP calling via read-colored de bruijn graphs," *Bioinform.*, vol. 36, no. 22-23, pp. 5275–5281, 2021.

[15] Thomas C. Conway and Andrew J. Bromage, "Succinct data structures for assembling large genomes," *Bioinform.*, vol. 27, no. 4, pp. 479–486, 2011.

[16] Patrick Putnam, Ge Zhang, and Philip A. Wilsey, "A comparison study of succinct data structures for use in GWAS," *BMC Bioinform.*, vol. 14, pp. 369, 2013.

[17] Guillaume Marçais, Brad Solomon, Rob Patro, and Carl Kingsford, "Sketching and sublinear data structures in genomics," *Annual Review of Biomedical Data Science*, vol. 2, pp. 93–118, 2019.

[18] Stephen A. Cook and Robert A. Reckhow, "Time-bounded random access machines," in *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg, Eds. 1972, pp. 73–80, ACM.

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

[20] David Lorge Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*, Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde, Eds. 1994, pp. 279–287, IEEE Computer Society / ACM Press.

[21] Simon Gog and Matthias Petri, "Optimized succinct data structures for massive data," *Softw. Pract. Exp.*, vol. 44, no. 11, pp. 1287–1314, 2014.

[22] Joshimar Cordova and Gonzalo Navarro, "Practical dynamic entropy-compressed bitvectors with applications," in *15th International Symposium, SEA 2016, Proceedings*. 2016, vol. 9685 of *Lecture Notes in Computer Science*, pp. 105–117, Springer.

[23] Saska Dönges, Simon J. Puglisi, and Rajeev Raman, "On dynamic bitvector implementations," in *Data Compression Conference, DCC 2022*, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, Eds. 2022, pp. 252–261, IEEE.

[24] Miguel E. Coimbra, Joana Hrotkó, Alexandre P. Francisco, Luís M. S. Russo, Guillermo de Bernardo, Susana Ladra, and Gonzalo Navarro, "A practical succinct dynamic graph representation," *Inf. Comput.*, vol. 285, no. Part, pp. 104862, 2022.

[25] Pawel Gawrychowski and Patrick K. Nicholson, "Optimal encodings for range top-k, selection, and min-max," in *ICALP 2015, Proceedings, Part I*, 2015, pp. 593–604.

[26] Wooyoung Park, Seungbum Jo, and Srinivasa Rao Satti, "Practical Implementation of Encoding Range Top-2 Queries," *The Computer Journal*, 2022.

[27] Seungbum Jo, Wooyoung Park, Kunihiko Sadakane, and Srinivasa Rao Satti, "Practical Implementations of Compressed RAM," in *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*. 2023, IEEE.

[28] Peter Bro Miltersen, "Cell probe complexity-a survey," in *Proceedings of the 19th conference on the foundations of software technology and theoretical computer science, advances in data structures workshop*. Citeseer, 1999, p. 2.

[29] Jean Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, no. 4, pp. 229–239, 1980.

[30] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan, "Scaling and related techniques for geometry problems," in *Proceedings of STOC 1984*, 1984, pp. 135–143.

[31] Dov Harel and Robert Endre Tarjan, "Fast algorithms for finding nearest common ancestors," *siam Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[32] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe, "Nearest common ancestors: a survey and a new distributed algorithm," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, Winnipeg, Manitoba, Canada, August 11-13, 2002*, Arnold L. Rosenberg and Bruce M. Maggs, Eds. 2002, pp. 258–264, ACM.

[33] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin, "Lowest common ancestors in trees and directed acyclic graphs," *J. Algorithms*, vol. 57, no. 2, pp. 75–94, 2005.

[34] Omer Berkman and Uzi Vishkin, "Recursive star-tree parallel data structure," *SIAM J. Comput.*, vol. 22, no. 2, pp. 221–242, 1993.

[35] Hao Yuan and Mikhail J. Atallah, "Data structures for range minimum queries in multidimensional arrays," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, Moses Charikar, Ed. 2010, pp. 150–160, SIAM.

[36] Guy Jacobson, "Space-efficient static trees and graphs," in *FOCS 1989*, 1989, pp. 549–554.

[37] Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao, "On space efficient two dimensional range minimum data structures," *Algorithmica*, vol. 63, no. 4, pp. 815–830, 2012.

[38] Kunihiko Sadakane, "Compressed suffix trees with full functionality," *Theory Comput. Syst.*, vol. 41, no. 4, pp. 589–607, 2007.

[39] Gonzalo Navarro and Kunihiko Sadakane, "Fully functional static and dynamic succinct trees," *ACM Trans. Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, 2014.

[40] Luís M. S. Russo, "Range minimum queries in minimal space," *Theor. Comput. Sci.*, vol. 909, pp. 19–38, 2022.

[41] Tomasz Marek Kowalski and Szymon Grabowski, "Faster range minimum queries," *Softw. Pract. Exp.*, vol. 48, no. 11, pp. 2043–2060, 2018.

[42] Pawel Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann, "Compressed range minimum queries," *Theor. Comput. Sci.*, vol. 812, pp. 39–48, 2020.

[43] J. Ian Munro and Sebastian Wild, "Entropy trees and range-minimum queries in optimal average-case space," *CoRR*, vol. abs/1903.02533, 2019.

[44] Rajeev Raman, "Encoding data structures," in *WALCOM 2015*, 2015, pp. 1–7.

[45] Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit, "Practical range minimum queries revisited," in *SEA 2017*, 2017, pp. 12:1–12:16.

[46] Héctor Ferrada and Gonzalo Navarro, "Improved range minimum queries," *J. Discrete Algorithms*, vol. 43, pp. 72–80, 2017.

[47] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica, "Succinct: Enabling queries on compressed data," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015.* 2015, pp. 337–350, USENIX Association.

[48] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica, "Blowfish: Dynamic storage-performance tradeoff in data stores," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, Katerina J. Argyraki and Rebecca Isaacs, Eds. 2016, pp. 485–500, USENIX Association.

[49] Nicola Prezza, "A framework of dynamic data structures for string processing," in *16th International Symposium on Experimental Algorithms, SEA 2017*, vol. 75 of *LIPIcs*, pp. 11:1–11:15.

[50] Miguel E. Coimbra, Alexandre P. Francisco, Luís M. S. Russo, Guillermo de Bernardo, Susana Ladra, and Gonzalo Navarro, "On dynamic succinct graph representations," in *Data Compression Conference, DCC 2020,*

Snowbird, UT, USA, March 24-27, 2020, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, Eds. 2020, pp. 213–222, IEEE.

[51] Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti, "Encodings for range selection and top-k queries," in *ESA 2013*, 2013, vol. 8125 of *Lecture Notes in Computer Science*, pp. 553–564.

[52] Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao, "Asymptotically optimal encodings of range data structures for selection and top-k queries," *ACM Trans. Algorithms*, vol. 13, no. 2, pp. 28:1–28:31, 2017.

[53] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 11:1–11:58, 2008.

[54] Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti, "Encoding range minima and range top-2 queries," *Philosophical Transactions of the Royal Society A*, vol. 372, no. 2016, pp. 20130131, 2014.

[55] Arash Farzan and J. Ian Munro, "A uniform paradigm to succinctly encode various families of trees," *Algorithmica*, vol. 68, no. 1, pp. 16–40, Jan 2014.

[56] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane, "Succinct trees in practice," in *Proceedings of ALENEX 2010*, 2010, pp. 84–97.

[57] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, 2005.

[58] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung, "Ultra-succinct representation of ordered trees with applications," *J. Comput. Syst. Sci.*, vol. 78, no. 2, pp. 619–631, 2012.

[59] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 43, 2007.

[60] J. Ian Munro, Venkatesh Raman, and Srinivasa Rao Satti, "Space efficient suffix trees," *J. Algorithms*, vol. 39, no. 2, pp. 205–222, 2001.

[61] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri, "From theory to practice: Plug and play with succinct data structures," in *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, Joachim Gudmundsson and Jyrki Katajainen, Eds. 2014, vol. 8504 of *Lecture Notes in Computer Science*, pp. 326–337, Springer.

[62] Luc Devroye, "On random cartesian trees," *Random Struct. Algorithms*, vol. 5, no. 2, pp. 305–328, 1994.

[63] Terry A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[64] David A Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[65] Pi-Chung Wang, Yuan-Rung Yang, Chun-Liang Lee, and Hung-Yi Chang, "A memory-efficient huffman decoding algorithm," in *19th International Conference on Advanced Information Networking and Applications (AINA 2005), 28-30 March 2005, Taipei, Taiwan.* 2005, pp. 475–479, IEEE Computer Society.

[66] Jamil As-ad, Mohibur Rahaman, Rashed Mustafa, Zinnia Sultana, and Lutfun Nahar, "An improved decoding technique for efficient huffman

coding," *Journal of Computer Science and Applications and Information Technology*, vol. 2, no. 1, pp. 1–5, 2017.

[67] Douglas Comer, "The ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[68] Armando Faz Hernández, "Yet another survey on simd instructions," 2013.

[69] Patrick Klitzke and Patrick K. Nicholson, "A general framework for dynamic succinct and compressed data structures," in *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016*. 2016, pp. 160–173, SIAM.

요약

본 논문에서는 다양한 공간 효율적인 자료구조에 대해 다룬다. 또한, 본 논문에서는 이 자료구조들을 실용적 관점에서 구현하고 실험을 통해 그 성능을 평가하였다. 본 논문에서 제안하는 자료구조는 우수한 공간과 시간 효율성에 더해서, 동일한 이론적 성능을 보여주며 더 간단한 보조 자료구조를 갖는다. 간단한 보조 자료구조는 소프트웨어 엔지니어들이 지속적인 유지보수를 수행할 수 있도록 도와준다. 본 논문은 다음 두 가지 문제를 고려한다. (1) 구간 Top-2 인코딩, (2) 접근, 변경 연산을 지원하는 동적 문자열. 순서를 갖는 원소들로 이루어진 배열에서, Top-2 질의는 주어진 질의 범위 안에서 첫 번째와 두 번째 원소를 반환한다. 또한, Top-2 인코딩 문제는 Top-2 질의를 효율적으로 지원하기 위해 주어진 배열을 인코딩하는 문제이다. 동적 압축 문자열 문제는 문자열을 압축된 상태로 유지하면서, 접근 연산과 변경 연산을 효율적으로 지원하는 문제이다.

구간 Top-2 질의 문제를 위해서, 본 논문은 두 가지 실용적 구현을 제안한다. 첫 번째 구현은 구간 Top-2 질의에 대해 효율적으로 답변하는 Davoodi. et. al. [1]의 자료구조에 기반을 둔다. 본 논문에서는 Davoodi. et. al. [1]의 자료구조를 변형하여 Davoodi. et. al. [1]의 자료구조와 거의 동일한 이론적 성능을 가지며 더 단순한 보조 자료구조를 가지는 구현을 제안하였다. 또한, 해당 자료구조 구현은 개선된 공간과 시간 효율성을 보여준다. 본 논문의 다른 구현은 2 x n 행렬에서의 Top-2 인코딩에 기반을 둔다. 해당 자료구조 구현은 개선된 인코딩 시간과 비슷한 공간 효율성을 보여준다.

동적 압축 문자열 문제를 위해서, 본 논문은 두 가지 실용적 구현을 제안한다. 각각의 구현은 Jansson et al. [4]의 Compressed RAM 연구와 Grossi et al. [5]의 연구에 기반을 둔다. 이 문제에서, 본 논문은 이전 연구들 [4], [5]보다 더 간단한 자료구조를 제안한다. 또한, 본 논문의 자료구조가 가지는 구현의 용이성은, 최적화의 여지를 더 많이 가지게 한다. 실험 결과는 본 논문의 자료구조가 입력 문자열의 엔트로피에 비례하는 공간 사용을 유지하면서, 효율적인 연산을 지원하는

것을 보여준다.

주요어: 구간 Top-2 질의, 구간 최대 질의, 카테시안 트리, 인코딩 모델. 공간 효율적인 인코딩, 공간 효율적인 자료구조, 동적 자료구조, 동적 문자열, 접근 질의, 교체 질의, 삽입 질의, 삭제 질의

학번: 2016-21203

Acknowledgements

I could not have undertaken this journey without my supervisor professor Srinivasa Rao Satti and professor Seungbum Jo for their academic guidance and consistent support. Professor Srinivasa Rao Satti provided special care for me to become a decent researcher. Professor Seungbum Jo instructed me about good practice in academic research.

I also would like to express my sincere gratitude to professor Kunsoo Park, professor Soonhoi Ha, professor Inbok Lee for supervising my work.

Professor Kunsoo Park not only provided knowledge and expertise, but also gave me professional skills. Professor Soonhoi Ha provided his plentiful experience, and gave me support in my school life. Professor Inbok Lee gave thoughtful comments and recommendations on this dissertation.

Lastly, I'd like to acknowledge lab alumni and members. Their belief in me gives me motivation during this process. I'd also like to thank my family for all the financial and emotional support.