저작자표시-비영리-변경금지 2.0 대한민국

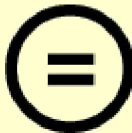
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 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

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

비트 문자열, 범위 질의 및 관련 문제들에 대한 공간 효율적인 인코딩

FEBRUARY 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Seungbum Jo

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

비트 문자열, 범위 질의 및 관련 문제들에 대한 공간 효율적인 인코딩

지도교수 Srinivasa Rao Satti

이 논문을 공학박사학위논문으로 제출함

2015 년 11 월

서울대학교 대학원

전기.컴퓨터 공학부

조 승 범

조 승 범의 박사학위논문을 인준함

2015 년 12 월

위 원 장	박 근 수	(인)
부위원장	Srinivasa Rao Satti	(인)
위 원	문 봉 기	(인)
위 원	강 유	(인)
위 원	이 인 복	(인)

Abstract

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

Seungbum Jo
Department of Electrical Engineerning and Computer Science
Collage of Engineering
The Graduate School
Seoul National University

In this thesis, we design and implement various space efficient data structures. Most of these structures use spaces close to the information-theoretic lower bound while supporting the queries efficiently. In particular, this thesis is concerned with the data structures for four problems: (i) supporting rank and select queries on compressed bit strings, (ii) nearest larger neighbor problem, (iii) simultaneous encodings for range and next/previous larger/smaller value queries, and (iv) range Top-k queries on two-dimensional arrays.

We first consider practical implementations of *compressed* bitvectors, which support rank and select operations on a given bit-string, while storing the bit-string in compressed form [45]. Our approach relies on *variable-to-fixed* encodings of the bit-string, an approach that has not yet been considered systematically for practical encodings of bitvectors. We show that this approach leads to fast practical implementations with low *redundancy* (i.e., the space used by the bitvector in addition to the compressed representation of the bit-string), and is a flexible and promising solution to the problem of supporting rank and select on moderately compressible bit-strings, such as those encountered in real-world applications.

Next, we propose space-efficient data structures for the nearest larger neighbor problem [44, 46]. Given a sequence of n elements from a total order, and a position in the sequence, the nearest larger neighbor (NLN) query returns the position of the element which is closest to the query position, and is larger than the element at the query position. The problem of finding all nearest larger neighbors has attracted interest due to its applications for parenthesis matching and in computational geometry [3, 4, 7]. We consider a data structure version of this problem, which is to preprocess a given sequence of elements to construct a data structure that can answer NLN queries efficiently. For one-dimensional arrays, we give time-space tradeoffs for the problem on *indexing model*. For two-dimensional arrays, we give an optimal encoding with constant query on *encoding model*.

We also propose space-efficient encodings which support various range queries, and previous and next smaller/larger value queries [47]. Given a sequence of n elements from a total order, we obtain a $4.088n + o(n)$-bit encoding that supports all these queries where n is the length of input array. For the case when we need to support all these queries in constant time, we give an encoding that takes $4.585n + o(n)$ bits. This improves the $5.08n + o(n)$-bit encoding obtained by encoding the colored $2d$-Min and $2d$-Max heaps proposed by Fischer [25]. We extend the original DFUDS [6] encoding of the colored $2d$-Min and $2d$-Max heap that supports the queries in constant time. Then, we combine the extended DFUDS of $2d$-Min heap and $2d$-Max heap using the Min-Max encoding of Gawrychowski and Nicholson [30] with some modifications. We also obtain encodings that take lesser space and support a subset of these queries.

Finally, we consider the various encodings that support range Top-k queries on a two-dimensional array containing elements from a total order. For an $m \times n$ array, we first propose an optimal encoding for answering one-sided Top-k queries, whose query range is restricted to $[1 \dots m][1 \dots a]$, for $1 \leq a \leq n$. Next, we propose an encoding for the general Top-k queries that takes $m^2 \lg \binom{(k+1)n}{n} +$

$m \lg m + o(n)$ bits. This generalizes the Top-k encoding of Gawrychowski and Nicholson [30].

Keywords: Space-efficient data structure, succinct data structure, encoding model, indexing model, bitvector, rank query, select query, nearest larger neighbor problem, range queries, next/previous larger query, range Top-k query
Student Number: 2011-30257

Contents

Abstract **i**

Contents **v**

List of Figures **ix**

List of Tables **xi**

Chapter 1 Introduction **1**

1.1 Computational model 2

1.1.1 Encoding and indexing models 2

1.2 Contribution of the thesis 3

1.3 Organization of the thesis 5

Chapter 2 Preliminaries **7**

Chapter 3 Compressed bit vectors based on variable-to-fixed encodings **10**

3.1 Introduction. 10

3.2 Bit-vectors using V2F coding 14

3.3 V2F compression algorithms for bit-strings 16

3.3.1 Tunstall code 16

3.3.2 Enumerative codes 19

3.3.3 LZW algorithm 23
3.3.4 Empirical evaluation of the compressors 23
3.4 Practical implementation of bitvectors based on V2F compression. 26
3.4.1 Testing Methodology 29
3.4.2 Results of Empirical Evaluation 33
3.5 Future works 35

Chapter 4 Space Efficient Data Structures for Nearest Larger Neighbor 39

4.1 Introduction 39
4.2 Indexing NLV queries on 1D arrays 43
4.3 Encoding NLN queries on 2D binary arrays 44
4.4 Encoding NLN queries for general 2D arrays 50
4.4.1 2D NLN in the encoding model – distinct case 50
4.4.2 2D NLN in the encoding model – general case 53
4.5 Open problems 63

Chapter 5 Simultaneous encodings for range and next/previous larger/smaller value queries 64

5.1 Introduction 64
5.2 Preliminaries 67
5.2.1 $2d$-Min heap 69
5.2.2 Encoding range min-max queries 72
5.3 Extended DFUDS for colored 2d-Min heap 75
5.4 Encoding colored 2d-Min and 2d-Max heaps 80
5.4.1 Combined data structure for $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$ . . . 82
5.4.2 Encoding colored $2d$-Min and $2d$-Max heaps using less space 88
5.5 Open problems 89

Chapter 6 Encoding Two-dimensional range Top-k queries **90**

6.1 Introduction 90

6.2 Encoding one-sided range Top-k queries on 2D array 92

6.3 Encoding general range Top-k queries on 2D array 95

6.4 Open problems 99

Chapter 7 Conculsion **100**

Bibliography **102**

요약 **112**

List of Figures

Figure 3.1 An example of an (ad-hoc) enumerative code. The graph is given on the top (leaves shown shaded) and the code-words, and their phrases, right. 20

Figure 3.2 Memory test 36

Figure 3.3 rank_1 test 36

Figure 3.4 Random select_1 test 37

Figure 3.5 Hard select_1 test 37

Figure 3.6 Mixed test 38

Figure 4.1 Suppose the nearest block that contains a **1** from the $(2,2)$-block is the $(4,3)$-block. Then $d(4,3)$ contains the blocks $(2,5),(3,4),(4,3)$ and $(5,2)$, in that order. We can find the nearest **1** in NE(2,2) using $\mathsf{RMQ}(2,3)$ on $D_{(4,3)}$ and $\mathsf{RMQ}(1,3)$ on $D_{(4,4)}$. 49

Figure 4.2 The positions of *useful* and *dummy* elements in a 6×6 array. In this example, the dummy elements (X's) are in the range [1..24] and the useful elements (O's) are in the range [25..26]. 51

Figure 4.3 Pointers in $encoding_{2D}$ and $encoding_{grid}$ 59

Figure 5.1 Colored $2d$-Min heap of A 71

Figure 5.2 Encoding of $2d$-Min heap and $2d$-Max heap of A 72
Figure 5.3 $D_{\mathsf{CMin(A)}}$, $\mathsf{pre_rank}_{\mathsf{CMin(A)}}$, $V_{min}[i]$, $\mathsf{node}_{V_{min}}$, $\mathsf{pre_select}_{\mathsf{CMin(A)}}$ and $\mathsf{node_color}_{\mathsf{CMin(A)}}$ for colored $2d$-Min heap 77
Figure 5.4 Data structure combining the colored $2d$-Min heap and colored $2d$-Max heap of A. C is represented in uncompressed form. 83

Figure 6.1 Top-k encoding of the 2D array A when $k = 2$ 97

List of Tables

Table 1.1 The summary of previous results and our results. C = number of codewords, ℓ = codeword size, RQ_{min} = {RMinQ, RLMinQ, RRMinQ, RkMinQ} and RQ_{max} = {RMaxQ, RLMaxQ, RRMaxQ, RkMaxQ}. 6

Table 3.1 Characteristics of the test files 24

Table 3.2 Compression ratios of the test files. 24

Table 3.3 Total phrase length of test files (as % of compressed output), excluding RLE codewords 29

Table 6.1 The summary of our results for Top-k queries on $m \times n$ 2D array. $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$ 92

Chapter 1

Introduction

The amount of data increases much faster than the capacity of the storage devices in recent days. Also the rapid growth of the mobile device market requires storing the large amount of the data into the limited space. To overcome this problem, one of the best solution is compressing the original data. For example, there are numerous text compression algorithms [73, 76] which can store the text data using much less space than its original size. However, the raw compressed data itself cannot support most of the queries (for example, random accesse and extracting arbitrary substrings from the LZ77-compressed [76] texts) without uncompressing the whole compressed data and this makes a huge bottleneck to answer the queries. In many areas of data science like *Real-time data analysis* or *BigData*, supporting queries efficiently for large data sets is a important issue. Therefore, storing data with compact size while supporting queries efficiently becomes a crucial issue.

One can define a *compressed (space-efficient) data structure* as a data structure which takes less space than the conventional data structure while supporting same set of queries. Let OPT be the minimum number of bits required to store the data while supporting the query (information-theoretic lower

bound). The compressed data structures are divided into three types as follows. (i) *Implicit* if the data structure uses $OPT+O(1)$ bits, (ii) *Succinct* if the data structure uses $OPT+o(OPT)$ bits and (iii) *Compact* if the data structure uses $O(OPT)$ bits. Since it is hard to design implicit data structure while supporting queries efficiently due to its space requirement, the main goal in the theoretical computer science area is usually maintain the size of the data structures as succinct while supporting the queries efficiently.

Succinct Data Structures were first introduced by Jacobson [42]. He showed how to represent the static trees and graphs while supporting the various queries efficiently. There are succinct data structures for various problems such as *Indexable dictionaries* [64, 69], *Permutations* [54], *Equivalence relations* [52] and *Range minimum queries* [10, 27].

1.1 Computational model

In this thesis, we assume a standard word-RAM model [53] as computational model. Word-RAM model is a variant of the classic RAM (random access machine) model [14] which is the computational model under the realistic assumption of a computer. In this model, each memory cell stores a *word* of size ω and we can read and write any cell in the memory in $O(1)$ time. Also, we can support 'C-style' arithmetic operations $(+, -, *, /, \%)$ and boolean operations $(\&, |, \wedge, \sim, <<, >>)$ on words in $O(1)$ time. Since each word needs to be large enough to store pointers and indices to access the data in practice, we set the word size $\omega = \Theta(\lg n)$ bits[1] for n input elements. We count space in terms of the number of bits used.

1.1.1 Encoding and indexing models

We consider the data structures in two different models that have been studied in the succinct data structures literature, namely the *indexing* and *encoding*

[1] We use $\lg n$ to denote $\log_2 n$

models. In both these models, the data structure is created after preprocessing the input data. In the indexing model, the queries can be answered by probing the data structure as well as the input data, whereas in the encoding model, the query algorithm cannot access the input data. The size of the data structure in the encoding model is also referred to as the *effective entropy* [35, 68] of the input data, with respect to the problem.

Suppose there is a set of input data S and set of all queries Q. If we can reconstruct any element in S from the answer to the queries in Q, encoding has no space advantage compared to indexing for answering the queries. But if the number of all possible answers on S induced by Q is significantly smaller than the size of S, then we can save some space by using the encoding model which doesn't need to store the original data.

1.2 Contribution of the thesis

In this thesis, we propose the following space-efficient data structures.

- **Bitvector based on variable-to-fixed encodings**: *Bitvector* is a data structure which supports rank and select operations on a given bit-string. In this thesis, we design *bitvectors based on variable-to-fixed compressed bit-string* (V2F bitvector). In the theoretical view, we show that regardless of the V2F compression algorithms, there exists a V2F bitvector which has low *redundancy* (that is, the difference in size between the V2F bitvector and the compressed bit-string is asymptotically smaller than the size of the compressed bit-string) and supports rank and select in constant time. We also give practical implementations of V2F bitvector and evaluate their practical performance with various existing implementations. The empirical evaluation shows that our V2F bitvector has low redundancy and supports rank and select queries efficiently compared to other previous implementations.

- **Encoding and indexing of the nearest Larger neighbor queries**: Given an elements in an array from a total order, *the nearest larger neighbor* (NLN) query returns the position of the element which is closest to the query position, and is larger than the element at the query position. We consider the NLN problem on one and two-dimensional arrays. For one-dimensional array of size n, we propose an $O((n/c)\lg c)$-bit index which supports NLN queries in $O(c)$ time, for any parameter $2 \leq c \leq n$, improving the structure of Fischer et al. [28]. For a $n \times n$ two-dimensional array, we first show that $\Theta(n^2)$ bits are necessary to encode NLN queries. Also, we give an optimal encoding which supports NLN queries in constant time on a two-dimensional array, improving the NLN encoding from Jayapaul et al. [44].

- **Simultaneous encodings of various range queries and next/previous larger/smaller value queries**: Given a sequence of n elements from a total order, we consider the encoding which supports range minimum query and its variants (RMinQ, RLMinQ, RRMinQ, RkMinQ), range maximum query and its variants (RMaxQ, RLMaxQ, RRMaxQ, RkMaxQ) and next/previous larger/smaller value queries (NLV, NSV, PLV, PSV). In this thesis, we obtain a $4.585n + o(n)$-bit encoding which supports all these queries in constant time for a sequence of size n. This improves the Fischer's $5.08n+o(n)$-bit encoding [25] which supports same set of queries in constant time. We also prove that if the query time is not concerned, we can obtain a $4.088n + o(n)$-bit encoding which supports all these queries.

- **Encoding of range Top-k queries on a two-dimensional array**: Given an elements in an array from a total order and a rectangular range in an array. *Range top-k* (Top-k) query returns the positions of k largest elements in the range, In this thesis, we consider various encodings which support Top-k queries on a two-dimensional array. This problem has not

been studied. For an $m \times n$ array, we first obtain an optimal encoding for one-sided Top-k queries whose query range is restricted to $[1 \ldots m][1 \ldots i]$, for $1 \leq i \leq n$. Also, we propose the $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$-bit encoding for Top-k queries on a two-dimensional array with any rectangular query ranges by extending the Top-k encoding on a one-dimensional array proposed by Gawrychowski and Nicholson [30]. In most of encodings do not support the Top-k queries in efficient query time.

The summary of previous results and our results for these data structures are in Table 1.1.

1.3 Organization of the thesis

The rest of this thesis is organized as follows. In Chapter 2, we introduce data structures for supporting rank and select queries which are basic operations on various space-efficient data structures. In Chapter 3, we describe compressed bit vectors based on variable-to-fixed encodings which have low redundancy in both theoretical and practical implementations. In Chapter 4 we consider the encoding and indexing data structures for Nearest Larger Neighbor (NLN) problem on one-dimensional and two-dimensional arrays. In Chapter 5, we propose encodings that support various range queries (range minimum, range maximum and their variants), and previous and next smaller/larger value queries. In Chapter 6, we propose the various encodings that supports Top-k queries. Finally in Chapter 7, we summarize the results in this thesis and give some open problems.

Input data	Query	Space Usage (bits)	Query time	Encoding/Indexing	Reference
Bit string $X[1 \ldots n]$	rank_1, select_1	$C\ell + O(C \log(n/C))$	$O(1)$	Encoding	This thesis
1D array $A[1 \ldots n]$, $1 \leq c \leq n$	PLV (NLN)	$O((n/c)\log c + (n \log n)/c^2)$	$O(c)$	Indexing	[44]
		$O((n/c)\lg c)$	$O(c)$	Indexing	This thesis
2D array $A[1 \ldots n][1 \ldots n]$	NLN	$O(n^2 \lg\lg n)$	$O(1)$	Encoding	[44]
		$O(n^2)$	$O(\lg\lg\lg n)$	Indexing	[44]
		$O(n^2)$	$O(1)$	encoding	This thesis
2D binary array $A[1 \ldots n][1 \ldots n]$, $1 \leq c \leq n^2$	NLN	$O(n^2/c)$	$O(c)$	Encoding	This thesis
1D array $A[1 \ldots n]$	RMinQ, PSV	$2n + o(n)$	$O(1)$	Encoding	[27]
	RQ_{min}, PSV NSV	$2.54n + o(n)$			[25]
	RMinQ, RMaxQ	$3n + o(n)$			[30]
	RQ_{min}, RQ_{max} PSV, NSV PLV, NLV	$4.585n + o(n)$			This thesis
2D array $A[1 \ldots m][1 \ldots n]$	One-sided, sorted Top-k	$n\left\lceil \lg(\sum_{i=0}^{\min(m,k)} \binom{m}{i}(\frac{k!}{(k-i)!}))\right\rceil$	$O(k)$	Encoding	This thesis
	Four-sided, unsorted Top-k	$O(mn \lg n)$			
	Four-sided, sorted Top-k	$m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$			

Table 1.1 The summary of previous results and our results. C = number of codewords, ℓ = codeword size, RQ_{min} = {RMinQ, RLMinQ, RRMinQ, RkMinQ} and RQ_{max} = {RMaxQ, RLMaxQ, RRMaxQ, RkMaxQ}.

Chapter 2

Preliminaries

In this chapter, we introduce data structures for answering rank and select queries, one of the fundamental problems in succinct data structures. These structures are used for the various space-efficient data structures proposed in this thesis.

Given a string $S[1 \dots n]$ over an alphabet Σ, rank and select are defined as follows.

- $\mathsf{rank}_\alpha(S, i)$: The number of occurrences of α in the first i positions of S, for any $\alpha \in \Sigma$.
- $\mathsf{select}_\alpha(S, i)$: The position of the i-th α in S, for any $\alpha \in \Sigma$.

In the thesis, we only consider the case when S is a *bit-string*, i.e, $\Sigma = \{0, 1\}$. We first introduce the following lemma from [69] that gives a succinct encoding of S.

Lemma 2.1 ([69]). *Let S be a string of length n containing m 1s. One can encode S using $\lg \binom{n}{m} + o(n)$ bits to support both $\mathsf{rank}_x(S, i)$ and $\mathsf{select}_x(S, i)$ in constant time, for $x \in \Sigma$. Also, one can decode any $\lg n$ consecutive bits in S in $O(1)$ time.*

Also, we use following lemmas from [37] that can be used to support rank and select operations on moderately dense bit strings (i.e., bit strings in which the number of zeros and ones is at most a poly-log factor smaller than the length of the string).

Lemma 2.2 ([37]). *Let S be a bit-string of length n containing m 1s. If $m \geq n/(\lg n)^c$, for some constant $c > 0$, one can support rank_1 and select_1 in $O(1)$ time using $\lg \binom{n}{m} + O(m)$ bits.*

Lemma 2.3 ([37]). *Given integer $n > m > 0$ such that $\min\{n - m, m\} \geq n/(\lg n)^c$ for some constant c, one can store a bit-string S with $n_0 \leq n - m$ 0s and $n_1 \leq m$ 1s, using $\lg \binom{n}{n-m} + O(\min\{n, n - m\})$ bits, such that select_0 and select_1 are supported in $O(1)$ time.*

Now we introduce another lemma from [62]. This lemma shows that if the number of ones is significantly less than the number of zeros, one can encode S using less space than the encoding described in Lemma 2.1 (but do not support queries in constant time).

Lemma 2.4 ([62]). *Let S be a bit-string of length n containing m 1s. One can encode S using $O(m \lg(n/m))$ bits such that rank_1 and select_1 can be supported in $O(n/m)$ time.*

One can generalize the rank and select queries as follows. Given a string $S[1 \dots n]$ and pattern string p over the alphabet Σ, $\mathsf{rank}_p(S, i)$ returns the number of occurrences of pattern p in the first i positions of S, and $\mathsf{select}_p(S, i)$ returns the position of the i-th occurrence of pattern p in S. Combining the results from [56] and [69], one can show the following lemma for generalized rank and select queries on a bit-string.

Lemma 2.5 ([56], [69]). *Let S be a bit-string of length n over the containing m 1s. One can encode S using $\lg \binom{n}{m} + o(n)$ bits to support both $\mathsf{rank}_p(S, i)$ and*

$\mathsf{select}_p(S, i)$ *in constant time, for any binary pattern* p *with length* $|p| \leq 1/2 \lg n$. *Also, one can decode any* $\lg n$ *consecutive bits in* S*, in constant time.*

Chapter 3

Compressed bit vectors based on variable-to-fixed encodings

3.1 Introduction.

A *bitvector* is a fundamental building block of many space-efficient data structures. As described in Chapter 2, given a bit-string X of *length* n with *weight* m (i.e., with m **1** bits), the aim is to pre-process X to support the following operations, for any $b \in \{0, 1\}$:

- $\mathsf{rank}_b(X, i)$ returns the number of occurrences of b in the first i positions of X.
- $\mathsf{select}_b(X, i)$ returns the position of the ith b in X.

These operations can be supported in $O(1)$ time using $n + o(n)$ bits of space [13]. If X is a (uniformly) random bit-string, it cannot be be compressed, and this space bound is therefore, in the worst case, optimal to within lower-order terms. However, bit-strings encountered in practical applications are often compressible, and many algorithmic applications use bitvectors on bit-strings that are constructed to be *sparse*—contain $m = o(n)$ **1**s—and such bit-strings are

compressible to $o(n)$ bits. Starting from the work of [64, 70], there is now a rich theory of *compressed* bitvectors, which aim to use space approaching that used by a compressed representation of the bit-string, for many different measures of compressibility[1]. The most basic measures of compressibility are *density-sensitive*, i.e. they depend only upon the length n and weight m of the bit-string. These are the *information-theoretic minimum*, $B(n,m) \stackrel{\text{def}}{=} \lceil \log \binom{n}{m} \rceil$ bits, and the *zeroth-order empirical entropy*, $H_0(X) \stackrel{\text{def}}{=} -\sum_{i=0}^{1} p_i \lg p_i$, where $p_1 = m/n$ and $p_0 = 1 - p_1$; the compressed bit-string size should then be $nH_0(X) + O(1)$ bits. Note that if $m = o(n)$ then $B(n,m) \approx nH_0(X) = o(n)$.

Instance-sensitive measures[2], where the compressibility of the string X is a function of X, are more diverse, and include the *k-th order empirical entropy* H_k and functions of the gaps between successive **1**s [40], or the size of the output produced by a grammar-based compressor to X. In general, such measures would show that a bit-string X is at least as compressible as a density-sensitive measure on X.

Previous Work Although there have been many papers on implementations of bitvectors [18, 17, 36, 38, 51, 74] (and some researchers have implemented bitvectors as part of more complex data structures), there are fewer papers on compressed bitvectors for sparse bit-strings. It should be noted that supporting $O(1)$-time rank/select operations using reasonable space is possible only when $m = n/(\log n)^{O(1)}$ [66]. In this range, even the density-sensitive measure gives $O(m \log(n/m)) = O(m \log\log n)$ bits, so a compressed bitvector is significantly smaller than either an uncompressed bitvector, which takes $\Theta(n)$ bits, or viewing X as the characteristic vector of a set and storing the set explicitly, which requires $O(m \log n)$ bits. Such moderately sparse bit-strings are also of great

[1] As is common in the area of succinct and compressed data structures, we focus on *empirical* measures, i.e., those that are a function of the bitstring X itself, rather than measures derived by postulating a probabilistic model for generating bit-strings.

[2] A related term, *data-aware*, is used in [40].

practical interest. One focus of this chapter is on representing such bit-strings.

The following authors have considered practical data structures for sparse bit-strings. Geary et al. [32] considered "uniformly" sparse bit-strings, but their techniques do not apply to general sparse bit-strings, and they do not perform a stand-alone evaluation of their bitvector. Gupta et al. [40] considered very sparse bit-strings, and showed that instance-sensitive measures related to the γ and δ codes outperform density-sensitive ones, but they did not report on moderately sparse bit-strings. Delpratt et al. [18] considered Golomb coding in the context of the select_1 operation. Okanohara and Sadakane [63] performed arguably the first comprehensive evaluation, but focused mostly on the density-sensitive measures. Navarro et al. [59] considered rank and select on grammar-compressed bit-strings, but do not provide a stand-alone evaluation. Navarro and Providel [58] also provide an implementation of compressed bitvectors. This, again, targets the density-sensitive measures. Very recently, Kärkkäinen et al. [49] presented a hybrid approach combining run-length encoding (RLE), raw encoding and explicit encoding, and showed good performance on a class of bit-strings obtained from text indexing applications.

Our results In this chapter we explore the use of *variable-to-fixed (V2F)* encodings of a bit-string, which have only been partially explored previously. Our results show that this approach leads to very compact and high-performance compressed bitvectors. Indeed, we give a theoretical basis for the low *redundancy* (wasted space) of the codes as well as that of the bitvector. An ℓ-bit V2F code partitions the input bit-string into a concatenation of variable-length *phrases*. Each phrase, except the last one, is constrained to belong to a given dictionary D of $\leq 2^{\ell}$ bit-strings; the last phrase is a non-null prefix of a dictionary entry. Once the input bit-string is parsed, each phrase is replaced by its position in the dictionary, stored as a ℓ-bit *codeword*. V2F codes are studied in the data compression literature due to their desirable properties such as

error-resilience, but it appears that there has not yet been a comprehensive investigation of V2F bitvectors. That said, the class of V2F codes is quite broad: it includes e.g. RLE and grammar-based compression, and it is possible that there are application-specific implementations of V2F bitvectors inside other data structures.

Our main conceptual contributions are as follows:

- We argue that in general, V2F coding is an effective approach to reduce the *redundancy* of the bitvector, or the difference between the compressed size of the bit-string and the size of the bit-vector data structure. The redundancy can dominate the space usage of compressed bitvectors: e.g. if $m = O(n/(\log n)^2)$, the space usage of the compressed bitvector of [70], which is $B(n, m) + O(n \log\log n / \log n)$ bits, is dominated by the redundancy. We show that for the density range of interest, V2F compressors give redundancy that is asymptotically smaller than the compressed size of the bit-string.

- In practice, we give an approach for density-sensitive encoding of a bitvector that has a significantly lower (intrinsic) redundancy over that of Navarro and Providel [58] by using *Tunstall* codes [73]. Furthermore, we show that the Tunstall code always achieves H_0 empirical entropy with low redundancy (previously this was known only for random inputs).

- We give a new class of *enumerative* V2F codes. These codes generalize both Khodak's code [50, 20], a close relative of the Tunstall code, and RLE. Finally, a hybrid enumerative code which combines Khodak's code with RLE achieves excellent compression performance, even on bit-strings that are relatively incompressible by density-sensitive measures.

- We argue, as does Vigna [74], that practical implementations of select based on the method of "sampling" must address the issue of *long gaps*,

which many implementations do not do. This is because in practice, guarding against a worst-case scenario for long gaps (using ideas which derive back to [13]) consumes a lot of space. Although it seems real-life bit-strings *can* have a number of reasonably long gaps, we note that the typical test (select a random **1**) is likely to give running times that are independent of the distribution of the underlying bit-vector. We propose a test that would "fairly" and "naturally" test the handling of a select implementation in the presence of long gaps, and show that implementations that do not guard against long gaps do indeed slow down.

Our implementation has been structured into two independent parts: a framework for rank and select, and a compressor-specific part that deals with individual codewords. This highlights the challenges faced by a V2F-based bitvector, and offers a lot of room for innovation with respect to how to deal with codewords. The fact that there is indeed room has already been hinted at in [58, 59], but we argue that reasonable performance is obtained by a default implementation in many cases.

The rest of this chapter is structured as follows. Section 3.2 describes a general result on supporting rank and select operations on bit-strings compressed using V2F schemes. In Section 3.3, we describe the V2F schemes that we use in the experimental evaluation. Section 3.4 describes the details of our implementation, and also the results from the experimental evaluation of V2F schemes. Section 5.5 contains some future directions.

3.2 Bit-vectors using V2F coding

As indicated earlier, the redundancy of a compressed bitvector targeting a particular compressibility measure is the difference between the size of the bit string under that compressibility measure and the size of the bitvector. Pătraşcu [65] showed that rank/select can be supported in $O(1)$ time using

$B(n,m) + n/(\log n)^{O(1)}$ bits, and that for $m = \Theta(n)$, this is optimal [67]. However, there is no evidence yet that the approach of [65] is feasible in practice. Another approach to low-redundancy compressed bitvectors achieves $B(n,m) + O(m(\log\log n)^2/(\log n))$ bits and $O(1)$ time for the range $m = n/(\log n)^{O(1)}$ [36, 64]. While the redundancy is not as low as Pătraşcu's, it is roughly a log factor less than the compressed bit-string – a very desirable feature. We now show that this holds in general for V2F codes under modest assumptions:

Theorem 3.1. *Given a bit-string X of n bits encoded as C codewords using a V2F code of ℓ bits each. Further assume that there is a data structure, which given a codeword c, supports* rank *and* select *in $O(1)$ time on the phrase $p(c)$ that the codeword c stands for. Then we can support $\mathsf{select}_1(X,i)$ and $\mathsf{rank}_1(X,i)$ in $O(1)$ time using $C\ell + O(C\log(n/C))$ bits, provided that $C = n/(\log n)^{O(1)}$.*

Proof. For any bit-string s, let $w(s)$ denote the weight of s, and for $i = 1,\ldots,C$, let c_i denote the i-th codeword, and let $m = w(X)$. The data structure consists of two bitvectors on the following bit-strings:

- the *ones distribution* bit-string $OD = \mathbf{0}^{w(p(c_1))}\mathbf{1}\mathbf{0}^{w(p(c_2))}\mathbf{1}\ldots\mathbf{0}^{w(p(c_C))}\mathbf{1}$.
- the *phrase size* bit-string $PS = \mathbf{1}\mathbf{0}^{|p(c_1)|-1}\mathbf{1}\mathbf{0}^{|p(c_2)|-1}\mathbf{1}\ldots\mathbf{1}\mathbf{0}^{|p(c_C)|-1}$.

It is easy to see that $|OD| = m + C$, $w(OD) = C$, $|PS| = n$ and $w(PS) = C$.

- To compute $\mathsf{select}_1(X,i)$, we first determine the number of codewords before the codeword in which the selected $\mathbf{1}$ lies as $j = \mathsf{rank}_1(OD, \mathsf{select}_0(OD,i))$. We then determine the total number of $\mathbf{1}$s in $c_1,\ldots,c_j$ as $k = \mathsf{select}_1(OD,j) - j$, and the start position of c_{j+1} in X as $d = \mathsf{select}_1(PS, j+1) - 1$. Finally, we select the $i-k$-th $\mathbf{1}$ in $p(c_{j+1})$, add d to the answer and return.

- To compute $\mathsf{rank}_1(X,i)$, we first find the codeword j in which the i-th position lies by $j = \mathsf{rank}_1(PS,i)$. We then determine d, the start position of c_j, and k, the number of $\mathbf{1}$s in $c_1,\ldots,c_{j-1}$, as before, and return $k + \mathsf{rank}_1(p(c_j), i-d)$.

We store OD using Lemma 2.3, which uses $O(C\log(m/C))$ bits. In addition, we pad OD to length n by adding zeros at the end (so that the condition in Lemma 2.2 applies), and store the resulting bit-string as well as PS using Lemma 2.2, which takes $O(C\log(n/C))$ bits. □

Remark 1. *We will typically choose* $\ell = \Theta(\log n)$ *bits. Thus, provided that* $\log(n/C) = o(\log n)$*, the redundancy will be smaller than the size of the compressed output, which is* $C\ell$ *bits.*

3.3 V2F compression algorithms for bit-strings

We now describe different V2F compression schemes that we use to compress the given bit-string X. Each of these schemes partitions X into a sequence of variable-length *phrases*. Each phrase, except the last one, belongs to a *dictionary* of size $M = 2^\ell$ that is constructed from the source string. The dictionary entries are also referred to as *code words*. The compressed representation of X simply consists of a sequence of ℓ-bit codes (from the dictionary) corresponding to each phrase. The only difference between various compression algorithms is the way in which they construct the dictionary.

3.3.1 Tunstall code

For a given phrase length L, the Tunstall code is designed to maximize $E[L]$, the expected number of source letters per phrase for a memoryless source [73]. Given an input bit-string X, the dictionary constructed by Tunstall's algorithm can be represented as a full binary tree T (i.e., every node has 0 or 2 children), which we refer to as the *Tunstall tree*. Each edge in T corresponds to a bit, and each phrase corresponds to a leaf in T. The phrase corresponding to a leaf u can be obtained by concatenating the symbols corresponding to the edges on the root-to-leaf path to u.

We now describe the algorithm to construct a Tunstall code for X with

$M = 2^\ell$ codewords. First, we define some terminology. Letting $n = |X|$, and m be the weight of X, define $p_0 = 1 - m/n$ and $p_1 = m/n$. The probability[3] of a bit-string $b_1 b_2 \ldots b_l$ is defined to be $\prod_{i=1}^{l} p_{b_i}$. Each leaf in T is labelled by the probability of the corresponding phrase, and each internal node is labelled by the sum of the probabilities of its children. The algorithm is as follows:

(1) Start with 2-level rooted tree with the root connected to two leaves, corresponding to **0** and **1**.

(2) Pick a leaf node which has the highest probability and grow two leaves on it.

(3) Repeat step (2) while the number of leaves in the tree is at most M.

It has long been known that the Tunstall code achieves zeroth-order entropy (defined appropriately) for random sources [73] and its redundancy[4] for random sources has been shown to be low [20]. We now show that the redundancy of the Tunstall code with respect to empirical entropy is also low.

Theorem 3.2. *Given a bit-string X with length n and weight m, suppose that it is encoded using a Tunstall code with $M = 2^\ell$ codewords, constructed taking $p_0 = 1-m/n$ and $p_1 = m/n$ as the probabilities of **0** and **1** respectively. Assume, without loss of generality, that $p_1 \leq p_0$ and further assume that $\ell = \Theta(\log n)$ and $\log(1/p_1) = o(\log n)$. Then $C\ell \leq nH_0(X) + O(nH_0(X)\log(1/p_1)/\ell)$.*

Proof. Say that a *final* leaf refers to a leaf of the Tunstall tree T at the end of the algorithm. Observe that the probabilities of the leaves of T at any stage of the algorithm add up to 1. Hence, while the number of leaves is less than M, there will always be a leaf with probability greater than $1/M$, so we will never expand

[1]This is not a probability in the true sense, of course, since we are dealing with a given fixed bit-string X.

[2]Here the term redundancy has been overloaded to refer to the size of the compressed output relative to the ideal compressed size.

a leaf with probability at most $1/M$. It follows that the minimum probability of a final leaf is greater than p_1/M. Let p^* be the maximum probability of any final leaf. Since all final leaves are created by expanding leaves with probability $\geq p^*$, and at least one final leaf must have probability $\leq 1/M$, it follows that $p^* p_1 \leq 1/M$ or $p^* \leq 1/(p_1 M)$.

Suppose that the output of parsing X according to the Tunstall code comprises C codewords $c_1, c_2, \ldots, c_C$. Let $\Pr(c_i)$ denote the probability of the phrase of c_i. Then $-\log \prod_{i=1}^{C} \Pr(c_i) = -\log(p_0^{n-m} p_1^m) = nH_0(X)$. However, $\prod_{i=1}^{C} \Pr(c_i) \leq (1/(p_1 M))^C$ from the above, which gives $nH_0(X) \geq C \log(p_1 M)$, or:

$$nH_0(X) + C \log(1/p_1) \geq C\ell \tag{3.1}$$

With the above assumption on p_1, it is not hard to verify that $C\ell = O(nH_0(X))$, and plugging this back into Equation (3.1) we get that $C\ell \leq nH_0(X) + O(nH_0(X) \log(1/p_1)/\ell)$. □

Remark 2. 1. *Since we assume* $\log(1/p_1) = o(\ell)$*, the redundancy is a lower-order term.*

2. *Note that a similar argument shows that* $C\ell \geq nH_0(X) - C\log(1/p_1)$*. In other words, the output of Tunstall coding is never much less than the empirical entropy.*

Theorems 3.1 and 3.2 allow us to obtain a small improvement in redundancy over the bitvector of [36, Thm 2], which previously had the lowest known redundancy of any bitvector that does not use the (fairly complex) technique of *informative encoding* [36] or its successors [65].

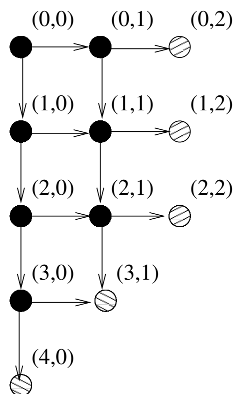
Corollary 3.1. *Let X be a bit-string with length n and weight m. There is a bit-vector that supports* rank_1 *and* select_1 *in $O(1)$ time when $m = n/(\log n)^{O(1)}$ and uses $nH_0(X) + O(m \frac{\log(n/m) \log\log n}{\log n})$ bits.*

Proof. Since $H_0(X) = O((m/n)\log(n/m))$, from Theorem 3.2 the output of the Tunstall coding occupies $nH_0(X) + O(m(\log(n/m))^2/\log n)$ bits. To augment it with rank_1 and select_1, we use Theorem 3.1. The additional data structures use $O(C\log(n/C)) = O\left(\frac{nH_0(X)}{\log n}\log\left(\frac{n\log n}{nH_0(X)}\right)\right)$ bits. Simplifying, we get that the redundancy of the bitvector is $O\left(m\frac{\log(n/m)}{\log n}(\log(n/m) + \log\log n)\right) = O(m\frac{\log(n/m)\log\log n}{\log n})$ bits.

Finally, it only remains to explain how to do $\mathsf{rank}/\mathsf{select}$ on an individual phrase in $O(1)$ time. Taking the notation of Theorem 3.1, we create the concatenated bit-string $p(0)p(1)\ldots p(2^\ell - 1)$. The maximum length L of an individual phrase must satisfy $(p_0)^L \geq p_1/M$, from which one can obtain that $L = O(n\log n/m)$. Since $n/m = O(\log n)^{O(1)}$, if we choose $\ell = (\log n)/2$, the bit-string containing the concatenated phrases will be of size $O(n^{1/2+\epsilon})$, for any positive constant $\epsilon < 1/2$. By building a bit-vector on this bit-string and furthermore explicitly storing the start of each phrase, as well as the cumulative numbers of **1**s in this bit-string (using $O(2^\ell \log n) = O(n^{1/2+\epsilon})$ bits), rank and select on individual phrases can be supported in $O(1)$ time. □

3.3.2 Enumerative codes

We define a class of *enumerative* codes as follows. An enumerative code can be specified as a (directed) graph on a subset of the vertices (i, j), for $i \geq 0$ and $j \geq 0$. A vertex (i, j) may either have no outgoing edges (be a *leaf*) or point to *both* vertices $(i + 1, j)$ and $(i, j + 1)$. Furthermore, a vertex (i, j) is *complete* if either it has indegree 2, or either i or j is 0 (and its indegree is 1); and *incomplete* otherwise. All incomplete vertices must be leaves. Finally, the vertex $(0, 0)$ is always in the graph. Given such a graph, the code is specified as follows. For every complete leaf (i, j) we allocate $\binom{i+j}{j}$ codewords, which code for all phrases with i **0**s and j **1**s. For every incomplete leaf (i, j), if its (sole) predecessor is $(i, j - 1)$ then we allocate all $\binom{i+j-1}{j-1}$ codewords, which code for all phrases with i **0**s and j **1**s that end with a **1**. If its predecessor is $(i - 1, j)$,



Codeword	Phrase	Vertex in the graph
0	11	(0,2)
1	011	(1,2)
2	101	
3	0011	(2,2)
4	0101	
5	1001	
6	0001	(3,1)
7	0010	
8	0100	
9	1000	
10	0000	(4,0)

Figure 3.1 An example of an (ad-hoc) enumerative code. The graph is given on the top (leaves shown shaded) and the codewords, and their phrases, right.

then we allocate all $\binom{i+j-1}{j}$ codewords, which code for all phrases with i **0**s and j **1**s that end with a **0** (see Fig. 3.1). Clearly, we must ensure that the total number of codewords is at most 2^ℓ.

Given such a graph, we parse the input-bit string as follows. Each phrase starts at $(0, 0)$. If we are currently at the non-leaf vertex (i, j), upon reading a **1**, we move to $(i, j+1)$; upon reading a **0**, we move to $(i+1, j)$. By construction, both these vertices are in the graph. If we are at a complete leaf (i, j) then we have so far read a phrase with i **0**s and j **1**s; since all possible $\binom{i+j}{j}$ such phrases have associated codewords, we choose the appropriate codeword, output it and restart from $(0, 0)$. Arriving at an incomplete leaf (i, j) from $(i, j-1)$, we must have read a phrase with i **0**s and j **1**s where the last bit is a **1**, so we output the appropriate codeword (the other case is similar), and restart from $(0, 0)$. We now give examples of enumerative codes.

RLE.

RLE is a special case of enumerative coding. To have codes for runs of **0**s and **1**s of length $1, \ldots, 2^{\ell-1}$, the corresponding graph contains the non-leaf

vertices $(0, i)$ and $(i, 0)$, and the leaf vertices $(1, i)$ and $(i, 1)$ for $i = 1, \ldots, 2^{\ell-1} - 1$, together with the leaf vertices $(0, 2^{\ell-1})$ and $(2^{\ell-1}, 0)$. A codeword is thus assigned to each phrase of the form $\mathbf{0}^i\mathbf{1}$ and $\mathbf{1}^i\mathbf{0}$ for $i = 1, \ldots, 2^{\ell-1} - 1$; and one each for $\mathbf{0}^{2^{\ell-1}}$ and $\mathbf{1}^{2^{\ell-1}}$.

Khodak Code.

The Khodak code [20] is is obtained by modifying Step (2) of the Tunstall algorithm in Section 3.3.1 to pick all the leaf nodes with highest probability and grow two leaves on all of them. It is known that every Khodak code is a Tunstall code, and that for the same dictionary size, the Khodak code has asymptotically the same average phrase length as the Tunstall code [20]. We show:

Theorem 3.3. *Any Khodak code is an enumerative code.*

Proof. We first prove an auxiliary lemma that implies that, when the probabilities of zero and one are not the same, the dictionary constructed by the Khodak algorithm is a subset of the dictionary constructed by the Tunstall algorithm – by observing that the order in which the leaves are expanded in both the algorithms is the same; but Khodak algorithm may stop earlier if there is not enough space to expand all the leaves with same probability.

Lemma 3.1. *For rational number $0 < d < 1, d \neq 1/2$, there are no nonnegative integers x, y, z, w such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$.*

Proof. Suppose that there exist nonnegative integers x, y, z, w such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$. Let $d = n/m$ for positive integers m and n, such that n and m are relatively prime. Without loss of generality, assume that $d > \frac{1}{2}$ and $z + w \geq x + y$. Then it is easy to argue that $z \geq x$ and $y \geq w$. Thus, $m^{z+w-x-y}(m-n)^{y-w} = n^{z-x}$. If m is even, then left side is even while right side is odd as n is relatively prime to m. If m is odd and n is even, then left

side is odd while right side is even. Finally, if both m and n are odd, the left side is even while the right side is odd. $\square$

We now prove Theorem 3.3. Define T^k as the tree whose leaves represent the phrases of the Khodak code (similar to T in the Tunstall code). Next, let $T^k(i,j)$ be the set of all leaves in T^k which represent the phrases with i zeros and j ones. We say that $T^k(i,j)$ is complete if T^k contains all possible $(\binom{i+j}{i})$ phrases with i zeros and j ones (this is analogous to the definition of completeness of nodes in the enumerative codes). Now to prove Theorem 3.3, it is enough to prove the claim that if the Khodak algorithm expands the leaves in $T^k(i,j)$ then $T^k(i,j)$ is complete. The claim holds if the zero density is 1/2, because in this case, T^k is always a complete binary tree (and each expansion step expands all the leaves). Now we assume that the one density is strictly larger than the zero density. Since for every step in the Khodak algorithm, i and j for expanding $T^k(i,j)$ are uniquely determined by the Lemma 3.1, the claim can be proved by the induction on the number of expansion steps taken by the Khodak algorithm.

(Basis step) In the first step, we expand the leaf $T^k(0,1)$ which is complete.

(Inductive step) Assume the hypothesis that the claim is true if the number of steps is at most r. In the $r+1$ step, suppose we expand $T_k(i,j)$ which is not complete. Note that $T^k(i,j)$ is generated by expanding $T^k(i,j-1)$ or $T^k(i-1,j)$. Since both $T^k(i,j-1)$ and $T^k(i-1,j)$ are expanded before $r+1$-th step (because they have the smaller probability than $T^k(i,j)$), by induction hypothesis, they are complete. But if we expand $T^k(i,j-1)$ and $T^k(i-1,j)$ which are complete, $T^k(i,j)$ becomes complete, contradicting the assumption. $\square$

Hybrid Enumerative Coding.

To obtain better compression using enumerative encoding, we reserve a fraction of codewords for run-length codes, and use the remaining for the Khodak code-

words. The run-length codewords are divided among **0** runs and **1** runs based on the densities of **0**s and **1**s.

3.3.3 LZW algorithm

Lempel-Ziv-Welch (LZW) algorithm [76] is a well-known dictionary-based compression algorithm. The dictionary constructed by the LZW algorithm has no fixed bound on its size, and it is not stored as part of the compressed text as it can be reconstructed during decompression. However, since our approach uses a bounded-size dictionary (with M codewords), we modify the LZW algorithm as follows: We first construct the dictionary in one pass over the string, as in the normal LZW algorithm till its size is M, and use that to parse the whole string in a second pass. Also, unlike the original LZW algorithm, the modified algorithm requires both the compressed string as well as the dictionary for decompression.

3.3.4 Empirical evaluation of the compressors

We now describe the compression performance of the above algorithms. We set $\ell = 16$ so each dictionary has $2^{16} = 65536$ codewords. For implementing RLE and Hybrid algorithms, we determined the maximum length of runs of **0**s in the RLE part of the dictionary as the smaller of ($2^{15} \times$ density of **0**) and maximum length of runs of **0**s in the test file (the maximum length of **1**s in the RLE part is also determined in the same way).

Test files

Table 3.1 summarizes the characteristics of the bit-strings we used in our experiments. `factor9.6` and `proteins` are obtained by parsing two XML files, and outputting $\mathbf{0}^i\mathbf{1}$ when a text node of length i is encountered [18]. `Z-Accidents` and `Z-Pumsb2` are used in a data structure for mining frequent patterns from benchmark data sets [33]. `dblp_100` and `english_100` are the FM-indices [23]

Bit-string	Total Size (10^6 bits)	Density of **0**s	Max run-length of **0**s	Max run-length of **1**s
`factor9.6`	812.0	0.964	2927	1
`proteins`	374.9	0.900	27376	1
`Z-Accidents`	903.3	0.996	4,250,294	1,315
`Z-Pumsb2`	1661.1	0.999	1,138,613	7,774
`dblp_100`	680.8	0.629	5,252,073	3,115,460
`english_100`	784.3	0.710	2,142,856	743,383
`rand_dblp`	680.8	0.629	42	20
`rand_english`	784.3	0.710	50	17

Table 3.1 Characteristics of the test files

Bit-string	Tunstall	LZW	Enumerative code			H_0	*Logsum*
			Khodak	RLE	Hybrid		
`factor9.6`	0.242	**0.151**	0.241	0.573	0.228	0.223	0.236
`proteins`	0.466	**0.104**	0.475	1.585	0.546	0.466	0.484
`Z-Accidents`	0.045	0.035	0.046	0.058	**0.030**	0.041	0.111
`Z-Pumsb2`	0.007	0.006	0.007	0.007	**0.004**	0.008	0.097
`dblp_100`	0.975	0.145	0.975	0.369	**0.136**	0.952	0.201
`english_100`	0.869	**0.285**	0.869	0.771	0.306	0.868	0.305
`rand_dblp`	**0.956**	0.971	**0.956**	4.872	**0.956**	0.952	0.991
`rand_english`	**0.874**	0.891	**0.874**	4.146	**0.874**	0.868	0.910

Table 3.2 Compression ratios of the test files.

of the text files in Pizza&Chili Corpus [24]. We use the implementation of FM-index from fm-index++ [72]. **`rand_dblp`** and **`rand_english`** are generated at random, but setting their length and density to be the same as `dblp_100` and `english_100`, respectively. The test bit-strings can be classified into four types based on their properties. The bit-strings `factor 9.6` and `proteins` are fairly sparse but have relatively short runs of **0**s and **1**s. The bit-strings `Z-Accidents` and `Z-Pumsb2` are very sparse and have some very long runs of **0**s. While `dblp_100` and `english_100` are quite dense, they have long runs of **0**s and **1**s; obviously, their randomly generated analogues do not have such long runs.

Table 3.2 shows the compression ratio achieved by the compressors on the test bit-strings. We also give the H_0 values of the bit-strings and their *Logsum*

value, defined as follows. If we divide a given bit-string X of length n into fixed-size blocks B_i, $i = 1 \ldots \lceil n/63 \rceil$ of size 63 and each B_i has weight $m(i)$, $Logsum(X)$ is defined as $\frac{1}{n} \sum_{i=1}^{\lceil n/63 \rceil} (\log(\binom{63}{m(i)}) + 6)$. *Logsum* is an estimate of the standard density-sensitive approach to compressed bitvectors used in [70] and predecessors (referred to as RRR in what follows), based on the implementation of [58], which is optimized for low redundancy. We make the following observations:

- There is a negligible difference in compression ratio between the Tunstall and Khodak codes. While Tunstall/Khodak are sometimes better than H_0, the variation is small, as implied by Remark 2.

- *Logsum* is sometimes significantly better than H_0, e.g. in `dblp_100` and `english_100`. The reason is that all-**0** and all-**1** blocks (which occur frequently in these bit-strings) compress far better than would be suggested by the overall density of these bit-strings. However, the additive overhead of 6 bits per block means that *Logsum*'s performance is poor on bit-strings such as `Z-Pumsb2` and `Z-Accidents`, as well as the random bit-strings.

- Among the enumerative codes, Hybrid uniformly performed the best, even easily outperforming RLE on very sparse files. It is also often the overall best performer, but it does perform poorly relative to LZW on the XML bit-strings. We speculate that this is because in XML files, identical elements may have similar-length text nodes under them (e.g., a `zipcode` element will usually contain a text string of length 5) and LZW is able to capture such long-range patterns.

3.4 Practical implementation of bitvectors based on V2F compression.

We now describe our implementation (in C++), which follows the general approach used by many existing schemes such as that of [58], but with some modifications. The bit-vector class is (largely) independent of the compressor, and takes as input two files: one which contains the codewords and the phrases, and another which contains 16-bit codewords output by the compressor. The codewords output by the compressor are read into a codeword array, and the rest of the bit-vector has three parts: a $\mathsf{rank}/\mathsf{select}_1$ *index*, a table for scanning codewords, and finally a class that deals with rank/select operations on individual codewords. We now describe each in detail.

$\mathsf{rank}/\mathsf{select}_1$ index. For rank we divide the bit-string into *rank blocks* of size B, where the i-th block consists of the bits numbered iB through $(i+1)B-1$. For each block, we store the position of the first codeword that intersects the block, the weight at the start of that codeword, and the absolute position in the bit-string where that codeword begins. The default is $B = 1024$, but this can (and should) be varied according to the compressibility of the bit-string, so that each block (on average) spans a moderate number (say 30-50) of codewords. For select_1, we use the standard "sample and scan" approach [13] used by most select implementations including [74, 58, 34]. We choose a sampling parameter s and divide the bit-string into *select blocks*, where the i-th select block begins at the position of the is-th **1**, and scan this select block to answer $\mathsf{select}_1(j)$ queries for $j = is+1, \ldots, (i+1)s-1$ (Type 0 blocks). This approach does not guarantee a good time bound if the **1**s are distributed non-uniformly: in the worst case, one may need to scan $\Theta(n)$ bit positions. To mitigate this effect, we treat *long gaps* differently [13]: we choose a threshold LG, and whenever a select block is larger than LG, we store the positions of the **1**s in the block explicitly (Type 1 blocks). Even though the number of long gaps is at most n/LG, LG must be relatively

high as storing **1** positions is costly. In addition, if a long gap spans only a moderate number of codewords, it is treated as a Type 0 block. For Type 0 blocks, as with rank blocks, we store codeword/phrase alignment information, and cumulative information. We choose s satisfying $m/s = \Theta(n/B)$, so that the number of select and rank blocks is similar (so on average both rank and select queries scan similar numbers of codewords).

Scanning a Rank/Select Block. To perform a rank operation, or a select_1 on a Type 0 block, we need to scan a rank/select block to find the codeword that contains position i. The key loop in scanning a block is to (a) read a codeword at a time from the compressed bit string, (b) obtain (and accumulate) the length of its phrase and its weight, and (c) determine both the codeword where position i lies, and the offset of position i within that codeword. This is done by table lookup, and this gives rise to the most important constraint on the size of M: it must comfortably fit "into cache" (as the cache is likely to contain other data in real applications). On our machine, this suggests that ℓ should be limited to 16; the table then takes 512KB[5].

Long Gaps: a Theoretical View. In this paragraph, we illustrate the potential asymptotic gains by using V2F codes in terms of protecting against long gaps in the "sample and scan" approach to select_1. This illustration makes a number of mappings from current practical parameter choices to asymptotic functions, which by its very nature involves a certain amount of guesswork: we do not hope to convince everybody of these mappings.

We begin by assuming that most practical implementations can be viewed asymptotically as using a block size of $B = \Theta((\log n)^2)$ bits and work by accessing $O(1)$ random memory locations and scanning $\Theta(\log n)$ consecutive memory locations, where each location comprises $\Theta(\log n)$ bits. This is justified as there

[1]For the current compressors, no phrase can be longer than 2^ℓ bits, so this could be reduced to 256KB.

is evidence that due to address translation, the cost of a random memory access is $O(\log n)$ [48]. For simplicity we consider the case of a bit-string with weight $\Theta(n/\log n)$, i.e. one whose compressed size is $O(n \log\log n/\log n)$ bits, and assume that we wish to achieve a redundancy of $O(n/\log n)$ bits. A typical sampling factor would be $s = \log n$, so that the cost of pointers to the sampled locations is $O((m/s)\log n) = O(n/\log n)$ bits. We would choose the long gap parameter to be $L = \Theta((\log n)^3)$, so that the cost of storing the locations of the **1**s in the at most $O(n/L)$ long gaps is $O((n/L)s \log n) = O(n/\log n)$ bits. This makes the worst-case cost of scanning a gap which of length exactly L to be $O((\log n)^2)$. However, in (say) Tunstall or Khodak coding, a bit-string with length $L = \Theta((\log n)^3)$ with weight $s = O(\log n)$ is compressed to $O((\log n)^2)$ bits or $O(\log n)$ codewords, which can be scanned in $O(\log n)$ time. (Note that the compressed size of these L bits is more than the information-theoretic bound for these L bits, but the Tunstall code is based on the *global* density and encodes each **0** using $\log(n/(n-m)) = O(1/\log n)$ bits and each **1** using $\log(n/m) = O(\log\log n)$ bits.)

Implementation of Codeword Operations. Having located the codeword containing the answer, we perform an appropriate rank/select on its phrase. The default implementation of rank on a codeword concatenates all phrases into a bit-string similar to Corollary 3.1 and stores it in a bit-vector supporting rank [34], together with two words per codeword to allow rank on an individual phrase to be reduced to rank on the bit-vector. select on each phrase is done by explicitly storing the positions of the **1**s in the phrase in an array. We estimate the *fixed* overhead to be about 4 `ints` per codeword, or 1MB overall. However, a potentially major variable overhead is the size of the rank phrase bit-vector and the phrase select array.

An obvious optimization is that for codes known to comprise runs of **0**s or **1**s, indicated by an additional *type* field stored in the length/weight table,

File name	Khodak	LZW	Enumerative code	
			Khodak	Hybrid
factor9.6	2.15%	7.24%	2.15%	2.15%
proteins	1.26%	3.01%	1.23%	0.90%
Z-Accidents	38.36%	7.22%	38.30%	19.60%
Z-Pumsb2	668.61%	200.14%	667.98%	73.25%
dblp_100	0.16%	15.53%	0.16%	0.54%
english_100	0.17%	52.20%	0.17%	0.22%
rand_dblp	0.16%	0.16%	0.16%	0.16%
rand_english	0.17%	0.16%	0.17%	0.17%

Table 3.3 Total phrase length of test files (as % of compressed output), excluding RLE codewords

we directly (and trivially) answer rank and select queries on the corresponding phrase. Table 3.3 shows the size of the resulting rank phrase bit-vector (the phrase select array is usually smaller). As suggested by Corollary 3.1, for Khodak codes, the size of the dictionary is negligible for relatively high-density bit-vectors. The overhead is much larger for the Z-Accidents and Z-Pumsb2, though Hybrid codes, which have many RLE codes, have smaller dictionaries than Khodak codes. Nevertheless, for very sparse bit-strings, it is clear that this naive approach is inappropriate.

3.4.1 Testing Methodology

The code was written in C++, and compiled with g++ 4.8.3 with optimisation level 3, and tested on a 64-bit machine with 64GB RAM and an Intel Xeon E7450 6-core CPU clocked at 2.40GHz with 3×3 MB shared L2 caches and 12MB L3 cache, running Fedora Linux (kernel version 3.16.2). Tests were performed for the memory usage, and four tests for the speed of this structure, as follows.

Memory Test. To determine the true physical memory used by these data structures, we initialize them and then fork a process that allocates memory

equal to the physical memory of the machine, which will result in all other processes' pages to be swapped out. Putting the forked process to sleep, we then perform rank and select operations and then measure the resident memory of the process.

In this test, we implemented bitvector based on V2F codes in two ways - practical implementation described in this section and implementation based on Theorem 3.1. In the latter implementation (based on Theorem 3.1), we implemented OD and PS using RRR and sdarray which have low redundancy on dense and sparse bit-strings respectively. Since this implementation is not optimized for the speed tests, we only used the practical implementation for the other four tests.

We also measure the memory usage of our implementations by a self-reporting procedure which checks the total size of the main data structures using size reporting functions. Testing results shows that the measured memory size is larger than self-reported memory size because of the initial space used by OS and other variables in the program. But difference between them does not exceed 10MB in all test files.

rank_1 Test. To test the speed of rank, we perform $\mathsf{rank}_1(i)$ n times, for random $i \in 1..n$.

Random select_1 Test. Like the rank_1 test, this test performs $\mathsf{select}_1(i)$ n times, for i selected randomly from $1..m$. Although "sample-and-scan" approach does not guarantee a good time bound, if the bit-string has long gaps, several implementations, including `RSDic`, do not guard against long gaps. However, their performance for random select tests on random bit-vectors (which typically don't have long gaps) is good. Vigna [74] proposed testing on pathological bit-strings to determine whether an implementation had good worst-case select performance. We note, however, that essentially *regardless* of the input bit-

string, a random select test will not be able to distinguish between "sample-and-scan" bit-vectors, that deal with long gaps and those that don't. Specifically, observe that in any select block, the expected time taken to perform a select of one of the **1**s in this block, assuming a fairly even distribution of the **1**s within this block, is essentially *proportional to its length.* Since a random select accesses each select block with equal probability, it is not hard to see that the average running time of a random select is essentially *independent* of the distribution of select block lengths; i.e., a random select test is unlikely to distinguish between an easy bit-string and a pathological one. To address this issue, we propose a *hard* select test, described below.

Hard select_1 Test. We perform 2^{19} random rank_1 queries, and store the results in an array Q of the same size (with repetitions). We then repeat the following, n times: select a random index i in Q and perform $\mathsf{select}_1(Q[i]+1)$. Doing this will select a **1** in a select block with probability proportional to the length of the select block (since the argument of the rank query falls in a select block with probability proportional to its length), and thus focusses on the harder select queries in a bit-string.

Mixed Test. We initialise an array Q of size 2^{19} to values from random $\mathsf{rank}_1(i)$ as above. We cycle through the array and perform $\mathsf{select}(Q[i]+1,1)$ as above, but then do a $\mathsf{rank}_1(j)$ for a random index j, and store the result in $Q[i]$. Each such pair of rank and select operations is performend n times.

For our benchmarks, we choose the LZW code for XML bit-strings and Hybrid code for other bit-strings as F2V coders which gives the best compression ratio for their bit-strings. For comparison, we used Okanohara's rsdic code [61] (based on [58]), sdarray from Okanohara and Sadakane [63] and RRR from `sdsl-lite` [34]. We now describe the rsdic and sdarray bitvectors briefly.

The *Compressed Rank Select Dictionary*, rsdic is based on the structure proposed by Navarro and Providel [58]. It divides the bit-string X into fixed-sized blocks of length $t = \lceil (\lg n)/2 \rceil$. The set of all possible blocks are divided into classes based on the number of 1's in the block. Hence, each block can be identified by a pair (k, r), where k is the class number which is simply the weight of the block, and r is the index of the block in a table containing the set of all possible blocks in the class, in some canonical order, say, the lexicographic order. Thus, the representation of any block can be stored in $\lceil \lg(t+1) \rceil + \lceil \log \binom{t}{k} \rceil$ bits. Also, one can rebuild a block "on-the-fly" using its representation, without storing any additional precomputed tables. For the sequence of blocks constituting the given bit-string X, it stores the first components (i.e., the weights of the blocks) in an array K, using fixed size entries of $\lceil \lg(t+1) \rceil$ bits each; the second components of all the blocks in the sequence are concatenated and stored as a bitvector R. To enable fast access into R, it first groups every $\lfloor \lg n \rfloor$ consecutive blocks into a superblock. For every superblock, it then stores a pointer into R to point to the starting position of the representations corresponding to its blocks. In addition, we also store the rank up to the first bit in each superblock. To compute the rank for a given position, we first find the superblock containing the position, and do sequential search from the first block in the superblock. To support the select operation, we first perform a binary search to find the superblock containing the required position, and then scan the blocks within the superblock. The size of R can be shown to be at most $nH_0(X) + o(n)$ bits, and the size of K is $n\lceil \lg(t+1)/t \rceil = o(n)$ bits. Thus the space usage of rsdic is $nH_0(X) + o(n)$ bits.

Okanohara and Sadakane [63] proposed the sdarray which either stores an sarray when the given bit-string X is *sparse*, or a darray when X is *dense*. To describe the sarray, consider an array $x[0, \ldots, m-1]$ where $x[i]$ stores the position of the $(i+1)$-th 1-bit in X. We choose a parameter t, and store the lower $z = \lceil \lg t \rceil$ bits of each $x[i]$ in an array L such that $L[i] = x[i] \bmod t$. The

upper $w = \lfloor \lg(n/t) \rfloor$ bits of each $x[i]$ is encoded in unary to obtain a bit vector, H, of length $m+t$, along with auxiliary structures to support rank and select in $O(1)$ time on H. The operation select on X can be supported in constant time by finding the upper bits using the select operation on H, and accessing the lower bits from the array L. To support rank(i) on X, we first find a smallest element whose position is greater than $\lceil i/2^w \rceil \cdot 2^w$ using select on H and count number of ones sequentially from here using H and L. By choosing $t = 1.44m$, the total size of sarray becomes $1.92m + m(\lg(n/m)) + o(m)$ bits.

The construction of darray first divides the given bit-string X into blocks of L ones each, and constructs an array $P[0, \ldots, n/(L-1)]$ such that $P[i]$ stores the position of iL-th one in X. These blocks are represented based on their length. If the length of a block is more than $(\lg n)^4$, it is represented by storing the positions of all the ones in it. Otherwise, its representation consists of the position of every $(\lg n)$-th one in the block, using $L(\lg L)/\lg n$ bits. To support select(i), we first find the block that contains the answer using P. If this block is longer than $(\lg n)^4$, we can read the answer from its representation. Otherwise, we use the representation of the block to find a sequence of $(\lg n)$ positions, one of which corresponds to the required answer, and scan the sequence to find the answer. The *rank* operation is supported using an approach similar to that of rsdic. By choosing $L = (\lg n)^2$, the size of darray can be limited to $n + o(n)$ bits, including the bit-string X.

3.4.2 Results of Empirical Evaluation

Memory test. Practical implementation of bitvectors based on V2F used significantly less memory than the competition in most cases (see Fig. 3.2); the exception is sdarray with `Z-Accidents` and `Z-Pumsb2` and RRR with `rand_dblp`, `rand_english` and `english_100`. In the former case (for `Z-Accidents` and `Z-Pumsb2` files), despite the V2F compressed bit-string being significantly less than H_0, the compressed size is so small that the fixed overhead of the phrase

rank/select structure dominates. Also for latter three files, their V2F compression ratios are close to *Logsum*, and the overhead in the bitvector based on V2F implementations is more than that of RRR.

Although the redundancy in Theorem 3.1 is less than (little-oh of) the compressed output size in theory, for the implementation based on Theorem 3.1, the space overhead is $1 \sim 3$ times more than the compressed output size, in all the test files except Z-Accidents and Z-Pumsb2 (for which the compressed output size is significantly smaller). This is because the $O(\lg(n/C))$ term in the redundancy can be larger than the codeword size even though the value of $\lg(n/C)$ in these files is $4 \sim 6$, which is less than the codeword size.

rank_1 test. Generally speaking, apart from sdarray, which is not optimized for rank, all others are comparably fast. However, sdarray does better than rsdic on the Z-vectors, possibly because it fits in cache due to its much lower memory usage, and the V2F bitvectors and RRR both do relatively poorly on the random bit-strings (see Fig. 3.3).

Random select_1 test. As expected, sdarray is generally the fastest, but loses out a little on the FM-index files, as it cannot compress them. The V2F bitvector is the second-best, and is very close to the best, in most cases, but performs slightly worse on the random files (see Fig. 3.4). rsdic and RRR show significant weakness on the Z-vectors and XML bit-strings respectively.

Hard select_1 test. rsdic is the only bit-vector that does not guard against long gaps, and performs very poorly (up to 20 times slower) on three of the input files (see Fig. 3.5). V2F bitvectors do the hard select_1 test at roughly the same speed as the random select_1, and thus demonstrate their resilience.

Mixed test. The V2F bitvectors are the best overall performers in this test, since they show good performance for both the hard select test and the

rank test (see Fig. 3.6).

3.5 Future works

In this chapter, we consider theoretical and practical implementations of compressed bitvectors. There is much room for further investigation. For instance, the naive approach to operations on individual phrases, as well as the relatively simple approach to supporting rank/select, leads to an overhead that is rather high for highly compressible bit-strings (admittedly, these are so sparse as to test the boundaries of our stated aim of targeting "moderately compressible" bit-strings). This could be overcome by adhering more closely to the theoretical result, and making greater use of on-the-fly decoding also can be considerd. Apart from the Tunstall/Khodak/Enumerative codes, we have not explored V2F codes in any non-trivial way. Much more work is clearly possible along this axis as well.

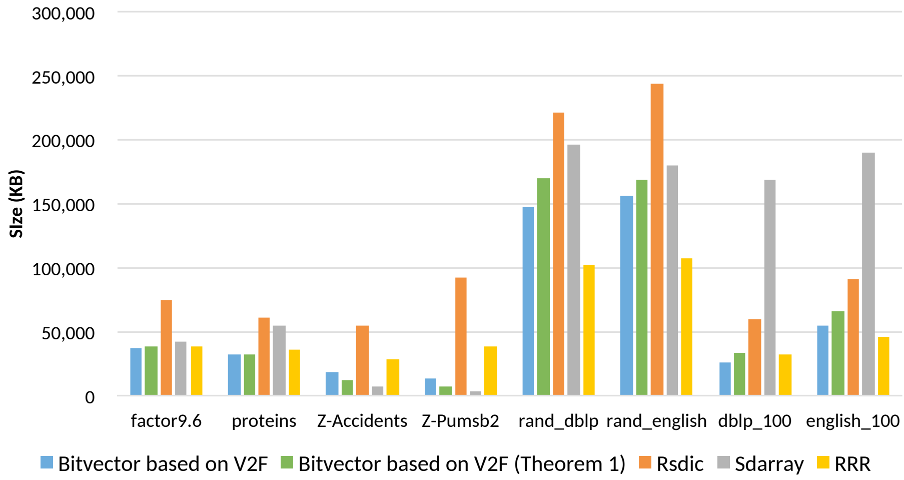


Figure 3.2 Memory test

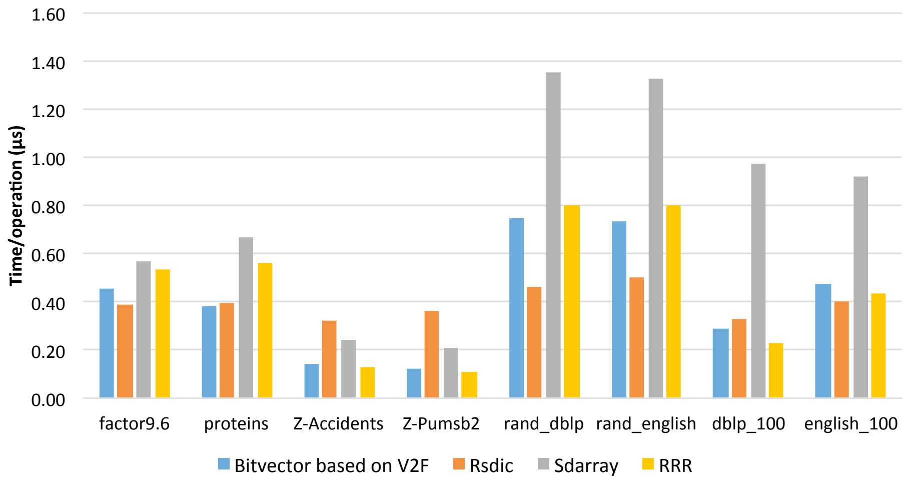


Figure 3.3 rank_1 test

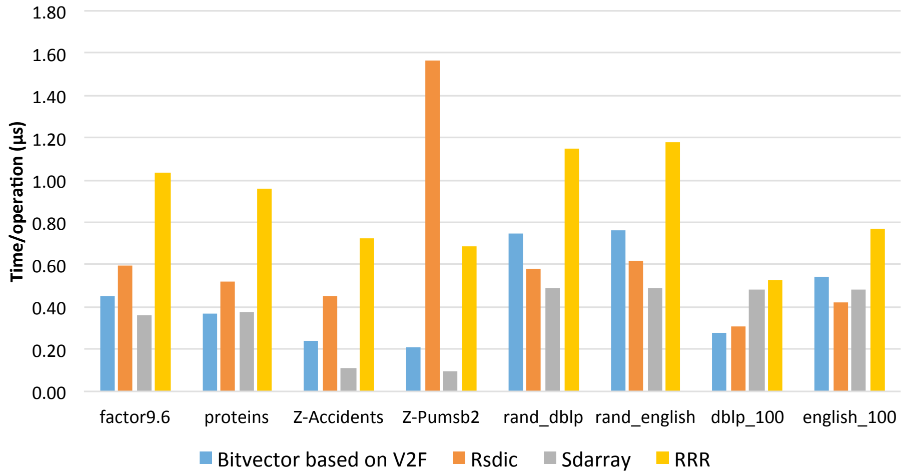


Figure 3.4 Random select_1 test

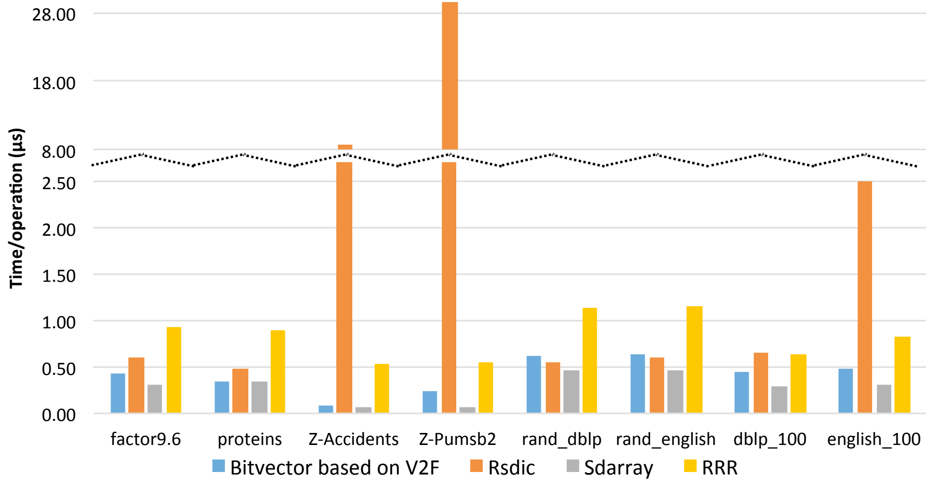


Figure 3.5 Hard select_1 test

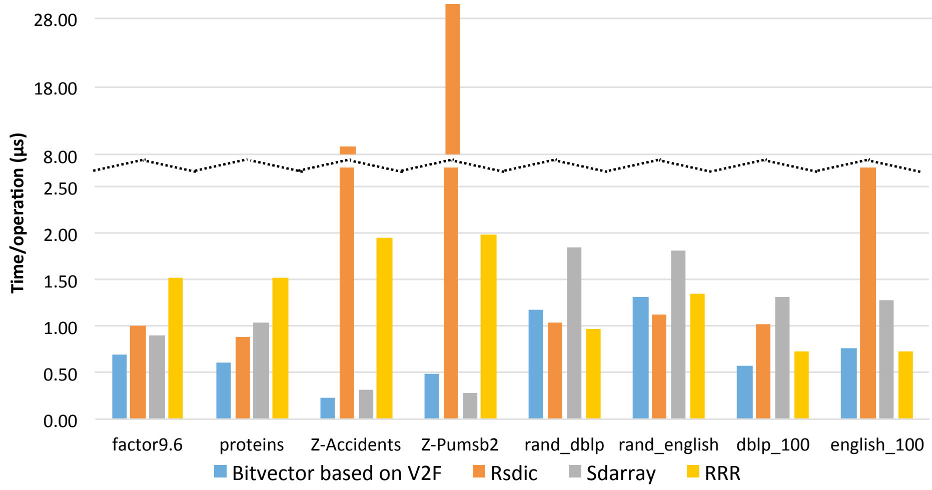


Figure 3.6 Mixed test

Chapter 4

Space Efficient Data Structures for Nearest Larger Neighbor

4.1 Introduction

Given a sequence of n elements from a totally ordered set, and a position in the sequence, the *nearest largest neighbor* (NLN) query asks for the position of an element which is closest to the query position, and is larger than the element at the query position. More formally, given an array $A[1 \dots n]$ of length n containing elements from a totally ordered set, and a position i in A, we define the query:

- $\mathsf{NLN}(i)$: return the index j such that $A[j] > A[i]$ and $|i - j| = \min\{k : A[i + k] > A[i] \textit{ or } A[i - k] > A[i] \text{ for } k > 0\}$. Ties are broken to the left, and if there is no element greater than the query element, the query returns the answer ∞.

In a similar way, we can the define NLV (*next larger value*), and PLV (*previous larger value*) queries, which return the position of the nearest larger neighbor to the right and left, respectively, of the query position. In a symmetric way,

one can also define nearest *smaller* neighbor problems. In this chapter, we will stick to the version that seeks the larger neighbors.

We exhibit connections between the NLN problem and the well-studied problem of supporting *range maximum* queries on a given array. Given an array A, the query $\mathsf{RMaxQ}(i, j)$ (range maximum query) returns a position k between i and j such that $A[k]$ is a maximum element among $A[i, \dots, j]$.

Two-dimensional NLN We also consider a natural extension of the NLN problem to two-dimensional arrays. Here, we define the NLN of a query position as the closest position in the two-dimensional (2D) array, in terms of the L_1 distance, that contains an element larger than the element at the query position. More formally, given a position (i, j) in $A[1 \dots n][1 \dots n]$, $\mathsf{NLN}((i, j)) = (i', j')$ such that $A[i', j'] > A[i, j],$ and $|i - i'| + |j - j'| = \min\{|x| + |y| : A[i + x, j + y] > A[i, j]\}$. If there is no element greater than the query element, the query returns the answer (∞, ∞).

Previous Work and Motivation These kinds of problems have attracted much attention. In addition to the data structuring problems, the *off-line* variants, usually called *All Nearest Larger Neighbors* (or similar), which consist in computing answers for all possible input positions, have also been studied. For example, Berkman et al. [7] gave efficient parallel algorithms for the one-dimensional (1D) off-line problem and showed their importance as a preprocessing routine for answering range minimum queries, triangulation algorithms, reconstructing a binary tree from its traversal orders and matching a sequence of balanced parentheses [7].

Fischer et al. [28] considered the problem of supporting NLV and PLV, and showed how a data structure supporting these two queries can be used in obtaining *entropy-bounded compressed suffix tree* representation. (They considered the min version of the problem instead of max, and named the operations NSV and

PSV, for the next and previous smaller values, respectively.) They considered the problem of supporting NLV and PLV in the indexing model, and obtained the following time-space tradeoff result. For any $1 \le c, \ell \le n$, one can use:

$$O\left(\frac{n}{c}\lg c + \ell\frac{n\lg\lg n}{\lg n} + \frac{n\lg n}{c^\ell}\right)$$

bits of space and answer queries in $O(c\ell)$ time. As given, they are unable to go below $O(n\lg\lg n/\lg n)$ space, and use more space than we do whenever $c = \omega(\lg n)$. As mentioned later, we improved the trade-off to $O((n/c)\lg c)$ bits with $O(c)$ time. To attain $O((n/c)\lg c)$ space for $c = (\lg n)^{\Omega(1)}$, one can choose $\ell = O(1)$ and obtain $O(c)$ time. For smaller values of c, the middle term in the space usage will never dominate for reasonable values of ℓ (clearly, we must always choose $c \ge 2$ and $\ell = O(\lg\lg n)$ in this range) and it suffices (and is optimal) to choose $\ell = O(\lg_c \lg n) = O(\lg\lg n - \lg\lg c)$. Thus, for any $c = O(\lg n)$, their running time for space $O((n/c)\lg c)$ is $O(c(\lg\lg n - \lg\lg c))$, and our solution is better for small enough c. Jayapaul et al. [44] gave a solution that uses $O((n/c)\log c + (n\log n)/c^2)$ bits and $O(c)$ time; this space usage equals ours for $c = \Omega(\log n/\log\log n)$ but is worse otherwise.

Fischer et al. [28] also gave an encoding that supports the PSV and NSV queries in constant time, using $4n + o(n)$ bits. The encoding size was later reduced to the optimal $2.54n + o(n)$ bits by Fischer [25]. Jayapaul et al. [44] observe that this can be further improved to $2n + o(n)$ bits if all the elements are distinct. For the case of binary sequences, the data structure version of the NLN problem can be solved by building an auxiliary structure to support rank and select queries on the bit-strings [69]. This uses $o(n)$ bits of extra space, in addition to the input array, and answers NLN (and also NLV and PLV) queries in $O(1)$ time.

Given a 2D array, Asano et al. [3] considered the *All Nearest Larger Neighbours* problem which asks for computing the NLN values for all the elements in the input array. They showed that this problem can be solved in $O(n^2\lg n)$

time (and more generally, for any d-dimensional array in $O(n^d \lg n)$ time). To the best of our knowledge, the data structure version of the 2D NLN problem, in which we are interested in constructing a data structure that answers online queries efficiently, has not been considered earlier.

Our results Our main results are as follows.

- For the case of 1D, we look at the problems in indexing model. We provide an algorithm that matches the tradeoff for NLN [44]. For NLV, our algorithm achieves the time-space product of $O((n/c) \lg c)$ (where the query takes $O(c)$ time) while the lower bound is $\Omega(n)$.
- For the 2D NLN problem in the encoding model, we first show that $\Omega(n^2)$ bits are necessary to encode the array to support NLN queries, even when all the elements are distinct. We then describe an asymptotically optimal $\Theta(n^2)$-bit encoding that answers queries in $O(1)$ time, even when all the elements are not distinct. One can achieve this result easily when all the elements in the array are distinct. However, distinctness is a strong assumption in these kinds of problems. For example, in the 1D case with distinct values, NLV and PLV are obtained relatively easily from the Cartesian tree, giving an $2n + o(n)$ bit-encoding. By contrast, if we do not assume distinctness, the optimal space is about $2.54n$ bits, and the data structure achieving this bound is also more complex [25]. Also, Asano et al. [3] remark that the off-line problem for any dimension is "simplified considerably" if one assumes distinctness.

As we note, in the 1D case, the NLV and PLV problems are closely connected to the RMaxQ problem. In the 1-D case, there is no asymptotic difference between the encoding complexity of RMaxQ and NLV/PLV. The 2D RMaxQ problem has received a great deal of attention lately [19, 10, 9, 8]. It is known that any 2-D RMaxQ encoding takes $\Omega(n^2 \lg n)$ bits [19, 10]; thus, our $\Theta(n^2)$-bit en-

coding for 2D NLN shows that encoding complexity of NLN is asymptotically different from RMaxQ in the 2D encoding scenario (unlike the 1-D case).

The rest of this chapter is structured as follows. Section 4.2, we describes an indexing for answering NLV queries on 1D arrays. In Section 4.3, we propose an optimal time-space tradeoff encoding for answering NLN queries on 2D binary arrays. In Section 4.4, we propose an encoding for answering NLN queries on 2D arrays which takes asymptotically optimal space and supports NLN queries in constant time. Finally, in Section 4.5, we give some open problems.

4.2 Indexing NLV queries on 1D arrays

In this section, we give a result for the NLV problem in the indexing model on 1D array. The approach follows closely the proof of Fischer et al. [28], which in turn adapts ideas from Jacobson's representation of balanced parentheses sequences [42], and is given in full for completeness.

We begin with the following lemma gives an encoding for answering NLV queries on 1D array.

Lemma 4.1 ([28, 27, 44])**.** *Given a 1D array A of size n, there exists a data structure in the encoding model that uses $2n + o(n)$ bits and solves NLV queries in $O(1)$ time.*

Now we state our result in this section.

Theorem 4.1. *Given a 1D array A of size n, there exists a data structure which supports NLV queries in the indexing model in $O(c)$ time using $O((n/c) \lg c)$ bits for any parameter $2 \leq c \leq n$.*

Proof. Divide A into n/c blocks of size c. For any value $1 \leq i \leq n$, if i and $\mathsf{NLV}(i)$ are in the same block, say that i is a *near* value, otherwise say that i is a *far* value. Consider a block B and suppose that one or more of its far values have an NLV in block B', for an arbitrary block B'. Then the leftmost far value

in B whose NLV is in B' is called a *pioneer*, and its NLV is called its *match*. It is known that there are $O(n/c)$ pioneers in A [42].

We maintain a bit-vector V in which the i-th bit is a 1 if $A[i]$ is a pioneer or a match of one, and 0 otherwise. This bit-vector has length n and contains $O(n/c)$ 1's, so by Lemma 2.4, we can store it in $O((n/c)\lg c)$ bits and perform $\mathsf{rank}/\mathsf{select}$ queries on it in $O(c)$ time. Next, we take the sequence S_p consisting of all pioneers and their matches. This sequence is of length $O(n/c)$. We represent this sequence using Lemma 4.1 using $O(n/c)$ bits, to support NLV queries in $O(1)$ time. We claim that if $k = \mathsf{NLV}(j)$ in S_p, then $\mathsf{select}_1(V,k) = \mathsf{NLV}(\mathsf{select}_1(V,j))$ in A. Suppose that this claim is not true. This means there is a pioneer i_p such that $\mathsf{NLV}(i_p)$ is the value between i_p and the match of i_p. It cannot be the case that i_p and $\mathsf{NLV}(i_p)$ are in the same block, since i_p is a far value. If i_p and $\mathsf{NLV}(i_p)$ are in different blocks, then $\mathsf{NLV}(i_p)$ is the match of i_p. So the claim is true.

To answer the query $\mathsf{NLV}(i)$, we first check to see if the answer is in the same block as i taking $O(c)$ time. If so, we are done. Else, (assuming wlog that $A[i]$ is not a pioneer value) we find the first pioneer p_i before position i by doing $\mathsf{rank}/\mathsf{select}$ on V. As $A[i] < A[p_i]$, $\mathsf{NLV}(i)$ is less than or equal to the match of p_i. Since i is the far value in this case, $\mathsf{NLV}(i)$ and $\mathsf{NLV}(p_i)$ are in the same block. We find the corresponding position of $\mathsf{NLV}(p_i)$ in S_p using the NLV encoding of S_p and find the $\mathsf{NLV}(p_i)$ using $\mathsf{rank}/\mathsf{select}$ on V. Finally we scan left from $\mathsf{NLV}(p_i)$ to find $\mathsf{NLV}(i)$. The overall time taken to answer the query is $O(c)$. $\square$

4.3 Encoding NLN queries on 2D binary arrays

In this section, we first give an optimal encoding for NLN, and using this obtain an time-space trade-off for an NLN index for a 2D binary array.

Theorem 4.2. *There is a data structure that takes $O(n^2)$ bits for any binary array $A[1\ldots\ n][1\ldots\ n]$, and supports NLN queries in $O(1)$ time.*

Proof. Given a query position p, we compute $\mathsf{NLN}(p)$ by computing the positions of the nearest larger values in all four *quadrants* induced by a vertical and a horizontal line passing through p, and then returning the closest of these four positions as the answer (the point p is included in all the four quadrants). Thus, it is enough to describe a structure that supports finding the position of the nearest larger value in (say) the upper-right quadrant; in the rest of this proof, we use NLN_{NE} to denote this.

Given a position $p = (i, j)$, let $q = (i', j')$ be its NLN_{NE} if there is a **1** in the upper-right quadrant of p. For each position (i, j), we give a label from the alphabet $\{R, C, D, O, Z\}$, depending on the answer for the query NLN_{NE}, as follows. The position (i, j) is labeled with:

- O ("One") if $A[p] = \mathbf{1}$ (the value at the position is **1**);
- R ("Row") if $i = i'$ (its answer is in the same row);
- C ("Column") if $j = j'$ (its answer is in the same column);
- D ("Diagonal") if $i < i'$ and $j < j'$ (its answer is not in the same row or column); and
- Z ("Zero") if $A[p] = 0$ and also there is no **1** in the upper-right quadrant of p.

Now, given a query position p, if the position p is labeled with O or Z, then we conclude that NLN_{NE} does not exist (in this quadrant). Otherwise, if the label is R, we can find its answer by following the positions $(i, j + k)$, for $k = 1, 2, \ldots$ (i.e., elements in the same row) till we reach a position with label O, and return that position as the answer. Also, one can easily show that all the intermediate positions cannot have label O. Analogously, if the label is C, then we follow the positions in the same column until we reach a position with label O and return that position. Finally, if the label is D, then we first follow the positions $(i + k, j + k)$, for $k = 1, 2, \ldots$ till we reach the first position

$(i+\ell, j+\ell)$ with a label different from D. The label of position $(i+\ell, j+\ell)$ can be O, R or C. If it is O, then we return that position as the answer. In the other two cases, we can find the answer by following the row or column as described above. The data structure simply stores the labels of all positions in the array (for each quadrant). In addition, to support queries faster, we build rank/select structures (over constant alphabet strings) for the encoding of each row, each column and each diagonal. By Lemma 2.5, the total space usage is clearly $O(n^2)$ bits. Now, queries can be supported in constant time by using rank/select to jump to the appropriate positions as described in the above procedures. $\square$

Now we describe an index for a given 2D binary array, in the bit-probe model, that uses $O(n^2/c)$ bits and supports NLN queries in $O(c)$ time. Since the indexing trade-off lower bound for the 1D case described in Theorem 4.2 also holds for higher dimensions, it follows that the achieved trade-off is optimal.

We begin by introducing some notation that will be used later. Suppose we divide an $n \times n$ array A into blocks of size $c \times c$, for $1 \leq c \leq n$, and divide each block into c sub-blocks of size $\sqrt{c} \times \sqrt{c}$. We define an (i, j)-block as the sub-array $A[(i-1)c+1 \ldots ic][(j-1)c \ldots jc]$ and an (i, j, k, l)-sub-block as the sub-array $A[(i-1)c+(k-1)\sqrt{c} \ldots (i-1)c+k\sqrt{c}][(j-1)c+(l-1)\sqrt{c} \ldots (j-1)c+l\sqrt{c}]$. For each (i, j)-block, we define eight regions, consisting of sets of blocks (some of which can be empty) as follows: the region

$N(i, j)$ consists of all (i, l)-blocks with $l > j$;

$S(i, j)$ consists of all (i, l)-blocks with $l < j$;

$E(i, j)$ contains all (k, j)-blocks with $k > i$;

$W(i, j)$ contains all (k, j)-blocks with $k < i$;

$NE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l > j$;

$NW(i, j)$ contains all (k, l)-blocks with $k < i$ and $l > j$;

$SE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l < j$; and

$SW(i, j)$ contains all (k, l)-blocks with $k < i$ and $l < j$.

Similarly, for each (i, j, k, l)-sub-block, we also define the regions $N_{i,j}(k, l)$, $S_{i,j}(k, l)$, $E_{i,j}(k, l)$, $W_{i,j}(k, l)$, $NE_{i,j}(k, l)$, $NW_{i,j}(k, l)$, $SE_{i,j}(k, l)$ and $SW_{i,j}(k, l)$ in the same way.

Theorem 4.3. *Given a binary array $A[1 \ldots\ n][1 \ldots\ n]$ one can construct an index of size $O(n^2/c)$ bits to support NLN queries in optimal $O(c)$ time, for any parameter c, where $1 \leq c \leq n$.*

Proof. We divide the array A into blocks and sub-blocks as mentioned earlier. We construct an $n/c \times n/c$ array $A'[1 \ldots n/c][1 \ldots n/c]$ such that $A'[i][j] = \mathbf{1}$ if there exists at least a single $\mathbf{1}$ in the (i, j)-block, and 0 otherwise. We also construct another $n/\sqrt{c} \times n/\sqrt{c}$ array $A''[1 \ldots n/\sqrt{c}][1 \ldots n/\sqrt{c}]$ such that $A''[i][j] = \mathbf{1}$ if there exists at least a single $\mathbf{1}$ in the $(\lfloor i/\sqrt{c} \rfloor, \lfloor j/\sqrt{c} \rfloor, i - \lfloor i/\sqrt{c} \rfloor \sqrt{c}, j - \lfloor j/\sqrt{c} \rfloor \sqrt{c})$-sub-block, and 0 otherwise .

Suppose the query q is in the (i, j, k, l)-sub-block. If $A''[i\sqrt{c}+k, j\sqrt{c}+l] = \mathbf{1}$, scanning $O(1)$ sub-blocks is enough to find the NLN of q, and this takes $O(c)$ time.

Now, consider the case when $A''[i\sqrt{c} + k, j\sqrt{c} + l] = 0$ but $A'[i, j] = \mathbf{1}$. In this case, it is clear that we can identify $O(c)$ sub-blocks in which the answer may lie – namely all the sub-blocks in its block, and the eight neighbouring blocks. We find the potential answer in each of the eight directions (E, W, N, S, NE, NW, SE, and SW), and then compare their positions to find the actual answer. To find the answer in E direction, we scan the bits in A'' that are to the right of the current sub-block, till we find a $\mathbf{1}$, say, in sub-block s. We then scan sub-block s, and the sub-block immediately to its right, to find the potential answer in this direction. Similarly, we can find the potential answers in the W, S, and N directions. Next, we find the nearest $\mathbf{1}$ to the query in the $NE_{i,j}(k, l)$ region. This element is the nearest $\mathbf{1}$ from the bottom-left corner of $(i, j, k + 1, l + 1)$-sub-block. The nearest $\mathbf{1}$ from the bottom-left corner of (a, b, c, d)-sub-block in the $NE_{a,b}(c, d)$ region is same as the the nearest $\mathbf{1}$ from

the bottom-left corners of one of these four sub-blocks: (1) (a, b, c, d)-sub-block (2) $(a, b, c+1, d)$-sub-block, (3) $(a, b, c, d+1)$-sub-block, or (4) $(a, b, c+1, d+1)$-sub-block. Therefore we encode each sub-blocks using 2 bits indicating the case it belongs to ((1), (2), (3) or (4)), which takes a total of $O(n^2/c)$ bits. Now, to find the answer in the NE direction, we scan $O(c)$ sub-blocks to find the sub-block which contains the nearest **1** from q in $NE(i, j, k, l)$. Once we find the corresponding sub-block, finding the nearest **1** from the bottom-left corner in the sub-block takes $O(c)$ time. We can find the nearest **1** in the $NW_{ij}(k, l)$, $SE_{ij}(k, l)$ and $SW_{ij}(k, l)$ regions in the same way. Then the NLN of q is the closest one among these eight candidates.

Finally, consider the case when $A'[i, j] = 0$. By storing the data structure of Theorem 4.2 for the array A' using $O(n^2/c^2)$ bits, we can find the nearest blocks in each direction to the query position which contains a **1**, in $O(1)$ time. Let one of these blocks be the (i', j')-block, let ℓ be the L_1 distance from (i, j) to (i', j') in A'. The value ℓc is an estimate (within an additive factor of $2c$) for the L_1 distance from q to its NLN. Assume, wlog, that (i', j') is in the $NE(i, j)$ region of A'. We first describe how to find the nearest **1** in the $NE(i, j)$ region. Define $d(i, j)$ as the sequence of blocks in the top-left to the bottom-right diagonal that contains the (i, j)-block (i.e., all the blocks (i', j') in A such that $i' + j' = i + j$), where the blocks are ordered in the increasing order of their i values. We store a 1-D array $D_{(i,j)}$ of size equal to $|d(i, j)| \leq n/c$, $D_{(i,j)}[m]$ is the distance from the bottom-left element of the m-th block in the sequence $d(i, j)$ to the nearest **1** in that block, and $2c + 1$ if there is no **1** in that block. Note that each block belongs to exactly one $D_{(i,j)}$, and hence the total size of all these $D_{(i,j)}$ arrays is $O((n^2/c^2) \lg c)$ bits. In addition, we also construct a linear-bit RMQ (range minimum query) data structure for each $D_{(i,j)}$ (using a total of $O(n^2/c^2)$ bits), so that RMQ queries can be supported in $O(1)$ time [27]. Now, we find the two potential blocks in the $NE(i, j)$ region that may have the nearest **1** from q by performing RMQs on $D_{(i',j')}$ and $D_{(i',j'+1)}$ among all the blocks that are

Figure 4.1 Suppose the nearest block that contains a **1** from the $(2,2)$-block is the $(4,3)$-block. Then $d(4,3)$ contains the blocks $(2,5),(3,4),(4,3)$ and $(5,2)$, in that order. We can find the nearest **1** in NE(2,2) using $\mathsf{RMQ}(2,3)$ on $D_{(4,3)}$ and $\mathsf{RMQ}(1,3)$ on $D_{(4,4)}$.

contained in the $NE(i,j)$ region (it is easy to see that they form a consecutive range). We then choose the closer one between these two from q. (Figure 4.1 shows an example.) Note that if (i',j') is in a different region from $NE(i,j)$, then we may not find any potential answer in $NE(i,j)$, as all the 'relevant' blocks in $d(i',j')$ and $d(i',j'+1)$ may be empty. We can find the nearest **1** in $NW(i,j)$, $SE(i,j)$ and $SW(i,j)$ in a similar way.

Next, we describe how to find the nearest **1** in the $N(i,j)$ region (finding the nearest **1** in the $S(i,j)$, $E(i,j)$ and $W(i,j)$ regions is analogous). For each position in the bottom row of an (a,b)-block with $A'[a,b] = \mathbf{1}$, we store two bits indicating whether its answer within the block is in (1) the same column (C), or (2) some column to the left (L), or (3) some column to the right (R). The query algorithm simply *follows* the L or R *pointers* till it reaches a C, and then scans the column upwards till it finds a **1** in that column. This takes $O(c \times n^2/c^2) = O(n^2/c)$ bits over all the blocks. This encoding enables us to find the closest **1** within the block from any column in the bottom row of that block in $O(c)$ time. Since ℓ is the L_1 distance between (i,j) and (i',j') in A',

we know that all the blocks $A[i, j-r]$, for $1 \leq r < \ell$ are empty (otherwise, we have a closer non-empty block than (i', j')). Let k be the column corresponding to the query position q. We claim that the closest **1** to q in the $N(i,j)$ region is closest **1** to the bottom row and column k of either the $(i, j+\ell)$-block or the $(i, j+\ell+1)$-block. These can be computed in $O(c)$ time using the above encoding, and then compared to find the required answer. Finally we can find NLN of q by comparing these eight candidate answers. □

The optimality of the trade-off follows from the lower bound of the following lemma.

Lemma 4.2 ([44]). *Given a 1D array of size n, any data structure which stores $O(n/c)$ bits and answers NLV (or NLN) queries in the indexing model, requires at least $\Omega(c)$ query time, for any $1 \leq c \leq n$.*

4.4 Encoding NLN queries for general 2D arrays

Consider an $n \times n$ 2D array $A[1 \ldots n][1 \ldots n]$. Given two positions (i, j) and (i', j') in A, we define $\mathsf{dist}((i,j),(i'j')) = |i - i'| + |j - j'|$. A trivial solution to the NLN problem in 2D array is to store $\mathsf{NLN}((i,j))$, for $1 \leq i, j \leq n$. This requires $O(n^2 \lg n)$ bits, and supports queries in $O(1)$ time. In the following, we obtain improved results for the 2D NLN in the encoding and indexing models, and also describe some trade-off results.

4.4.1 2D NLN in the encoding model – distinct case

When there is no restriction on the elements of the array, one can show an n^2-bit lower bound for NLN encoding (described in Section 4.4.2). Using a simple encoding method, one can prove that the same asymptotic lower bound applies even when all the elements of the array are distinct to obtain the following.

X	O	X	O	X	O
X	O	X	O	X	O
X	X	X	X	X	X
X	O	X	O	X	O
X	O	X	O	X	O
X	X	X	X	X	X

O : Useful

X : Dummy

Figure 4.2 The positions of *useful* and *dummy* elements in a 6×6 array. In this example, the dummy elements (X's) are in the range [1..24] and the useful elements (O's) are in the range [25..26].

Theorem 4.4. *Any data structure which supports* NLN *queries on an $n \times n$ array $A[1 \dots n][1 \dots n]$ in encoding model requires at least $n^2/6$ bits, even when all the elements in A are distinct.*

Proof. Without loss of generality, we assume that n is a multiple of 6. We first define a set $\mathcal{A}$ of $2^{n^2/6}$ 2D arrays, and then show that the answers to the NLN queries in any array $A \in \mathcal{A}$ can be used to distinguish A from $\mathcal{A} \setminus \{A\}$. This proves that encoding for an arbitrary array in $\mathcal{A}$ requires at least $\lg(|\mathcal{A}|) = n^2/6$ bits in the worst case.

Each array $A \in \mathcal{A}$ contains elements from the set $\{1, 2, \dots, n^2\}$, where each element appears in the array exactly once. To describe the arrays in $\mathcal{A}$, we partition the elements of each array into *useful* and *dummy* elements. The positions $(3i + 1, 2j)$ and $(3i + 2, 2j)$, for $0 \leq i < n/3$ and $1 \leq j \leq n/2$, contain useful elements, and the remaining $2n^2/3$ positions contain the dummy elements (see Figure 4.2). We assign the elements from 1 to $2n^2/3$ to the positions corresponding to the dummy elements, in row-major order. Also, we first assign the elements from $2n^2/3 + 1$ to n^2 to the positions corresponding to the useful

elements, in row-major order. Let A_0 denote this array. We now obtain the $2^{n^2/6}$ arrays in $\mathcal{A}$ by repeatedly taking a pair of adjacent useful elements and flipping them.

Consider any two arrays A and A' in $\mathcal{A}$. We know that for at least one pair of adjacent positions $(3i+1, 2j)$ and $(3i+2, 2j)$, for some $0 \leq i < n/3$ and $1 \leq j \leq n/2$, we will have $A[3i+1, 2j] < A[3i+2, 2j]$ while $A'[3i+1, 2j] > A'[3i+2, 2j]$ or vice versa, and hence their NLN answers are distinct. Therefore, given the answers to the NLN queries of all adjacent pairs of useful elements, we can distinguish the array A from $\mathcal{A} \setminus \{A\}$. □

We now obtain an asymptotically optimal upper bound for 2D NLN encoding for the distinct case. The proof is based on ideas from Asano and Kirkpatrick [4].

Lemma 4.3. *A 2D array $A[1 \dots n][1 \dots n]$ can be encoded using $O(n^2)$ bits to support NLN queries, provided all elements are distinct.*

Proof. The main idea is to divide the array recursively into blocks of geometrically increasing size, and store the NLN values of all elements, except the largest element and the elements whose answers are stored at a previous level, in each block explicitly. The following argument shows that this requires $O(n^2)$ bits overall.

In the first level, we divide A into $n^2/4$ blocks of size 2×2 each. Except for the largest element in each 2×2 block, the distance of NLN answer for the other three elements are bounded by 2. In general, at level k, we divide A into $n^2/4^k$ blocks of size $2^k \times 2^k$ each. In each of these $2^k \times 2^k$-sized blocks, there are four elements left for which we need to store the answer to their NLN queries. For three of these four elements, which do not correspond to the maximum value in the block, we store their answers at level k. Since the distance to the NLN answer for these three elements is bounded by 2^{k+2}, we can store these answers using $O(k)$ bits. Thus the total space usage is bounded by $\sum_{k=1}^{\lg n} (3n^2/4^k) * O(k) = O(n^2)$ bits. □

We now describe another $O(n^2)$-bit encoding for the 2D NLN problem that supports queries in constant time when the elements are distinct.

Theorem 4.5. *A 2D array $A[1 \dots n][1 \dots n]$ can be encoded using $O(n^2)$ bits to support NLN queries in $O(1)$ time, provided all elements are distinct.*

Proof. The encoding is a small variant of the encoding described in the proof of Lemma 4.3. For each position in A, in some canonical order (say, row-major order), we write down the relative position (i.e., the distance from the position to its answer in horizontal and vertical directions) of its NLN answer. We use a variable-length encoding, such as γ-code or δ-code [21], to write these answers. The proof of Lemma 4.3 implies that the sum of the lengths of all these answers is $O(n^2)$. We also store an indexable bit vector [69] indicating the starting positions of each code. This enables us to find the position where the answer to a given query starts and ends, in constant time. □

4.4.2 2D NLN in the encoding model – general case

It is easy to see that for any two distinct $n \times n$ binary arrays can be distinguished by looking at the NLN answers at every positions. In other words, any two distinct binary arrays must have distinct NLN encodings. This shows an n^2-bit lower bound for NLN encoding in the general case, In this section, we give an encoding which supports NLN queries in a 2D array with $O(n^2)$ bits in the general case. Before starting the 2D case, we consider the 1D case first. Jayapaul et al. [44] showed how to encode an array A with n distinct elements using $O(n)$ bits to answer NLN queries. We give an alternate proof, that is similar to the proof of Lemma 4.3.

Lemma 4.4. *There exists an encoding of an array $A[1 \dots\ n]$ that uses $O(n)$ bits while supporting NLN queries, provided all elements are distinct.*

Proof. We write down the sequence $d(1), d(2), \dots, d(n)$ explicitly, where $d(i) = n$ if $A[i]$ is the maximum element of A, and $d(i) = |i - \mathsf{NLN}(i)|$ otherwise,

for $1 \leq i \leq n$, together with a sequence of n bits that indicate if $i < \mathsf{NLN}(i)$ or $i > \mathsf{NLN}(i)$. Thus, it is enough to show that $\sum_{i=1}^{n} \lg d(i) = O(n)$. Since all the elements in A are distinct, there are at most $n/2^k$ elements for which $d(i) \geq 2^k$, for any $1 \leq k \leq \lg n$. From this observation, it follows that there are $O(n/2^k)$ elements for which $2^k \leq d(i) < 2^{k+1}$, and hence $\sum_{i=1}^{n} \lg d(i) \leq \sum_{k=1}^{\lg n}(O(n/2^k) \cdot O(k)) = O(n)$. $\square$

We now describe a simple modification of the above encoding that can be used to support NLN queries even when the elements are not distinct. Queries are not supported in constant time with this encoding. Note that one can use the encoding of Fischer [25] to obtain a linear-bit (in fact, a $2.54n$-bit) encoding which supports NLN queries in constant time. However, in contrast to Fischer's encoding, the new approach stores explict pointers from one array position to another, and we use the space cost of these explicit pointers to upper bound the space usage of the pointers stored in the proof of Theorem 4.6.

Instead of encoding the NLN of a position i as in Lemma 4.4, we encode the distance between i and the nearest value which is $\geq A[i]$ in the same direction as $\mathsf{NLN}(i)$. Formally, we define $d_l(i) = i - (\max_{j<i, A[j] \geq A[i]} j)$ and $d_r(i) = (\min_{j>i, A[j] \geq A[i]} j) - i$ and $d(i) = d_l(i)$ if $\mathsf{NLN}(i) < i$ and $d(i) = d_r(i)$ otherwise. For each i, we encode $d(i)$ (using a variable-length encoding) and store a bit indicating whether $d(i) = d_r(i)$ or $d(i) = d_l(i)$, and view this as a "pointer" to $j = i + d_r(i)$ or $j = i - d_l(i)$ respectively. Finally, we also store a bit indicating whether or not $A[i] = A[j]$. With this encoding, $\mathsf{NLN}(i)$ can be easily found by following the $d(\cdot)$ "pointers" from i until we reach a position that is greater than $A[i]$. We refer to this encoding of a 1D array as $encoding_{1D}$.

The following lemma shows that this encoding uses $O(n)$ bits[1]:

Lemma 4.5. *For a 1D array $A[1\dots n]$, $encoding_{1D}$ takes $O(n)$ bits.*

Proof. For A, $encoding_{1D}$ consists of two bit strings of length $O(n)$, and a sequence of variable-length encodings storing the values $d(1), d(2), \dots, d(n)$. Let $D = \sum_{i=1}^{n} \lg d(i)$. To prove the lemma, it is enough to show that $D = O(n)$.

We first create a new array A' with all distinct elements, and bound the value D using the size of the NLN encoding of A'. Consider the array $A'[1\dots n]$ of size n, where $A'[i] = A[i] + \epsilon i$ if $\mathsf{NLN}(i) > i$ and $A'[i] = A[i] - \epsilon i$ if $\mathsf{NLN}(i) < i$ for some $\epsilon > 0$. If we set ϵ small enough then if $A[i] > A[j]$ for some i, j then $A'[i] > A'[j]$ as well, but all elements in A' are distinct. So if we define $d'(i)$, NLN' and D' on A' analogously to $d(i)$, NLN, and D on A, $D' = \sum_{i=1}^{n} \lg d'(i) = O(n)$ by Lemma 4.4. We now show that $D \leq 2D'$.

For a subset S of $U = \{1, 2, \dots, n\}$, we define D_S as $\sum_{i \in S} \lg d(i)$, (and D'_S analogously). To prove that $D \leq 2D'$, we partition the set U into disjoint subsets, and show that $D_S \leq 2D'_S$ for every subset S in the partition. To define the partitions of U, we first extend the array A such that $A[0] = A[n+1] = \infty$. Now, each subset in the partition of U contains a set of indices $i_1, \dots i_{r-1}$ where $0 \leq i_0 < i_1 < \dots < i_r \leq n+1$ with $r > 1$ is a maximal sequence of indices such that $A[i_0] > A[i_1]$, $A[i_{r-1}] < A[i_r]$, $A[i_1] = A[i_2] = \dots = A[i_{r-1}]$ and for all $i_0 < j < i_r$, $A[j] < A[i_1]$ if $j \notin \{i_1, \dots, i_{r-1}\}$. It is easy to show that this collection of subsets form a partition of U, i.e., they are pairwise disjoint and cover U.

Let i_k be the index such that $\mathsf{NLN}(i_l) = i_0$ for all $0 < l \leq k$ and $\mathsf{NLN}(i_l) = i_r$ for all $k < l \leq r-1$. Then by the definition of A', for all $k < l \leq r-1$,

[1] Note that this encoding cannot be obtained by simply breaking ties among equal elements in some arbitrary fashion and applying Lemma 4.4. For example, if $A[i] = A[i+1]$ and $A[i-t]$ and $A[i+1+t]$ for some $t > 1$ are the nearest larger values, then in the current encoding, neither $A[i]$ nor $A[i+1]$ would point to one another. If we break ties then either $A[i]$ points to $A[i+1]$ or $A[i+1]$ points to $A[i]$.

$\mathsf{NLN}'(i_l) = i_{l+1}$ so $d(i_l) = d'(i_l)$. For the elements to the left of i_k, we can consider the case that there exist $0 < m \leq k$ such that $\mathsf{NLN}'(i_l) = i_{l-1}$ for all $0 < l \leq m-1$ and $\mathsf{NLN}'(i_l) = i_{k+1}$ for $m \leq l \leq k$. Then:

$$
\begin{aligned}
D_S - D'_S = & \sum_{j=1}^{r-1} \lg d(i_j) - \sum_{j=1}^{r-1} \lg d'(i_j) \\
= & \left(\sum_{j=1}^{m-1} \lg d(i_j) + \sum_{j=m}^{k} \lg(i_j - i_{j-1}) + \sum_{j=k+1}^{r-1} \lg d(i_j) \right) \\
- & \left(\sum_{j=1}^{m-1} \lg d'(i_j) + \sum_{j=m}^{k} \lg(i_{k+1} - i_j) + \sum_{j=k+1}^{r-1} \lg d'(i_j) \right) \\
= & \sum_{j=m}^{k} \lg(i_j - i_{j-1}) - \sum_{j=m}^{k} \lg(i_{k+1} - i_j) \\
\leq & \lg(i_m - i_{m-1}) - \lg(i_{k+1} - i_k) \\
& (\because i_j - i_{j-1} \leq i_{k+1} - i_{j-1} \text{ for all } m \leq j \leq k) \\
\leq & \lg(i_m - i_{m-1}) \leq \lg(i_m - i_0) \leq \lg(i_r - i_m) \quad (\because \mathsf{NLN}(i_m) = i_0) \\
\leq & \lg(i_{k+1} - i_m) + \sum_{j=k+1}^{r-1} \lg(i_{j+1} - i_j) \text{ (by the concavity of lg function)} \\
\leq & \sum_{j=1}^{r-1} \lg d'(i_j) = D'_S
\end{aligned}
$$

□

We now extend this encoding to encode NLNs for a 2D array $A[1 \ldots n][1 \ldots n]$ that answers NLN queries. We call this encoding scheme *encoding*$_{2D}$. We then show that *encoding*$_{2D}$ takes $O(n^2)$ bits (in Theorem 4.6).

In *encoding*$_{2D}$, each (i, j) "points to" another location (i', j'), such that $A[i', j'] \geq A[i, j]$, as follows: $|i - i'|$ is encoded using $O(1 + \lg |i' - i|)$ (the *row cost* of the pointer) and $|j - j'|$ is coded using $O(1 + \lg |j' - j|)$ bits (the *column cost* of the pointer), the direction from (i, j) to (i', j') is given using two bits, and finally one extra bit indicates whether or not $A[i', j'] > A[i, j]$. Now we explain how to specify the pointers. Pick an element $A[i, j]$ and without loss

of generality assume that $\mathsf{NLN}(i,j) = (i^*, j^*)$ with $i^* \geq i, j^* \geq j$. We choose pointers as follows:

Case (1) Let $i' > i$ be the smallest value such that $i' \leq i^*$ and $A[i,j] = A[i',j]$. If i' exists, then we store a pointer from (i,j) to (i',j) and set the extra bit to 0.

Case (2) If there exists no i' such that $A[i,j] = A[i',j]$. for $i < i' \leq i^*$, then let $j' > j$ be the smallest value such that $j' \leq j^*$ and $A[i,j] = A[i,j']$. If j' exists, we store a pointer from (i,j) to (i,j') and set the extra bit to 0.

Case (3) If there exists no i' such that $A[i,j] = A[i',j]$. for $i < i' \leq i^*$, and also if there exists no j' such that $A[i,j] = A[i,j']$. for $j < j' \leq j^*$, then we store a pointer from (i,j) to (i^*, j^*) and set the extra bit to 1.

To obtain $\mathsf{NLN}(i,j)$, we follow pointers starting from (i,j) until we follow one with the extra bit set to 1, and return the position pointed to by this pointer.

We now show that the above procedure computes $\mathsf{NLN}(i,j)$, for all $1 \leq i,j \leq n$. The proof is by induction on k, the distance between (i,j) and $\mathsf{NLN}(i,j) = (i^*, j^*)$. The base case $k = 1$ follows directly from case 3) above.

Assume the induction hypothesis holds for all NLNs at distance $\leq k$, and choose an (i,j) such that $\mathsf{NLN}(i,j) = (i^*, j^*)$ and $dist((i,j),(i^*,j^*)) = k+1$. Assume, without loss of generality, that the pointer from (i,j) has its extra bit set to 0 (otherwise, the induction step is trivial) and it points to (i',j) with $i' > i$. Assume that $\mathsf{NLN}(i',j) = (x,y) \neq (i^*, j^*)$, and $dist((x,y),(i,j))$ is greater than $dist((x,y),(i^*,j^*))$. Since $A[i',j] = A[i,j]$, $dist((i',j),(x,y)) \leq dist((i',j),(i^*,j^*)) < dist((i,j),(i^*,j^*)) = k+1$. By the induction hypothesis, following pointers from (i',j) leads to (x,y). Now:

$$\begin{aligned} dist((i,j),(x,y)) &= dist((i,j),(i',j)) + dist((i',j),(x,y)) \\ &\leq dist((i,j),(i',j)) + dist((i',j),(i^*,j^*)) \; (\because \mathsf{NLN}(i',j) = (x,y)) \\ &= dist((i,j),(i^*,j^*)), \end{aligned}$$

contradicting the assumption that $dist((x, y), (i, j)) > dist((x, y), (i^*, j^*))$.

Theorem 4.6. *There exists an encoding of a 2D array $A[1 \dots n][1 \dots n]$ that supports* NLN *queries, using $O(n^2)$ bits.*

Proof. We show that $encoding_{2D}$, described earlier, takes $O(n^2)$ bits. To upper bound the space, we first describe an encoding, called $encoding_{grid}$ as follows. We encode each column and each row of A using $encoding_{1D}$, using $O(n^2)$ bits. These pointers are called *grid* pointers. However, the maximal values in each row and column do not have pointers by Lemma 4.5, as their NLN is not defined. So, in addition, for each row r which has (locally) maximum values in columns $i_1 < \dots < i_k$, we store *extra* pointers from (i_j, r) to (i_{j+1}, r) and vice versa for $j = 0, \dots, k$, taking $i_0 = 0$ and $i_{k+1} = n + 1$. The space taken by these extra pointers is $O(\lg i_1 + \sum_{j=2}^{k-1} \lg (i_j - i_{j-1}) + \lg (n + 1 - i_k)) = O(n)$ bits for row r. We do this for all rows and columns, at a cost of $O(n^2)$ bits overall.

Although $encoding_{grid}$ does not encode NLN, we use it to upper bound the space used by $encoding_{2D}$. Let a *grid pointer* and a *2D pointer* refer to a pointer in $encoding_{grid}$ and $encoding_{2D}$ respectively. For any 2D pointer, the cost of encoding it can be upper-bounded by the cost of encoding (one or more) grid pointers. Each grid pointer will be used $O(1)$ times this way. Below, we show how to upper bound all Case (2) 2D pointers and the column cost of all Case (3) 2D pointers by grid pointers in rows, using each grid pointer at most thrice. The costs of Case (1) 2D pointers and the column cost of Case (3) 2D pointers can similarly be bounded by the costs of grid pointers in the columns. This will prove the theorem.

We consider a fixed location (i, j), and assume wlog that $\mathsf{NLN}(i, j) = (i^*, j^*)$ with $i^* \geq i$ and $j^* > j$ (if $j^* = j$ then the pointer from (i, j) will have column distance 0 and there is nothing to bound). There are four cases to consider (see Figure 4.3).

Case (a) Let $j' > j$ be the minimum index such that $A[i, j'] \geq A[i, j]$. Suppose

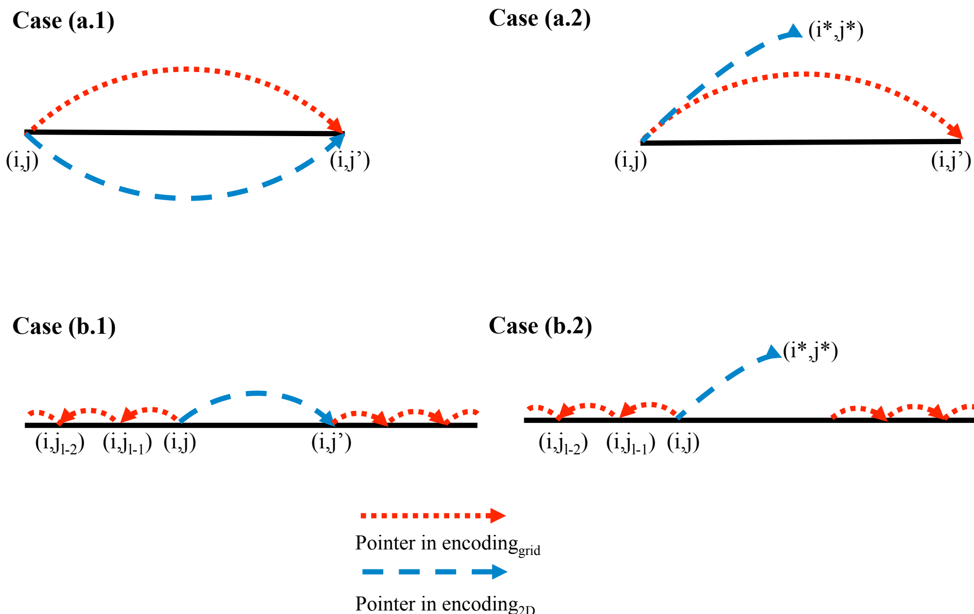


Figure 4.3 Pointers in $encoding_{2D}$ and $encoding_{grid}$

that j' exists and there is a grid pointer from (i, j) to (i, j') or vice versa. There are two sub-cases:

(a.1) The 2D pointer from (i, j) points to (i, j'). We use the cost of this grid pointer to upper bound the cost of the 2D pointer. Observe that if there is a 2D pointer from (i, j) to (i, j'), there cannot be a 2D pointer from (i, j') to (i, j), so the grid pointer is used for upper-bounding only once in this case.

(a.2) The 2D pointer from (i, j) points to (i^*, j^*). Observe that $j' \geq j^*$, since otherwise either (i, j') is a larger value that is closer than (i^*, j^*), a contradiction, or we would have a Case (2) 2D pointer from (i, j) to (i, j'). The pointer between (i, j) and (i, j') will only be charged twice for upper-bounding in this case.

Case (b) Either (i) $A[i, j] > A[i, j']$ for all $j' > j$, or (ii) there exists a $j' > j$

such that $A[i,j] \leq A[i,j']$, and there are no grid pointers either from (i,j) to (i,j') or vice versa. As before, we consider two sub-cases.

(b.1) Suppose the 2D pointer from (i,j) points to (i,j'), where $j' > j$ is the smallest index such that $A[i,j'] \geq A[i,j]$. If $A[i,j]$ is a maximal value in row i, the cost of the pointer is upper-bounded by the extra pointer between (i,j) and (i,j'). If not, the absence of grid pointers between (i,j) and (i,j') implies that the NLN of (i,j) in the i-th row is (i,j_0) for some $j_0 < j$. Note that $|j_0 - j| \geq \mathsf{dist}((i,j),(i^*,j^*))$ (otherwise $\mathsf{NLN}(i,j)$ would be (i,j_0)). The path p between (i,j) and (i,j_0) in $encoding_{grid}$ may comprise a number of grid edges. We can bound the cost of the 2D edge from (i,j) to (i,j') by the total cost of the grid edges on the path p consisting of the elements $j = j_l, j_{l-1}, \ldots, j_1, j_0$ (omitting the row number for brevity).[2] Note that for any $0 < k < l$, no 2D pointer from (i,j_k) can end up in Case (b), so this path can only be used twice to upper-bound the cost of a 2D edge: once from (i,j) and once (possibly) from (i,j_0).

(b.2) Suppose the 2D pointer from (i,j) points to (i^*,j^*). If $A[i,j]$ is a maximal value in row i, then if j' exists, then it must be the case that $j' > j^*$, and the row cost of the 2D pointer is bounded by the extra pointer between (i,j) and (i,j'). On the other hand, if j' does not exist, then the row cost of the 2D pointer is bounded by the extra pointer from (i,j) to $(i,n+1)$.

If $A[i,j]$ is not maximal, then arguing as above, we see that the NLN of (i,j) in the i-th row is (i,j_0) for some $j_0 < j$, that $|j_0 - j| \geq |j - j^*|$, and so we can upper-bound the row cost of this 2D pointer by the total cost of all the grid pointers between j and j_0, and each of these

[1]Since the log function is concave, the sum of the costs of the path p is no less than the cost of a single edge from (i,j) to (i,j_0).

grid pointers is used at most twice (once each for the pointers out of (i, j) and (i, j_0) in Case (b) to upper bound a 2D pointer.

□

We now describe an $O(n^2)$-bit encoding that supports NLN queries in constant time on a 2D array.

Theorem 4.7. *There exists an encoding of a 2D array $A[1 \ldots n][1 \ldots n]$ that uses $O(n^2)$ bits while supporting NLN queries in $O(1)$ time.*

Proof. We first divide A into blocks of size $b \times b$, and divide each block into sub-blocks of size $s \times s$. For each position (i, j) in the array, we say that the four locations $(i+1, j)$, $(i-1, j)$, $(i, j+1)$ and $(i, j-1)$ (even if they are outside the array range) are its *neighbors*. We say that a location (i, j) is a boundary location with respect to a block (sub-block) if one of its neighbors is not in the same block (sub-block). Note that there are $O(b)$ ($O(s)$) boundary elements in each block (sub-block). For each block, we store a $b \times b$ bitmap of size b^2 bits, such that the (i, j)-th bit, for $1 \leq i, j \leq b$, stores a 1 if the corresponding element in that position is a maximum element in that block, and stores a 0 otherwise. We also store similar bitmaps for each sub-block, using s^2 bits for each sub-block.

For each boundary position (i, j) in a block B, we store the nearest position to (i, j) whose value is larger than the maximum element in B. This takes $O(b \lg n)$ bits for each block. Also, for a sub-block B' in a block B, if B' does not contain the maximum element in B, then for each boundary position (i, j) in B', we store the nearest position to (i, j) whose value is larger than the maximum element in B'. Since the distance to this position is at most $2b$, it takes $O(s \lg b)$ bits for each sub-block to store this information.

Finally, for a position (i, j) in a sub-block B', if $A[i, j]$ is not the maximum element in B', then its NLN always exists in the sub-array A' of A, of size

$5s \times 5s$ such that $B' = A'[2s+1,\ldots,3s][2s+1,\ldots,3s]$ (i.e., B' is the center sub-block in the $5s \times 5s$ sub-array A'). Theorem 4.6 shows that this sub-array A' can be encoded using λs^2 bits, for some positive constant λ, to support NLN queries. For each sub-block B' in A, we store the encoding (of Theorem 4.6) of the corresponding sub-array A'. Over all the sub-blocks, this takes $O(n^2)$ bits.

In addition, we construct a precomputed table which we store as a two-dimensional array. The first dimension is indexed by all possible bit-strings of length λs^2, and the second dimension is indexed by all possible positions in a sub-block. The (e, p)-th entry in this array stores the NLN of the position p in sub-block B' within the $5s \times 5s$ sub-array A' (with B as its center sub-block) whose encoding is the bit string e.

We now describe the query algorithm. Consider the query position (i, j), and let B (B') be the block (sub-block) that contains (i, j). We first check whether (i, j) is a position of the maximal element in B' in $O(1)$ time using the bitmap defined above. If $A[i, j]$ is not a maximal element in B', then we use the precomputed table to find the answer, in $O(1)$ time. If $A[i, j]$ is the maximum element in B' but not in B, then we can answer the query in $O(1)$ time by comparing the distance between (i, j) and stored positions on the four boundary positions in B' which have the same row or the column positions as (i, j) and choose the nearest one. If $A[i, j]$ is a maximal element in B, we can find its NLN in $O(1)$ time by a similar procedure as above case, by looking at the four boundary positions in B with same row or column index as (i, j). Thus, in all cases, the queries can be supported in $O(1)$ time. The overall space usage is $O(n^2/b^2 \times (b^2 + b \lg n) + n^2/s^2 \times (s^2 + s \lg b) + s^2 2^{\lambda s^2} \lg s^2)$ bits. By choosing $b = \lg n$ and $s = c\sqrt{\lg n}$, for some small constant c (chosen appropriately), the overall space usage becomes $O(n^2)$ bits. □

4.5 Open problems

Our main contribution is a systematic study of data structures for NLV on 1D arrays in the indexing model, and NLN on 2D arrays in the encoding model.

We suggest the following open problems for future works.

- Is there a data structure that takes less than $2.54n + o(n)$ bits and can answer NLN queries in a one dimensional array in constant time in the general case (when elements may repeat) in the encoding model?
- For a 1D array, is there an index for NLV that uses $O(n/c)$ bits and supports queries in $O(c)$ query time?
- For a 2D array, is there an index for NLN that uses $O(n^2/c)$ bits and supports queries in $O(c)$ query time?

Chapter 5

Simultaneous encodings for range and next/previous larger/smaller value queries

5.1 Introduction

Given an array $A[1 \dots n]$ of n elements from a total order. For $1 \leq i \leq j \leq n$, suppose that there are m (l) positions $i \leq p_1 \leq \cdots \leq p_m \leq j$ ($i \leq q_1 \leq \cdots \leq q_l \leq j$) in A which are the positions of minimum (maximum) values between $A[i]$ and $A[j]$. Then we can define various range minimum (maximum) queries as follows.

- Range Minimum Query ($\mathsf{RMinQ}_A(i,j)$) : Return an arbitrary position among $p_1, \dots, p_m$.

- Range Leftmost Minimum Query ($\mathsf{RLMinQ}_A(i,j)$) : Return p_1.

- Range Rightmost Minimum Query ($\mathsf{RRMinQ}_A(i,j)$) : Return p_j.

- Range k-th Minimum Query ($\mathsf{RkMinQ}_A(i,j)$) : Return p_k (for $1 \leq k \leq m$).

- Range Maximum Query ($\mathsf{RMaxQ}_A(i,j)$) : Return an arbitrary position among $q_1, \ldots, q_l$.
- Range Leftmost Maximum Query ($\mathsf{RLMaxQ}_A(i,j)$) : Return q_1.
- Range Rightmost Maximum Query ($\mathsf{RRMaxQ}_A(i,j)$) : Return q_l.
- Range k-th Maximum Query ($\mathsf{RkMaxQ}_A(i,j)$) : Return q_k (for $1 \leq k \leq l$).

Also for $1 \leq i \leq n$, we consider following additional queries on A.

- Previous Smaller Value ($\mathsf{PSV}_A(i)$) : $\max(j : j < i, A[j] < A[i])$.
- Next Smaller Value ($\mathsf{NSV}_A(i)$) : $\min(j : j > i, A[j] < A[i])$.
- Previous Larger Value ($\mathsf{PLV}_A(i)$) : $\max(j : j < i, A[j] > A[i])$.
- Next Larger Value ($\mathsf{NLV}_A(i)$) : $\min(j : j > i, A[j] > A[i])$.

For define above four queries formally, we assume that $A[0] = A[n+1] = -\infty$ for $\mathsf{PSV}_A(i)$ and $\mathsf{NSV}_A(i)$. Similarly we assume that $A[0] = A[n+1] = \infty$ for $\mathsf{PLV}_A(i)$ and $\mathsf{NLV}_A(i)$.

Our aim is to obtain space-efficient encodings that support these queries efficiently.

Previous Work The range minimum/maximum problem has been well-studied in the literature. It is well-known [5] that finding RMinQ_A can be transformed to the problem of finding the LCA (Lowest Common Ancestor) between (the nodes corresponding to) the two query positions in the Cartesian tree constructed on A. Furthermore, since different topological structures of the Cartesian tree on A give rise to different set of answers for RMinQ_A on A, one can obtain an information-theoretic lower bound of $2n - \Theta(\lg n)$ bits on the encoding of A that answers RMinQ queries. Sadakane [71] proposed the $4n + o(n)$-bit encoding with constant query time for RMinQ_A problem using the balanced parentheses (BP) [55] of the Cartesian tree of A with some additional nodes.

Fischer and Heun [27] introduced the $2d$-Min heap, which is a variant of the Cartesian tree, and showed how to encode it using the Depth first unary degree sequence (DFUDS) [6] representation in $2n + o(n)$ bits which supports RMinQ_A queries in constant time. Davoodi et al. show that same $2n + o(n)$-bit encoding with constant query time can be obtained by encoding the Cartesian trees.[16]. For RkMinQ_A, Fischer and Heun [26] defined the *approximate range median of minima query* problem which returns a position RkMinQ_A for some $\frac{1}{16}m \leq k \leq \frac{15}{16}m$, and proposed an encoding that uses $2.54n + o(n)$ bits and supports the *approximate* RMinQ_A queries in constant time, using a *Super Cartesian tree.*

For PSV_A and NSV_A, if all elements in A are distinct, then $2n + o(n)$ bits are enough to answer the queries in constant time, by using the $2d$-Min heap of Fischer and Heun [27]. For the general case, Fischer [25] proposed the *colored* $2d$*-Min heap*, and proposed an optimal $2.54n + o(n)$-bit encoding which can answer PSV_A and NSV_A in constant time.

One can support both RMinQ_A and RMaxQ_A in constant time trivially using the encodings for RMinQ_A and RMaxQ_A queries, using a total of $4n + o(n)$ bits. Gawrychowski and Nicholson reduce this space to $3n + o(n)$ bits while maintaining constant time query time [30]. Their scheme also can support PSV_A and PLV_A in constant time when there are no consecutive equal elements in A.

Our results In this chapter, we first extend the original DFUDS [6] for colored 2d-Min(Max) heap that supports the queries in constant time. Then, we combine the extended DFUDS of $2d$-Min heap and $2d$-Max heap using Gawrychowski and Nicholson's Min-Max encoding [30] with some modifications. As a result, we obtain the following non-trivial encodings that support a wide range of queries.

Theorem 5.1. *An array $A[1 \dots n]$ containing n elements from a total order can be encoded using*

(a) *at most* $3.17n+o(n)$ *bits to support* RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A, *and* PLV_A *queries;*

(b) *at most* $3.322n + o(n)$ *bits to support the queries in (a) in constant time;*

(c) *at most* $4.088n + o(n)$ *bits to support* RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A, NSV_A, RMaxQ_A, RRMaxQ_A, RLMaxQ_A, RkMaxQ_A, PLV_A *and* NLV_A *queries; and*

(d) *at most* $4.585n + o(n)$ *bits to support the queries in (c) in constant time.*

If the array contains no two consecutive equal elements, then (a) and (b) take $3n + o(n)$ *bits, and (c) and (d) take* $4n + o(n)$ *bits.*

This chapter organized as follows. Section 5.2 introduces various data structures that we use later in our encodings. In Section 5.3, we describe the encoding of colored $2d$-Min heap by extending the DFUDS of $2d$-Min heap. This encoding uses a distinct approach from the encoding of the colored $2d$-Min heap by Fischer [25]. Finally, in Section 5.4, we combine the encoding of this colored $2d$-Min heap and Gawrychowski and Nicholson's Min-Max encoding [30] with some modifications, to obtain our main result (Theorem 5.1).

5.2 Preliminaries

We first introduce some useful data structures that we use to encode various bit vectors and balanced parenthesis sequences.

Balanced parenthesis sequences Given a string $S[1 \dots n]$ over the alphabet $\Sigma = \{'(', ')'\}$, if S is balanced and $S[i]$ is an open (close) parenthesis, then we can define $\mathsf{findopen}_S(i)$ ($\mathsf{findclose}_S(i)$) which returns the position of the matching close (open) parenthesis to $S[i]$. Now we introduce the lemma from Munro and Raman [55].

Lemma 5.1 ([55]). *Let S be a balanced parenthesis sequence of length n. If one can access any $\lg n$-bit subsequence of S in constant time, Then both $\mathsf{findopen}_S(i)$ and $\mathsf{findclose}_S(i)$ can be supported in constant time with $o(n)$-bit additional space.*

Depth first unary degree sequence *Depth first unary degree sequence (DFUDS)* is one of the well-known methods for representing ordinal trees [6]. It consists of a balanced sequence of open and closed parentheses, which can be defined inductively as follows. If the tree consists of the single node, its DFUDS is '()'. Otherwise, if the ordinal tree T has k subtrees $T_1 \dots T_k$, then its DFUDS, D_T is the sequence $(^{k+1}\ )d_{T_1} \dots d_{T_k}$ (i.e., $k+1$ open parentheses followed by a close parenthesis concatenated with the 'partial' DFUDS sequences $d_{T_1} \dots d_{T_k}$) where d_{T_i}, for $1 \leq i \leq k$, is the DFUDS of the subtree T_i (i.e., D_{T_i}) with the first open parenthesis removed. From the above construction, it is easy to prove by induction that if T has n nodes, then the size of D_T is $2n$ bits. The following lemma shows that DFUDS representation can be used to support various navigational operations on the tree efficiently.

Lemma 5.2 ([1], [6], [43]). *Given an ordinal tree T on n nodes with DFUDS sequence D_T, one can construct an auxiliary structure of size $o(n)$ bits to support the following operations in constant time: for any two nodes x and y in T,*

- $\mathsf{parent}_T(x)$ *: Label of the parent node of node x.*
- $\mathsf{degree}_T(x)$ *: Degree of node x.*
- $\mathsf{depth}_T(x)$ *: Depth of node x (The depth of the root node is 0).*
- $\mathsf{subtree_size}_T(x)$ *: Size of the subtree of T which has the x as the root node.*
- $\mathsf{next_sibling}_T(x)$ *: The label of the next sibling of the node x.*
- $\mathsf{child}_T(x, i)$ *: Label of the i-th child of the node x.*
- $\mathsf{child_rank}_T(x)$ *: Number of siblings left to the node x.*
- $\mathsf{la}_T(x, i)$ *: Label of the level ancestor of node x at depth i.*
- $\mathsf{lca}_T(x, y)$ *: Label of the least common ancestor of node x and y.*

- $\mathsf{pre_rank}_T(i)$: *The preorder rank of the node in* T *corresponding to* $D_T[i]$.
- $\mathsf{pre_select}_T(x)$: *The first position of node with preorder rank* x *in* D_T.

We use the following lemma to bound the space usage of the data structures described in Section 5.4.

Lemma 5.3. *Given two positive integers* a *and* n*, and a nonnegative integer* $k \le n$*,* $\lg \binom{n}{k} + a(n-k) \le n \lg (2^a + 1)$.

Proof. By raising both sides to the power of 2, it is enough to prove that $\binom{n}{k} 2^{(a(n-k))} \le (2^a+1)^n$. We prove the lemma by induction on n and k. In the base case, when $n = 1$ and $k = 0$, the claim holds since $2^a < (2^a + 1)$. Now suppose that $\binom{n'}{k'} 2^{a(n'-k')} \le (2^a+1)^{n'}$ for all $0 < n' \le n$ and $0 \le k' \le k$. Then

$$\begin{aligned}\binom{n+1}{k} 2^{a(n+1-k)} &= \left(\binom{n}{k} + \binom{n}{k-1}\right) 2^{a(n+1-k)} \\ &\le 2^a(2^a+1)^n + (2^a+1)^n = (2^a+1)^{n+1} \text{ by induction hypothesis.}\end{aligned}$$

Also by induction hypothesis,

$$\begin{aligned}\binom{n}{k+1} 2^{a(n-(k+1))} &= \left(\binom{n-1}{k} + \binom{n-1}{k+1}\right) 2^{a(n-(k+1))} \\ &\le (2^a+1)^{n-1} \left(1 + \frac{\binom{n-1}{k+1}(2^{a(n-1-k)})}{(2^a+1)^{n-1}}\right)\end{aligned}$$

Since $\binom{n-1}{k+1} 2^{a(n-1-k)} < 2^a(2^a+1)^{n-1} (\because (2^a+1)^{n-1} = \sum_{m=0}^{n-1} \binom{n-1}{m} 2^{a(n-1-m)})$,

$$(2^a+1)^{n-1} \left(1 + \frac{\binom{n-1}{k+1}(2^{a(n-1-k)})}{(2^a+1)^{n-1}}\right) < (2^a+1)^{n-1}(1+2^a) = (2^a+1)^n.$$

Therefore the above inequality still holds when $n' = n+1$ or $k' = k+1$, which proves the lemma. □

5.2.1 2d-Min heap

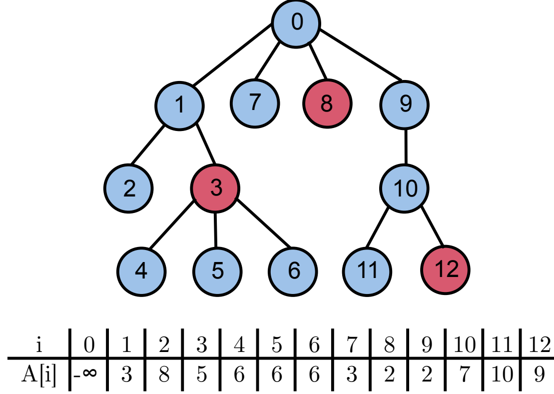
The 2d-Min heap [27] on A, denoted by $\mathsf{Min(A)}$, is designed to encode the answers of $\mathsf{RMinQ}_A(i,j)$ efficiently. We can also define the 2d-Max heap on A

($\mathsf{Max(A)}$) analogously. $\mathsf{Min(A)}$ is an ordered labeled tree with $n+1$ nodes labeled with $0 \ldots n$. Each node in $\mathsf{Min(A)}$ is labeled by its preorder rank and each label corresponds to a position in A. We extend the array $A[1 \ldots n]$ to $A[0 \ldots n]$ with $A[0] = -\infty$. In the labeled tree, the node x denotes the node labeled x. For every vertex i, except for the root node, its parent node is (labeled with) $\mathsf{PSV}_A(i)$.

Using the operations in Lemma 5.2, Fischer and Heun [27] showed that $\mathsf{RMinQ}_A(i, j)$ can be answered in constant time using $D_{\mathsf{Min(A)}}$. If the elements in A are not distinct, $\mathsf{RMinQ}_A(i, j)$ returns the $\mathsf{RRMinQ}_A(i, j)$.

Fischer and Heun [27] also proposed a linear-time stack-based algorithm to construct $D_{\mathsf{Min(A)}}$. Their algorithm maintains a min-stack consisting of a decreasing sequence of elements from top to the bottom. The elements of A are pushed into the min-stack from right to left and before pushing the element $A[i]$, all the elements from the stack that are larger than $A[i]$ are popped. Starting with an empty string, the algorithm constructs a sequence S as described below. Whenever k elements are popped from the stack and then an element is pushed into the stack, $(^k)$ is prepended to S. Finally, after pushing $A[1]$ into the stack, if the stack contains m elements, then $(^{m+1})$ is prepended to S. One can show that this sequence S is the same as the DFUDS sequence $D_{\mathsf{Min(A)}}$. Analogously, one can construct $D_{\mathsf{Max(A)}}$ using a similar stack-based algorithm.

Colored 2d-Min heap From the definition of $2d$-Min heap, it is easy show that $\mathsf{PSV}_A(i)$, for $1 \leq i \leq n$, is the label corresponding to the parent of the node labeled i in $\mathsf{Min(A)}$. Thus, using the encoding of Lemma 5.2 using $2n+o(n)$ bits, one can support the $\mathsf{PSV}_A(i)$ queries in constant time. A straightforward way to support $\mathsf{NSV}_A(i)$ is to construct the $2d$-Min heap structure for the reverse of the array A, and encode it using an additional $2n+o(n)$ bits. Therefore one can encode all answers of PSV_A and NSV_A using $4n+o(n)$ bits with constant query time. To reduce this size, Fischer proposed the *colored $2d$-Min heap* [25]. This



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	-∞	3	8	5	6	6	6	3	2	2	7	10	9

Figure 5.1 Colored $2d$-Min heap of A

has the same structure as normal $2d$-Min heap, and in addition, the vertices are colored either red or blue. Suppose there is a parent node x in the colored $2d$-Min heap with its children $x_1 \dots x_k$. Then for $1 < i \leq k$, node x_i is colored red if $A[x_i] < A[x_{i-1}]$, and all the other nodes are colored blue (see Figure 5.1). We define the operation $\mathsf{NRS}(x_i)$ which returns the leftmost red sibling to the right (i.e., next red sibling) of x_i.

The following lemma can be used to support $\mathsf{NSV}_A(i)$ efficiently using the colored $2d$-Min heap representation.

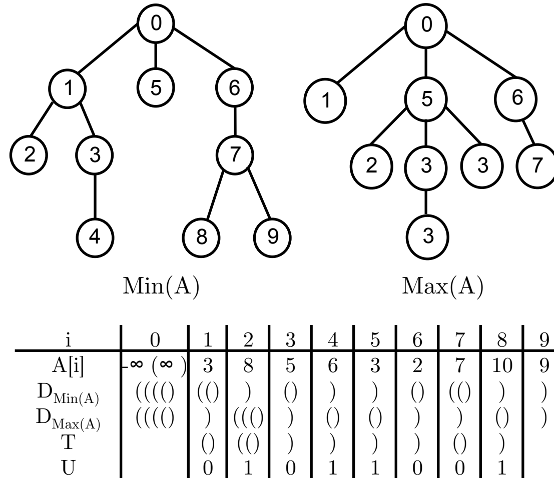
Lemma 5.4 ([25])**.** *Let CMin(A) be the colored $2d$-Min heap on A. Suppose there is a parent node x in CMin(A) with its children $x_1 \dots x_k$. Then for $1 \leq i \leq k$,*

$$\mathsf{NSV}_A(x_i) = \begin{cases} \mathsf{NRS}(x_i) & \text{if } \mathsf{NRS}(x_i) \text{ exists,} \\ x_k + \mathsf{subtree_size}(x_k) & \text{otherwise.} \end{cases}$$

If all the elements in A are distinct, then a $2n + o(n)$-bit encoding of $\mathsf{Min(A)}$ is enough to support RMinQ_A, PSV_A and NSV_A with constant query time. In the general case, Fischer proposed an optimal $2.54n + o(n)$-bit encoding of colored $2d$-Min heap on A using TC-encoding [22]. This encoding also supports two additional operations, namely *modified* $\mathsf{child}_{\mathsf{CMin(A)}}(x, i)$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}(x)$,

which answer the i-th red child of node x and the number of red siblings to the left of node x, respectively, in constant time. Using these operations, one can also support RLMinQ_A and RkMinQ_A in constant time.

5.2.2 Encoding range min-max queries



i	0	1	2	3	4	5	6	7	8	9
A[i]	-∞ (∞)	3	8	5	6	3	2	7	10	9
$D_{Min(A)}$	(((()	(()	)	()	)	)	()	(()	)	)
$D_{Max(A)}$	(((()	)	((()	)	()	()	)	)	()	)
T		()	(()	)	)	)	)	()	)	
U		0	1	0	1	1	0	0	1	

Figure 5.2 Encoding of $2d$-Min heap and $2d$-Max heap of A

One can support both RMinQ_A and RMaxQ_A in constant time by encoding both $\mathsf{Min(A)}$ and $\mathsf{Max(A)}$ separately using $4n + o(n)$ bits. Gawrychowski and Nicholson [30] described an alternate encoding that uses only $3n + o(n)$ bits while still supporting the queries in $O(1)$ time. There are two key observations which are used in obtaining this structure:

1. If we can access any $\lg n$-bit substring of $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ on $O(1)$ time, we can still support both queries in $O(1)$ time, using an additional $o(n)$ bits;

2. To generate $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ using Fischer and Heun's stack-based algorithm, in each step we push an element into both the min-stack and

max-stack, and pop a certain number of elements from exactly one of the stacks (assuming that $A[i] \neq A[i+1]$, for all i, where $1 \leq i < n$).

Now we describe the overall encoding in [30] briefly. The structure consists of two bit strings T and U along with various auxiliary structures. For $1 \leq i < n$, if k elements are popped from the min (max)-stack when we push $A[i](1 \leq i < n)$ into both the stacks (from right to left), we prepend $(^{k-1}$) and 0(1) to the currently generated T and U respectively. Initially, when $i = n$, both min and max stacks push ')' so we do not prepend anything to both strings. But we can recover it easily because this is the last ')' in common. Finally, after pushing $A[1]$ into both the stacks, we pop the remaining elements from them, and store the number of these popped elements in min and max stack explicitly using $\lg n$ bits. One can show that the size of T is at most $2n$ bits, and that of U is $n-1$ bits. Thus the total space usage is at most $3n$ bits. See Algorithm 1 for the pseudocode, and Figure 5.2 for an example.

To recover any $\lg n$-bit substring, $D_{\mathsf{Min(A)}}[d_1 \dots d_{\lg n}]$, in constant time we construct the following auxiliary structures. We first divide $D_{\mathsf{Min(A)}}$ into blocks of size $\lg n$, and for the starting position of each block, store its corresponding position in T. For this purpose, we construct a bit string B_{min} of length at most $2n$ such that $B_{min}[i] = 1$ if and only if $T[i]$ corresponds to the start position of the ith-block in $D_{\mathsf{Min(A)}}$. We encode B_{min} using the representation of Lemma 2.5 which takes $o(n)$ bits since the number of ones in B_{min} is $2n/\lg n$. Then if d_1 belongs to the i-th block, it is enough to recover the i-th and the $(i+1)$-st blocks in the worst case.

Now, to recover the i-th block of $D_{\mathsf{Min(A)}}$, we first compute the distance between i-th and $(i+1)$-st 1's in B_{min}. If this distance is less than $c \lg n$ for some fixed constant $c > 9$, we call it a *min-good block*, otherwise, we call it a *min-bad block*. We can recover a min-good block in $D_{\mathsf{Min(A)}}$ in $O(c)$ time using a $o(n)$-bit pre-computed table indexed by all possible strings of length $\lg n/4$ bits for T and U (we can find the position corresponding to the i-th block

Algorithm 1 Construction algorithm for T and U

```
1: Initialize T to ')', and U to ε.
2: Initialize Min-stack and Max-stack as empty stacks
3: Push A[n] into Min-stack and Max-stack.
4: for i := n − 1 to 1 do
5:     counter = 0
6:     if A[i] < A[i − 1] then
7:         Push A[i] into Max-stack
8:         while ((Min-stack is not empty) & (Top of Min-stack > A[i])) do
9:             Pop Min-stack
10:            counter = counter + 1
11:        end while
12:        Push A[i] into Min-stack
13:        Prepend (^(counter−1)) to T and 0 to U
14:    else // A[i] > A[i − 1]
15:        Push A[i] into Min-stack
16:        while ((Max-stack is not empty) & (Top of Max-stack < A[i])) do
17:            Pop Max-stack
18:            counter = counter + 1
19:        end while
20:        Push A[i] into Max-stack
21:        Prepend (^(counter−1)) to T and 1 to U
22:    end if
23: end for
```

in U in constant time), which stores the appropriate $O(\lg n)$ bits of $D_{\mathsf{Min(A)}}$ obtained from them (see [30] for details). For min-bad blocks, we store the answers explicitly. This takes $(2n/(c \lg n)) \cdot \lg n = 2n/c$ additional bits. To save this additional space, we store the min-bad blocks in compressed form using the property that any min-bad block in $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ cannot overlap more than $4 \lg n$ bits in T, (since any $2 \lg n$ consecutive bits in T consist of at least $\lg n$ bits from either $D_{\mathsf{Min(A)}}$ or $D_{\mathsf{Max(A)}}$). So, for $c > 9$ we can save more than $\lg n$ bits by compressing the remaining $(c-4)\lg n$ bits in T corresponding to each min-bad block in $D_{\mathsf{Min(A)}}$. Thus, we can reconstruct any $\lg n$-bit substring of $D_{\mathsf{Min(A)}}$ (and $D_{\mathsf{Max(A)}}$) in constant time, using a total of $3n + o(n)$ bits.

We first observe that if there is a position i, for $1 \leq i < n$ such that $A[i] = A[i+1]$, we cannot decode the '$)'$ in T which corresponds to $A[i]$ only using T and U since we do not pop any elements from both min and max stacks when we push $A[i]$ into both stacks. Gawrychowski and Nicholson [30] handle this case by defining an ordering between equal elements (for example, by breaking the ties based on their positions). But this ordering does not help us in supporting the PSV and PLV queries. We describe how to handle the case when there are repeated (consecutive) elements in A, to answer the PSV and PLV queries.

Gawrychowski and Nicholson [30] also show that any encoding that supports both RMinQ_A and RMaxQ_A cannot use less than $3n - \Theta(\lg n)$ bits for sufficiently large n (even if all elements in A are distinct).

5.3 Extended DFUDS for colored 2d-Min heap

In this section, we describe an encoding of colored $2d$-Min heap on A ($\mathsf{CMin(A)}$) using at most $3n + o(n)$ bits while supporting RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A and NSV_A in constant time. This is done by storing the color information of the nodes using a bit string of length at most n, in addition to the DFUDS representation of $\mathsf{CMin(A)}$. We can also encode the colored $2d$-Max

heap in a similar way. In the worst case, this representation uses more space than the colored $2d$-Min heap encoding of Fischer [25], but the advantage is that it separates the tree encoding from the color information. We later describe how to combine the tree encodings of the $2d$-Min heap and $2d$-Max heap, and (separately) also combine the color information of the two trees, to reduce the overall space.

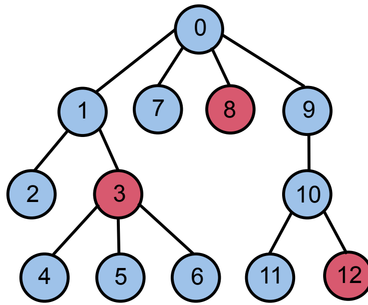
Now we describe the main encoding of $\mathsf{CMin(A)}$. The encoding consists of two parts: $D_{\mathsf{CMin(A)}}$ and V_{min}. The sequence $D_{\mathsf{CMin(A)}}$ is same as $D_{\mathsf{Min(A)}}$, the DFUDS representation of $\mathsf{CMin(A)}$, which takes $2n + o(n)$ bits and supports the operations in Lemma 5.2 in constant time.

The bit string V_{min} stores the color information of all nodes in $\mathsf{CMin(A)}$, except the nodes which are the leftmost children of their parents (the color of these nodes is always blue), as follows. Suppose there are p nodes in $\mathsf{CMin(A)}$, for $1 \leq p \leq n$, which are the leftmost children of their parents. Then we define the bit string $V_{min}[0 \ldots n-p]$ as follows. For $1 \leq i \leq n-p$, $V_{min}[i]$ stores 0 if the color of the node

$$\mathsf{node}_{V_{min}}(i) = \mathsf{pre_rank}_{\mathsf{CMin(A)}}(\mathsf{findclose}_{D_{\mathsf{CMin(A)}}}(\mathsf{select}_{((}(D_{\mathsf{CMin(A)}}, i+1)) + 1)$$

in $\mathsf{CMin(A)}$ is red, and 1 otherwise. This follows from the observation that if there is an $i, 1 \leq i < 2n-1$ such that $D_{\mathsf{CMin(A)}}[i] =$ '$($' and $D_{\mathsf{CMin(A)}}[i+1] =$ '$)$', then $D_{\mathsf{CMin(A)}}[i+2]$ corresponds to the node which is the leftmost child of the node $\mathsf{pre_rank}_{\mathsf{CMin(A)}}(D_{\mathsf{CMin(A)}}[i])$, so we skip these nodes by counting the pattern '$(\, ($' in $D_{\mathsf{CMin(A)}}$. Also, we set $V_{min}[0] = 1$, which corresponds to the first open parenthesis in $D_{\mathsf{CMin(A)}}$. For example, for $\mathsf{CMin(A)}$ in Figure 5.1, we store the node 3's color in $V_{min}[4]$. This is becuase $\mathsf{select}_{((}(D_{\mathsf{CMin(A)}}, 5) = 7$, $\mathsf{findclose}_{D_{\mathsf{CMin(A)}}}(7) + 1 = 11$ and $\mathsf{pre_rank}_{\mathsf{CMin(A)}}(11) = 3$ (see Figure 5.3). We define the bit string V_{max} in a similar way.

The following lemma shows that encoding $\mathsf{Min(A)}$ and V_{min} separately, using at most $3n + o(n)$ bits, has the same functionality as the $\mathsf{CMin(A)}$ encoding of



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	-∞	3	8	5	6	6	6	3	2	2	7	10	9

$D_{\mathsf{CMin(A)}}$	(	(	(	(	(	)	(	(	)	)	(	(	(	)	)	)	)	)	)	(	)	(	(	)	)	)
pre_rank$_{\mathsf{CMin(A)}}$	0	0	0	0	0	0	1	1	1	2	3	3	3	3	4	5	6	7	8	9	9	10	10	10	11	12

V_{min}	1	1	0	1	0	1	1	0
node$_{V_{min}}$	-	9	8	7	3	6	5	12

pre_select$_{\mathsf{CMin(A)}}$	1	7	10	11	15	16	17	18	19	20	22	25	26
node_color$_{\mathsf{CMin(A)}}$	-	-	-	4	-	6	5	3	2	1	-	-	7

Figure 5.3 $D_{\mathsf{CMin(A)}}$, pre_rank$_{\mathsf{CMin(A)}}$, $V_{min}[i]$, node$_{V_{min}}$, pre_select$_{\mathsf{CMin(A)}}$ and node_color$_{\mathsf{CMin(A)}}$ for colored $2d$-Min heap

Fischer [25], which only takes $2.54n + o(n)$ bits.

Lemma 5.5. *For an array $A[1 \dots n]$ of length n, there is an encoding for A which takes at most $3n + o(n)$ bits and supports RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A and NSV_A in constant time.*

Proof. The encoding consists of the $2n + o(n)$-bit encoding of $\mathsf{Min(A)}$ encoded using structure of Lemma 5.2, together with the bit string V_{min} that stores the color information of the nodes in $\mathsf{CMin(A)}$. We use a $o(n)$-bit auxiliary structure to support the $\mathsf{rank}/\mathsf{select}$ queries on V_{min} in constant time. Since the size of V_{min} is at most n bits, the total space of the encoding is at most $3n + o(n)$ bits.

To define the correspondence between the nodes in $\mathsf{CMin(A)}$ and the positions in the bit string V_{min}, we define the following operation. For $0 \leq i \leq n$, we define $\mathsf{node_color}_{\mathsf{CMin(A)}}(i)$ as the position of V_{min} that stores the color of the node i in $\mathsf{CMin(A)}$. This can be computed in constant time, using $o(n)$ bits, by

$$\mathsf{node_color}_{\mathsf{CMin(A)}}(i) = \begin{cases} \text{undefined} & \text{if } \mathsf{child_rank}_{\mathsf{CMin(A)}}(i) = 0 \\ \mathsf{rank}_{((}(D_{\mathsf{CMin(A)}}, c) - 1 & \text{otherwise} \end{cases}$$

where $c = \mathsf{findopen}_{D_{\mathsf{CMin(A)}}}(\mathsf{pre_select}_{\mathsf{CMin(A)}}(i) - 1)$ (note that $\mathsf{node_color}_{\mathsf{CMin(A)}}$ is the inverse operation of $\mathsf{node}_{V_{min}}$).

Now we describe how to support the queries in constant time. Fischer and Heun [27] showed that $\mathsf{RMinQ}_A(i, j)$ can be answered in constant time using $D_{\mathsf{CMin(A)}}$. In fact, they return the position $\mathsf{RRMinQ}_A(i, j)$ as the answer to $\mathsf{RMinQ}_A(i, j)$. Also, as mentioned earlier, $\mathsf{PSV}_A(i) = \mathsf{parent}_{\mathsf{CMin(A)}}(i)$, and hence can be answered in constant time. Therefore, it is enough to describe how to find $\mathsf{RLMinQ}_A(i, j)$, $\mathsf{RkMinQ}_A(i, j)$ and $\mathsf{NSV}_A(i)$ in constant time.

$\mathsf{RLMinQ}_A(i, j)$: As shown by Fischer and Huen [27], all corresponding values of left siblings of the node $\mathsf{RRMinQ}_A(i, j)$ in A are at least $A[\mathsf{RRMinQ}_A(i, j)]$ (i.e., the values of the siblings are in the non-increasing order, from left to right). Also,

for a child node m of any of the left siblings of the node $\mathsf{RRMinQ}_A(i,j)$, $A[m] > A[\mathsf{RRMinQ}_A(i,j)]$. Therefore, the position $\mathsf{RLMinQ}_A(i,j)$ corresponds to one of the left siblings of the node whose position corresponds to $\mathsf{RRMinQ}_A(i,j)$.

We first check whether the color of the node $\mathsf{RRMinQ}_A(i,j)$ is red or not using V_{min}. If $V_{min}[\mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j))] = 0$ then $\mathsf{RLMinQ}_A(i,j)$ is equal to $\mathsf{RRMinQ}_A(i,j)$. If not, we find the node $\mathsf{leftmost}(i,j)$ which is the left-most sibling of the node $\mathsf{RRMinQ}_A(i,j)$ between the nodes in $[i \dots j]$. $\mathsf{leftmost}(i,j)$ can be found in constant time by computing the depth of node i and comparing this value with d_{right}, the depth of the node $\mathsf{RRMinQ}_A(i,j)$. More specifically,

$$\mathsf{leftmost}(i,j) = \begin{cases} i & \text{if } \mathsf{depth}_{\mathsf{CMin(A)}}(i) = d_{right}. \\ \mathsf{next_sibling}_{\mathsf{CMin(A)}}(\mathsf{la}_{\mathsf{CMin(A)}}(i, d_{right})) & \text{otherwise.} \end{cases}$$

In the next step, find the leftmost blue sibling n_v such that there is no red sibling between n_v and $\mathsf{RRMinQ}_A(i,j)$. This can be found in constant time by first finding the index v using the equation

$$v = \mathsf{select}_0(V_{min}, \mathsf{rank}_0(V_{min}, \mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j))) + 1) - 1$$

and then finding the node n_v using $n_v = \mathsf{node}_{V_{min}}(v)$. If $\mathsf{child_rank}_{\mathsf{CMin(A)}}(n_v) \leq \mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{leftmost}(i,j))$ or $\mathsf{child_rank}_{\mathsf{CMin(A)}}(n_v) = 1$ (this is the case that $\mathsf{leftmost}(i,j)$ can be the the lestmost sibling), then $\mathsf{RLMinQ}_A(i,j) = \mathsf{leftmost}(i,j)$. Otherwise, $\mathsf{RLMinQ}_A(i,j) = n_v$.

$\mathsf{RkMinQ}_A(i,j)$: This query can be answered in constant time by returning the k-th sibling (in the left-to-right order) of $\mathsf{RLMinQ}_A(i,j)$, if it exists. More formally, if $\mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j)) - \mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{RLMinQ}_A(i,j))$ is at least $k-1$, then $\mathsf{RkMinQ}_A(i,j)$ exists; and in this case, $\mathsf{RkMinQ}_A(i,j)$ can be computed in constant time by computing

$$\mathsf{child}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j)), \mathsf{RLMinQ}_A(i,j) + k - 1).$$

$\mathsf{NSV}_A(i)$: By Lemma 5.4, it is enough to show how to support $\mathsf{NRS}(i)$ in

constant time (note that we can support $\mathsf{subtree_size}$ in constant time using Lemma 5.2). If node i is the rightmost sibling, then $\mathsf{NRS}(i)$ does not exist. Otherwise we define v' as $\mathsf{select}_0(V_{min}, \mathsf{rank}_0(V_{min}, \mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{next_sibling}(i))))$. Let $n_{v'} = \mathsf{node}_{V_{min}}(v')$. If the parent of $n_{v'}$ is same as the parent of i, then $\mathsf{NRS}(i) = n_{v'}$; otherwise $\mathsf{NRS}(i)$ does not exist. Finally, if $\mathsf{NRS}(i)$ does not exist, we compute the node r which is the rightmost sibling of the node i can be found by

$$\mathsf{child}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(i), \mathsf{degree}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(i)) - 1).$$

Then $\mathsf{NSV}_A(i) = r + \mathsf{subtree_size}_{\mathsf{CMin(A)}}(r)$. All these operations can be done in constant time. □

5.4 Encoding colored 2d-Min and 2d-Max heaps

In this section, we describe our encodings for supporting various subsets of operations, proving the results stated in Theorem 5.1. As mentioned in Section 5.2.1, the TC-encoding of the colored $2d$-Min heap of Fischer [25] can answer RMinQ_A, RRMinQ_A, PSV_A and NSV_A queries in $O(1)$ time, using $2.54n + o(n)$ bits. The following lemma shows that we can also support the queries RLMinQ_A and RkMinQ_A using the same structure.

Lemma 5.6. *For an array $A[1 \dots n]$ of length n, RLMinQ_A, RkMinQ_A can be answered in constant time by the TC-encoding of colored $2d$-Min heap.*

Proof. Fischer [25] defined two operations, which are modifications of the child and $\mathsf{child_rank}$, as follows:

- $\mathsf{mchild}_{\mathsf{CMin(A)}}(x, i)$ - returns the i-th red child of node x in CMin(A), and
- $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}(x)$ - returns the number of red siblings to the left of node x in CMin(A).

He showed that the TC-encoding of the colored $2d$-Min heap can support $\mathsf{mchild}_{\mathsf{CMin(A)}}(x, i)$ and $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}(x)$ in constant time. Also, since the TC-encoding supports $\mathsf{depth}_{\mathsf{CMin(A)}}$, $\mathsf{next_sibling}_{\mathsf{CMin(A)}}$, $\mathsf{la}_{\mathsf{CMin(A)}}$, $\mathsf{child}_{\mathsf{CMin(A)}}$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}$ in constant time on ordinal trees [41], we can support $\mathsf{leftmost}(i, j)$ (defined in the proof of the Lemma 5.5) in constant time. For answering $\mathsf{RLMinQ}_A(i, j)$, we first find the previous red sibling l of $\mathsf{RRMinQ}_A(i, j)$ using $\mathsf{mchild}_{\mathsf{CMin(A)}}$ and $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}$. If such a node l exists, we compare the child ranks of $\mathsf{next_sibling}_{\mathsf{CMin(A)}}(l)$ and $\mathsf{leftmost}(i, j)$, and return the node with the larger rank value as the answer. $\mathsf{RkMinQ}_A(i, j)$ can be answered by returning the k-th sibling (in the left-to-right order) of $\mathsf{RLMinQ}_A(i, j)$ using $\mathsf{child}_{\mathsf{CMin(A)}}$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}$, if it exists. □

By storing a similar TC-encoding of colored $2d$-Max heap, in addition to the structure of Lemma 5.6, we can support all the operations mentioned in Theorem 5.1(c) in $O(1)$ time. This uses a total space of $5.08n + o(n)$ bits. We now describe alternative encodings to reduce the overall space usage.

More specifically, we show that a combined encoding of $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$, using at most $3.17n + o(n)$ bits, can be used to answer RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A, and PLV_A queries (Theorem 5.1(a)). To support the queries in constant time, we use a less space-efficient data structure that encodes the same structures, using at most $3.322n+o(n)$ bits (Theorem 5.1(b)). Similarly, a combined encoding of $D_{\mathsf{CMin(A)}}$, $D_{\mathsf{CMax(A)}}$, V_{min} and V_{max} using at most $4.088n + o(n)$ bits can be used to answer RLMinQ_A, RkMinQ_A, NSV_A, RLMaxQ_A, RkMaxQ_A, and NLV_A queries in addition (Theorem 5.1(c)). Again, to support the queries in constant time, we design a less space-efficient data structure using at most $4.58n + o(n)$ bits (Theorem 5.1(d)).

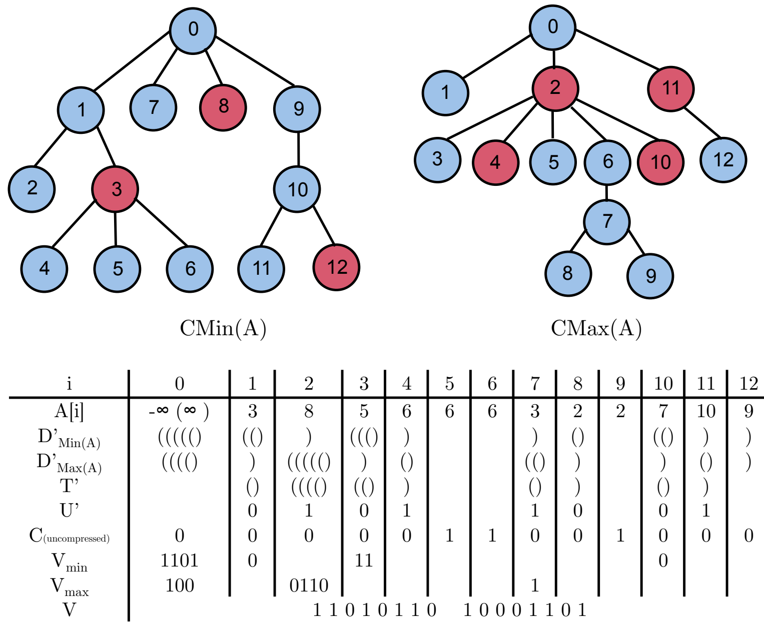
In the following, we first describe the data structure of Theorem 5.1(b) followed by the structure for Theorem 5.1(d). Next we describe the encodings of Theorem 5.1(a) and Theorem 5.1(c).

5.4.1 Combined data structure for $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$

As mentioned in Section 5.2.2, the encoding of Gawrychowski and Nicholson [30] consists of two bit strings U and T of total length at most $3n$, along with the encodings of B_{min}, B_{max} and a few additional auxiliary structures of total size $o(n)$ bits. In this section, we denote this encoding by E. To encode the DFUDS sequences of $\mathsf{CMin(A)}$ and $\mathsf{CMax(A)}$ in a single structure, we use E with some modifications, which we denote by E'. As described in Section 5.2.2, encoding scheme of Gawrychowski and Nicholson cannot be used (as it is) to support the PSV and PLV queries if there is a position i, for $1 \leq i < n$ such that $A[i] = A[i+1]$. To support these queries, we define an additional bit string $C[1 \dots n]$ such that $C[1] = 0$, and for $1 < i \leq n$, $C[i] = 1$ iff $A[i-1] = A[i]$. If the bit string C has k ones in it, then we represent C using $\lg \binom{n}{k} + o(n)$ bits while supporting rank, select queries and decoding any $\lg n$ consecutive bits in C in constant time, using Lemma 2.5. We also define a new array $A'[0 \dots n-k]$ by setting $A'[0] = A[0]$, and for $0 < i \leq n-k$, $A'[i] = A[\mathsf{select}_0(C, i)]$. (Note that A' has no consecutive repeated elements.) In addition, we define another sequence $D'_{\mathsf{CMin(A)}}$ of size $2n-k$ as follows. Suppose $D_{\mathsf{CMin(A')}} = (^{\delta_1}\ ) \dots (^{\delta_{n-k}}\ )$, for some $0 \leq \delta_1 \dots \delta_n \leq n-k$, then we set $D'_{\mathsf{CMin(A)}} = (^{\delta_1+\epsilon_1}\ ) \dots (^{\delta_{n-k}+\epsilon_{n-k}}\ )$, where $\delta_i + \epsilon_i$ is the number of elements popped when $A[i]$ is pushed into the min-stack of A, for $1 \leq i \leq n-k$. (Analogously, we define $D'_{\mathsf{CMax(A)}}$.)

The encoding E' defined on A consists of two bit strings U' and T', along with C, B'_{min}, B'_{max} and additional auxiliary structures (as in E). Let U and T be the bit strings in E defined on A'. Then U' is same as U in E, and size of U' is $n-k-1$ bits. To obtain T', we add some additional open parentheses to T as follows. Suppose $T = (^{\delta_1}\ )(^{\delta_2}\ ) \dots (^{\delta_{n-k}}\ )$, where $0 \leq \delta_i \leq n-k$ for $1 \leq i \leq n-k$. Then $T' = (^{\delta_1+\epsilon_1}\ ) \dots (^{\delta_{n-k}+\epsilon_{n-k}}\ )$, where $\delta_i + \epsilon_i$ is the number of elements are popped when $A[i]$ is pushed into the min or max stack of A, for $1 \leq i \leq n-k$ (see Figure 5.4 for an example). Since the length of T is at most

$2(n-k)$, and $|T'|-|T| = \sum_{i=1}^{n-k} \epsilon_i \leq k$, the size of T' is at most $2n-k$ bits. The encodings of B'_{min} and B'_{max} are defined on $D'_{\mathsf{CMin(A)}}$, $D'_{\mathsf{CMax(A)}}$ and T', similar to B_{min} and B_{max} in E. The total size of the encodings of the modified B'_{min} and B'_{max} is $o(n)$ bits. All the other auxiliary structures use $o(n)$ bits. Although we use E' instead of E, we can use the decoding algorithm in E without any modifications because all the properties used in the algorithm still hold even though T' has additional open parentheses compared to T. Therefore from E' we can reconstruct any $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ or $D'_{\mathsf{CMax(A)}}$ in constant time, and thus we can support rank and select on these strings in constant time with $o(n)$ additional structures by Lemma 2.5.



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	-∞ (∞)	3	8	5	6	6	6	3	2	2	7	10	9
$D'_{Min(A)}$	((((()	(()	)	((()	)			)	()		(()	)	)
$D'_{Max(A)}$	(((()	)	((((()	)	()			(()	)		)	()	)
T'		()	(((()	(()	)			()	)		()	)	
U'		0	1	0	1			1	0		0	1	
$C_{(uncompressed)}$	0	0	0	0	0	1	1	0	0	1	0	0	0
V_{min}	1101	0		11							0		
V_{max}	100		0110					1					
V	1 1 0 1 0 1 1 0 1 0 0 0 1 1 0 1												

Figure 5.4 Data structure combining the colored $2d$-Min heap and colored $2d$-Max heap of A. C is represented in uncompressed form.

Decoding $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$

We use the following auxiliary structures to decode $D_{\mathsf{CMin(A)}}$ from $D'_{\mathsf{CMin(A)}}$ and C. For this, we first define a correspondence between $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$ as follows. Note that both $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$ have the same number of open parentheses, but $D'_{\mathsf{CMin(A)}}$ has fewer close parentheses than $D_{\mathsf{CMin(A)}}$. The ith open parenthesis in $D_{\mathsf{CMin(A)}}$ corresponds to the ith open parenthesis in $D'_{\mathsf{CMin(A)}}$. Suppose there are ℓ and ℓ' $(\leq \ell)$ close parentheses between the ith and the $(i+1)$st open parentheses in $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$, respectively. Then the last ℓ' close parentheses in $D_{\mathsf{CMin(A)}}$ correspond, in that order, to the ℓ' close parentheses in $D'_{\mathsf{CMin(A)}}$; the remaining close parentheses in $D_{\mathsf{CMin(A)}}$ do not have a corresponding position in $D'_{\mathsf{CMin(A)}}$.

We construct three bit strings P_{min}, Q_{min} and R_{min} of lengths $2n - k$, $\lceil 2n/\lg n \rceil$ and $\lceil 2n/\lg n \rceil$, respectively, as follows. For $1 \leq i \leq \lceil 2n/\lg n \rceil$, if the position $i \lg n$ in $D_{\mathsf{CMin(A)}}$ has its corresponding position j in $D'_{\mathsf{CMin(A)}}$, then we set $P_{min}[j] = 1$, $Q_{min}[i] = 0$ and $R_{min}[i] = 0$. If position $i \lg n$ in $D_{\mathsf{CMin(A)}}$ has no corresponding position in $D'_{\mathsf{CMin(A)}}$ but for some k_i where $1 \leq k_i < \lg n$, suppose there is a leftmost position $q = i \lg n + k_i$ which has its corresponding position j in $D'_{\mathsf{CMin(A)}}$. Then we set $P_{min}[j] = 1$, $Q_{min}[i] = 1$ and $R_{min}[i] = 0$. Finally, if all positions between $i \lg n$ and $(i+1) \lg n$ in $D_{\mathsf{CMin(A)}}$ have no corresponding position in $D'_{\mathsf{CMin(A)}}$, then we set $Q_{min}[i] = 1$ and $R_{min}[i] = 1$. In remaining positions for P_{min}, Q_{min} and R_{min}, we set their values as 0. We also store the values, k_i explicitly, for $1 \leq i \leq \lceil 2n/\lg n \rceil$, whenever they are defined (as in the second case). Since $k_i < \lg n$, we can store all the k_i values explicitly using at most $2n \lg \lg n / \lg n = o(n)$ bits.

Since the bit strings P_{min}, Q_{min} and R_{min} have at most $2n/\lg n$ 1's each, they can be represented using the structure of Lemma 2.5, taking $o(n)$ bits while supporting rank and select queries in constant time. We define P_{max}, Q_{max}, R_{max} in the same way, and represent them analogously.

In addition to these bit strings, we construct two pre-computed tables. In the rest of this section, we refer to the parenthesis strings (such as $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$) also as bit strings. To describe these tables, we first define two functions f and f', each of which takes two bit strings s and c as parameters, and returns a bit string of length at most $|s| + |c|$, as follows.

$$\begin{cases} f(s, \epsilon) = s \\ f(\epsilon, c) = \epsilon \\ f(s, 1 \cdot c_1) =) \cdot f(s, c_1) \\ f((^{\delta}\ ) \cdot s_1, 0 \cdot c_1) = (^{\delta} \cdot) \cdot f(s_1, c_1) \end{cases} \begin{cases} f'(s, \epsilon) = s \\ f'(\epsilon, c) = \epsilon \\ f'(s, c_1 \cdot 1) = f'(s, c_1) \cdot) \\ f'(s_1 \cdot (^{\delta}\ ), c_1 \cdot 0) = f'(s_1, c_1) \cdot (^{\delta} \cdot) \end{cases}$$

One can easily show that if s is a substring of $D'_{\mathsf{CMin(A)}}$ and c is a substring of C whose starting (ending) position corresponds to the starting (ending) position in s, then $f(s, c)$ ($f'(s, c)$) returns the substring of $D_{\mathsf{CMin(A)}}$ whose starting (ending) position corresponds to the starting (ending) position in s,

We construct a pre-computed table T_f that, for each possible choice of bit strings s and c of length $(1/4)\lg n$, stores the bit string $f(s, c)$. These pre-computed tables can be used to decode a substring of $D_{\mathsf{CMin(A)}}$ given a substring of $D'_{\mathsf{CMin(A)}}$ (denoted s) and a substring of C whose bits correspond to s. The total space usage of T_f is $2^{(1/4)\lg n} \cdot 2^{(1/4)\lg n} \cdot ((1/2)\lg n) = o(n)$ bits. We can also construct $T_{f'}$ defined analogous to T_f using $o(n)$ bits.

Now we describe how to decode $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ in constant time. (We can decode $\lg n$ consecutive bits of $D_{\mathsf{CMax(A)}}$ in a similar way.) Suppose we divide $D_{\mathsf{CMin(A)}}$ into blocks of size $\lg n$. As described in Section 5.2.2, it is enough to show that for $1 \leq i \leq \lceil 2n/\lg n \rceil$, we can decode i-th block of $D_{\mathsf{CMin(A)}}$ in constant time. First, we check the value of the $R_{min}[i]$. If $R_{min}[i] = 1$, then the i-th block in $D_{\mathsf{CMin(A)}}$ consists of a sequence of $\lg n$ consecutive close parentheses. Otherwise, there are two cases depending on the value of $Q_{min}[i]$. We compute the position p which is a position in $D'_{\mathsf{CMin(A)}}$ (it's exact correspondence in $D_{\mathsf{CMin(A)}}$ depends on the value of the bit $Q_{min}[i]$), and

then compute the position c_p in C which corresponds to p in $D'_{\mathsf{CMin(A)}}$, using the following equations:

$$p = \mathsf{select}_1(P_{min}, i - \mathsf{rank}_1(R_{min}, i))$$

$$c_p = \begin{cases} \mathsf{select}_0(C, \mathsf{rank}_)(D'_{\mathsf{CMin(A)}}, p)) & \text{if } D'_{\mathsf{CMin(A)}}[p] =')' \\ \mathsf{select}_0(C, \mathsf{rank}_)(D'_{\mathsf{CMin(A)}}, p) + 1) & \text{otherwise} \end{cases}$$

Case (1) $\boldsymbol{Q_{min}[i] = 0}$. In this case, we take the $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ starting from p, and the $\lg n$ consecutive bits of C starting from the position c_p (padding at the end with zeros if necessary). Using these bit strings, we can decode the i-block in $D_{\mathsf{CMin(A)}}$ by looking up T_f with these substrings (a constant number of times, until the pre-computed table generates the required $\lg n$ bits). Since the position p corresponds to the starting position of the i-th block in $D_{\mathsf{CMin(A)}}$ in this case, we can decode the i-th block of $D_{\mathsf{CMin(A)}}$ in constant time.

Case (2) $\boldsymbol{Q_{min}[i] = 1}$. First we decode $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ whose starting position corresponds to the position p using the same procedure as in Case (1). Let S_1 be this bit string. Next, we take the $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ ending with position p, and the $\lg n$ consecutive bits of C ending with position c_p (padding at the beginning with zeros if necessary). Then we can decode the $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ whose ending position corresponds to the p by looking up $T_{f'}$ (a constant number of times) with these substrings. Let S_2 be this bit string. By concatenating S_1 and S_2, we obtain a $2\lg n$-bit substring of $D_{\mathsf{CMin(A)}}$ which contains the starting position of the i-th block of $D_{\mathsf{CMin(A)}}$ (since the starting position of the i-th block in $D_{\mathsf{CMin(A)}}$, and the position which corresponds to p differ by at most $\lg n$). Finally, we can obtain the i-th block in $D_{\mathsf{CMin(A)}}$ by skipping the first $\lg n - k_i$ bits in $S_1 \cdot S_2$, and taking $\lg n$ consecutive bits from there.

From the encoding described above, we can decode any $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$ in constant time. Therefore by Lemma 5.5, we can answer all queries supported by CMin(A) and CMax(A) (without using the color information) in constant time. If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, then the size of C is $\lg \binom{n}{k} + o(n)$ bits, and the size of E' on A is $3n - 2k + o(n)$ bits. All other auxiliary bit strings and tables take $o(n)$ bits. Therefore, by the Lemma 5.3, we can encode A using $3n - 2k + \lg \binom{n}{k} + o(n) \leq ((1 + \lg 5)n + o(n) < 3.322n + o(n)$ bits. Also, this encoding supports the queries in Theorem 5.1(b) (namely RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A and PLV_A, which do not need the color information) in constant time. This proves Theorem 5.1(b).

Note that if $k = 0$ (i.e, there are no consecutive equal elements), E' on A is same as E on A. Therefore, we can support all the queries in Theorem 5.1(b) using $3n + o(n)$ bits with constant query time.

Encoding V_{min} and V_{max}

We simply concatenate V_{max} and V_{min} on A and store it as bitstring V, and store the length of V_{min} using $\lg n$ bits (see V in Figure 5.4). If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, Fischer and Heun's stack based algorithm [27] does not pop any elements from both stacks when these k elements and $A[n]$ are pushed into them. Before pushing any of the remaining elements into the min- and max-stacks, we pop some elements from exactly one of the stacks. Also after pushing $A[1]$ into both the stacks, we pop the remaining elements from the stacks in the final step. Suppose the n elements are popped from the min-stack during p runs of pop operations. Then, it is easy show that the elements are popped from the max-stack during $n - k - p$ runs of pop operations. Also, p $(n - k - p)$ is the number of leftmost children in CMin(A) (CMax(A)) since each run of pop operations generates exactly one open parenthesis whose matched closing parenthesis corresponds to the leftmost child

in $\mathsf{CMin(A)}$ ($\mathsf{CMax(A)}$). As described in Section 5.3, the size of V_{min} is $n-p+1$ bits, and that of V_{max} is $p+k+1$ bits. Thus, the total size of V is $n+k+2$ bits.

Therefore, we can decode any $\lg n$-bit substring of V_{min} or V_{max} in constant time using V and the length of V_{min}. By combining these structures with the encoding of Theorem 5.1(b), we can support the queries in Theorem 5.1(d) (namely, the queries RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A, NSV_A RMaxQ_A, RRMaxQ_A, RLMaxQ_A, RkMaxQ_A, PLV_A and NLV_A) in constant time. By Lemma 5.3, the total space of these structures is $4n - k + \lg \binom{n}{k} + o(n) \leq ((3 + \lg 3)n + o(n) < 4.585n + o(n)$ bits. This proves Theorem 5.1(d).

Note that if $k = 0$ (i.e, there are no consecutive equal elements), E' on A is same as E on A, and the size of V is $n + 2$ bits. Therefore we can support all the queries in Theorem 5.1(d) using $4n + o(n)$ bits with constant query time.

5.4.2 Encoding colored $2d$-Min and $2d$-Max heaps using less space

In this section, we give new encodings that prove Theorem 5.1(a) and Theorem 5.1(c), which use less space but take more query time than the previous encodings. To prove Theorem 5.1(a), we maintain the encoding E' on A, with the modification that instead of T' (which takes at most $2n - k$ bits), we store the bit string T (which takes at most $2(n - k)$ bits) which is obtained by constructing the encoding E on A'. Note that $f(s, c)$ is well-defined when s and c are substrings of $D_{\mathsf{CMin(A')}}$ and C, respectively. If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, then the total size of the encoding is at most $3(n-k) + \lg \binom{n}{k} + o(n) \leq n \lg 9 + o(n) < 3.17n + o(n)$ bits. If we can reconstruct the sequences $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$, by Lemma 5.5, we can support all the required queries. We now describe how to decode the entire $D_{\mathsf{CMin(A)}}$ using this encoding. (Decoding $D_{\mathsf{CMax(A)}}$ can be done analogously.)

Once we decode the sequence $D_{\mathsf{CMin(A')}}$, we reconstruct the sequence $D_{\mathsf{CMin(A)}}$

by scanning the sequences $D_{\mathsf{CMin(A')}}$ and C from left to right, and using the lookup table T_f. Note that $f(D_{\mathsf{CMin(A')}}, C)$ looses some open parentheses in $D_{\mathsf{CMin(A)}}$ whose matched close parentheses are not in $D_{\mathsf{CMin(A')}}$ but in $f(D_{\mathsf{CMin(A')}}, C)$. So when we add m consecutive close parentheses from the r-th position in $D_{\mathsf{CMin(A')}}$ in decoding with T_f, we add m more open parentheses before the position $pos = \mathsf{findopen}_{D_{\mathsf{Min(A')}}}(r-1)$. This proves Theorem 5.1(a).

To prove Theorem 5.1(c), we combine the concatenated sequence of V_{min} and V_{max} on A' whose total size is $n-k+2$ bits to the above encoding. Then we can reconstruct V_{min} on A by adding m extra 1's before $V_{min}[\mathsf{rank}_{((}(D_{\mathsf{Min(A')}}, pos)]$ when m consecutive close parentheses are added from the r-th position in $D_{\mathsf{CMin(A')}}$ while decoding with T_f. (Reconstructing V_{max} on A can be done in a similar way.) The space usage of this encoding is $4(n-k) + \lg \binom{n}{k} + o(n) \leq n \lg 17 + o(n) < 4.088n + o(n)$ bits. This proves Theorem 5.1(c).

5.5 Open problems

In this chpater, we obtained space-efficient encodings that support a large set of range and previous/next smaller/larger value queries.

We suggest the following open problems for future works.

- Can we support the queries in the Theorem 5.1(c) in $O(1)$ time using at most $4.088n + o(n)$ bits?
- As described in Section 5.2, Gawrychowski and Nicholson [30] show that any encoding that supports both RMinQ_A and RMaxQ_A requires at least $3n - \Theta(\lg n)$ bits. Can we obtain an improved lower bound in the case when we need to support the queries in Theorem 5.1(a)?
- Can we prove a lower bound that is strictly more than $3n$ bits for any encoding that supports the queries in Theorem 5.1(c)?

Chapter 6

Encoding Two-dimensional range Top-k queries

6.1 Introduction

Given a one-dimensional (1D) array $A[1 \dots n]$ from a total order and $1 \leq k \leq n$, the *Range Top-k query on A* ($\mathsf{Top}\text{-}k(i, j, A), 1 \leq i, j \leq n$) returns the positions of k largest values in $A[i \dots j]$. We can extend this query to the two-dimensional (2D) array case. Given a 2D array $A[1 \dots m][1 \dots n]$, from a total order and $1 \leq k \leq mn$, the *Top-k query on A* ($\mathsf{Top}\text{-}k(i, j, a, b, A), 1 \leq i, j \leq m, 1 \leq a, b \leq n$) returns the positions of k largest values in $A[i \dots j][a \dots b]$. Without loss of generality, we assume that all elements in A are distinct by ordering equal elements in the lexicographic order of their positions. Also, if the k positions of a $\mathsf{Top}\text{-}k$ query are reported in sorted order of the corresponding values, we refer to the query as *sorted* $\mathsf{Top}\text{-}k$ query; and refer to it as *unsorted* $\mathsf{Top}\text{-}k$ query, otherwise. For $1 \leq i, j \leq m$ and $1 \leq a, b \leq n$, we can also classify $\mathsf{Top}\text{-}k$ queries on 2D array by its range as follows.

- 1-sided query : The query range is $[1 \dots m][1 \dots b]$.

- 4-sided query : The query range is $[i \dots j][a \dots b]$.

We can also consider 2-sided and 3-sided queries which correspond to the ranges $[1 \dots j][1 \dots a]$ and $[1 \dots j][a \dots b]$ respectively. We consider how to support the Top-k queries in the encoding model.

In the rest of the chapter, we assume that for Top-k encodings, k is at most the size of the array (either 1D or 2D). Also, unless otherwise mentioned, we assume that all Top-k queries are sorted Top-k queries.

Previous Work. Encoding Top-k queries on 1D array has been widely studied in the recent years. For a 1D array $A[\dots n]$, Chan and Wilkinson [12] proposed a data structure that uses $\Theta(n)$ words and answers selection queries (i.e., selecting the k-th largest element) in $O(\lg k / \lg \lg n)$ time. Grossi et al. [39] considered the Top-k encoding problem, and obtained an $O(n \lg \kappa)$-bit encoding which can answer the Top-k queries for any $k \le \kappa$ in $O(\kappa)$ time or alternately, using $O(n \lg^2 \kappa)$ bits with $O(k)$ query time. (They also considered one-sided Top-k query, they proposed $n \lg k + O(n)$-bit encoding with $O(k)$ query time.) The space usage of this encoding was improved to $O(n \lg \kappa)$ bits, maintaining the $O(k)$ query time, by Navarro et al. [60]. Recently, Gawrychowski and Nicholson [31] proposed an $(k+1)nH_0(1/(k+1)) + o(n)$-bit[1] encoding for Top-k queries and showed that at least $(k+1)nH_0(1/(k+1))(1-o(1))$ bits are required to encode Top-k queries.

To the best of our knowledge, there are no results for range Top-k queries for 2D array with general k. For $k = 1$, the Top-k query is same as the *Range Maximum Query (RMaxQ)*, which has been well-studied for 1D as well as for 2D arrays. For a 2D $m \times n$ array, Brodal et al. [10] proposed an $O(nm \min(m, \lg n))$-bit encoding which answers RMaxQ queries in $O(1)$ time. Brodal et al. [8] improved the space bound to the optimal $O(nm \lg m)$ bits, although this encoding does not support the queries efficiently.

[1] $H_0(x) = x \lg(1/x) + (1-x) \lg(1/(1-x))$

Array size	Query range	Space	Query time
$m \times n$	one-sided	$n \lceil \lg T \rceil$ bits	-
$m \times n$	four-sided	$O(mn \lg n)$ bits	$O(k)$
$m \times n$	four-sided	$m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ bits	-

Table 6.1 The summary of our results for Top-k queries on $m \times n$ 2D array. $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$

Our results. For an $m \times n$ 2D array A, we first obtain an $n \lceil \lg T \rceil$-bit encoding for answering one-sided Top-k queries, where $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$. We then show that any encoding that supports Top-k queries on A must use at least $n \lg T$ bits.

Next, we observe that there exists an $O(mn \lg n)$-bit data structure which answers 4-sided Top-k queries on A in $O(k)$ time by combining the results of [11] and [10]. We then propose an $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$-bit encoding for 4-sided Top-k queries on A, by extending the Top-k encoding of Gawrychowski and Nicholson for 1D arrays [31]. The summary of our results are in Table 6.1.

This chapter organized as follows. Section 6.2 gives an lower and upper bound for encoding one-sided Top-k queries on 2D array. In Section 6.3 we propose two encodings for answering 4-sided Top-k queries on 2D array. Finally, in Section 6.4, we give some open problems.

6.2 Encoding one-sided range Top-k queries on 2D array

In this section, we consider the encoding of one-sided Top-k queries on a 2D array $A[1 \dots m][1 \dots n]$. We first introduce the encoding by simply extending the encoding of one-sided Top-k queries for 1D array proposed by Grossi et al. [39]. Next we propose an optimal encoding for one-sided Top-k queries on A.

For a 1D array $A'[1 \dots n]$, one can define another 1D array $X[1 \dots n]$ such

as $X[i] = i$ for $1 \leq i \leq k$ and for $k < i \leq n$, $X[i] = X[i']$ if there exist a position $i' < i$ such that $A'[i]$ is larger than $A'[i']$ which is the k-th largest value in $A'[1 \dots i-1]$, and $X[i] = k+1$ otherwise. One can answer the $\mathsf{Top}\text{-}k(1, i, A')$ by finding the rightmost occurrence of every element $1 \dots k$ in $X[1 \dots i]$. By representing X (along with some additional auxiliary structures) using $n \lg k + O(n)$ bits, Grossi et al. [39] obtained an encoding which supports 1-sided $\mathsf{Top}\text{-}k$ queries on A' in $O(k)$ time.

For a 2D array A, one can encode A to support one-sided $\mathsf{Top}\text{-}k$ queries by writing down the values of A in column-major order into a 1D array, and using the encoding described above – resulting in the following encoding.

Proposition 6.1. *A 2D array $A[1 \dots m][1 \dots n]$ can be encoded using $mn \lg k + O(n)$ bits to support one-sided $\mathsf{Top}\text{-}k$ queries in $O(k)$ time.*

Now we describe an optimal encoding of A which supports one-sided $\mathsf{Top}\text{-}k$ queries. For 1D array $A'[1 \dots n]$, we can define another 1D array $B'[1 \dots n]$ such that for $1 \leq i \leq n$, $B'[i] = l$ if $A'[i]$ is the l-th largest element in $A'[1 \dots i]$ with $l \leq k$, and $B'[i] = k+1$ otherwise. Then we answer the $\mathsf{Top}\text{-}k(1, i, A')$ query as follows. We first find the rightmost position $p_1 \leq i$ such that $B'[p_1] \leq k$. Then we find the positions $p_2 > p_3 \cdots > p_k$ such that for $2 \leq j \leq k$, p_j is the rightmost position in $A'[1 \dots p_{j-1} - 1]$ with $B'[p_j] \leq k - j + 1$. Finally, we return the positions $p_1, p_2, \dots, p_k$. Therefore by storing B' using $n \lceil \lg (k+1) \rceil$ bits, we can answer the one-sided $\mathsf{Top}\text{-}k$ queries on A'. Also we can sort $A'[p_1], \dots, A'[p_k]$ using the property that for $1 \leq b < a \leq k$, $A'[p_a] < A'[p_b]$ if and only if one of the following two conditions hold: (i) $B'[p_a] \geq B'[p_b]$, or (ii) $B'[p_a] < B'[p_b]$ and there exist $q = B'[p_b] - B'[p_a]$ positions $j_1, j_2, \dots, j_q$ such that $p_a < j_1 < \cdots < j_q < p_b$ and $B'[j_r] \leq B'[p_a]$ for $1 \leq r \leq q$.

We can extend this encoding for the one-sided $\mathsf{Top}\text{-}k$ queries on a 2D array A. For $1 \leq j \leq n$, we first define the elements of j-th column in A as $a_{1j} \dots a_{mj}$. Then we define the sequence $S_j = s_{1j} \dots s_{mj}$ such that for $1 \leq i \leq m$, $s_{ij} = l$

if a_{ij} is the l-th largest element in $A[1\dots m][1\dots j]$ with $l \leq k$ and $s_{ij} = k+1$ otherwise. Since there exist $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$ possible S_i sequences, we can store $S^A = S_1 \dots S_n$ using $n\lceil \lg T \rceil$ bits and we can answer the one-sided $\mathsf{Top}\text{-}k(1, m, 1, j)$ queries on A by the following procedure.

1. Find the rightmost column q, for some $q \leq j$, such that S_q has $\ell > 0$ elements $s_{p_1 q}, \dots, s_{p_\ell q}$ where $s_{p_1 q} < \dots < s_{p_\ell q} < k+1$. If $\ell > k$, we return the positions of $A[p_1][q] \dots A[p_k][q]$ as the answers of the query, and stop. If $\ell \leq k$, we return the positions of $A[p_1][q] \dots A[p_\ell][q]$.

2. Repeat Step 1 by setting k to $k - \ell$, and j to $q-1$.

We can return the positions in the sorted order of their corresponding values similar to the 1D array case. This encoding takes less space than the encoding in the Proposition 6.1 since $mn \lg k = n \lg \sum_{i=0}^{m} \binom{m}{i}(k-1)^i \geq n \lg T$. The following theorem shows that the space usage of this encoding is essentially optimal for answering one-sided $\mathsf{Top}\text{-}k$ queries on A.

Theorem 6.1. *Any encoding of a 2D array $A[1\dots m][1\dots n]$ that supports one-sided $\mathsf{Top}\text{-}k$ queries requires $n \lg T$ bits, where $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$.*

Proof. Suppose there are two distinct sequences $S^A = S_1 \dots S_i$ and $S^{A'} = S'_1 \dots S'_i$ which give one-sided $\mathsf{Top}\text{-}k$ encodings of 2D arrays A and A', respectively. For $1 \leq b \leq n$, if $S_b \neq S'_b$ then $\mathsf{Top}\text{-}k(1, m, 1, b, A) \neq \mathsf{Top}\text{-}k(1, m, 1, b, A')$ by the definition of S^A and $S^{A'}$. Since for an $m \times n$ array, there are T^n distinct sequences $S^{A_1} \dots S^{A_{T^n}}$, it is enough to prove that for $1 \leq q \leq T^n$, each $S^{A_q} = S^q_1 \dots S^q_n$ has an array A such that $S^A = S^{A_q}$.

Without loss of generality, suppose that all elements in A come from the set $L = \{1, \dots, mn\}$. Then we can reconstruct A from the rightmost column using S^{A_q} as follows. If $s^q_{jn} \leq k$, for $1 \leq j \leq m$, we assign the s^q_{jn}-th largest element in L to $A[j][n]$. After we assign all values in the rightmost column with $s^q_{jn} \leq k$, we discard all assigned values from L, move to $(n-1)$-th column

and repeat the procedure. After we assign all values in A whose corresponding values in S^{A_q} are smaller than $k+1$, we assign the remaining values in L to remaining positions in A_q which are not assigned yet. Thus for any $1 \le b \le n$, if S^q_b has $\ell > 0$ elements $s_{p_1 b}, \dots, s_{p_\ell b}$ where $s_{p_1 b} < \cdots < s_{p_\ell b} < k+1$, then the b-th column in A contains ℓ-largest elements in $A[1 \dots m][1 \dots b]$ by the above procedure. This shows that $S^A = S^{A_q}$. □

6.3 Encoding general range Top-k queries on 2D array

In this section, we give an encoding which supports general Top-k queries on 2D array. For an $m \times n$ 2D array, we first introduce an $O(mn \lg n)$-bit encoding which supports Top-k query in $O(k)$ time by using the RMaxQ encoding of Brodal et al. [8].

Proposition 6.2. *A 2D array $A[1 \dots m][1 \dots n]$ can be encoded using $O(mn \lg n)$ bits to support unsorted Top-$k(i, j, a, b, A)$ in $O(k)$ time for $1 \le a, b \le m$ and $1 \le i, j \le n$.*

Proof. We use a data structure similar to the one outlined in [11] (based on Frederikson's heap selection algorithm [29]) for answering unsorted Top-k queries in 1D array[2]. First encode A using $O(mn \lg n)$ bits to support RMaxQ (range maximum) queries in constant time for the any rectangular range in A. This encoding also supports finding the rank (i.e., the position in sorted order) of any element in A in $O(1)$ time [10]. Next, let $x = A[x_1][x_2]$ be the maximum value in $A[i \dots j][a \dots b]$, which can be found using an RMaxQ query on A. Then consider the 4-ary heap obtained by the following procedure. The root of the heap is x, and its four subtrees are formed by recursively constructing the 4-ary heap on the sub-arrays $A[i \dots x_1 - 1][a \dots b]$, $A[x_1 + 1 \dots j][a \dots b]$, $A[x_1][a \dots x_2 - 1]$ and $A[x_1][x_2 + 1 \dots b]$, respectively. Now, we can find the k largest elements in

[2]Brodal et al. [11] also give another structure to answer sorted Top-k queries, with the same time and space bounds.

the above 4-ary heap in $O(k)$ time using the algorithm proposed by Frederickson [29] (note that this algorithm only builds a heap with $O(k)$ nodes which is a connected subgraph of the above 4-ary heap). $\square$

We now introduce another encoding to support $\mathsf{Top}\text{-}k$ queries on an $m \times n$ 2D array A. This encoding extends the optimal $\mathsf{Top}\text{-}k$ encoding of Gawrychowski and Nicholson [31] for a 1D array. This encoding does not support the queries efficiently. Compared to the encoding of Proposition 6.2, this encoding uses less space when $n = \Omega(k^m)$. We first review the Gawrychowski and Nicholson [31]'s optimal $\mathsf{Top}\text{-}k$ encoding for 1D array, and show how to extend this encoding to the 2D array case.

For a given 1D array $A'[1 \dots n]$, we define the sequence of arrays $S^{A'} = S^{A'}_1 \dots S^{A'}_n$, where for $1 \le j \le n$ and $1 \le i \le j$, $S^{A'}_j$ is an array of size j defined as follows.

$$S^{A'}_j[i] = \begin{cases} p & \text{if there are } p\ (< k) \text{ elements larger than } A'[i] \text{ in } A'[i+1 \dots j] \\ k & \text{otherwise} \end{cases}$$

See Figure 6.1 for an example.

If $S^{A'}_j[i] < k$, we call $A[i]$ in $A[1 \dots j]$ as *active*, otherwise $A[i]$ is *inactive* in $A[1 \dots j]$.

Gawrychowski and Nicholson [31] show that for $1 \le i, j \le n$, $\mathsf{Top}\text{-}k(i, j, A')$ can be answered using $S^{A'}_j[i \dots j]$. They obtained a $\lg \binom{(k+1)n}{n} + o(n)$-bit encoding of $S^{A'}$ by representing $\delta^{A'}_1 \dots \delta^{A'}_{n-1}$ (where $\delta^{A'}_i = \sum_{l=1}^{i+1} S^{A'}_{i+1}[l] - \sum_{l=1}^{i} S^{A'}_i[l]$) in unary, and compressing the sequence using the Lemma 2.1. Since $\sum_{i-1}^{n-1} \delta^{A'}_i \le kn$, the unary sequence has kn zeros and n ones. The following lemma states their result for 1D arrays.

Lemma 6.1 ([31])**.** *Given an 1D array $A[1 \dots n]$, there is an encoding of A using $\lg \binom{(k+1)n}{n} + o(n)$ bits which supports $\mathsf{Top}\text{-}k$ queries.*

We now describe how to extend this encoding to a 2D $m \times n$ array A. For $1 \le i \le m$, let $A_i[1 \dots n]$ be the array of the i-th row in A. Then we first

A_1	3	7	8	2	6	4
A_2	6	4	10	3	5	2

$S_1^{A_1}$	0					
$S_2^{A_1}$	1	0				
$S_3^{A_1}$	2	1	0			
$S_4^{A_1}$	2	1	0	0		
$S_5^{A_1}$	2	2	0	1	0	
$S_6^{A_1}$	2	2	0	2	0	0

$S_1^{A_2}$	0					
$S_2^{A_2}$	0	0				
$S_3^{A_2}$	1	1	0			
$S_4^{A_2}$	1	1	0	0		
$S_5^{A_2}$	1	2	0	1	0	
$S_6^{A_2}$	1	2	0	1	0	0

$I_1^{(1,2)}$	1					
$I_2^{(1,2)}$	2	0				
$I_3^{(1,2)}$	2	1	1			
$I_4^{(1,2)}$	2	1	1	1		
$I_5^{(1,2)}$	2	1	1	2	0	
$I_6^{(1,2)}$	2	1	1	2	0	0

$I_1^{(2,1)}$	0					
$I_1^{(2,1)}$	1	0				
$I_1^{(2,1)}$	2	1	0			
$I_1^{(2,1)}$	2	1	0	0		
$I_1^{(2,1)}$	2	2	0	1	0	
$I_1^{(2,1)}$	2	2	0	2	0	0

Figure 6.1 Top-k encoding of the 2D array A when $k = 2$

maintain the Top-k encoding of $A_1 \dots A_m$ using Lemma 6.1, and this takes $m \lg \binom{(k+1)n}{n} + o(n)$ bits. In addition, for every $1 \le i \ne j \le m$, we define the sequence of arrays, $I^{(i,j)} = I_1^{(i,j)} \dots I_n^{(i,j)}$. For $1 \le r \le n$, $I_r^{(i,j)}$ is an array of size r defined as follows.

$$I_r^{(i,j)}[s] = \begin{cases} p & \text{if } i > j \text{ and there are } p\ (< k) \text{ elements which are} \\ & \text{larger than } A_i[s] \text{ in } A_j[s+1 \dots r] \\ q & \text{if } i < j \text{ and there are } q\ (< k) \text{ elements which are} \\ & \text{larger than } A_i[s] \text{ in } A_j[s \dots r] \\ k & \text{otherwise (if there are} \ge k \text{ elements, in the above two cases)} \end{cases}$$

See Figure 6.1 for an example.

We can answer the Top-$k(i, j, a, b, A)$ queries as follows. We first define the 1D array $B[1 \dots b(j-i+1)]$ by writing down the values of $A[i \dots j][1 \dots b]$ in column-major order. Then we observe that Top-$k(i, j, a, b, A)$ can be answered using $S^B_{b(j-i+1)}[a(j-i+1)+1 \dots b(j-i+1)]$.

The following lemma shows that we can compute the values in $S^B_{b(j-i+1)}$ using $S^{A_1} \dots S^{A_m}$ and all the arrays $I_b^{(c,d)}$, for $1 \le c \ne d \le m$.

Lemma 6.2. *Given a 2D array $A[1\dots m][1\dots n]$, for $1 \le i \le j \le m$ and $1 \le b \le n$, let $B[1\dots q]$ be the 1D array of size $q = (j-i+1)b$ obtained by writing the elements of $A[i\dots j][1\dots b]$ in column-major order. Also, for any $1 \le s \le q$, let (s_{row}, s_{col}) be the position corresponding $B[s]$ in A (which can be computed using $s_{col} = \lceil s/(j-i+1) \rceil$ and $s_{row} = s-(s_{col}-1)*(j-i+1)+(i-1)$). Then*

$$S_q^B[s] = \max(k, (S_b^{A_{s_{row}}}[s_{col}] + \sum_{i \le \ell \le j, \ell \neq s_{row}} I_b^{(s_{row},\ell)}[s_{col}])).$$

Proof. It is enough to count the number of elements in B (i.e., in $A[i\dots j][a\dots b]$) which are larger than $B[s]$ (i.e., $A[s_{row}][s_{col}]$) in $B[s+1\dots q]$ (i.e., the corresponding elements in A). Let L be the set of these elements. If $|L| \ge k$, then $S_q^B[s] = k$. In the following, we describe how to compute $S_q^B[s]$ when $|L| < k$.

From the definition of $S_b^{A_{s_{row}}}$, it follows that the number of elements in L which are in row s_{row} is $S_b^{A_{s_{row}}}[s_{col}]$.. Also, for any row $\ell \neq s_{row}$, $I_b^{(s_{row},\ell)}[s_{col}]$ is the number of elements in L that belong to row ℓ. From all these values, we can compute $|L|$. □

By Lemma 6.2, we can answer the Top-k queries on A using the Top-k encodings of all the rows $A_1, \dots, A_m$, together with all the arrays $I^{(i,j)}$, for all $1 \le i \neq j \le m$. Since we can recover the order of all active elements in the prefix of i-th row using S^{A_i} [31], we can decode $I_p^{(i,j)}$ using $I_{p-1}^{(i,j)}$ and $\gamma_p^{ij} = \sum_{l=1}^{p} I_p^{(i,j)}[l] - \sum_{l=1}^{p-1} I_{p-1}^{(i,j)}[l]$ by the following procedure, for $p > 1$.

1. Append 0 to $I_{p-1}^{(i,j)}$. Let this array be $J_{p-1}^{(i,j)}$.
2. Find the positions of $\gamma_{p-1}^{(i,j)}$ smallest active values in $A_i[1\dots p]$ using S^{A_i}, and increase the values of $J_{p-1}^{(i,j)}$ in these positions by 1.

Therefore, using $I_1^{(i,j)}$, and $\gamma_2^{(i,j)}, \dots, \gamma_n^{(i,j)}$, we can decode $I^{(i,j)}$. Since the sum $\sum_{\ell=2}^{\ell=n} \gamma_\ell^{(i,j)}$ is at most kn, we can encode all the arrays $I^{(i,j)}$ (for all possible $i \neq j$) using $m(m-1)\lg\binom{(k+1)n}{n} + o(n)$ bits (by converting $\gamma_\ell^{(i,j)}$'s into unary, as in the encoding of Lemma 6.1). Also, to encode $I_1^{(i,j)}$ for $i < j$ (note that if

$i > j$, $I_1^{(i,j)}$ is always 0), we need to store the ordering of all elements in the first column, which takes $m \lg m$ bits. This gives a proof of the following theorem.

Theorem 6.2. *Given a 2D array $A[1 \dots m][1 \dots n]$, there is an encoding of A using $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ bits which can answer the Top-k queries.*

6.4 Open problems

In this chapter, we obtained encodings which answer Top-k query on 2D array.

We suggest the following open problems for future works.

- Can we support efficient query time on our proposed encodings of Theorem 6.1 and Theorem 6.2?

- For 2 and 3-sided query, can we obtain an encoding which uses less space than the 4-sided Top-k queries on 2D array?

- Is the effective entropy of unsorted Top-k queries smaller than the effective entropy of sorted Top-k queries on 2D arrays?

Chapter 7

Conculsion

In this thesis, we proposed various space-efficient data structures that answer rank and select on bit strings, NLN queries, range queries and next/previous larger/smaller values simultaneously, and Range Top-k queries on two-dimensional array. Most of our data structures not only require less space than existing data structures, but also support queries efficiently .

In Chapter 3, we are aware, carefully investigated V2F compressors as a basis for bitvectors. We have shown how V2F bitvectors can lead to simple bitvectors with low redundancy. Empirical testing of an implementation, which albeit differs considerably from the theoretical proposals, shows that low memory usage and good, robust speed performance can be obtained via V2F compressors.

In Chapter 4, we proposed data structures for NLV in one-dimensional arrays in indexing model, and NLN in two-dimensional arrays, in the encoding models. For two-dimensional arrays we obtained a data structure that uses asymptotically optimal space and supports NLN queries in constant time.

In Chapter 5, we obtained space-efficient encodings that support a large set of range and previous/next smaller/larger value queries. The encodings that

support the queries in constant time take more space than the ones that do not support the queries in constant time.

Finally, in Chapter 6, we obtained encodings which answer $\mathsf{Top}\text{-}k$ query on two-dimensional array. In particular, for $m \times n$ two-dimensional array, we proposed an optimal encoding when the query is one-sided.

In addition to the open problems in each chapter, we give the following general open problems which can be considered for all problems in this thesis.

- In this thesis, all data structures give exact answers for queries. Different from exact queries, for $\epsilon > 0$, *ϵ-approximate query* returns the answer between T and ϵT where T is an exact answer of the query. For example, $(1+\epsilon)$-approximate $\mathsf{NLN}(i)$ on $A[1 \ldots n]$ returns the position j such that $A[j] > A[i]$ and $|j - i| \leq (1+\epsilon)|\mathsf{NLN}(i) - i|$. There are several studies about approximate queries such as nearest neighbor problem [2] and range minima in the middle [26]. Can data structures in this thesis use less space or query time for answering the approximate queries instead of exact queries?

- All data structures proposed in this thesis only works on a one or two-dimensional array. Extending these data structures for general n-dimensional array can be considered as an open problem. For example, Yuan and Atallah [75], and Davoodi et al. [15] obtained encodings which support RMinQ queries on n-dimensional arrays.

  More interesting open problem is that we can generalize an n-dimensional array into an *n-dimensional grid*. For example, we can consider the two-dimensional array as a special case of two-dimensional grid such that all grid points have an assigned value. Navarro et al. proposed data structures for various range queries on two-dimensional grid [57], but they did not considered the problems in terms of *density*, i.e, the ratio between the total number of grid points and the grid points which have an assigned

value. In many practical cases, the grid is *sparse*, that is, only few grid points have an assigned value. Can we design efficient data structures where input data is an n-dimensional grid and obtain better space or query time when the grid is sparse?

Bibliography

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97, 2010.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] T. Asano, S. Bereg, and D. G. Kirkpatrick. Finding nearest larger neighbors. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2009.

[4] T. Asano and D. G. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *WADS*, volume 8037 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2013.

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000, Proceedings*, pages 88–94, 2000.

[6] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[7] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 14(3):344–370, 1993.

[8] G. S. Brodal, A. Brodnik, and P. Davoodi. The encoding complexity of two dimensional range minimum data structures. In *ESA 2013, 2013. Proceedings*, pages 229–240, 2013.

[9] G. S. Brodal, P. Davoodi, M. Lewenstein, R. Raman, and S. S. Rao. Two dimensional range minimum queries and Fibonacci lattices. In *ESA*, pages 217–228, 2012.

[10] G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.

[11] G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *ISAAC 2009, Proceedings*, pages 173–182, 2009.

[12] T. M. Chan and B. T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013,2013*, pages 241–251, 2013.

[13] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996.

[14] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[15] P. Davoodi, J. Iacono, G. M. Landau, and M. Lewenstein. Range minimum query indexes in higher dimensions. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, pages 149–159, 2015.

[16] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *COCOON 2012, Proceedings*, pages 396–407, 2012.

[17] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *WEA*, volume 4007 of *LNCS*, pages 134–145. Springer, 2006.

[18] O. Delpratt, N. Rahman, and R. Raman. Compressed prefix sums. In *SOFSEM (1)*, volume 4362 of *LNCS*, pages 235–247. Springer, 2007.

[19] E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[20] M. Drmota, Y. A. Reznik, and W. Szpankowski. Tunstall code, khodak variations, and random walks. *IEEE Transactions on Information Theory*, 56(6):2928–2937, 2010.

[21] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

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

[23] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 269–278, 2001.

[24] P. Ferragina and G. Navarro. Pizza&chili corpus. `http://pizzachili.dcc.uchile.cl/texts.html`.

[25] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

[26] J. Fischer and V. Heun. Finding range minima in the middle: Approximations and applications. *Mathematics in Computer Science*, 3(1):17–30, 2010.

[27] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[28] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[29] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

[30] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range min-max and top-k. *CoRR*, abs/1411.6581, 2014.

[31] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range top-k k , selection, and min-max. In *ICALP 2015, Proceedings, Part I*, pages 593–604, 2015.

[32] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[33] B. Goethals and M. Zaki. Fimi: Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/`, 2004.

[34] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. *CoRR*, abs/1311.1249, 2013.

[35] M. J. Golin, J. Iacono, D. Krizanc, R. Raman, and S. S. Rao. Encoding 2d range maximum queries. In *ISAAC*, volume 7074 of *LNCS*, pages 180–189. Springer, 2011.

[36] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *ESA*, volume 4698 of *LNCS*, pages 371–382. Springer, 2007.

[37] A. Golynski, A. Orlandi, R. Raman, and S. S. Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.

[38] R. Gonzáalez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[39] R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. R. Satti. Encodings for range selection and top-k queries. In *Algorithms - ESA 2013*, volume 8125 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2013.

[40] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.

[41] M. He, J. I. Munro, and S. R. Satti. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms*, 8(4):42, 2012.

[42] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989.

[43] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

[44] V. Jayapaul, S. Jo, V. Raman, and S. R. Satti. Space efficient data structures for nearest larger neighbor. In *Combinatorial Algorithms - 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014*, pages 176–187, 2014.

[45] S. Jo, S. Joannou, D. Okanohara, R. Raman, and S. R. Satti. Compressed bit vectors based on variable-to-fixed encodings. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, page 409, 2014.

[46] S. Jo, R. Raman, and S. R. Satti. Compact encodings and indexes for the nearest larger neighbor problem. In *WALCOM 2015*, pages 53–64, 2015.

[47] S. Jo and S. R. Satti. Simultaneous encodings for range and next/previous larger/smaller value queries. In *Computing and Combinatorics - 21st International Conference, COCOON 2015, Beijing, China, August 4-6, 2015, Proceedings*, pages 648–660, 2015.

[48] T. Jurkiewicz and K. Mehlhorn. The cost of address translation. In P. Sanders and N. Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 148–162. SIAM, 2013.

[49] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 302–311, 2014.

[50] G. L. Khodak. Connection between redundancy and average delay of fixed-length coding. In *All-Union Conf. Problems of Theoretical Cybernetics*, 1969. (in Russian).

[51] D. K. Kim, J. C. Na, J. E. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. In *WEA*, volume 3503 of *LNCS*, pages 315–327. Springer, 2005.

[52] M. Lewenstein, J. I. Munro, and V. Raman. Succinct data structures for representing equivalence classes. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Proceedings*, pages 502–512, 2013.

[53] P. B. Miltersen. Cell probe complexity - a survey. *FSTTCS*, 1999.

[54] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[55] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[56] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[57] G. Navarro, Y. Nekrich, and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Theor. Comput. Sci.*, 482:60–72, 2013.

[58] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *SEA*, volume 7276 of *LNCS*, pages 295–306. Springer, 2012.

[59] G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *SEA*, volume 6630 of *LNCS*, pages 193–205. Springer, 2011.

[60] G. Navarro, R. Raman, and S. R. Satti. Asymptotically optimal encodings for range selection. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014,2014*, pages 291–301, 2014.

[61] D. Okanohara. Compressed rank select dictionary. `https://code.google.com/p/rsdic/`, 2012. [Online; accessed 1-March-2013].

[62] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*, 2007.

[63] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007.

[64] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[65] M. Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008.

[66] M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC*, pages 232–240. ACM, 2006.

[67] M. Patrascu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *SODA*, pages 117–122. SIAM, 2010.

[68] R. Raman. Encoding data structures. In *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, pages 1–7, 2015.

[69] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007.

[70] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 25pp, 2007.

[71] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[72] Y. Tabei. fmindex-plus-plus. `https://code.google.com/p/fmindex-plus-plus/`.

[73] B. P. Tunstall. *Synthesis of noiseless compression codes.* PhD thesis, Georgia Tech, 1967.

[74] S. Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *LNCS*, pages 154–168. Springer, 2008.

[75] H. Yuan and M. 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*, pages 150–160, 2010.

[76] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

요약

본 논문에서는 다양한 범위 질의 및 관련문제들을 해결하는 공간 효율적인 자료 구조들을 디자인 및 구현하였다. 본 논문에서 제안한 대부분의 자료 구조들은 정보 엔트로피에 가까운 적은 공간 만을 차지하면서도 효과적인 질의 시간을 지원한다. 세부적으로 본 논문에서는 다음 네 가지 문제 ((i) 압축된 비트 문자열 상에서 rank 와 select 질의를 지원하는 문제, (ii) nearest larger neighbor 질의 문제, (iii) 여러 범위 질의 및, next/previous larger/smaller value 질의 문제, (iv) 이차원 배열 상에서의 Top-k 질의 문제) 을 해결하는 공간 효율적인 자료 구조에 대해 연구하였다.

본 논문에서는 우선 압축된 비트 벡터를 실질적으로 구현하였다 [45]. 비트 벡터는 비트 문자열 상에서 rank 와 select 질의를 지원하는 자료 구조를 뜻한다. 본 논문에서는 이전까지 체계적으로 연구 되지 않았던 *V2F(variable-to-fixed)* 압축 알고리즘으로 압축되어진 비트 문자열 상에서 비트 벡터를 구현하는 방법에 대해 연구하였다. 본 논문은 이러한 접근 방식이 실제 상황에서 빠른 질의를 지원함과 동시에 적은 여분 공간 (압축 된 비트 문자열을 제외한 비트 벡터의 공간) 을 차지한다는 것을 보였다. 본 논문에서 제안한 비트 벡터는 다양한 방법으로 압축된 비트 문자열 상에서 rank 와 select 질의를 지원하는 효과적이면서도 실용적인 방안을 제공한다.

이어서 본 논문에서는 nearest larger neighbor 문제를 해결하는 공간 효율적인 자료 구조에 대해 연구하였다 [44, 46]. 전순서가 주어진 n개의 원소를 가지는 일차원 배열이 있을 때, nearest larger neighbor (NLN) 질의는 배열 상의 어느 한 위치가 질의로 주어졌을 때, 질의와 가장 가까운 곳에 위치하면서 질의보다 큰 값을 가진 배열 상의 원소의 위치를 반환한다. 배열상에 모든 원소들의 NLN 질의에 답하는 문제는 괄호 매칭 이나 계산 기하학 관련 문제들에 활용될 수 있기에 큰 주목을 받고 있다 [3, 4, 7]. 본 논문에서는 이러한 NLN 질의를 빠른 시간 안에 풀 수 있는 공간 효율적인 자료 구조들에 대해 연구하였다. 우선 본

논문은 인덱싱 모델 하에서 일차원 배열에서 질의 시간과 사용 공간 사이에 tradeoff 를 가지는 자료 구조를 제안하였으며 인코딩 모델 하에서 이차원 배열에서 최적에 가까운 공간을 사용하면서 상수 시간 안에 NLN 질의에 답할 수 있는 자료 구조를 제안하였다.

또한 본 논문에서는 다양한 범위 질의들(범위 최소 질의, 범위 최대 질의 및 이들에 대한 확장 질의) 과 함께 next/previous larger/smaller value 질의를 답할 수 있는 공간 효율적인 자료 구조에 대해 연구하였다 [47]. 전순서가 주어진 n개의 원소를 가지는 일차원 배열이 있을 때, 본 논문에서는 $4.088n + o(n)$ 비트의 공간을 사용하면서 위에 주어진 모든 질의들에 답할 수 있는 자료 구조를 제안하였다. 또한 본 논문에서는 $4.585n+o(n)$ 비트의 공간을 사용하면서 위에 주어진 모든 질의들을 상수 시간 안에 답할 수 있는 자료 구조를 제안하였다. 본 논문이 제안한 자료 구조는 기존의 Fischer 에 의해 연구 된 $5.08n+o(n)$ 비트의 공간을 사용하는 자료구조에 비해 적은 공간을 차지한다 [25]. 본 논문에서는 우선 색칠 된 $2d$-Min heap 과 색칠 된 $2d$-Max heap 를 인코딩 하기 위해 기존의 DFUDS [6] 인코딩 기법을 확장한 다음, Gawrychowski 와 Nicholson 이 $2d$-Min heap 과 $2d$-Max heap 을 동시에 인코딩 하기 위해 제안한 자료 구조를 수정하여 색칠 된 $2d$-Min heap 과 색칠 된 $2d$-Max heap 을 동시에 인코딩 할 수 있음을 보였다. 본 논문은 또한 위의 질의들 중 일부를 지원하면서 $4.088n + o(n)$ 비트보다 더 적은 공간을 사용하는 자료 구조를 제안하였다.

마지막으로 본 논문에서는 전순서가 주어진 이차원 배열 상에서 Top-k 질의에 답할 수 있는 다양한 자료 구조들에 대해 연구하였다. $m\times n$ 이차원 배열에서 본 논문은 질의 범위가 $[1\dots m][1\dots a]$ $(1 \le a \le n)$ 로 제한 되었을 때 최적의 공간을 사용하는 자료 구조를 제안하였다. 또한 본 논문은 Gawrychowski 와 Nicholson 이 제안한 일차원 배열상에서 Top-k 질의를 지원하는 자료 구조를 확장하여 $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ 비트의 공간을 사용하면서 일반적인 Top-k 질의를 지원하는 자료 구조를 제안하였다.

주요어: 공간 효율적인 자료구조, 간결한 자료구조, 인코딩 모델, 인덱싱 모델, 비트벡터, rank 질의, select 질의, nearest larger neighbor 문제, 범위 질의,

next/previous larger 질의, 범의 Top-k 질의

학번: 2011-30257

저작자표시-비영리-변경금지 **2.0** 대한민국

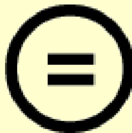
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 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

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

비트 문자열, 범위 질의 및 관련 문제들에 대한 공간 효율적인 인코딩

FEBRUARY 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Seungbum Jo

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

비트 문자열, 범위 질의 및 관련 문제들에 대한 공간 효율적인 인코딩

지도교수 Srinivasa Rao Satti

이 논문을 공학박사학위논문으로 제출함

2015 년 11 월

서울대학교 대학원

전기.컴퓨터 공학부

조 승 범

조 승 범의 박사학위논문을 인준함

2015 년 12 월

위 원 장	박 근 수	(인)
부위원장	Srinivasa Rao Satti	(인)
위 원	문 봉 기	(인)
위 원	강 유	(인)
위 원	이 인 복	(인)

Abstract

Space Efficient Encodings for Bit-strings, Range queries and Related Problems

Seungbum Jo
Department of Electrical Engineerning and Computer Science
Collage of Engineering
The Graduate School
Seoul National University

In this thesis, we design and implement various space efficient data structures. Most of these structures use spaces close to the information-theoretic lower bound while supporting the queries efficiently. In particular, this thesis is concerned with the data structures for four problems: (i) supporting rank and select queries on compressed bit strings, (ii) nearest larger neighbor problem, (iii) simultaneous encodings for range and next/previous larger/smaller value queries, and (iv) range Top-k queries on two-dimensional arrays.

We first consider practical implementations of *compressed* bitvectors, which support rank and select operations on a given bit-string, while storing the bit-string in compressed form [45]. Our approach relies on *variable-to-fixed* encodings of the bit-string, an approach that has not yet been considered systematically for practical encodings of bitvectors. We show that this approach leads to fast practical implementations with low *redundancy* (i.e., the space used by the bitvector in addition to the compressed representation of the bit-string), and is a flexible and promising solution to the problem of supporting rank and select on moderately compressible bit-strings, such as those encountered in real-world applications.

Next, we propose space-efficient data structures for the nearest larger neighbor problem [44, 46]. Given a sequence of n elements from a total order, and a position in the sequence, the nearest larger neighbor (NLN) query returns the position of the element which is closest to the query position, and is larger than the element at the query position. The problem of finding all nearest larger neighbors has attracted interest due to its applications for parenthesis matching and in computational geometry [3, 4, 7]. We consider a data structure version of this problem, which is to preprocess a given sequence of elements to construct a data structure that can answer NLN queries efficiently. For one-dimensional arrays, we give time-space tradeoffs for the problem on *indexing model*. For two-dimensional arrays, we give an optimal encoding with constant query on *encoding model*.

We also propose space-efficient encodings which support various range queries, and previous and next smaller/larger value queries [47]. Given a sequence of n elements from a total order, we obtain a $4.088n + o(n)$-bit encoding that supports all these queries where n is the length of input array. For the case when we need to support all these queries in constant time, we give an encoding that takes $4.585n + o(n)$ bits. This improves the $5.08n + o(n)$-bit encoding obtained by encoding the colored $2d$-Min and $2d$-Max heaps proposed by Fischer [25]. We extend the original DFUDS [6] encoding of the colored $2d$-Min and $2d$-Max heap that supports the queries in constant time. Then, we combine the extended DFUDS of $2d$-Min heap and $2d$-Max heap using the Min-Max encoding of Gawrychowski and Nicholson [30] with some modifications. We also obtain encodings that take lesser space and support a subset of these queries.

Finally, we consider the various encodings that support range $\mathsf{Top}\text{-}k$ queries on a two-dimensional array containing elements from a total order. For an $m \times n$ array, we first propose an optimal encoding for answering one-sided $\mathsf{Top}\text{-}k$ queries, whose query range is restricted to $[1 \dots m][1 \dots a]$, for $1 \leq a \leq n$. Next, we propose an encoding for the general $\mathsf{Top}\text{-}k$ queries that takes $m^2 \lg \binom{(k+1)n}{n} +$

$m \lg m + o(n)$ bits. This generalizes the Top-k encoding of Gawrychowski and Nicholson [30].

Keywords: Space-efficient data structure, succinct data structure, encoding model, indexing model, bitvector, rank query, select query, nearest larger neighbor problem, range queries, next/previous larger query, range Top-k query
Student Number: 2011-30257

Contents

Abstract i

Contents v

List of Figures ix

List of Tables xi

Chapter 1 Introduction 1

1.1 Computational model 2

1.1.1 Encoding and indexing models 2

1.2 Contribution of the thesis 3

1.3 Organization of the thesis 5

Chapter 2 Preliminaries 7

Chapter 3 Compressed bit vectors based on variable-to-fixed encodings 10

3.1 Introduction. 10

3.2 Bit-vectors using V2F coding 14

3.3 V2F compression algorithms for bit-strings 16

3.3.1 Tunstall code 16

3.3.2 Enumerative codes 19

3.3.3 LZW algorithm 23
3.3.4 Empirical evaluation of the compressors 23
3.4 Practical implementation of bitvectors based on V2F compression. 26
3.4.1 Testing Methodology 29
3.4.2 Results of Empirical Evaluation 33
3.5 Future works 35

Chapter 4 Space Efficient Data Structures for Nearest Larger Neighbor **39**
4.1 Introduction 39
4.2 Indexing NLV queries on 1D arrays 43
4.3 Encoding NLN queries on 2D binary arrays 44
4.4 Encoding NLN queries for general 2D arrays 50
4.4.1 2D NLN in the encoding model – distinct case 50
4.4.2 2D NLN in the encoding model – general case 53
4.5 Open problems 63

Chapter 5 Simultaneous encodings for range and next/previous larger/smaller value queries **64**
5.1 Introduction 64
5.2 Preliminaries 67
5.2.1 2d-Min heap 69
5.2.2 Encoding range min-max queries 72
5.3 Extended DFUDS for colored 2d-Min heap 75
5.4 Encoding colored 2d-Min and 2d-Max heaps 80
5.4.1 Combined data structure for $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$ . . . 82
5.4.2 Encoding colored 2d-Min and 2d-Max heaps using less space 88
5.5 Open problems 89

Chapter 6 Encoding Two-dimensional range Top-k queries 90

6.1 Introduction 90

6.2 Encoding one-sided range Top-k queries on 2D array 92

6.3 Encoding general range Top-k queries on 2D array 95

6.4 Open problems 99

Chapter 7 Conculsion 100

Bibliography 102

요약 112

List of Figures

Figure 3.1 An example of an (ad-hoc) enumerative code. The graph is given on the top (leaves shown shaded) and the code-words, and their phrases, right. 20

Figure 3.2 Memory test 36

Figure 3.3 rank_1 test 36

Figure 3.4 Random select_1 test 37

Figure 3.5 Hard select_1 test 37

Figure 3.6 Mixed test 38

Figure 4.1 Suppose the nearest block that contains a **1** from the $(2,2)$-block is the $(4,3)$-block. Then $d(4,3)$ contains the blocks $(2,5),(3,4),(4,3)$ and $(5,2)$, in that order. We can find the nearest **1** in NE(2,2) using $\mathsf{RMQ}(2,3)$ on $D_{(4,3)}$ and $\mathsf{RMQ}(1,3)$ on $D_{(4,4)}$. 49

Figure 4.2 The positions of *useful* and *dummy* elements in a 6×6 array. In this example, the dummy elements (X's) are in the range [1..24] and the useful elements (O's) are in the range [25..26]. 51

Figure 4.3 Pointers in $encoding_{2D}$ and $encoding_{grid}$ 59

Figure 5.1 Colored $2d$-Min heap of A 71

Figure 5.2 Encoding of $2d$-Min heap and $2d$-Max heap of A 72

Figure 5.3 $D_{\mathsf{CMin(A)}}$, $\mathsf{pre_rank}_{\mathsf{CMin(A)}}$, $V_{min}[i]$, $\mathsf{node}_{V_{min}}$, $\mathsf{pre_select}_{\mathsf{CMin(A)}}$ and $\mathsf{node_color}_{\mathsf{CMin(A)}}$ for colored $2d$-Min heap 77

Figure 5.4 Data structure combining the colored $2d$-Min heap and colored $2d$-Max heap of A. C is represented in uncompressed form. 83

Figure 6.1 Top-k encoding of the 2D array A when $k = 2$ 97

List of Tables

Table 1.1 The summary of previous results and our results. C = number of codewords, ℓ = codeword size, RQ_{min} = {RMinQ, RLMinQ, RRMinQ, RkMinQ} and RQ_{max} = {RMaxQ, RL-MaxQ, RRMaxQ, RkMaxQ}. 6

Table 3.1 Characteristics of the test files 24

Table 3.2 Compression ratios of the test files. 24

Table 3.3 Total phrase length of test files (as % of compressed output), excluding RLE codewords 29

Table 6.1 The summary of our results for Top-k queries on $m \times n$ 2D array. $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$ 92

Chapter 1

Introduction

The amount of data increases much faster than the capacity of the storage devices in recent days. Also the rapid growth of the mobile device market requires storing the large amount of the data into the limited space. To overcome this problem, one of the best solution is compressing the original data. For example, there are numerous text compression algorithms [73, 76] which can store the text data using much less space than its original size. However, the raw compressed data itself cannot support most of the queries (for example, random accesse and extracting arbitrary substrings from the LZ77-compressed [76] texts) without uncompressing the whole compressed data and this makes a huge bottleneck to answer the queries. In many areas of data science like *Real-time data analysis* or *BigData*, supporting queries efficiently for large data sets is a important issue. Therefore, storing data with compact size while supporting queries efficiently becomes a crucial issue.

One can define a *compressed (space-efficient) data structure* as a data structure which takes less space than the conventional data structure while supporting same set of queries. Let OPT be the minimum number of bits required to store the data while supporting the query (information-theoretic lower

bound). The compressed data structures are divided into three types as follows. (i) *Implicit* if the data structure uses $OPT + O(1)$ bits, (ii) *Succinct* if the data structure uses $OPT + o(OPT)$ bits and (iii) *Compact* if the data structure uses $O(OPT)$ bits. Since it is hard to design implicit data structure while supporting queries efficiently due to its space requirement, the main goal in the theoretical computer science area is usually maintain the size of the data structures as succinct while supporting the queries efficiently.

Succinct Data Structures were first introduced by Jacobson [42]. He showed how to represent the static trees and graphs while supporting the various queries efficiently. There are succinct data structures for various problems such as *Indexable dictionaries* [64, 69], *Permutations* [54], *Equivalence relations* [52] and *Range minimum queries* [10, 27].

1.1 Computational model

In this thesis, we assume a standard word-RAM model [53] as computational model. Word-RAM model is a variant of the classic RAM (random access machine) model [14] which is the computational model under the realistic assumption of a computer. In this model, each memory cell stores a *word* of size ω and we can read and write any cell in the memory in $O(1)$ time. Also, we can support 'C-style' arithmetic operations $(+, -, *, /, \%)$ and boolean operations $(\&, |, \wedge, \sim, <<, >>)$ on words in $O(1)$ time. Since each word needs to be large enough to store pointers and indices to access the data in practice, we set the word size $\omega = \Theta(\lg n)$ bits[1] for n input elements. We count space in terms of the number of bits used.

1.1.1 Encoding and indexing models

We consider the data structures in two different models that have been studied in the succinct data structures literature, namely the *indexing* and *encoding*

[1]We use $\lg n$ to denote $\log_2 n$

models. In both these models, the data structure is created after preprocessing the input data. In the indexing model, the queries can be answered by probing the data structure as well as the input data, whereas in the encoding model, the query algorithm cannot access the input data. The size of the data structure in the encoding model is also referred to as the *effective entropy* [35, 68] of the input data, with respect to the problem.

Suppose there is a set of input data S and set of all queries Q. If we can reconstruct any element in S from the answer to the queries in Q, encoding has no space advantage compared to indexing for answering the queries. But if the number of all possible answers on S induced by Q is significantly smaller than the size of S, then we can save some space by using the encoding model which doesn't need to store the original data.

1.2 Contribution of the thesis

In this thesis, we propose the following space-efficient data structures.

- **Bitvector based on variable-to-fixed encodings**: *Bitvector* is a data structure which supports rank and select operations on a given bit-string. In this thesis, we design *bitvectors based on variable-to-fixed compressed bit-string* (V2F bitvector). In the theoretical view, we show that regardless of the V2F compression algorithms, there exists a V2F bitvector which has low *redundancy* (that is, the difference in size between the V2F bitvector and the compressed bit-string is asymptotically smaller than the size of the compressed bit-string) and supports rank and select in constant time. We also give practical implementations of V2F bitvector and evaluate their practical performance with various existing implementations. The empirical evaluation shows that our V2F bitvector has low redundancy and supports rank and select queries efficiently compared to other previous implementations.

- **Encoding and indexing of the nearest Larger neighbor queries**: Given an elements in an array from a total order, *the nearest larger neighbor* (NLN) query returns the position of the element which is closest to the query position, and is larger than the element at the query position. We consider the NLN problem on one and two-dimensional arrays. For one-dimensional array of size n, we propose an $O((n/c)\lg c)$-bit index which supports NLN queries in $O(c)$ time, for any parameter $2 \leq c \leq n$, improving the structure of Fischer et al. [28]. For a $n \times n$ two-dimensional array, we first show that $\Theta(n^2)$ bits are necessary to encode NLN queries. Also, we give an optimal encoding which supports NLN queries in constant time on a two-dimensional array, improving the NLN encoding from Jayapaul et al. [44].

- **Simultaneous encodings of various range queries and next/previous larger/smaller value queries**: Given a sequence of n elements from a total order, we consider the encoding which supports range minimum query and its variants (RMinQ, RLMinQ, RRMinQ, RkMinQ), range maximum query and its variants (RMaxQ, RLMaxQ, RRMaxQ, RkMaxQ) and next/previous larger/smaller value queries (NLV, NSV, PLV, PSV). In this thesis, we obtain a $4.585n + o(n)$-bit encoding which supports all these queries in constant time for a sequence of size n. This improves the Fischer's $5.08n + o(n)$-bit encoding [25] which supports same set of queries in constant time. We also prove that if the query time is not concerned, we can obtain a $4.088n + o(n)$-bit encoding which supports all these queries.

- **Encoding of range Top-k queries on a two-dimensional array**: Given an elements in an array from a total order and a rectangular range in an array. *Range top-k* (Top-k) query returns the positions of k largest elements in the range, In this thesis, we consider various encodings which support Top-k queries on a two-dimensional array. This problem has not

been studied. For an $m \times n$ array, we first obtain an optimal encoding for one-sided $\mathsf{Top}\text{-}k$ queries whose query range is restricted to $[1 \dots m][1 \dots i]$, for $1 \leq i \leq n$. Also, we propose the $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$-bit encoding for $\mathsf{Top}\text{-}k$ queries on a two-dimensional array with any rectangular query ranges by extending the $\mathsf{Top}\text{-}k$ encoding on a one-dimensional array proposed by Gawrychowski and Nicholson [30]. In most of encodings do not support the $\mathsf{Top}\text{-}k$ queries in efficient query time.

The summary of previous results and our results for these data structures are in Table 1.1.

1.3 Organization of the thesis

The rest of this thesis is organized as follows. In Chapter 2, we introduce data structures for supporting rank and select queries which are basic operations on various space-efficient data structures. In Chapter 3, we describe compressed bit vectors based on variable-to-fixed encodings which have low redundancy in both theoretical and practical implementations. In Chapter 4 we consider the encoding and indexing data structures for Nearest Larger Neighbor (NLN) problem on one-dimensional and two-dimensional arrays. In Chapter 5, we propose encodings that support various range queries (range minimum, range maximum and their variants), and previous and next smaller/larger value queries. In Chapter 6, we propose the various encodings that supports $\mathsf{Top}\text{-}k$ queries. Finally in Chapter 7, we summarize the results in this thesis and give some open problems.

Input data	Query	Space Usage (bits)	Query time	Encoding/Indexing	Reference
Bit string $X[1 \dots n]$	rank_1, select_1	$C\ell + O(C \log(n/C))$	$O(1)$	Encoding	This thesis
1D array $A[1 \dots n]$, $1 \le c \le n$	PLV (NLN)	$O((n/c) \log c + (n \log n)/c^2)$	$O(c)$	Indexing	[44]
		$O((n/c) \lg c)$	$O(c)$	Indexing	This thesis
2D array $A[1 \dots n][1 \dots n]$	NLN	$O(n^2 \lg \lg n)$	$O(1)$	Encoding	[44]
		$O(n^2)$	$O(\lg \lg \lg n)$	Indexing	[44]
		$O(n^2)$	$O(1)$	encoding	This thesis
2D binary array $A[1 \dots n][1 \dots n]$, $1 \le c \le n^2$	NLN	$O(n^2/c)$	$O(c)$	Encoding	This thesis
1D array $A[1 \dots n]$	RMinQ, PSV	$2n + o(n)$	$O(1)$	Encoding	[27]
	RQ_{min}, PSV NSV	$2.54n + o(n)$			[25]
	RMinQ, RMaxQ	$3n + o(n)$			[30]
	RQ_{min}, RQ_{max} PSV, NSV PLV, NLV	$4.585n + o(n)$			This thesis
2D array $A[1 \dots m][1 \dots n]$	One-sided, sorted Top-k	$n \left\lceil \lg(\sum_{i=0}^{\min(m,k)} \binom{m}{i} (\frac{k!}{(k-i)!})) \right\rceil$		Encoding	This thesis
	Four-sided, unsorted Top-k	$O(mn \lg n)$	$O(k)$		
	Four-sided, sorted Top-k	$m^2 \lg \binom{(k+1)n}{n} +$ $m \lg m + o(n)$			

Table 1.1 The summary of previous results and our results. C = number of codewords, ℓ = codeword size, RQ_{min} = {RMinQ, RLMinQ, RRMinQ, RkMinQ} and RQ_{max} = {RMaxQ, RLMaxQ, RRMaxQ, RkMaxQ}.

Chapter 2

Preliminaries

In this chapter, we introduce data structures for answering rank and select queries, one of the fundamental problems in succinct data structures. These structures are used for the various space-efficient data structures proposed in this thesis.

Given a string $S[1 \dots n]$ over an alphabet Σ, rank and select are defined as follows.

- $\mathsf{rank}_\alpha(S, i)$: The number of occurrences of α in the first i positions of S, for any $\alpha \in \Sigma$.
- $\mathsf{select}_\alpha(S, i)$: The position of the i-th α in S, for any $\alpha \in \Sigma$.

In the thesis, we only consider the case when S is a *bit-string*, i.e, $\Sigma = \{0, 1\}$. We first introduce the following lemma from [69] that gives a succinct encoding of S.

Lemma 2.1 ([69])**.** *Let S be a string of length n containing m 1s. One can encode S using $\lg \binom{n}{m} + o(n)$ bits to support both $\mathsf{rank}_x(S, i)$ and $\mathsf{select}_x(S, i)$ in constant time, for $x \in \Sigma$. Also, one can decode any $\lg n$ consecutive bits in S in $O(1)$ time.*

Also, we use following lemmas from [37] that can be used to support rank and select operations on moderately dense bit strings (i.e., bit strings in which the number of zeros and ones is at most a poly-log factor smaller than the length of the string).

Lemma 2.2 ([37])**.** *Let S be a bit-string of length n containing m 1s. If $m \geq n/(\lg n)^c$, for some constant $c > 0$, one can support rank_1 and select_1 in $O(1)$ time using $\lg \binom{n}{m} + O(m)$ bits.*

Lemma 2.3 ([37])**.** *Given integer $n > m > 0$ such that $\min\{n - m, m\} \geq n/(\lg n)^c$ for some constant c, one can store a bit-string S with $n_0 \leq n - m$ 0s and $n_1 \leq m$ 1s, using $\lg \binom{n}{n-m} + O(\min\{n, n-m\})$ bits, such that select_0 and select_1 are supported in $O(1)$ time.*

Now we introduce another lemma from [62]. This lemma shows that if the number of ones is significantly less than the number of zeros, one can encode S using less space than the encoding described in Lemma 2.1 (but do not support queries in constant time).

Lemma 2.4 ([62])**.** *Let S be a bit-string of length n containing m 1s. One can encode S using $O(m \lg(n/m))$ bits such that rank_1 and select_1 can be supported in $O(n/m)$ time.*

One can generalize the rank and select queries as follows. Given a string $S[1 \ldots n]$ and pattern string p over the alphabet Σ, $\mathsf{rank}_p(S, i)$ returns the number of occurrences of pattern p in the first i positions of S, and $\mathsf{select}_p(S, i)$ returns the position of the i-th occurrence of pattern p in S. Combining the results from [56] and [69], one can show the following lemma for generalized rank and select queries on a bit-string.

Lemma 2.5 ([56], [69])**.** *Let S be a bit-string of length n over the containing m 1s. One can encode S using $\lg \binom{n}{m} + o(n)$ bits to support both $\mathsf{rank}_p(S, i)$ and*

$\mathsf{select}_p(S, i)$ *in constant time, for any binary pattern* p *with length* $|p| \leq 1/2 \lg n$. *Also, one can decode any* $\lg n$ *consecutive bits in* S, *in constant time.*

Chapter 3

Compressed bit vectors based on variable-to-fixed encodings

3.1 Introduction.

A *bitvector* is a fundamental building block of many space-efficient data structures. As described in Chapter 2, given a bit-string X of *length* n with *weight* m (i.e., with m **1** bits), the aim is to pre-process X to support the following operations, for any $b \in \{0, 1\}$:

- $\mathsf{rank}_b(X, i)$ returns the number of occurrences of b in the first i positions of X.
- $\mathsf{select}_b(X, i)$ returns the position of the ith b in X.

These operations can be supported in $O(1)$ time using $n + o(n)$ bits of space [13]. If X is a (uniformly) random bit-string, it cannot be be compressed, and this space bound is therefore, in the worst case, optimal to within lower-order terms. However, bit-strings encountered in practical applications are often compressible, and many algorithmic applications use bitvectors on bit-strings that are constructed to be *sparse*—contain $m = o(n)$ **1**s—and such bit-strings are

compressible to $o(n)$ bits. Starting from the work of [64, 70], there is now a rich theory of *compressed* bitvectors, which aim to use space approaching that used by a compressed representation of the bit-string, for many different measures of compressibility[1]. The most basic measures of compressibility are *density-sensitive*, i.e. they depend only upon the length n and weight m of the bit-string. These are the *information-theoretic minimum*, $B(n, m) \stackrel{\text{def}}{=} \lceil \log \binom{n}{m} \rceil$ bits, and the *zeroth-order empirical entropy*, $H_0(X) \stackrel{\text{def}}{=} -\sum_{i=0}^{1} p_i \lg p_i$, where $p_1 = m/n$ and $p_0 = 1 - p_1$; the compressed bit-string size should then be $nH_0(X) + O(1)$ bits. Note that if $m = o(n)$ then $B(n, m) \approx nH_0(X) = o(n)$.

Instance-sensitive measures[2], where the compressibility of the string X is a function of X, are more diverse, and include the *k-th order empirical entropy* H_k and functions of the gaps between successive **1**s [40], or the size of the output produced by a grammar-based compressor to X. In general, such measures would show that a bit-string X is at least as compressible as a density-sensitive measure on X.

Previous Work Although there have been many papers on implementations of bitvectors [18, 17, 36, 38, 51, 74] (and some researchers have implemented bitvectors as part of more complex data structures), there are fewer papers on compressed bitvectors for sparse bit-strings. It should be noted that supporting $O(1)$-time rank/select operations using reasonable space is possible only when $m = n/(\log n)^{O(1)}$ [66]. In this range, even the density-sensitive measure gives $O(m \log(n/m)) = O(m \log \log n)$ bits, so a compressed bitvector is significantly smaller than either an uncompressed bitvector, which takes $\Theta(n)$ bits, or viewing X as the characteristic vector of a set and storing the set explicitly, which requires $O(m \log n)$ bits. Such moderately sparse bit-strings are also of great

[1] As is common in the area of succinct and compressed data structures, we focus on *empirical* measures, i.e., those that are a function of the bitstring X itself, rather than measures derived by postulating a probabilistic model for generating bit-strings.

[2] A related term, *data-aware*, is used in [40].

practical interest. One focus of this chapter is on representing such bit-strings.

The following authors have considered practical data structures for sparse bit-strings. Geary et al. [32] considered "uniformly" sparse bit-strings, but their techniques do not apply to general sparse bit-strings, and they do not perform a stand-alone evaluation of their bitvector. Gupta et al. [40] considered very sparse bit-strings, and showed that instance-sensitive measures related to the γ and δ codes outperform density-sensitive ones, but they did not report on moderately sparse bit-strings. Delpratt et al. [18] considered Golomb coding in the context of the select_1 operation. Okanohara and Sadakane [63] performed arguably the first comprehensive evaluation, but focused mostly on the density-sensitive measures. Navarro et al. [59] considered rank and select on grammar-compressed bit-strings, but do not provide a stand-alone evaluation. Navarro and Providel [58] also provide an implementation of compressed bitvectors. This, again, targets the density-sensitive measures. Very recently, Kärkkäinen et al. [49] presented a hybrid approach combining run-length encoding (RLE), raw encoding and explicit encoding, and showed good performance on a class of bit-strings obtained from text indexing applications.

Our results In this chapter we explore the use of *variable-to-fixed (V2F)* encodings of a bit-string, which have only been partially explored previously. Our results show that this approach leads to very compact and high-performance compressed bitvectors. Indeed, we give a theoretical basis for the low *redundancy* (wasted space) of the codes as well as that of the bitvector. An ℓ-bit V2F code partitions the input bit-string into a concatenation of variable-length *phrases*. Each phrase, except the last one, is constrained to belong to a given dictionary D of $\leq 2^\ell$ bit-strings; the last phrase is a non-null prefix of a dictionary entry. Once the input bit-string is parsed, each phrase is replaced by its position in the dictionary, stored as a ℓ-bit *codeword*. V2F codes are studied in the data compression literature due to their desirable properties such as

error-resilience, but it appears that there has not yet been a comprehensive investigation of V2F bitvectors. That said, the class of V2F codes is quite broad: it includes e.g. RLE and grammar-based compression, and it is possible that there are application-specific implementations of V2F bitvectors inside other data structures.

Our main conceptual contributions are as follows:

- We argue that in general, V2F coding is an effective approach to reduce the *redundancy* of the bitvector, or the difference between the compressed size of the bit-string and the size of the bit-vector data structure. The redundancy can dominate the space usage of compressed bitvectors: e.g. if $m = O(n/(\log n)^2)$, the space usage of the compressed bitvector of [70], which is $B(n, m) + O(n \log\log n/\log n)$ bits, is dominated by the redundancy. We show that for the density range of interest, V2F compressors give redundancy that is asymptotically smaller than the compressed size of the bit-string.

- In practice, we give an approach for density-sensitive encoding of a bitvector that has a significantly lower (intrinsic) redundancy over that of Navarro and Providel [58] by using *Tunstall* codes [73]. Furthermore, we show that the Tunstall code always achieves H_0 empirical entropy with low redundancy (previously this was known only for random inputs).

- We give a new class of *enumerative* V2F codes. These codes generalize both Khodak's code [50, 20], a close relative of the Tunstall code, and RLE. Finally, a hybrid enumerative code which combines Khodak's code with RLE achieves excellent compression performance, even on bit-strings that are relatively incompressible by density-sensitive measures.

- We argue, as does Vigna [74], that practical implementations of select based on the method of "sampling" must address the issue of *long gaps*,

which many implementations do not do. This is because in practice, guarding against a worst-case scenario for long gaps (using ideas which derive back to [13]) consumes a lot of space. Although it seems real-life bit-strings *can* have a number of reasonably long gaps, we note that the typical test (select a random **1**) is likely to give running times that are independent of the distribution of the underlying bit-vector. We propose a test that would "fairly" and "naturally" test the handling of a select implementation in the presence of long gaps, and show that implementations that do not guard against long gaps do indeed slow down.

Our implementation has been structured into two independent parts: a framework for rank and select, and a compressor-specific part that deals with individual codewords. This highlights the challenges faced by a V2F-based bitvector, and offers a lot of room for innovation with respect to how to deal with codewords. The fact that there is indeed room has already been hinted at in [58, 59], but we argue that reasonable performance is obtained by a default implementation in many cases.

The rest of this chapter is structured as follows. Section 3.2 describes a general result on supporting rank and select operations on bit-strings compressed using V2F schemes. In Section 3.3, we describe the V2F schemes that we use in the experimental evaluation. Section 3.4 describes the details of our implementation, and also the results from the experimental evaluation of V2F schemes. Section 5.5 contains some future directions.

3.2 Bit-vectors using V2F coding

As indicated earlier, the redundancy of a compressed bitvector targeting a particular compressibility measure is the difference between the size of the bit string under that compressibility measure and the size of the bitvector. Pătraşcu [65] showed that rank/select can be supported in $O(1)$ time using

$B(n,m)+n/(\log n)^{O(1)}$ bits, and that for $m=\Theta(n)$, this is optimal [67]. However, there is no evidence yet that the approach of [65] is feasible in practice. Another approach to low-redundancy compressed bitvectors achieves $B(n,m)+O(m(\log\log n)^2/(\log n))$ bits and $O(1)$ time for the range $m=n/(\log n)^{O(1)}$ [36, 64]. While the redundancy is not as low as Pătraşcu's, it is roughly a log factor less than the compressed bit-string – a very desirable feature. We now show that this holds in general for V2F codes under modest assumptions:

Theorem 3.1. *Given a bit-string X of n bits encoded as C codewords using a V2F code of ℓ bits each. Further assume that there is a data structure, which given a codeword c, supports* rank *and* select *in $O(1)$ time on the phrase $p(c)$ that the codeword c stands for. Then we can support $\mathsf{select}_1(X,i)$ and $\mathsf{rank}_1(X,i)$ in $O(1)$ time using $C\ell+O(C\log(n/C))$ bits, provided that $C=n/(\log n)^{O(1)}$.*

Proof. For any bit-string s, let $w(s)$ denote the weight of s, and for $i=1,\ldots,C$, let c_i denote the i-th codeword, and let $m=w(X)$. The data structure consists of two bitvectors on the following bit-strings:

- the *ones distribution* bit-string $OD=\mathbf{0}^{w(p(c_1))}\mathbf{1}\mathbf{0}^{w(p(c_2))}\mathbf{1}\ldots\mathbf{0}^{w(p(c_C))}\mathbf{1}$.
- the *phrase size* bit-string $PS=\mathbf{1}\mathbf{0}^{|p(c_1)|-1}\mathbf{1}\mathbf{0}^{|p(c_2)|-1}\mathbf{1}\ldots\mathbf{1}\mathbf{0}^{|p(c_C)|-1}$.

It is easy to see that $|OD|=m+C$, $w(OD)=C$, $|PS|=n$ and $w(PS)=C$.

- To compute $\mathsf{select}_1(X,i)$, we first determine the number of codewords before the codeword in which the selected $\mathbf{1}$ lies as $j=\mathsf{rank}_1(OD,\mathsf{select}_0(OD,i))$. We then determine the total number of $\mathbf{1}$s in $c_1,\ldots,c_j$ as $k=\mathsf{select}_1(OD,j)-j$, and the start position of c_{j+1} in X as $d=\mathsf{select}_1(PS,j+1)-1$. Finally, we select the $i-k$-th $\mathbf{1}$ in $p(c_{j+1})$, add d to the answer and return.

- To compute $\mathsf{rank}_1(X,i)$, we first find the codeword j in which the i-th position lies by $j=\mathsf{rank}_1(PS,i)$. We then determine d, the start position of c_j, and k, the number of $\mathbf{1}$s in $c_1,\ldots,c_{j-1}$, as before, and return $k+\mathsf{rank}_1(p(c_j),i-d)$.

We store OD using Lemma 2.3, which uses $O(C\log(m/C))$ bits. In addition, we pad OD to length n by adding zeros at the end (so that the condition in Lemma 2.2 applies), and store the resulting bit-string as well as PS using Lemma 2.2, which takes $O(C\log(n/C))$ bits. □

Remark 1. *We will typically choose* $\ell = \Theta(\log n)$ *bits. Thus, provided that* $\log(n/C) = o(\log n)$*, the redundancy will be smaller than the size of the compressed output, which is* $C\ell$ *bits.*

3.3 V2F compression algorithms for bit-strings

We now describe different V2F compression schemes that we use to compress the given bit-string X. Each of these schemes partitions X into a sequence of variable-length *phrases*. Each phrase, except the last one, belongs to a *dictionary* of size $M = 2^\ell$ that is constructed from the source string. The dictionary entries are also referred to as *code words*. The compressed representation of X simply consists of a sequence of ℓ-bit codes (from the dictionary) corresponding to each phrase. The only difference between various compression algorithms is the way in which they construct the dictionary.

3.3.1 Tunstall code

For a given phrase length L, the Tunstall code is designed to maximize $E[L]$, the expected number of source letters per phrase for a memoryless source [73]. Given an input bit-string X, the dictionary constructed by Tunstall's algorithm can be represented as a full binary tree T (i.e., every node has 0 or 2 children), which we refer to as the *Tunstall tree*. Each edge in T corresponds to a bit, and each phrase corresponds to a leaf in T. The phrase corresponding to a leaf u can be obtained by concatenating the symbols corresponding to the edges on the root-to-leaf path to u.

We now describe the algorithm to construct a Tunstall code for X with

$M = 2^\ell$ codewords. First, we define some terminology. Letting $n = |X|$, and m be the weight of X, define $p_0 = 1 - m/n$ and $p_1 = m/n$. The probability[3] of a bit-string $b_1 b_2 \ldots b_l$ is defined to be $\prod_{i=1}^{l} p_{b_i}$. Each leaf in T is labelled by the probability of the corresponding phrase, and each internal node is labelled by the sum of the probabilities of its children. The algorithm is as follows:

(1) Start with 2-level rooted tree with the root connected to two leaves, corresponding to **0** and **1**.

(2) Pick a leaf node which has the highest probability and grow two leaves on it.

(3) Repeat step (2) while the number of leaves in the tree is at most M.

It has long been known that the Tunstall code achieves zeroth-order entropy (defined appropriately) for random sources [73] and its redundancy[4] for random sources has been shown to be low [20]. We now show that the redundancy of the Tunstall code with respect to empirical entropy is also low.

Theorem 3.2. *Given a bit-string X with length n and weight m, suppose that it is encoded using a Tunstall code with $M = 2^\ell$ codewords, constructed taking $p_0 = 1-m/n$ and $p_1 = m/n$ as the probabilities of **0** and **1** respectively. Assume, without loss of generality, that $p_1 \leq p_0$ and further assume that $\ell = \Theta(\log n)$ and $\log(1/p_1) = o(\log n)$. Then $C\ell \leq nH_0(X) + O(nH_0(X)\log(1/p_1)/\ell)$.*

Proof. Say that a *final* leaf refers to a leaf of the Tunstall tree T at the end of the algorithm. Observe that the probabilities of the leaves of T at any stage of the algorithm add up to 1. Hence, while the number of leaves is less than M, there will always be a leaf with probability greater than $1/M$, so we will never expand

[1]This is not a probability in the true sense, of course, since we are dealing with a given fixed bit-string X.

[2]Here the term redundancy has been overloaded to refer to the size of the compressed output relative to the ideal compressed size.

a leaf with probability at most $1/M$. It follows that the minimum probability of a final leaf is greater than p_1/M. Let p^* be the maximum probability of any final leaf. Since all final leaves are created by expanding leaves with probability $\geq p^*$, and at least one final leaf must have probability $\leq 1/M$, it follows that $p^* p_1 \leq 1/M$ or $p^* \leq 1/(p_1 M)$.

Suppose that the output of parsing X according to the Tunstall code comprises C codewords $c_1, c_2, \ldots, c_C$. Let $\Pr(c_i)$ denote the probability of the phrase of c_i. Then $-\log \prod_{i=1}^{C} \Pr(c_i) = -\log(p_0^{n-m} p_1^m) = nH_0(X)$. However, $\prod_{i=1}^{C} \Pr(c_i) \leq (1/(p_1 M))^C$ from the above, which gives $nH_0(X) \geq C \log(p_1 M)$, or:

$$nH_0(X) + C\log(1/p_1) \geq C\ell \tag{3.1}$$

With the above assumption on p_1, it is not hard to verify that $C\ell = O(nH_0(X))$, and plugging this back into Equation (3.1) we get that $C\ell \leq nH_0(X) + O(nH_0(X)\log(1/p_1)/\ell)$. □

Remark 2. 1. *Since we assume* $\log(1/p_1) = o(\ell)$*, the redundancy is a lower-order term.*

2. *Note that a similar argument shows that* $C\ell \geq nH_0(X) - C\log(1/p_1)$*. In other words, the output of Tunstall coding is never much less than the empirical entropy.*

Theorems 3.1 and 3.2 allow us to obtain a small improvement in redundancy over the bitvector of [36, Thm 2], which previously had the lowest known redundancy of any bitvector that does not use the (fairly complex) technique of *informative encoding* [36] or its successors [65].

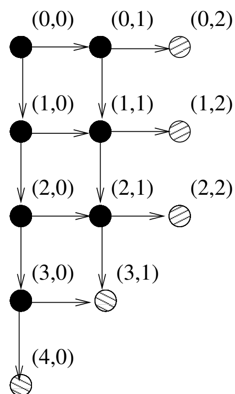
Corollary 3.1. *Let* X *be a bit-string with length* n *and weight* m*. There is a bit-vector that supports* rank_1 *and* select_1 *in* $O(1)$ *time when* $m = n/(\log n)^{O(1)}$ *and uses* $nH_0(X) + O(m\frac{\log(n/m)\log\log n}{\log n})$ *bits.*

Proof. Since $H_0(X) = O((m/n)\log(n/m))$, from Theorem 3.2 the output of the Tunstall coding occupies $nH_0(X) + O(m(\log(n/m))^2/\log n)$ bits. To augment it with rank_1 and select_1, we use Theorem 3.1. The additional data structures use $O(C\log(n/C)) = O\left(\frac{nH_0(X)}{\log n}\log\left(\frac{n\log n}{nH_0(X)}\right)\right)$ bits. Simplifying, we get that the redundancy of the bitvector is $O\left(m\frac{\log(n/m)}{\log n}(\log(n/m) + \log\log n)\right) = O(m\frac{\log(n/m)\log\log n}{\log n})$ bits.

Finally, it only remains to explain how to do $\mathsf{rank}/\mathsf{select}$ on an individual phrase in $O(1)$ time. Taking the notation of Theorem 3.1, we create the concatenated bit-string $p(0)p(1)\ldots p(2^\ell - 1)$. The maximum length L of an individual phrase must satisfy $(p_0)^L \geq p_1/M$, from which one can obtain that $L = O(n\log n/m)$. Since $n/m = O(\log n)^{O(1)}$, if we choose $\ell = (\log n)/2$, the bit-string containing the concatenated phrases will be of size $O(n^{1/2+\epsilon})$, for any positive constant $\epsilon < 1/2$. By building a bit-vector on this bit-string and furthermore explicitly storing the start of each phrase, as well as the cumulative numbers of **1**s in this bit-string (using $O(2^\ell \log n) = O(n^{1/2+\epsilon})$ bits), rank and select on individual phrases can be supported in $O(1)$ time. □

3.3.2 Enumerative codes

We define a class of *enumerative* codes as follows. An enumerative code can be specified as a (directed) graph on a subset of the vertices (i, j), for $i \geq 0$ and $j \geq 0$. A vertex (i, j) may either have no outgoing edges (be a *leaf*) or point to *both* vertices $(i + 1, j)$ and $(i, j + 1)$. Furthermore, a vertex (i, j) is *complete* if either it has indegree 2, or either i or j is 0 (and its indegree is 1); and *incomplete* otherwise. All incomplete vertices must be leaves. Finally, the vertex $(0, 0)$ is always in the graph. Given such a graph, the code is specified as follows. For every complete leaf (i, j) we allocate $\binom{i+j}{j}$ codewords, which code for all phrases with i **0**s and j **1**s. For every incomplete leaf (i, j), if its (sole) predecessor is $(i, j - 1)$ then we allocate all $\binom{i+j-1}{j-1}$ codewords, which code for all phrases with i **0**s and j **1**s that end with a **1**. If its predecessor is $(i - 1, j)$,



Codeword	Phrase	Vertex in the graph
0	11	(0,2)
1	011	(1,2)
2	101	
3	0011	(2,2)
4	0101	
5	1001	
6	0001	(3,1)
7	0010	
8	0100	
9	1000	
10	0000	(4,0)

Figure 3.1 An example of an (ad-hoc) enumerative code. The graph is given on the top (leaves shown shaded) and the codewords, and their phrases, right.

then we allocate all $\binom{i+j-1}{j}$ codewords, which code for all phrases with i **0**s and j **1**s that end with a **0** (see Fig. 3.1). Clearly, we must ensure that the total number of codewords is at most 2^{ℓ}.

Given such a graph, we parse the input-bit string as follows. Each phrase starts at $(0,0)$. If we are currently at the non-leaf vertex (i,j), upon reading a **1**, we move to $(i,j+1)$; upon reading a **0**, we move to $(i+1,j)$. By construction, both these vertices are in the graph. If we are at a complete leaf (i,j) then we have so far read a phrase with i **0**s and j **1**s; since all possible $\binom{i+j}{j}$ such phrases have associated codewords, we choose the appropriate codeword, output it and restart from $(0,0)$. Arriving at an incomplete leaf (i,j) from $(i,j-1)$, we must have read a phrase with i **0**s and j **1**s where the last bit is a **1**, so we output the appropriate codeword (the other case is similar), and restart from $(0,0)$. We now give examples of enumerative codes.

RLE.

RLE is a special case of enumerative coding. To have codes for runs of **0**s and **1**s of length $1,\ldots,2^{\ell-1}$, the corresponding graph contains the non-leaf

vertices $(0,i)$ and $(i,0)$, and the leaf vertices $(1,i)$ and $(i,1)$ for $i = 1, \ldots, 2^{\ell-1} - 1$, together with the leaf vertices $(0, 2^{\ell-1})$ and $(2^{\ell-1}, 0)$. A codeword is thus assigned to each phrase of the form $\mathbf{0}^i\mathbf{1}$ and $\mathbf{1}^i\mathbf{0}$ for $i = 1, \ldots, 2^{\ell-1} - 1$; and one each for $\mathbf{0}^{2^{\ell-1}}$ and $\mathbf{1}^{2^{\ell-1}}$.

Khodak Code.

The Khodak code [20] is is obtained by modifying Step (2) of the Tunstall algorithm in Section 3.3.1 to pick all the leaf nodes with highest probability and grow two leaves on all of them. It is known that every Khodak code is a Tunstall code, and that for the same dictionary size, the Khodak code has asymptotically the same average phrase length as the Tunstall code [20]. We show:

Theorem 3.3. *Any Khodak code is an enumerative code.*

Proof. We first prove an auxiliary lemma that implies that, when the probabilities of zero and one are not the same, the dictionary constructed by the Khodak algorithm is a subset of the dictionary constructed by the Tunstall algorithm – by observing that the order in which the leaves are expanded in both the algorithms is the same; but Khodak algorithm may stop earlier if there is not enough space to expand all the leaves with same probability.

Lemma 3.1. *For rational number $0 < d < 1, d \neq 1/2$, there are no nonnegative integers x, y, z, w such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$.*

Proof. Suppose that there exist nonnegative integers x, y, z, w such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$. Let $d = n/m$ for positive integers m and n, such that n and m are relatively prime. Without loss of generality, assume that $d > \frac{1}{2}$ and $z + w \geq x + y$. Then it is easy to argue that $z \geq x$ and $y \geq w$. Thus, $m^{z+w-x-y}(m-n)^{y-w} = n^{z-x}$. If m is even, then left side is even while right side is odd as n is relatively prime to m. If m is odd and n is even, then left

side is odd while right side is even. Finally, if both m and n are odd, the left side is even while the right side is odd. □

We now prove Theorem 3.3. Define T^k as the tree whose leaves represent the phrases of the Khodak code (similar to T in the Tunstall code). Next, let $T^k(i,j)$ be the set of all leaves in T^k which represent the phrases with i zeros and j ones. We say that $T^k(i,j)$ is complete if T^k contains all possible $(\binom{i+j}{i})$ phrases with i zeros and j ones (this is analogous to the definition of completeness of nodes in the enumerative codes). Now to prove Theorem 3.3, it is enough to prove the claim that if the Khodak algorithm expands the leaves in $T^k(i,j)$ then $T^k(i,j)$ is complete. The claim holds if the zero density is 1/2, because in this case, T^k is always a complete binary tree (and each expansion step expands all the leaves). Now we assume that the one density is strictly larger than the zero density. Since for every step in the Khodak algorithm, i and j for expanding $T^k(i,j)$ are uniquely determined by the Lemma 3.1, the claim can be proved by the induction on the number of expansion steps taken by the Khodak algorithm.

(Basis step) In the first step, we expand the leaf $T^k(0,1)$ which is complete.

(Inductive step) Assume the hypothesis that the claim is true if the number of steps is at most r. In the $r+1$ step, suppose we expand $T_k(i,j)$ which is not complete. Note that $T^k(i,j)$ is generated by expanding $T^k(i,j-1)$ or $T^k(i-1,j)$. Since both $T^k(i,j-1)$ and $T^k(i-1,j)$ are expanded before $r+1$-th step (because they have the smaller probability than $T^k(i,j)$), by induction hypothesis, they are complete. But if we expand $T^k(i,j-1)$ and $T^k(i-1,j)$ which are complete, $T^k(i,j)$ becomes complete, contradicting the assumption. □

Hybrid Enumerative Coding.

To obtain better compression using enumerative encoding, we reserve a fraction of codewords for run-length codes, and use the remaining for the Khodak code-

words. The run-length codewords are divided among **0** runs and **1** runs based on the densities of **0**s and **1**s.

3.3.3 LZW algorithm

Lempel-Ziv-Welch (LZW) algorithm [76] is a well-known dictionary-based compression algorithm. The dictionary constructed by the LZW algorithm has no fixed bound on its size, and it is not stored as part of the compressed text as it can be reconstructed during decompression. However, since our approach uses a bounded-size dictionary (with M codewords), we modify the LZW algorithm as follows: We first construct the dictionary in one pass over the string, as in the normal LZW algorithm till its size is M, and use that to parse the whole string in a second pass. Also, unlike the original LZW algorithm, the modified algorithm requires both the compressed string as well as the dictionary for decompression.

3.3.4 Empirical evaluation of the compressors

We now describe the compression performance of the above algorithms. We set $\ell = 16$ so each dictionary has $2^{16} = 65536$ codewords. For implementing RLE and Hybrid algorithms, we determined the maximum length of runs of **0**s in the RLE part of the dictionary as the smaller of (2^{15} $\times$ density of **0**) and maximum length of runs of **0**s in the test file (the maximum length of **1**s in the RLE part is also determined in the same way).

Test files

Table 3.1 summarizes the characteristics of the bit-strings we used in our experiments. `factor9.6` and `proteins` are obtained by parsing two XML files, and outputting $\mathbf{0}^i\mathbf{1}$ when a text node of length i is encountered [18]. `Z-Accidents` and `Z-Pumsb2` are used in a data structure for mining frequent patterns from benchmark data sets [33]. `dblp_100` and `english_100` are the FM-indices [23]

Bit-string	Total Size (10^6 bits)	Density of **0**s	Max run-length of **0**s	Max run-length of **1**s
`factor9.6`	812.0	0.964	2927	1
`proteins`	374.9	0.900	27376	1
`Z-Accidents`	903.3	0.996	4,250,294	1,315
`Z-Pumsb2`	1661.1	0.999	1,138,613	7,774
`dblp_100`	680.8	0.629	5,252,073	3,115,460
`english_100`	784.3	0.710	2,142,856	743,383
`rand_dblp`	680.8	0.629	42	20
`rand_english`	784.3	0.710	50	17

Table 3.1 Characteristics of the test files

Bit-string	Tunstall	LZW	Enumerative code			H_0	*Logsum*
			Khodak	RLE	Hybrid		
`factor9.6`	0.242	**0.151**	0.241	0.573	0.228	0.223	0.236
`proteins`	0.466	**0.104**	0.475	1.585	0.546	0.466	0.484
`Z-Accidents`	0.045	0.035	0.046	0.058	**0.030**	0.041	0.111
`Z-Pumsb2`	0.007	0.006	0.007	0.007	**0.004**	0.008	0.097
`dblp_100`	0.975	0.145	0.975	0.369	**0.136**	0.952	0.201
`english_100`	0.869	**0.285**	0.869	0.771	0.306	0.868	0.305
`rand_dblp`	**0.956**	0.971	**0.956**	4.872	**0.956**	0.952	0.991
`rand_english`	**0.874**	0.891	**0.874**	4.146	**0.874**	0.868	0.910

Table 3.2 Compression ratios of the test files.

of the text files in Pizza&Chili Corpus [24]. We use the implementation of FM-index from fm-index++ [72]. `rand_dblp` and `rand_english` are generated at random, but setting their length and density to be the same as `dblp_100` and `english_100`, respectively. The test bit-strings can be classified into four types based on their properties. The bit-strings `factor 9.6` and `proteins` are fairly sparse but have relatively short runs of **0**s and **1**s. The bit-strings `Z-Accidents` and `Z-Pumsb2` are very sparse and have some very long runs of **0**s. While `dblp_100` and `english_100` are quite dense, they have long runs of **0**s and **1**s; obviously, their randomly generated analogues do not have such long runs.

Table 3.2 shows the compression ratio achieved by the compressors on the test bit-strings. We also give the H_0 values of the bit-strings and their *Logsum*

value, defined as follows. If we divide a given bit-string X of length n into fixed-size blocks B_i, $i = 1 \ldots \lceil n/63 \rceil$ of size 63 and each B_i has weight $m(i)$, $Logsum(X)$ is defined as $\frac{1}{n}\sum_{i=1}^{\lceil n/63 \rceil}(\log(\binom{63}{m(i)}) + 6)$. *Logsum* is an estimate of the standard density-sensitive approach to compressed bitvectors used in [70] and predecessors (referred to as RRR in what follows), based on the implementation of [58], which is optimized for low redundancy. We make the following observations:

- There is a negligible difference in compression ratio between the Tunstall and Khodak codes. While Tunstall/Khodak are sometimes better than H_0, the variation is small, as implied by Remark 2.

- *Logsum* is sometimes significantly better than H_0, e.g. in `dblp_100` and `english_100`. The reason is that all-**0** and all-**1** blocks (which occur frequently in these bit-strings) compress far better than would be suggested by the overall density of these bit-strings. However, the additive overhead of 6 bits per block means that *Logsum*'s performance is poor on bit-strings such as `Z-Pumsb2` and `Z-Accidents`, as well as the random bit-strings.

- Among the enumerative codes, Hybrid uniformly performed the best, even easily outperforming RLE on very sparse files. It is also often the overall best performer, but it does perform poorly relative to LZW on the XML bit-strings. We speculate that this is because in XML files, identical elements may have similar-length text nodes under them (e.g., a `zipcode` element will usually contain a text string of length 5) and LZW is able to capture such long-range patterns.

3.4 Practical implementation of bitvectors based on V2F compression.

We now describe our implementation (in C++), which follows the general approach used by many existing schemes such as that of [58], but with some modifications. The bit-vector class is (largely) independent of the compressor, and takes as input two files: one which contains the codewords and the phrases, and another which contains 16-bit codewords output by the compressor. The codewords output by the compressor are read into a codeword array, and the rest of the bit-vector has three parts: a $\mathsf{rank/select}_1$ *index*, a table for scanning codewords, and finally a class that deals with rank/select operations on individual codewords. We now describe each in detail.

$\mathsf{rank/select}_1$ index. For rank we divide the bit-string into *rank blocks* of size B, where the i-th block consists of the bits numbered iB through $(i+1)B-1$. For each block, we store the position of the first codeword that intersects the block, the weight at the start of that codeword, and the absolute position in the bit-string where that codeword begins. The default is $B = 1024$, but this can (and should) be varied according to the compressibility of the bit-string, so that each block (on average) spans a moderate number (say 30-50) of codewords. For select_1, we use the standard "sample and scan" approach [13] used by most select implementations including [74, 58, 34]. We choose a sampling parameter s and divide the bit-string into *select blocks*, where the i-th select block begins at the position of the is-th **1**, and scan this select block to answer $\mathsf{select}_1(j)$ queries for $j = is+1, \ldots, (i+1)s-1$ (Type 0 blocks). This approach does not guarantee a good time bound if the **1**s are distributed non-uniformly: in the worst case, one may need to scan $\Theta(n)$ bit positions. To mitigate this effect, we treat *long gaps* differently [13]: we choose a threshold LG, and whenever a select block is larger than LG, we store the positions of the **1**s in the block explicitly (Type 1 blocks). Even though the number of long gaps is at most n/LG, LG must be relatively

high as storing **1** positions is costly. In addition, if a long gap spans only a moderate number of codewords, it is treated as a Type 0 block. For Type 0 blocks, as with rank blocks, we store codeword/phrase alignment information, and cumulative information. We choose s satisfying $m/s = \Theta(n/B)$, so that the number of select and rank blocks is similar (so on average both rank and select queries scan similar numbers of codewords).

Scanning a Rank/Select Block. To perform a rank operation, or a select_1 on a Type 0 block, we need to scan a rank/select block to find the codeword that contains position i. The key loop in scanning a block is to (a) read a codeword at a time from the compressed bit string, (b) obtain (and accumulate) the length of its phrase and its weight, and (c) determine both the codeword where position i lies, and the offset of position i within that codeword. This is done by table lookup, and this gives rise to the most important constraint on the size of M: it must comfortably fit "into cache" (as the cache is likely to contain other data in real applications). On our machine, this suggests that ℓ should be limited to 16; the table then takes 512KB[5].

Long Gaps: a Theoretical View. In this paragraph, we illustrate the potential asymptotic gains by using V2F codes in terms of protecting against long gaps in the "sample and scan" approach to select_1. This illustration makes a number of mappings from current practical parameter choices to asymptotic functions, which by its very nature involves a certain amount of guesswork: we do not hope to convince everybody of these mappings.

We begin by assuming that most practical implementations can be viewed asymptotically as using a block size of $B = \Theta((\log n)^2)$ bits and work by accessing $O(1)$ random memory locations and scanning $\Theta(\log n)$ consecutive memory locations, where each location comprises $\Theta(\log n)$ bits. This is justified as there

[1]For the current compressors, no phrase can be longer than 2^ℓ bits, so this could be reduced to 256KB.

is evidence that due to address translation, the cost of a random memory access is $O(\log n)$ [48]. For simplicity we consider the case of a bit-string with weight $\Theta(n/\log n)$, i.e. one whose compressed size is $O(n \log\log n/\log n)$ bits, and assume that we wish to achieve a redundancy of $O(n/\log n)$ bits. A typical sampling factor would be $s = \log n$, so that the cost of pointers to the sampled locations is $O((m/s)\log n) = O(n/\log n)$ bits. We would choose the long gap parameter to be $L = \Theta((\log n)^3)$, so that the cost of storing the locations of the **1**s in the at most $O(n/L)$ long gaps is $O((n/L)s \log n) = O(n/\log n)$ bits. This makes the worst-case cost of scanning a gap which of length exactly L to be $O((\log n)^2)$. However, in (say) Tunstall or Khodak coding, a bit-string with length $L = \Theta((\log n)^3)$ with weight $s = O(\log n)$ is compressed to $O((\log n)^2)$ bits or $O(\log n)$ codewords, which can be scanned in $O(\log n)$ time. (Note that the compressed size of these L bits is more than the information-theoretic bound for these L bits, but the Tunstall code is based on the *global* density and encodes each **0** using $\log(n/(n-m)) = O(1/\log n)$ bits and each **1** using $\log(n/m) = O(\log\log n)$ bits.)

Implementation of Codeword Operations. Having located the codeword containing the answer, we perform an appropriate rank/select on its phrase. The default implementation of rank on a codeword concatenates all phrases into a bit-string similar to Corollary 3.1 and stores it in a bit-vector supporting rank [34], together with two words per codeword to allow rank on an individual phrase to be reduced to rank on the bit-vector. select on each phrase is done by explicitly storing the positions of the **1**s in the phrase in an array. We estimate the *fixed* overhead to be about 4 `ints` per codeword, or 1MB overall. However, a potentially major variable overhead is the size of the rank phrase bit-vector and the phrase select array.

An obvious optimization is that for codes known to comprise runs of **0**s or **1**s, indicated by an additional *type* field stored in the length/weight table,

File name	Khodak	LZW	Enumerative code	
			Khodak	Hybrid
factor9.6	2.15%	7.24%	2.15%	2.15%
proteins	1.26%	3.01%	1.23%	0.90%
Z-Accidents	38.36%	7.22%	38.30%	19.60%
Z-Pumsb2	668.61%	200.14%	667.98%	73.25%
dblp_100	0.16%	15.53%	0.16%	0.54%
english_100	0.17%	52.20%	0.17%	0.22%
rand_dblp	0.16%	0.16%	0.16%	0.16%
rand_english	0.17%	0.16%	0.17%	0.17%

Table 3.3 Total phrase length of test files (as % of compressed output), excluding RLE codewords

we directly (and trivially) answer rank and select queries on the corresponding phrase. Table 3.3 shows the size of the resulting rank phrase bit-vector (the phrase select array is usually smaller). As suggested by Corollary 3.1, for Khodak codes, the size of the dictionary is negligible for relatively high-density bit-vectors. The overhead is much larger for the Z-Accidents and Z-Pumsb2, though Hybrid codes, which have many RLE codes, have smaller dictionaries than Khodak codes. Nevertheless, for very sparse bit-strings, it is clear that this naive approach is inappropriate.

3.4.1 Testing Methodology

The code was written in C++, and compiled with g++ 4.8.3 with optimisation level 3, and tested on a 64-bit machine with 64GB RAM and an Intel Xeon E7450 6-core CPU clocked at 2.40GHz with 3×3 MB shared L2 caches and 12MB L3 cache, running Fedora Linux (kernel version 3.16.2). Tests were performed for the memory usage, and four tests for the speed of this structure, as follows.

Memory Test. To determine the true physical memory used by these data structures, we initialize them and then fork a process that allocates memory

equal to the physical memory of the machine, which will result in all other processes' pages to be swapped out. Putting the forked process to sleep, we then perform rank and select operations and then measure the resident memory of the process.

In this test, we implemented bitvector based on V2F codes in two ways - practical implementation described in this section and implementation based on Theorem 3.1. In the latter implementation (based on Theorem 3.1), we implemented OD and PS using RRR and sdarray which have low redundancy on dense and sparse bit-strings respectively. Since this implementation is not optimized for the speed tests, we only used the practical implementation for the other four tests.

We also measure the memory usage of our implementations by a self-reporting procedure which checks the total size of the main data structures using size reporting functions. Testing results shows that the measured memory size is larger than self-reported memory size because of the initial space used by OS and other variables in the program. But difference between them does not exceed 10MB in all test files.

rank_1 Test. To test the speed of rank, we perform $\mathsf{rank}_1(i)$ n times, for random $i \in 1..n$.

Random select_1 Test. Like the rank_1 test, this test performs $\mathsf{select}_1(i)$ n times, for i selected randomly from $1..m$. Although "sample-and-scan" approach does not guarantee a good time bound, if the bit-string has long gaps, several implementations, including `RSDic`, do not guard against long gaps. However, their performance for random select tests on random bit-vectors (which typically don't have long gaps) is good. Vigna [74] proposed testing on pathological bit-strings to determine whether an implementation had good worst-case select performance. We note, however, that essentially *regardless* of the input bit-

string, a random select test will not be able to distinguish between "sample-and-scan" bit-vectors, that deal with long gaps and those that don't. Specifically, observe that in any select block, the expected time taken to perform a select of one of the **1**s in this block, assuming a fairly even distribution of the **1**s within this block, is essentially *proportional to its length.* Since a random select accesses each select block with equal probability, it is not hard to see that the average running time of a random select is essentially *independent* of the distribution of select block lengths; i.e., a random select test is unlikely to distinguish between an easy bit-string and a pathological one. To address this issue, we propose a *hard* select test, described below.

Hard select_1 Test. We perform 2^{19} random rank_1 queries, and store the results in an array Q of the same size (with repetitions). We then repeat the following, n times: select a random index i in Q and perform $\mathsf{select}_1(Q[i]+1)$. Doing this will select a **1** in a select block with probability proportional to the length of the select block (since the argument of the rank query falls in a select block with probability proportional to its length), and thus focusses on the harder select queries in a bit-string.

Mixed Test. We initialise an array Q of size 2^{19} to values from random $\mathsf{rank}_1(i)$ as above. We cycle through the array and perform $\mathsf{select}(Q[i]+1,1)$ as above, but then do a $\mathsf{rank}_1(j)$ for a random index j, and store the result in $Q[i]$. Each such pair of rank and select operations is performend n times.

For our benchmarks, we choose the LZW code for XML bit-strings and Hybrid code for other bit-strings as F2V coders which gives the best compression ratio for their bit-strings. For comparison, we used Okanohara's rsdic code [61] (based on [58]), sdarray from Okanohara and Sadakane [63] and RRR from `sdsl-lite` [34]. We now describe the rsdic and sdarray bitvectors briefly.

The *Compressed Rank Select Dictionary*, rsdic is based on the structure proposed by Navarro and Providel [58]. It divides the bit-string X into fixed-sized blocks of length $t = \lceil (\lg n)/2 \rceil$. The set of all possible blocks are divided into classes based on the number of 1's in the block. Hence, each block can be identified by a pair (k, r), where k is the class number which is simply the weight of the block, and r is the index of the block in a table containing the set of all possible blocks in the class, in some canonical order, say, the lexicographic order. Thus, the representation of any block can be stored in $\lceil \lg(t+1) \rceil + \lceil \log \binom{t}{k} \rceil$ bits. Also, one can rebuild a block "on-the-fly" using its representation, without storing any additional precomputed tables. For the sequence of blocks constituting the given bit-string X, it stores the first components (i.e., the weights of the blocks) in an array K, using fixed size entries of $\lceil \lg(t+1) \rceil$ bits each; the second components of all the blocks in the sequence are concatenated and stored as a bitvector R. To enable fast access into R, it first groups every $\lfloor \lg n \rfloor$ consecutive blocks into a superblock. For every superblock, it then stores a pointer into R to point to the starting position of the representations corresponding to its blocks. In addition, we also store the rank up to the first bit in each superblock. To compute the rank for a given position, we first find the superblock containing the position, and do sequential search from the first block in the superblock. To support the select operation, we first perform a binary search to find the superblock containing the required position, and then scan the blocks within the superblock. The size of R can be shown to be at most $nH_0(X) + o(n)$ bits, and the size of K is $n\lceil \lg(t+1)/t \rceil = o(n)$ bits. Thus the space usage of rsdic is $nH_0(X) + o(n)$ bits.

Okanohara and Sadakane [63] proposed the sdarray which either stores an sarray when the given bit-string X is *sparse*, or a darray when X is *dense*. To describe the sarray, consider an array $x[0, \ldots, m-1]$ where $x[i]$ stores the position of the $(i+1)$-th 1-bit in X. We choose a parameter t, and store the lower $z = \lceil \lg t \rceil$ bits of each $x[i]$ in an array L such that $L[i] = x[i] \bmod t$. The

upper $w = \lfloor \lg(n/t) \rfloor$ bits of each $x[i]$ is encoded in unary to obtain a bit vector, H, of length $m+t$, along with auxiliary structures to support rank and select in $O(1)$ time on H. The operation select on X can be supported in constant time by finding the upper bits using the select operation on H, and accessing the lower bits from the array L. To support rank(i) on X, we first find a smallest element whose position is greater than $\lceil i/2^w \rceil \cdot 2^w$ using select on H and count number of ones sequentially from here using H and L. By choosing $t = 1.44m$, the total size of sarray becomes $1.92m + m(\lg(n/m)) + o(m)$ bits.

The construction of darray first divides the given bit-string X into blocks of L ones each, and constructs an array $P[0, \ldots, n/(L-1)]$ such that $P[i]$ stores the position of iL-th one in X. These blocks are represented based on their length. If the length of a block is more than $(\lg n)^4$, it is represented by storing the positions of all the ones in it. Otherwise, its representation consists of the position of every $(\lg n)$-th one in the block, using $L(\lg L)/\lg n$ bits. To support select(i), we first find the block that contains the answer using P. If this block is longer than $(\lg n)^4$, we can read the answer from its representation. Otherwise, we use the representation of the block to find a sequence of $(\lg n)$ positions, one of which corresponds to the required answer, and scan the sequence to find the answer. The *rank* operation is supported using an approach similar to that of rsdic. By choosing $L = (\lg n)^2$, the size of darray can be limited to $n + o(n)$ bits, including the bit-string X.

3.4.2 Results of Empirical Evaluation

Memory test. Practical implementation of bitvectors based on V2F used significantly less memory than the competition in most cases (see Fig. 3.2); the exception is sdarray with `Z-Accidents` and `Z-Pumsb2` and RRR with `rand_dblp`, `rand_english` and `english_100`. In the former case (for `Z-Accidents` and `Z-Pumsb2` files), despite the V2F compressed bit-string being significantly less than H_0, the compressed size is so small that the fixed overhead of the phrase

rank/select structure dominates. Also for latter three files, their V2F compression ratios are close to *Logsum*, and the overhead in the bitvector based on V2F implementations is more than that of RRR.

Although the redundancy in Theorem 3.1 is less than (little-oh of) the compressed output size in theory, for the implementation based on Theorem 3.1, the space overhead is $1 \sim 3$ times more than the compressed output size, in all the test files except `Z-Accidents` and `Z-Pumsb2` (for which the compressed output size is significantly smaller). This is because the $O(\lg(n/C))$ term in the redundancy can be larger than the codeword size even though the value of $\lg(n/C)$ in these files is $4 \sim 6$, which is less than the codeword size.

rank$_1$ test. Generally speaking, apart from sdarray, which is not optimized for rank, all others are comparably fast. However, sdarray does better than rsdic on the Z-vectors, possibly because it fits in cache due to its much lower memory usage, and the V2F bitvectors and RRR both do relatively poorly on the random bit-strings (see Fig. 3.3).

Random select$_1$ test. As expected, sdarray is generally the fastest, but loses out a little on the FM-index files, as it cannot compress them. The V2F bitvector is the second-best, and is very close to the best, in most cases, but performs slightly worse on the random files (see Fig. 3.4). rsdic and RRR show significant weakness on the Z-vectors and XML bit-strings respectively.

Hard select$_1$ test. rsdic is the only bit-vector that does not guard against long gaps, and performs very poorly (up to 20 times slower) on three of the input files (see Fig. 3.5). V2F bitvectors do the hard select$_1$ test at roughly the same speed as the random select$_1$, and thus demonstrate their resilience.

Mixed test. The V2F bitvectors are the best overall performers in this test, since they show good performance for both the hard select test and the

rank test (see Fig. 3.6).

3.5 Future works

In this chapter, we consider theoretical and practical implementations of compressed bitvectors. There is much room for further investigation. For instance, the naive approach to operations on individual phrases, as well as the relatively simple approach to supporting rank/select, leads to an overhead that is rather high for highly compressible bit-strings (admittedly, these are so sparse as to test the boundaries of our stated aim of targeting "moderately compressible" bit-strings). This could be overcome by adhering more closely to the theoretical result, and making greater use of on-the-fly decoding also can be considerd. Apart from the Tunstall/Khodak/Enumerative codes, we have not explored V2F codes in any non-trivial way. Much more work is clearly possible along this axis as well.

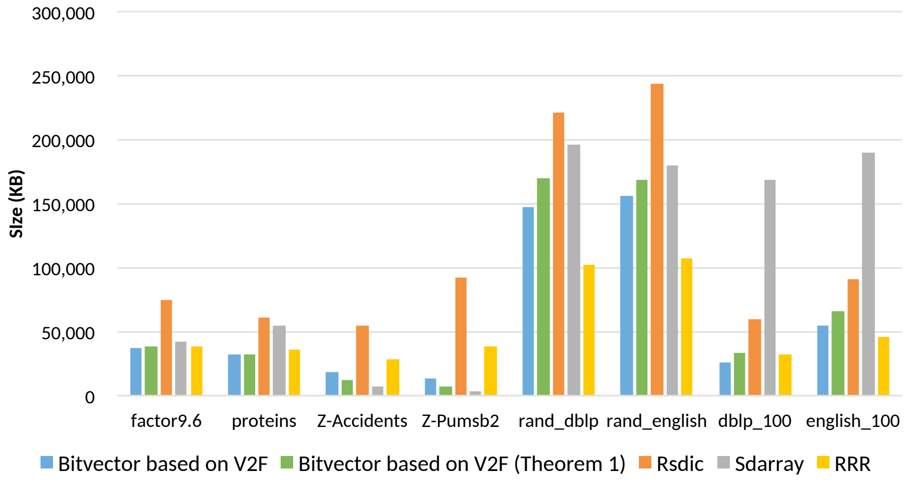


Figure 3.2 Memory test

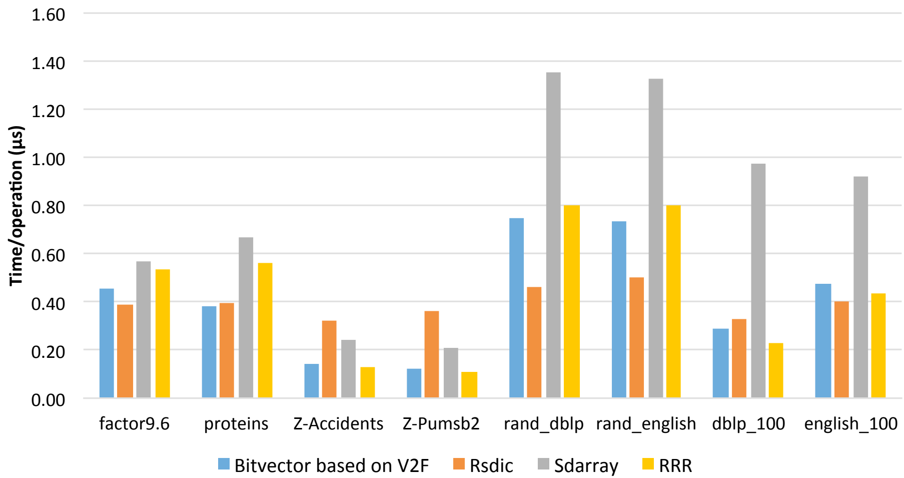


Figure 3.3 rank_1 test

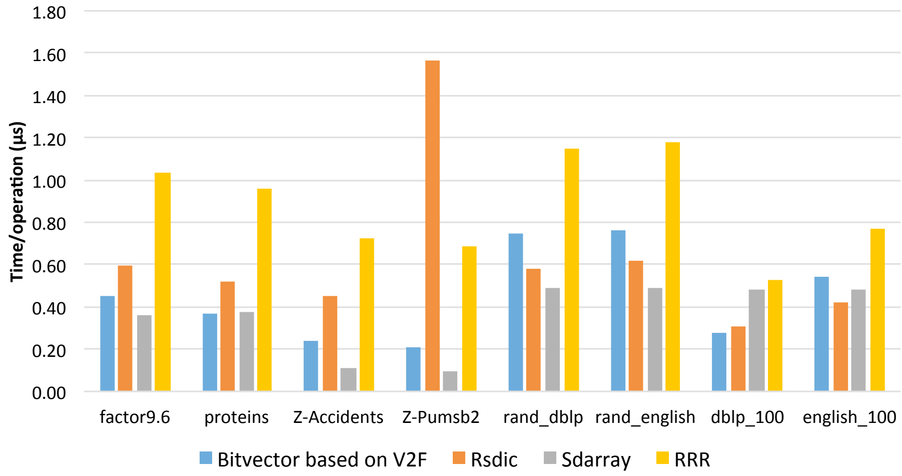


Figure 3.4 Random select_1 test

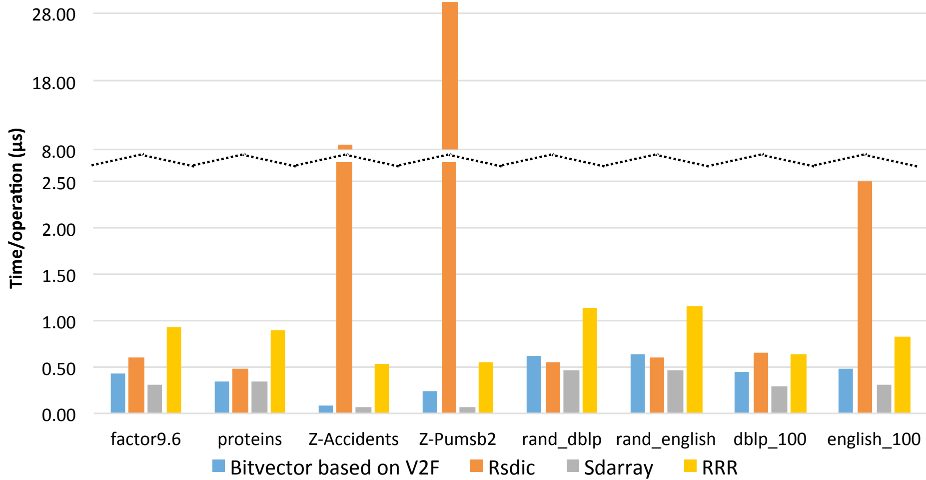


Figure 3.5 Hard select_1 test

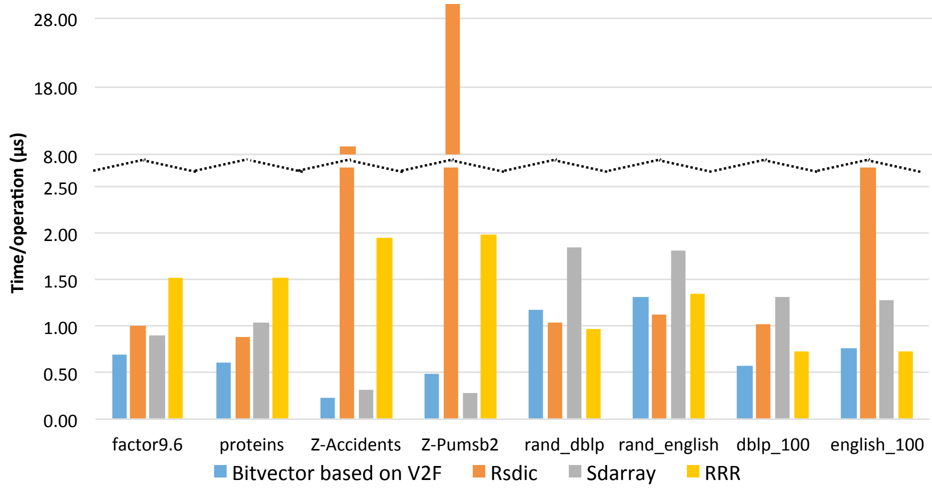


Figure 3.6 Mixed test

Chapter 4

Space Efficient Data Structures for Nearest Larger Neighbor

4.1 Introduction

Given a sequence of n elements from a totally ordered set, and a position in the sequence, the *nearest largest neighbor* (NLN) query asks for the position of an element which is closest to the query position, and is larger than the element at the query position. More formally, given an array $A[1 \dots n]$ of length n containing elements from a totally ordered set, and a position i in A, we define the query:

- $\mathsf{NLN}(i)$: return the index j such that $A[j] > A[i]$ and $|i - j| = \min\{k : A[i+k] > A[i] \text{ } or \text{ } A[i-k] > A[i] \text{ for } k > 0\}$. Ties are broken to the left, and if there is no element greater than the query element, the query returns the answer ∞.

In a similar way, we can the define NLV (*next larger value*), and PLV (*previous larger value*) queries, which return the position of the nearest larger neighbor to the right and left, respectively, of the query position. In a symmetric way,

one can also define nearest *smaller* neighbor problems. In this chapter, we will stick to the version that seeks the larger neighbors.

We exhibit connections between the NLN problem and the well-studied problem of supporting *range maximum* queries on a given array. Given an array A, the query $\mathsf{RMaxQ}(i, j)$ (range maximum query) returns a position k between i and j such that $A[k]$ is a maximum element among $A[i, \ldots, j]$.

Two-dimensional NLN We also consider a natural extension of the NLN problem to two-dimensional arrays. Here, we define the NLN of a query position as the closest position in the two-dimensional (2D) array, in terms of the L_1 distance, that contains an element larger than the element at the query position. More formally, given a position (i, j) in $A[1 \ldots n][1 \ldots n]$, $\mathsf{NLN}((i, j)) = (i', j')$ such that $A[i', j'] > A[i, j]$, and $|i - i'| + |j - j'| = \min\{|x| + |y| : A[i + x, j + y] > A[i, j]\}$. If there is no element greater than the query element, the query returns the answer (∞, ∞).

Previous Work and Motivation These kinds of problems have attracted much attention. In addition to the data structuring problems, the *off-line* variants, usually called *All Nearest Larger Neighbors* (or similar), which consist in computing answers for all possible input positions, have also been studied. For example, Berkman et al. [7] gave efficient parallel algorithms for the one-dimensional (1D) off-line problem and showed their importance as a preprocessing routine for answering range minimum queries, triangulation algorithms, reconstructing a binary tree from its traversal orders and matching a sequence of balanced parentheses [7].

Fischer et al. [28] considered the problem of supporting NLV and PLV, and showed how a data structure supporting these two queries can be used in obtaining *entropy-bounded compressed suffix tree* representation. (They considered the min version of the problem instead of max, and named the operations NSV and

PSV, for the next and previous smaller values, respectively.) They considered the problem of supporting NLV and PLV in the indexing model, and obtained the following time-space tradeoff result. For any $1 \le c, \ell \le n$, one can use:

$$O\left(\frac{n}{c}\lg c + \ell\frac{n \lg\lg n}{\lg n} + \frac{n \lg n}{c^{\ell}}\right)$$

bits of space and answer queries in $O(c\ell)$ time. As given, they are unable to go below $O(n \lg\lg n / \lg n)$ space, and use more space than we do whenever $c = \omega(\lg n)$. As mentioned later, we improved the trade-off to $O((n/c)\lg c)$ bits with $O(c)$ time. To attain $O((n/c)\lg c)$ space for $c = (\lg n)^{\Omega(1)}$, one can choose $\ell = O(1)$ and obtain $O(c)$ time. For smaller values of c, the middle term in the space usage will never dominate for reasonable values of ℓ (clearly, we must always choose $c \ge 2$ and $\ell = O(\lg\lg n)$ in this range) and it suffices (and is optimal) to choose $\ell = O(\lg_c \lg n) = O(\lg\lg n - \lg\lg c)$. Thus, for any $c = O(\lg n)$, their running time for space $O((n/c)\lg c)$ is $O(c(\lg\lg n - \lg\lg c))$, and our solution is better for small enough c. Jayapaul et al. [44] gave a solution that uses $O((n/c)\log c + (n \log n)/c^2)$ bits and $O(c)$ time; this space usage equals ours for $c = \Omega(\log n / \log\log n)$ but is worse otherwise.

Fischer et al. [28] also gave an encoding that supports the PSV and NSV queries in constant time, using $4n + o(n)$ bits. The encoding size was later reduced to the optimal $2.54n + o(n)$ bits by Fischer [25]. Jayapaul et al. [44] observe that this can be further improved to $2n + o(n)$ bits if all the elements are distinct. For the case of binary sequences, the data structure version of the NLN problem can be solved by building an auxiliary structure to support rank and select queries on the bit-strings [69]. This uses $o(n)$ bits of extra space, in addition to the input array, and answers NLN (and also NLV and PLV) queries in $O(1)$ time.

Given a 2D array, Asano et al. [3] considered the *All Nearest Larger Neighbours* problem which asks for computing the NLN values for all the elements in the input array. They showed that this problem can be solved in $O(n^2 \lg n)$

time (and more generally, for any d-dimensional array in $O(n^d \lg n)$ time). To the best of our knowledge, the data structure version of the 2D NLN problem, in which we are interested in constructing a data structure that answers online queries efficiently, has not been considered earlier.

Our results Our main results are as follows.

- For the case of 1D, we look at the problems in indexing model. We provide an algorithm that matches the tradeoff for NLN [44]. For NLV, our algorithm achieves the time-space product of $O((n/c) \lg c)$ (where the query takes $O(c)$ time) while the lower bound is $\Omega(n)$.
- For the 2D NLN problem in the encoding model, we first show that $\Omega(n^2)$ bits are necessary to encode the array to support NLN queries, even when all the elements are distinct. We then describe an asymptotically optimal $\Theta(n^2)$-bit encoding that answers queries in $O(1)$ time, even when all the elements are not distinct. One can achieve this result easily when all the elements in the array are distinct. However, distinctness is a strong assumption in these kinds of problems. For example, in the 1D case with distinct values, NLV and PLV are obtained relatively easily from the Cartesian tree, giving an $2n + o(n)$ bit-encoding. By contrast, if we do not assume distinctness, the optimal space is about $2.54n$ bits, and the data structure achieving this bound is also more complex [25]. Also, Asano et al. [3] remark that the off-line problem for any dimension is "simplified considerably" if one assumes distinctness.

As we note, in the 1D case, the NLV and PLV problems are closely connected to the RMaxQ problem. In the 1-D case, there is no asymptotic difference between the encoding complexity of RMaxQ and NLV/PLV. The 2D RMaxQ problem has received a great deal of attention lately [19, 10, 9, 8]. It is known that any 2-D RMaxQ encoding takes $\Omega(n^2 \lg n)$ bits [19, 10]; thus, our $\Theta(n^2)$-bit en-

coding for 2D NLN shows that encoding complexity of NLN is asymptotically different from RMaxQ in the 2D encoding scenario (unlike the 1-D case).

The rest of this chapter is structured as follows. Section 4.2, we describes an indexing for answering NLV queries on 1D arrays. In Section 4.3, we propose an optimal time-space tradeoff encoding for answering NLN queries on 2D binary arrays. In Section 4.4, we propose an encoding for answering NLN queries on 2D arrays which takes asymptotically optimal space and supports NLN queries in constant time. Finally, in Section 4.5, we give some open problems.

4.2 Indexing NLV queries on 1D arrays

In this section, we give a result for the NLV problem in the indexing model on 1D array. The approach follows closely the proof of Fischer et al. [28], which in turn adapts ideas from Jacobson's representation of balanced parentheses sequences [42], and is given in full for completeness.

We begin with the following lemma gives an encoding for answering NLV queries on 1D array.

Lemma 4.1 ([28, 27, 44]). *Given a 1D array A of size n, there exists a data structure in the encoding model that uses $2n + o(n)$ bits and solves NLV queries in $O(1)$ time.*

Now we state our result in this section.

Theorem 4.1. *Given a 1D array A of size n, there exists a data structure which supports NLV queries in the indexing model in $O(c)$ time using $O((n/c)\lg c)$ bits for any parameter $2 \le c \le n$.*

Proof. Divide A into n/c blocks of size c. For any value $1 \le i \le n$, if i and $\mathsf{NLV}(i)$ are in the same block, say that i is a *near* value, otherwise say that i is a *far* value. Consider a block B and suppose that one or more of its far values have an NLV in block B', for an arbitrary block B'. Then the leftmost far value

in B whose NLV is in B' is called a *pioneer*, and its NLV is called its *match*. It is known that there are $O(n/c)$ pioneers in A [42].

We maintain a bit-vector V in which the i-th bit is a 1 if $A[i]$ is a pioneer or a match of one, and 0 otherwise. This bit-vector has length n and contains $O(n/c)$ 1's, so by Lemma 2.4, we can store it in $O((n/c)\lg c)$ bits and perform rank/select queries on it in $O(c)$ time. Next, we take the sequence S_p consisting of all pioneers and their matches. This sequence is of length $O(n/c)$. We represent this sequence using Lemma 4.1 using $O(n/c)$ bits, to support NLV queries in $O(1)$ time. We claim that if $k = \mathsf{NLV}(j)$ in S_p, then $\mathsf{select}_1(V,k) = \mathsf{NLV}(\mathsf{select}_1(V,j))$ in A. Suppose that this claim is not true. This means there is a pioneer i_p such that $\mathsf{NLV}(i_p)$ is the value between i_p and the match of i_p. It cannot be the case that i_p and $\mathsf{NLV}(i_p)$ are in the same block, since i_p is a far value. If i_p and $\mathsf{NLV}(i_p)$ are in different blocks, then $\mathsf{NLV}(i_p)$ is the match of i_p. So the claim is true.

To answer the query $\mathsf{NLV}(i)$, we first check to see if the answer is in the same block as i taking $O(c)$ time. If so, we are done. Else, (assuming wlog that $A[i]$ is not a pioneer value) we find the first pioneer p_i before position i by doing rank/select on V. As $A[i] < A[p_i]$, $\mathsf{NLV}(i)$ is less than or equal to the match of p_i. Since i is the far value in this case, $\mathsf{NLV}(i)$ and $\mathsf{NLV}(p_i)$ are in the same block. We find the corresponding position of $\mathsf{NLV}(p_i)$ in S_p using the NLV encoding of S_p and find the $\mathsf{NLV}(p_i)$ using rank/select on V. Finally we scan left from $\mathsf{NLV}(p_i)$ to find $\mathsf{NLV}(i)$. The overall time taken to answer the query is $O(c)$. □

4.3 Encoding NLN queries on 2D binary arrays

In this section, we first give an optimal encoding for NLN, and using this obtain an time-space trade-off for an NLN index for a 2D binary array.

Theorem 4.2. *There is a data structure that takes $O(n^2)$ bits for any binary array $A[1\dots\ n][1\dots\ n]$, and supports NLN queries in $O(1)$ time.*

Proof. Given a query position p, we compute $\mathsf{NLN}(p)$ by computing the positions of the nearest larger values in all four *quadrants* induced by a vertical and a horizontal line passing through p, and then returning the closest of these four positions as the answer (the point p is included in all the four quadrants). Thus, it is enough to describe a structure that supports finding the position of the nearest larger value in (say) the upper-right quadrant; in the rest of this proof, we use NLN_{NE} to denote this.

Given a position $p = (i, j)$, let $q = (i', j')$ be its NLN_{NE} if there is a $\mathbf{1}$ in the upper-right quadrant of p. For each position (i, j), we give a label from the alphabet $\{R, C, D, O, Z\}$, depending on the answer for the query NLN_{NE}, as follows. The position (i, j) is labeled with:

- O ("One") if $A[p] = \mathbf{1}$ (the value at the position is $\mathbf{1}$);
- R ("Row") if $i = i'$ (its answer is in the same row);
- C ("Column") if $j = j'$ (its answer is in the same column);
- D ("Diagonal") if $i < i'$ and $j < j'$ (its answer is not in the same row or column); and
- Z ("Zero") if $A[p] = 0$ and also there is no $\mathbf{1}$ in the upper-right quadrant of p.

Now, given a query position p, if the position p is labeled with O or Z, then we conclude that NLN_{NE} does not exist (in this quadrant). Otherwise, if the label is R, we can find its answer by following the positions $(i, j + k)$, for $k = 1, 2, \ldots$ (i.e., elements in the same row) till we reach a position with label O, and return that position as the answer. Also, one can easily show that all the intermediate positions cannot have label O. Analogously, if the label is C, then we follow the positions in the same column until we reach a position with label O and return that position. Finally, if the label is D, then we first follow the positions $(i + k, j + k)$, for $k = 1, 2, \ldots$ till we reach the first position

$(i+\ell, j+\ell)$ with a label different from D. The label of position $(i+\ell, j+\ell)$ can be O, R or C. If it is O, then we return that position as the answer. In the other two cases, we can find the answer by following the row or column as described above. The data structure simply stores the labels of all positions in the array (for each quadrant). In addition, to support queries faster, we build rank/select structures (over constant alphabet strings) for the encoding of each row, each column and each diagonal. By Lemma 2.5, the total space usage is clearly $O(n^2)$ bits. Now, queries can be supported in constant time by using rank/select to jump to the appropriate positions as described in the above procedures. □

Now we describe an index for a given 2D binary array, in the bit-probe model, that uses $O(n^2/c)$ bits and supports NLN queries in $O(c)$ time. Since the indexing trade-off lower bound for the 1D case described in Theorem 4.2 also holds for higher dimensions, it follows that the achieved trade-off is optimal.

We begin by introducing some notation that will be used later. Suppose we divide an $n \times n$ array A into blocks of size $c \times c$, for $1 \leq c \leq n$, and divide each block into c sub-blocks of size $\sqrt{c} \times \sqrt{c}$. We define an (i, j)-block as the sub-array $A[(i-1)c+1 \ldots ic][(j-1)c \ldots jc]$ and an (i, j, k, l)-sub-block as the sub-array $A[(i-1)c+(k-1)\sqrt{c} \ldots (i-1)c+k\sqrt{c}][(j-1)c+(l-1)\sqrt{c} \ldots (j-1)c+l\sqrt{c}]$. For each (i, j)-block, we define eight regions, consisting of sets of blocks (some of which can be empty) as follows: the region

$N(i, j)$ consists of all (i, l)-blocks with $l > j$;

$S(i, j)$ consists of all (i, l)-blocks with $l < j$;

$E(i, j)$ contains all (k, j)-blocks with $k > i$;

$W(i, j)$ contains all (k, j)-blocks with $k < i$;

$NE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l > j$;

$NW(i, j)$ contains all (k, l)-blocks with $k < i$ and $l > j$;

$SE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l < j$; and

$SW(i, j)$ contains all (k, l)-blocks with $k < i$ and $l < j$.

Similarly, for each (i, j, k, l)-sub-block, we also define the regions $N_{i,j}(k, l)$, $S_{i,j}(k, l)$, $E_{i,j}(k, l)$, $W_{i,j}(k, l)$, $NE_{i,j}(k, l)$, $NW_{i,j}(k, l)$, $SE_{i,j}(k, l)$ and $SW_{i,j}(k, l)$ in the same way.

Theorem 4.3. *Given a binary array $A[1 \ldots\ n][1 \ldots\ n]$ one can construct an index of size $O(n^2/c)$ bits to support* NLN *queries in optimal $O(c)$ time, for any parameter c, where $1 \leq c \leq n$.*

Proof. We divide the array A into blocks and sub-blocks as mentioned earlier. We construct an $n/c \times n/c$ array $A'[1 \ldots n/c][1 \ldots n/c]$ such that $A'[i][j] = \mathbf{1}$ if there exists at least a single $\mathbf{1}$ in the (i, j)-block, and 0 otherwise. We also construct another $n/\sqrt{c} \times n/\sqrt{c}$ array $A''[1 \ldots n/\sqrt{c}][1 \ldots n/\sqrt{c}]$ such that $A''[i][j] = \mathbf{1}$ if there exists at least a single $\mathbf{1}$ in the $(\lfloor i/\sqrt{c} \rfloor, \lfloor j/\sqrt{c} \rfloor, i - \lfloor i/\sqrt{c} \rfloor \sqrt{c}, j - \lfloor j/\sqrt{c} \rfloor \sqrt{c})$-sub-block, and 0 otherwise .

Suppose the query q is in the (i, j, k, l)-sub-block. If $A''[i\sqrt{c}+k, j\sqrt{c}+l] = \mathbf{1}$, scanning $O(1)$ sub-blocks is enough to find the NLN of q, and this takes $O(c)$ time.

Now, consider the case when $A''[i\sqrt{c} + k, j\sqrt{c} + l] = 0$ but $A'[i, j] = \mathbf{1}$. In this case, it is clear that we can identify $O(c)$ sub-blocks in which the answer may lie – namely all the sub-blocks in its block, and the eight neighbouring blocks. We find the potential answer in each of the eight directions (E, W, N, S, NE, NW, SE, and SW), and then compare their positions to find the actual answer. To find the answer in E direction, we scan the bits in A'' that are to the right of the current sub-block, till we find a $\mathbf{1}$, say, in sub-block s. We then scan sub-block s, and the sub-block immediately to its right, to find the potential answer in this direction. Similarly, we can find the potential answers in the W, S, and N directions. Next, we find the nearest $\mathbf{1}$ to the query in the $NE_{i,j}(k, l)$ region. This element is the nearest $\mathbf{1}$ from the bottom-left corner of $(i, j, k + 1, l + 1)$-sub-block. The nearest $\mathbf{1}$ from the bottom-left corner of (a, b, c, d)-sub-block in the $NE_{a,b}(c, d)$ region is same as the the nearest $\mathbf{1}$ from

the bottom-left corners of one of these four sub-blocks: (1) (a,b,c,d)-sub-block (2) $(a,b,c+1,d)$-sub-block, (3) $(a,b,c,d+1)$-sub-block, or (4) $(a,b,c+1,d+1)$-sub-block. Therefore we encode each sub-blocks using 2 bits indicating the case it belongs to ((1), (2), (3) or (4)), which takes a total of $O(n^2/c)$ bits. Now, to find the answer in the NE direction, we scan $O(c)$ sub-blocks to find the sub-block which contains the nearest **1** from q in $NE(i,j,k,l)$. Once we find the corresponding sub-block, finding the nearest **1** from the bottom-left corner in the sub-block takes $O(c)$ time. We can find the nearest **1** in the $NW_{ij}(k,l)$, $SE_{ij}(k,l)$ and $SW_{ij}(k,l)$ regions in the same way. Then the NLN of q is the closest one among these eight candidates.

Finally, consider the case when $A'[i,j] = 0$. By storing the data structure of Theorem 4.2 for the array A' using $O(n^2/c^2)$ bits, we can find the nearest blocks in each direction to the query position which contains a **1**, in $O(1)$ time. Let one of these blocks be the (i',j')-block, let ℓ be the L_1 distance from (i,j) to (i',j') in A'. The value ℓc is an estimate (within an additive factor of $2c$) for the L_1 distance from q to its NLN. Assume, wlog, that (i',j') is in the $NE(i,j)$ region of A'. We first describe how to find the nearest **1** in the $NE(i,j)$ region. Define $d(i,j)$ as the sequence of blocks in the top-left to the bottom-right diagonal that contains the (i,j)-block (i.e., all the blocks (i',j') in A such that $i'+j' = i+j$), where the blocks are ordered in the increasing order of their i values. We store a 1-D array $D_{(i,j)}$ of size equal to $|d(i,j)| \le n/c$, $D_{(i,j)}[m]$ is the distance from the bottom-left element of the m-th block in the sequence $d(i,j)$ to the nearest **1** in that block, and $2c+1$ if there is no **1** in that block. Note that each block belongs to exactly one $D_{(i,j)}$, and hence the total size of all these $D_{(i,j)}$ arrays is $O((n^2/c^2)\lg c)$ bits. In addition, we also construct a linear-bit RMQ (range minimum query) data structure for each $D_{(i,j)}$ (using a total of $O(n^2/c^2)$ bits), so that RMQ queries can be supported in $O(1)$ time [27]. Now, we find the two potential blocks in the $NE(i,j)$ region that may have the nearest **1** from q by performing RMQs on $D_{(i',j')}$ and $D_{(i',j'+1)}$ among all the blocks that are

Figure 4.1 Suppose the nearest block that contains a **1** from the $(2,2)$-block is the $(4,3)$-block. Then $d(4,3)$ contains the blocks $(2,5),(3,4),(4,3)$ and $(5,2)$, in that order. We can find the nearest **1** in NE(2,2) using $\mathsf{RMQ}(2,3)$ on $D_{(4,3)}$ and $\mathsf{RMQ}(1,3)$ on $D_{(4,4)}$.

contained in the $NE(i,j)$ region (it is easy to see that they form a consecutive range). We then choose the closer one between these two from q. (Figure 4.1 shows an example.) Note that if (i',j') is in a different region from $NE(i,j)$, then we may not find any potential answer in $NE(i,j)$, as all the 'relevant' blocks in $d(i',j')$ and $d(i',j'+1)$ may be empty. We can find the nearest **1** in $NW(i,j)$, $SE(i,j)$ and $SW(i,j)$ in a similar way.

Next, we describe how to find the nearest **1** in the $N(i,j)$ region (finding the nearest **1** in the $S(i,j)$, $E(i,j)$ and $W(i,j)$ regions is analogous). For each position in the bottom row of an (a,b)-block with $A'[a,b] = \mathbf{1}$, we store two bits indicating whether its answer within the block is in (1) the same column (C), or (2) some column to the left (L), or (3) some column to the right (R). The query algorithm simply *follows* the L or R *pointers* till it reaches a C, and then scans the column upwards till it finds a **1** in that column. This takes $O(c \times n^2/c^2) = O(n^2/c)$ bits over all the blocks. This encoding enables us to find the closest **1** within the block from any column in the bottom row of that block in $O(c)$ time. Since ℓ is the L_1 distance between (i,j) and (i',j') in A',

we know that all the blocks $A[i, j-r]$, for $1 \leq r < \ell$ are empty (otherwise, we have a closer non-empty block than (i', j')). Let k be the column corresponding to the query position q. We claim that the closest **1** to q in the $N(i,j)$ region is closest **1** to the bottom row and column k of either the $(i, j+\ell)$-block or the $(i, j+\ell+1)$-block. These can be computed in $O(c)$ time using the above encoding, and then compared to find the required answer. Finally we can find NLN of q by comparing these eight candidate answers. $\square$

The optimality of the trade-off follows from the lower bound of the following lemma.

Lemma 4.2 ([44]). *Given a 1D array of size n, any data structure which stores $O(n/c)$ bits and answers NLV (or NLN) queries in the indexing model, requires at least $\Omega(c)$ query time, for any $1 \leq c \leq n$.*

4.4 Encoding NLN queries for general 2D arrays

Consider an $n \times n$ 2D array $A[1 \dots n][1 \dots n]$. Given two positions (i,j) and (i', j') in A, we define $\mathsf{dist}((i,j),(i'j')) = |i-i'| + |j-j'|$. A trivial solution to the NLN problem in 2D array is to store $\mathsf{NLN}((i,j))$, for $1 \leq i, j \leq n$. This requires $O(n^2 \lg n)$ bits, and supports queries in $O(1)$ time. In the following, we obtain improved results for the 2D NLN in the encoding and indexing models, and also describe some trade-off results.

4.4.1 2D NLN in the encoding model – distinct case

When there is no restriction on the elements of the array, one can show an n^2-bit lower bound for NLN encoding (described in Section 4.4.2). Using a simple encoding method, one can prove that the same asymptotic lower bound applies even when all the elements of the array are distinct to obtain the following.

X	O	X	O	X	O
X	O	X	O	X	O
X	X	X	X	X	X
X	O	X	O	X	O
X	O	X	O	X	O
X	X	X	X	X	X

O : Useful

X : Dummy

Figure 4.2 The positions of *useful* and *dummy* elements in a 6×6 array. In this example, the dummy elements (X's) are in the range [1..24] and the useful elements (O's) are in the range [25..26].

Theorem 4.4. *Any data structure which supports* NLN *queries on an $n \times n$ array $A[1 \ldots n][1 \ldots n]$ in encoding model requires at least $n^2/6$ bits, even when all the elements in A are distinct.*

Proof. Without loss of generality, we assume that n is a multiple of 6. We first define a set $\mathcal{A}$ of $2^{n^2/6}$ 2D arrays, and then show that the answers to the NLN queries in any array $A \in \mathcal{A}$ can be used to distinguish A from $\mathcal{A} \setminus \{A\}$. This proves that encoding for an arbitrary array in $\mathcal{A}$ requires at least $\lg(|\mathcal{A}|) = n^2/6$ bits in the worst case.

Each array $A \in \mathcal{A}$ contains elements from the set $\{1, 2, \ldots, n^2\}$, where each element appears in the array exactly once. To describe the arrays in $\mathcal{A}$, we partition the elements of each array into *useful* and *dummy* elements. The positions $(3i + 1, 2j)$ and $(3i + 2, 2j)$, for $0 \leq i < n/3$ and $1 \leq j \leq n/2$, contain useful elements, and the remaining $2n^2/3$ positions contain the dummy elements (see Figure 4.2). We assign the elements from 1 to $2n^2/3$ to the positions corresponding to the dummy elements, in row-major order. Also, we first assign the elements from $2n^2/3 + 1$ to n^2 to the positions corresponding to the useful

elements, in row-major order. Let A_0 denote this array. We now obtain the $2^{n^2/6}$ arrays in $\mathcal{A}$ by repeatedly taking a pair of adjacent useful elements and flipping them.

Consider any two arrays A and A' in $\mathcal{A}$. We know that for at least one pair of adjacent positions $(3i+1, 2j)$ and $(3i+2, 2j)$, for some $0 \leq i < n/3$ and $1 \leq j \leq n/2$, we will have $A[3i+1, 2j] < A[3i+2, 2j]$ while $A'[3i+1, 2j] > A'[3i+2, 2j]$ or vice versa, and hence their NLN answers are distinct. Therefore, given the answers to the NLN queries of all adjacent pairs of useful elements, we can distinguish the array A from $\mathcal{A} \setminus \{A\}$. □

We now obtain an asymptotically optimal upper bound for 2D NLN encoding for the distinct case. The proof is based on ideas from Asano and Kirkpatrick [4].

Lemma 4.3. *A 2D array $A[1 \ldots n][1 \ldots n]$ can be encoded using $O(n^2)$ bits to support NLN queries, provided all elements are distinct.*

Proof. The main idea is to divide the array recursively into blocks of geometrically increasing size, and store the NLN values of all elements, except the largest element and the elements whose answers are stored at a previous level, in each block explicitly. The following argument shows that this requires $O(n^2)$ bits overall.

In the first level, we divide A into $n^2/4$ blocks of size 2×2 each. Except for the largest element in each 2×2 block, the distance of NLN answer for the other three elements are bounded by 2. In general, at level k, we divide A into $n^2/4^k$ blocks of size $2^k \times 2^k$ each. In each of these $2^k \times 2^k$-sized blocks, there are four elements left for which we need to store the answer to their NLN queries. For three of these four elements, which do not correspond to the maximum value in the block, we store their answers at level k. Since the distance to the NLN answer for these three elements is bounded by 2^{k+2}, we can store these answers using $O(k)$ bits. Thus the total space usage is bounded by $\sum_{k=1}^{\lg n}(3n^2/4^k) * O(k) = O(n^2)$ bits. □

We now describe another $O(n^2)$-bit encoding for the 2D NLN problem that supports queries in constant time when the elements are distinct.

Theorem 4.5. *A 2D array $A[1 \ldots n][1 \ldots n]$ can be encoded using $O(n^2)$ bits to support NLN queries in $O(1)$ time, provided all elements are distinct.*

Proof. The encoding is a small variant of the encoding described in the proof of Lemma 4.3. For each position in A, in some canonical order (say, row-major order), we write down the relative position (i.e., the distance from the position to its answer in horizontal and vertical directions) of its NLN answer. We use a variable-length encoding, such as γ-code or δ-code [21], to write these answers. The proof of Lemma 4.3 implies that the sum of the lengths of all these answers is $O(n^2)$. We also store an indexable bit vector [69] indicating the starting positions of each code. This enables us to find the position where the answer to a given query starts and ends, in constant time. □

4.4.2 2D NLN in the encoding model – general case

It is easy to see that for any two distinct $n \times n$ binary arrays can be distinguished by looking at the NLN answers at every positions. In other words, any two distinct binary arrays must have distinct NLN encodings. This shows an n^2-bit lower bound for NLN encoding in the general case, In this section, we give an encoding which supports NLN queries in a 2D array with $O(n^2)$ bits in the general case. Before starting the 2D case, we consider the 1D case first. Jayapaul et al. [44] showed how to encode an array A with n distinct elements using $O(n)$ bits to answer NLN queries. We give an alternate proof, that is similar to the proof of Lemma 4.3.

Lemma 4.4. *There exists an encoding of an array $A[1 \ldots\ n]$ that uses $O(n)$ bits while supporting NLN queries, provided all elements are distinct.*

Proof. We write down the sequence $d(1), d(2), \ldots, d(n)$ explicitly, where $d(i) = n$ if $A[i]$ is the maximum element of A, and $d(i) = |i - \mathsf{NLN}(i)|$ otherwise,

for $1 \le i \le n$, together with a sequence of n bits that indicate if $i < \mathsf{NLN}(i)$ or $i > \mathsf{NLN}(i)$. Thus, it is enough to show that $\sum_{i=1}^{n} \lg d(i) = O(n)$. Since all the elements in A are distinct, there are at most $n/2^k$ elements for which $d(i) \ge 2^k$, for any $1 \le k \le \lg n$. From this observation, it follows that there are $O(n/2^k)$ elements for which $2^k \le d(i) < 2^{k+1}$, and hence $\sum_{i=1}^{n} \lg d(i) \le \sum_{k=1}^{\lg n}(O(n/2^k) \cdot O(k)) = O(n)$. □

We now describe a simple modification of the above encoding that can be used to support NLN queries even when the elements are not distinct. Queries are not supported in constant time with this encoding. Note that one can use the encoding of Fischer [25] to obtain a linear-bit (in fact, a $2.54n$-bit) encoding which supports NLN queries in constant time. However, in contrast to Fischer's encoding, the new approach stores explict pointers from one array position to another, and we use the space cost of these explicit pointers to upper bound the space usage of the pointers stored in the proof of Theorem 4.6.

Instead of encoding the NLN of a position i as in Lemma 4.4, we encode the distance between i and the nearest value which is $\ge A[i]$ in the same direction as $\mathsf{NLN}(i)$. Formally, we define $d_l(i) = i - (\max_{j<i, A[j] \ge A[i]} j)$ and $d_r(i) = (\min_{j>i, A[j] \ge A[i]} j) - i$ and $d(i) = d_l(i)$ if $\mathsf{NLN}(i) < i$ and $d(i) = d_r(i)$ otherwise. For each i, we encode $d(i)$ (using a variable-length encoding) and store a bit indicating whether $d(i) = d_r(i)$ or $d(i) = d_l(i)$, and view this as a "pointer" to $j = i + d_r(i)$ or $j = i - d_l(i)$ respectively. Finally, we also store a bit indicating whether or not $A[i] = A[j]$. With this encoding, $\mathsf{NLN}(i)$ can be easily found by following the $d(\cdot)$ "pointers" from i until we reach a position that is greater than $A[i]$. We refer to this encoding of a 1D array as $encoding_{1D}$.

The following lemma shows that this encoding uses $O(n)$ bits[1]:

Lemma 4.5. *For a 1D array $A[1 \dots n]$, encoding$_{1D}$ takes $O(n)$ bits.*

Proof. For A, *encoding*$_{1D}$ consists of two bit strings of length $O(n)$, and a sequence of variable-length encodings storing the values $d(1), d(2), \dots, d(n)$. Let $D = \sum_{i=1}^{n} \lg d(i)$. To prove the lemma, it is enough to show that $D = O(n)$.

We first create a new array A' with all distinct elements, and bound the value D using the size of the NLN encoding of A'. Consider the array $A'[1 \dots n]$ of size n, where $A'[i] = A[i] + \epsilon i$ if $\mathsf{NLN}(i) > i$ and $A'[i] = A[i] - \epsilon i$ if $\mathsf{NLN}(i) < i$ for some $\epsilon > 0$. If we set ϵ small enough then if $A[i] > A[j]$ for some i, j then $A'[i] > A'[j]$ as well, but all elements in A' are distinct. So if we define $d'(i)$, NLN' and D' on A' analogously to $d(i)$, NLN, and D on A, $D' = \sum_{i=1}^{n} \lg d'(i) = O(n)$ by Lemma 4.4. We now show that $D \leq 2D'$.

For a subset S of $U = \{1, 2, \dots, n\}$, we define D_S as $\sum_{i \in S} \lg d(i)$, (and D'_S analogously). To prove that $D \leq 2D'$, we partition the set U into disjoint subsets, and show that $D_S \leq 2D'_S$ for every subset S in the partition. To define the partitions of U, we first extend the array A such that $A[0] = A[n+1] = \infty$. Now, each subset in the partition of U contains a set of indices $i_1, \dots i_{r-1}$ where $0 \leq i_0 < i_1 < \cdots < i_r \leq n+1$ with $r > 1$ is a maximal sequence of indices such that $A[i_0] > A[i_1]$, $A[i_{r-1}] < A[i_r]$, $A[i_1] = A[i_2] = \cdots = A[i_{r-1}]$ and for all $i_0 < j < i_r$, $A[j] < A[i_1]$ if $j \notin \{i_1, \dots, i_{r-1}\}$. It is easy to show that this collection of subsets form a partition of U, i.e., they are pairwise disjoint and cover U.

Let i_k be the index such that $\mathsf{NLN}(i_l) = i_0$ for all $0 < l \leq k$ and $\mathsf{NLN}(i_l) = i_r$ for all $k < l \leq r-1$. Then by the definition of A', for all $k < l \leq r-1$,

[1] Note that this encoding cannot be obtained by simply breaking ties among equal elements in some arbitrary fashion and applying Lemma 4.4. For example, if $A[i] = A[i+1]$ and $A[i-t]$ and $A[i+1+t]$ for some $t > 1$ are the nearest larger values, then in the current encoding, neither $A[i]$ nor $A[i+1]$ would point to one another. If we break ties then either $A[i]$ points to $A[i+1]$ or $A[i+1]$ points to $A[i]$.

$\mathsf{NLN}'(i_l) = i_{l+1}$ so $d(i_l) = d'(i_l)$. For the elements to the left of i_k, we can consider the case that there exist $0 < m \le k$ such that $\mathsf{NLN}'(i_l) = i_{l-1}$ for all $0 < l \le m-1$ and $\mathsf{NLN}'(i_l) = i_{k+1}$ for $m \le l \le k$. Then:

$$
\begin{aligned}
D_S - D'_S = & \sum_{j=1}^{r-1} \lg d(i_j) - \sum_{j=1}^{r-1} \lg d'(i_j) \\
= & \left(\sum_{j=1}^{m-1} \lg d(i_j) + \sum_{j=m}^{k} \lg(i_j - i_{j-1}) + \sum_{j=k+1}^{r-1} \lg d(i_j) \right) \\
- & \left(\sum_{j=1}^{m-1} \lg d'(i_j) + \sum_{j=m}^{k} \lg(i_{k+1} - i_j) + \sum_{j=k+1}^{r-1} \lg d'(i_j) \right) \\
= & \sum_{j=m}^{k} \lg(i_j - i_{j-1}) - \sum_{j=m}^{k} \lg(i_{k+1} - i_j) \\
\le & \lg(i_m - i_{m-1}) - \lg(i_{k+1} - i_k) \\
& (\because i_j - i_{j-1} \le i_{k+1} - i_{j-1} \text{ for all } m \le j \le k) \\
\le & \lg(i_m - i_{m-1}) \le \lg(i_m - i_0) \le \lg(i_r - i_m) \quad (\because \mathsf{NLN}(i_m) = i_0) \\
\le & \lg(i_{k+1} - i_m) + \sum_{j=k+1}^{r-1} \lg(i_{j+1} - i_j) \text{ (by the concavity of lg function)} \\
\le & \sum_{j=1}^{r-1} \lg d'(i_j) = D'_S
\end{aligned}
$$

□

We now extend this encoding to encode NLNs for a 2D array $A[1 \ldots n][1 \ldots n]$ that answers NLN queries. We call this encoding scheme *encoding*$_{2D}$. We then show that *encoding*$_{2D}$ takes $O(n^2)$ bits (in Theorem 4.6).

In *encoding*$_{2D}$, each (i, j) "points to" another location (i', j'), such that $A[i', j'] \ge A[i, j]$, as follows: $|i - i'|$ is encoded using $O(1 + \lg |i' - i|)$ (the *row cost* of the pointer) and $|j - j'|$ is coded using $O(1 + \lg |j' - j|)$ bits (the *column cost* of the pointer), the direction from (i, j) to (i', j') is given using two bits, and finally one extra bit indicates whether or not $A[i', j'] > A[i, j]$. Now we explain how to specify the pointers. Pick an element $A[i, j]$ and without loss

of generality assume that $\mathsf{NLN}(i,j) = (i^*, j^*)$ with $i^* \geq i, j^* \geq j$. We choose pointers as follows:

Case (1) Let $i' > i$ be the smallest value such that $i' \leq i^*$ and $A[i,j] = A[i',j]$. If i' exists, then we store a pointer from (i,j) to (i',j) and set the extra bit to 0.

Case (2) If there exists no i' such that $A[i,j] = A[i',j]$. for $i < i' \leq i^*$, then let $j' > j$ be the smallest value such that $j' \leq j^*$ and $A[i,j] = A[i,j']$. If j' exists, we store a pointer from (i,j) to (i,j') and set the extra bit to 0.

Case (3) If there exists no i' such that $A[i,j] = A[i',j]$. for $i < i' \leq i^*$, and also if there exists no j' such that $A[i,j] = A[i,j']$. for $j < j' \leq j^*$, then we store a pointer from (i,j) to (i^*,j^*) and set the extra bit to 1.

To obtain $\mathsf{NLN}(i,j)$, we follow pointers starting from (i,j) until we follow one with the extra bit set to 1, and return the position pointed to by this pointer.

We now show that the above procedure computes $\mathsf{NLN}(i,j)$, for all $1 \leq i,j \leq n$. The proof is by induction on k, the distance between (i,j) and $\mathsf{NLN}(i,j) = (i^*,j^*)$. The base case $k=1$ follows directly from case 3) above.

Assume the induction hypothesis holds for all NLNs at distance $\leq k$, and choose an (i,j) such that $\mathsf{NLN}(i,j) = (i^*,j^*)$ and $dist((i,j),(i^*,j^*)) = k+1$. Assume, without loss of generality, that the pointer from (i,j) has its extra bit set to 0 (otherwise, the induction step is trivial) and it points to (i',j) with $i' > i$. Assume that $\mathsf{NLN}(i',j) = (x,y) \neq (i^*,j^*)$, and $dist((x,y),(i,j))$ is greater than $dist((x,y),(i^*,j^*))$. Since $A[i',j] = A[i,j]$, $dist((i',j),(x,y)) \leq dist((i',j),(i^*,j^*)) < dist((i,j),(i^*,j^*)) = k+1$. By the induction hypothesis, following pointers from (i',j) leads to (x,y). Now:

$$\begin{aligned}
dist((i,j),(x,y)) &= dist((i,j),(i',j)) + dist((i',j),(x,y)) \\
&\leq dist((i,j),(i',j)) + dist((i',j),(i^*,j^*)) \ (\because \mathsf{NLN}(i',j) = (x,y)) \\
&= dist((i,j),(i^*,j^*)),
\end{aligned}$$

contradicting the assumption that $dist((x,y),(i,j)) > dist((x,y),(i^*,j^*))$.

Theorem 4.6. *There exists an encoding of a 2D array* $A[1 \dots n][1 \dots n]$ *that supports* NLN *queries, using* $O(n^2)$ *bits.*

Proof. We show that $encoding_{2D}$, described earlier, takes $O(n^2)$ bits. To upper bound the space, we first describe an encoding, called $encoding_{grid}$ as follows. We encode each column and each row of A using $encoding_{1D}$, using $O(n^2)$ bits. These pointers are called *grid* pointers. However, the maximal values in each row and column do not have pointers by Lemma 4.5, as their NLN is not defined. So, in addition, for each row r which has (locally) maximum values in columns $i_1 < \dots < i_k$, we store *extra* pointers from (i_j, r) to (i_{j+1}, r) and vice versa for $j = 0, \dots, k$, taking $i_0 = 0$ and $i_{k+1} = n+1$. The space taken by these extra pointers is $O(\lg i_1 + \sum_{j=2}^{k-1} \lg(i_j - i_{j-1}) + \lg(n+1-i_k)) = O(n)$ bits for row r. We do this for all rows and columns, at a cost of $O(n^2)$ bits overall.

Although $encoding_{grid}$ does not encode NLN, we use it to upper bound the space used by $encoding_{2D}$. Let a *grid pointer* and a *2D pointer* refer to a pointer in $encoding_{grid}$ and $encoding_{2D}$ respectively. For any 2D pointer, the cost of encoding it can be upper-bounded by the cost of encoding (one or more) grid pointers. Each grid pointer will be used $O(1)$ times this way. Below, we show how to upper bound all Case (2) 2D pointers and the column cost of all Case (3) 2D pointers by grid pointers in rows, using each grid pointer at most thrice. The costs of Case (1) 2D pointers and the column cost of Case (3) 2D pointers can similarly be bounded by the costs of grid pointers in the columns. This will prove the theorem.

We consider a fixed location (i,j), and assume wlog that $\mathsf{NLN}(i,j) = (i^*, j^*)$ with $i^* \geq i$ and $j^* > j$ (if $j^* = j$ then the pointer from (i,j) will have column distance 0 and there is nothing to bound). There are four cases to consider (see Figure 4.3).

Case (a) Let $j' > j$ be the minimum index such that $A[i,j'] \geq A[i,j]$. Suppose

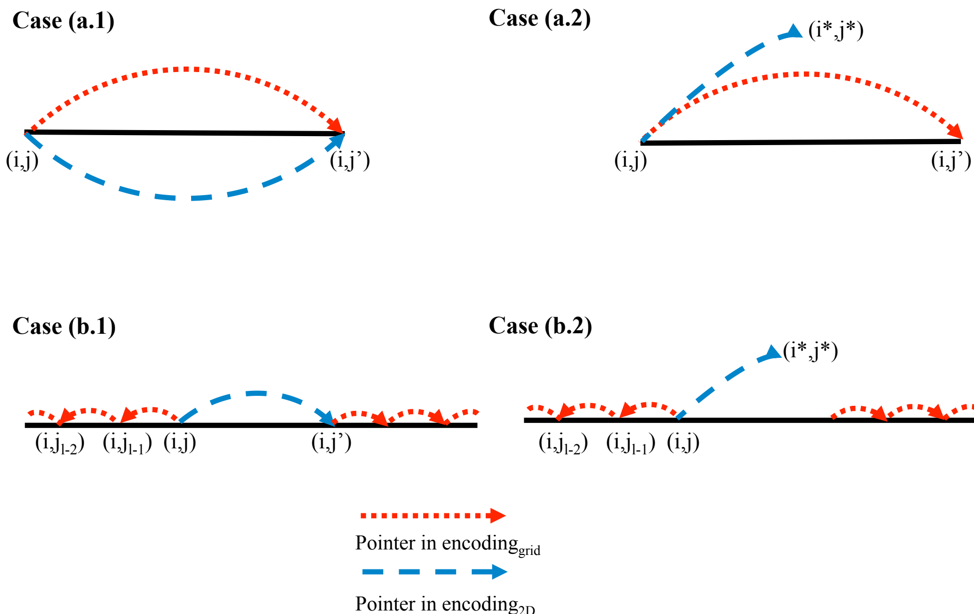


Figure 4.3 Pointers in $encoding_{2D}$ and $encoding_{grid}$

that j' exists and there is a grid pointer from (i, j) to (i, j') or vice versa. There are two sub-cases:

(a.1) The 2D pointer from (i, j) points to (i, j'). We use the cost of this grid pointer to upper bound the cost of the 2D pointer. Observe that if there is a 2D pointer from (i, j) to (i, j'), there cannot be a 2D pointer from (i, j') to (i, j), so the grid pointer is used for upper-bounding only once in this case.

(a.2) The 2D pointer from (i, j) points to (i^*, j^*). Observe that $j' \geq j^*$, since otherwise either (i, j') is a larger value that is closer than (i^*, j^*), a contradiction, or we would have a Case (2) 2D pointer from (i, j) to (i, j'). The pointer between (i, j) and (i, j') will only be charged twice for upper-bounding in this case.

Case (b) Either (i) $A[i, j] > A[i, j']$ for all $j' > j$, or (ii) there exists a $j' > j$

such that $A[i,j] \leq A[i,j']$, and there are no grid pointers either from (i,j) to (i,j') or vice versa. As before, we consider two sub-cases.

(b.1) Suppose the 2D pointer from (i,j) points to (i,j'), where $j' > j$ is the smallest index such that $A[i,j'] \geq A[i,j]$. If $A[i,j]$ is a maximal value in row i, the cost of the pointer is upper-bounded by the extra pointer between (i,j) and (i,j'). If not, the absence of grid pointers between (i,j) and (i,j') implies that the NLN of (i,j) in the i-th row is (i,j_0) for some $j_0 < j$. Note that $|j_0 - j| \geq \mathsf{dist}((i,j),(i^*,j^*))$ (otherwise $\mathsf{NLN}(i,j)$ would be (i,j_0)). The path p between (i,j) and (i,j_0) in $encoding_{grid}$ may comprise a number of grid edges. We can bound the cost of the 2D edge from (i,j) to (i,j') by the total cost of the grid edges on the path p consisting of the elements $j = j_l, j_{l-1}, \ldots, j_1, j_0$ (omitting the row number for brevity).[2] Note that for any $0 < k < l$, no 2D pointer from (i,j_k) can end up in Case (b), so this path can only be used twice to upper-bound the cost of a 2D edge: once from (i,j) and once (possibly) from (i,j_0).

(b.2) Suppose the 2D pointer from (i,j) points to (i^*,j^*). If $A[i,j]$ is a maximal value in row i, then if j' exists, then it must be the case that $j' > j^*$, and the row cost of the 2D pointer is bounded by the extra pointer between (i,j) and (i,j'). On the other hand, if j' does not exist, then the row cost of the 2D pointer is bounded by the extra pointer from (i,j) to $(i,n+1)$.

If $A[i,j]$ is not maximal, then arguing as above, we see that the NLN of (i,j) in the i-th row is (i,j_0) for some $j_0 < j$, that $|j_0 - j| \geq |j - j^*|$, and so we can upper-bound the row cost of this 2D pointer by the total cost of all the grid pointers between j and j_0, and each of these

[1]Since the log function is concave, the sum of the costs of the path p is no less than the cost of a single edge from (i,j) to (i,j_0).

grid pointers is used at most twice (once each for the pointers out of (i, j) and (i, j_0) in Case (b) to upper bound a 2D pointer.

□

We now describe an $O(n^2)$-bit encoding that supports NLN queries in constant time on a 2D array.

Theorem 4.7. *There exists an encoding of a 2D array $A[1 \ldots n][1 \ldots n]$ that uses $O(n^2)$ bits while supporting NLN queries in $O(1)$ time.*

Proof. We first divide A into blocks of size $b \times b$, and divide each block into sub-blocks of size $s \times s$. For each position (i, j) in the array, we say that the four locations $(i+1, j)$, $(i-1, j)$, $(i, j+1)$ and $(i, j-1)$ (even if they are outside the array range) are its *neighbors*. We say that a location (i, j) is a boundary location with respect to a block (sub-block) if one of its neighbors is not in the same block (sub-block). Note that there are $O(b)$ ($O(s)$) boundary elements in each block (sub-block). For each block, we store a $b \times b$ bitmap of size b^2 bits, such that the (i, j)-th bit, for $1 \leq i, j \leq b$, stores a 1 if the corresponding element in that position is a maximum element in that block, and stores a 0 otherwise. We also store similar bitmaps for each sub-block, using s^2 bits for each sub-block.

For each boundary position (i, j) in a block B, we store the nearest position to (i, j) whose value is larger than the maximum element in B. This takes $O(b \lg n)$ bits for each block. Also, for a sub-block B' in a block B, if B' does not contain the maximum element in B, then for each boundary position (i, j) in B', we store the nearest position to (i, j) whose value is larger than the maximum element in B'. Since the distance to this position is at most $2b$, it takes $O(s \lg b)$ bits for each sub-block to store this information.

Finally, for a position (i, j) in a sub-block B', if $A[i, j]$ is not the maximum element in B', then its NLN always exists in the sub-array A' of A, of size

$5s \times 5s$ such that $B' = A'[2s+1, \ldots, 3s][2s+1, \ldots, 3s]$ (i.e., B' is the center sub-block in the $5s \times 5s$ sub-array A'). Theorem 4.6 shows that this sub-array A' can be encoded using λs^2 bits, for some positive constant λ, to support NLN queries. For each sub-block B' in A, we store the encoding (of Theorem 4.6) of the corresponding sub-array A'. Over all the sub-blocks, this takes $O(n^2)$ bits.

In addition, we construct a precomputed table which we store as a two-dimensional array. The first dimension is indexed by all possible bit-strings of length λs^2, and the second dimension is indexed by all possible positions in a sub-block. The (e, p)-th entry in this array stores the NLN of the position p in sub-block B' within the $5s \times 5s$ sub-array A' (with B as its center sub-block) whose encoding is the bit string e.

We now describe the query algorithm. Consider the query position (i, j), and let B (B') be the block (sub-block) that contains (i, j). We first check whether (i, j) is a position of the maximal element in B' in $O(1)$ time using the bitmap defined above. If $A[i, j]$ is not a maximal element in B', then we use the precomputed table to find the answer, in $O(1)$ time. If $A[i, j]$ is the maximum element in B' but not in B, then we can answer the query in $O(1)$ time by comparing the distance between (i, j) and stored positions on the four boundary positions in B' which have the same row or the column positions as (i, j) and choose the nearest one. If $A[i, j]$ is a maximal element in B, we can find its NLN in $O(1)$ time by a similar procedure as above case, by looking at the four boundary positions in B with same row or column index as (i, j). Thus, in all cases, the queries can be supported in $O(1)$ time. The overall space usage is $O(n^2/b^2 \times (b^2 + b \lg n) + n^2/s^2 \times (s^2 + s \lg b) + s^2 2^{\lambda s^2} \lg s^2)$ bits. By choosing $b = \lg n$ and $s = c\sqrt{\lg n}$, for some small constant c (chosen appropriately), the overall space usage becomes $O(n^2)$ bits. □

4.5 Open problems

Our main contribution is a systematic study of data structures for NLV on 1D arrays in the indexing model, and NLN on 2D arrays in the encoding model.

We suggest the following open problems for future works.

- Is there a data structure that takes less than $2.54n + o(n)$ bits and can answer NLN queries in a one dimensional array in constant time in the general case (when elements may repeat) in the encoding model?
- For a 1D array, is there an index for NLV that uses $O(n/c)$ bits and supports queries in $O(c)$ query time?
- For a 2D array, is there an index for NLN that uses $O(n^2/c)$ bits and supports queries in $O(c)$ query time?

Chapter 5

Simultaneous encodings for range and next/previous larger/smaller value queries

5.1 Introduction

Given an array $A[1 \ldots n]$ of n elements from a total order. For $1 \leq i \leq j \leq n$, suppose that there are m (l) positions $i \leq p_1 \leq \cdots \leq p_m \leq j$ ($i \leq q_1 \leq \cdots \leq q_l \leq j$) in A which are the positions of minimum (maximum) values between $A[i]$ and $A[j]$. Then we can define various range minimum (maximum) queries as follows.

- Range Minimum Query ($\mathsf{RMinQ}_A(i,j)$) : Return an arbitrary position among $p_1, \ldots, p_m$.

- Range Leftmost Minimum Query ($\mathsf{RLMinQ}_A(i,j)$) : Return p_1.

- Range Rightmost Minimum Query ($\mathsf{RRMinQ}_A(i,j)$) : Return p_j.

- Range k-th Minimum Query ($\mathsf{RkMinQ}_A(i,j)$) : Return p_k (for $1 \leq k \leq m$).

- Range Maximum Query ($\mathsf{RMaxQ}_A(i,j)$) : Return an arbitrary position among $q_1, \ldots, q_l$.
- Range Leftmost Maximum Query ($\mathsf{RLMaxQ}_A(i,j)$) : Return q_1.
- Range Rightmost Maximum Query ($\mathsf{RRMaxQ}_A(i,j)$) : Return q_l.
- Range k-th Maximum Query ($\mathsf{RkMaxQ}_A(i,j)$) : Return q_k (for $1 \le k \le l$).

Also for $1 \le i \le n$, we consider following additional queries on A.

- Previous Smaller Value ($\mathsf{PSV}_A(i)$) : $\max(j : j < i, A[j] < A[i])$.
- Next Smaller Value ($\mathsf{NSV}_A(i)$) : $\min(j : j > i, A[j] < A[i])$.
- Previous Larger Value ($\mathsf{PLV}_A(i)$) : $\max(j : j < i, A[j] > A[i])$.
- Next Larger Value ($\mathsf{NLV}_A(i)$) : $\min(j : j > i, A[j] > A[i])$.

For define above four queries formally, we assume that $A[0] = A[n+1] = -\infty$ for $\mathsf{PSV}_A(i)$ and $\mathsf{NSV}_A(i)$. Similarly we assume that $A[0] = A[n+1] = \infty$ for $\mathsf{PLV}_A(i)$ and $\mathsf{NLV}_A(i)$.

Our aim is to obtain space-efficient encodings that support these queries efficiently.

Previous Work The range minimum/maximum problem has been well-studied in the literature. It is well-known [5] that finding RMinQ_A can be transformed to the problem of finding the LCA (Lowest Common Ancestor) between (the nodes corresponding to) the two query positions in the Cartesian tree constructed on A. Furthermore, since different topological structures of the Cartesian tree on A give rise to different set of answers for RMinQ_A on A, one can obtain an information-theoretic lower bound of $2n - \Theta(\lg n)$ bits on the encoding of A that answers RMinQ queries. Sadakane [71] proposed the $4n + o(n)$-bit encoding with constant query time for RMinQ_A problem using the balanced parentheses (BP) [55] of the Cartesian tree of A with some additional nodes.

Fischer and Heun [27] introduced the $2d$-Min heap, which is a variant of the Cartesian tree, and showed how to encode it using the Depth first unary degree sequence (DFUDS) [6] representation in $2n + o(n)$ bits which supports RMinQ_A queries in constant time. Davoodi et al. show that same $2n + o(n)$-bit encoding with constant query time can be obtained by encoding the Cartesian trees.[16]. For RkMinQ_A, Fischer and Heun [26] defined the *approximate range median of minima query* problem which returns a position RkMinQ_A for some $\frac{1}{16}m \leq k \leq \frac{15}{16}m$, and proposed an encoding that uses $2.54n + o(n)$ bits and supports the *approximate* RMinQ_A queries in constant time, using a *Super Cartesian tree.*

For PSV_A and NSV_A, if all elements in A are distinct, then $2n + o(n)$ bits are enough to answer the queries in constant time, by using the $2d$-Min heap of Fischer and Heun [27]. For the general case, Fischer [25] proposed the *colored $2d$-Min heap*, and proposed an optimal $2.54n + o(n)$-bit encoding which can answer PSV_A and NSV_A in constant time.

One can support both RMinQ_A and RMaxQ_A in constant time trivially using the encodings for RMinQ_A and RMaxQ_A queries, using a total of $4n + o(n)$ bits. Gawrychowski and Nicholson reduce this space to $3n + o(n)$ bits while maintaining constant time query time [30]. Their scheme also can support PSV_A and PLV_A in constant time when there are no consecutive equal elements in A.

Our results In this chapter, we first extend the original DFUDS [6] for colored 2d-Min(Max) heap that supports the queries in constant time. Then, we combine the extended DFUDS of $2d$-Min heap and $2d$-Max heap using Gawrychowski and Nicholson's Min-Max encoding [30] with some modifications. As a result, we obtain the following non-trivial encodings that support a wide range of queries.

Theorem 5.1. *An array $A[1 \ldots n]$ containing n elements from a total order can be encoded using*

(a) at most $3.17n+o(n)$ *bits to support* RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A, *and* PLV_A *queries;*

(b) at most $3.322n+o(n)$ *bits to support the queries in (a) in constant time;*

(c) at most $4.088n+o(n)$ *bits to support* RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A, NSV_A, RMaxQ_A, RRMaxQ_A, RLMaxQ_A, RkMaxQ_A, PLV_A *and* NLV_A *queries; and*

(d) at most $4.585n+o(n)$ *bits to support the queries in (c) in constant time.*

If the array contains no two consecutive equal elements, then (a) and (b) take $3n+o(n)$ *bits, and (c) and (d) take* $4n+o(n)$ *bits.*

This chapter organized as follows. Section 5.2 introduces various data structures that we use later in our encodings. In Section 5.3, we describe the encoding of colored $2d$-Min heap by extending the DFUDS of $2d$-Min heap. This encoding uses a distinct approach from the encoding of the colored $2d$-Min heap by Fischer [25]. Finally, in Section 5.4, we combine the encoding of this colored $2d$-Min heap and Gawrychowski and Nicholson's Min-Max encoding [30] with some modifications, to obtain our main result (Theorem 5.1).

5.2 Preliminaries

We first introduce some useful data structures that we use to encode various bit vectors and balanced parenthesis sequences.

Balanced parenthesis sequences Given a string $S[1\dots n]$ over the alphabet $\Sigma = \{'(', ')'\}$, if S is balanced and $S[i]$ is an open (close) parenthesis, then we can define $\mathsf{findopen}_S(i)$ ($\mathsf{findclose}_S(i)$) which returns the position of the matching close (open) parenthesis to $S[i]$. Now we introduce the lemma from Munro and Raman [55].

Lemma 5.1 ([55]). *Let S be a balanced parenthesis sequence of length n. If one can access any $\lg n$-bit subsequence of S in constant time, Then both $\mathsf{findopen}_S(i)$ and $\mathsf{findclose}_S(i)$ can be supported in constant time with $o(n)$-bit additional space.*

Depth first unary degree sequence *Depth first unary degree sequence (DFUDS)* is one of the well-known methods for representing ordinal trees [6]. It consists of a balanced sequence of open and closed parentheses, which can be defined inductively as follows. If the tree consists of the single node, its DFUDS is '()'. Otherwise, if the ordinal tree T has k subtrees $T_1 \dots T_k$, then its DFUDS, D_T is the sequence $(^{k+1}\ )d_{T_1} \dots d_{T_k}$ (i.e., $k+1$ open parentheses followed by a close parenthesis concatenated with the 'partial' DFUDS sequences $d_{T_1} \dots d_{T_k}$) where d_{T_i}, for $1 \leq i \leq k$, is the DFUDS of the subtree T_i (i.e., D_{T_i}) with the first open parenthesis removed. From the above construction, it is easy to prove by induction that if T has n nodes, then the size of D_T is $2n$ bits. The following lemma shows that DFUDS representation can be used to support various navigational operations on the tree efficiently.

Lemma 5.2 ([1], [6], [43]). *Given an ordinal tree T on n nodes with DFUDS sequence D_T, one can construct an auxiliary structure of size $o(n)$ bits to support the following operations in constant time: for any two nodes x and y in T,*

- $\mathsf{parent}_T(x)$ *: Label of the parent node of node x.*
- $\mathsf{degree}_T(x)$ *: Degree of node x.*
- $\mathsf{depth}_T(x)$ *: Depth of node x (The depth of the root node is 0).*
- $\mathsf{subtree_size}_T(x)$ *: Size of the subtree of T which has the x as the root node.*
- $\mathsf{next_sibling}_T(x)$ *: The label of the next sibling of the node x.*
- $\mathsf{child}_T(x, i)$ *: Label of the i-th child of the node x.*
- $\mathsf{child_rank}_T(x)$ *: Number of siblings left to the node x.*
- $\mathsf{la}_T(x, i)$ *: Label of the level ancestor of node x at depth i.*
- $\mathsf{lca}_T(x, y)$ *: Label of the least common ancestor of node x and y.*

- *pre_rank*$_T(i)$: *The preorder rank of the node in* T *corresponding to* $D_T[i]$.
- *pre_select*$_T(x)$: *The first position of node with preorder rank* x *in* D_T.

We use the following lemma to bound the space usage of the data structures described in Section 5.4.

Lemma 5.3. *Given two positive integers* a *and* n*, and a nonnegative integer* $k \leq n$*,* $\lg \binom{n}{k} + a(n-k) \leq n \lg (2^a+1)$.

Proof. By raising both sides to the power of 2, it is enough to prove that $\binom{n}{k} 2^{(a(n-k))} \leq (2^a+1)^n$. We prove the lemma by induction on n and k. In the base case, when $n=1$ and $k=0$, the claim holds since $2^a < (2^a+1)$. Now suppose that $\binom{n'}{k'} 2^{a(n'-k')} \leq (2^a+1)^{n'}$ for all $0 < n' \leq n$ and $0 \leq k' \leq k$. Then

$$\binom{n+1}{k} 2^{a(n+1-k)} = \left(\binom{n}{k} + \binom{n}{k-1}\right) 2^{a(n+1-k)}$$
$$\leq 2^a(2^a+1)^n + (2^a+1)^n = (2^a+1)^{n+1} \text{ by induction hypothesis.}$$

Also by induction hypothesis,

$$\binom{n}{k+1} 2^{a(n-(k+1))} = \left(\binom{n-1}{k} + \binom{n-1}{k+1}\right) 2^{a(n-(k+1))}$$
$$\leq (2^a+1)^{n-1} \left(1 + \frac{\binom{n-1}{k+1}(2^{a(n-1-k)})}{(2^a+1)^{n-1}}\right)$$

Since $\binom{n-1}{k+1} 2^{a(n-1-k)} < 2^a(2^a+1)^{n-1} (\because (2^a+1)^{n-1} = \sum_{m=0}^{n-1} \binom{n-1}{m} 2^{a(n-1-m)})$,

$$(2^a+1)^{n-1} \left(1 + \frac{\binom{n-1}{k+1}(2^{a(n-1-k)})}{(2^a+1)^{n-1}}\right) < (2^a+1)^{n-1}(1+2^a) = (2^a+1)^n.$$

Therefore the above inequality still holds when $n' = n+1$ or $k' = k+1$, which proves the lemma. □

5.2.1 2d-Min heap

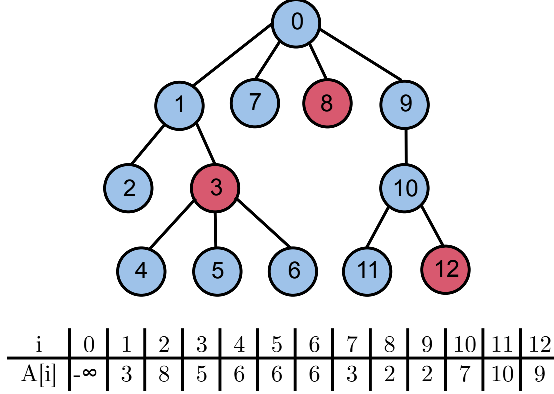
The 2d-Min heap [27] on A, denoted by $\mathsf{Min(A)}$, is designed to encode the answers of $\mathsf{RMinQ}_A(i,j)$ efficiently. We can also define the 2d-Max heap on A

(Max(A)) analogously. Min(A) is an ordered labeled tree with $n+1$ nodes labeled with $0 \dots n$. Each node in Min(A) is labeled by its preorder rank and each label corresponds to a position in A. We extend the array $A[1 \dots n]$ to $A[0 \dots n]$ with $A[0] = -\infty$. In the labeled tree, the node x denotes the node labeled x. For every vertex i, except for the root node, its parent node is (labeled with) $\mathsf{PSV}_A(i)$.

Using the operations in Lemma 5.2, Fischer and Heun [27] showed that $\mathsf{RMinQ}_A(i,j)$ can be answered in constant time using $D_{\mathsf{Min(A)}}$. If the elements in A are not distinct, $\mathsf{RMinQ}_A(i,j)$ returns the $\mathsf{RRMinQ}_A(i,j)$.

Fischer and Heun [27] also proposed a linear-time stack-based algorithm to construct $D_{\mathsf{Min(A)}}$. Their algorithm maintains a min-stack consisting of a decreasing sequence of elements from top to the bottom. The elements of A are pushed into the min-stack from right to left and before pushing the element $A[i]$, all the elements from the stack that are larger than $A[i]$ are popped. Starting with an empty string, the algorithm constructs a sequence S as described below. Whenever k elements are popped from the stack and then an element is pushed into the stack, $(^k)$ is prepended to S. Finally, after pushing $A[1]$ into the stack, if the stack contains m elements, then $(^{m+1})$ is prepended to S. One can show that this sequence S is the same as the DFUDS sequence $D_{\mathsf{Min(A)}}$. Analogously, one can construct $D_{\mathsf{Max(A)}}$ using a similar stack-based algorithm.

Colored 2d-Min heap From the definition of $2d$-Min heap, it is easy show that $\mathsf{PSV}_A(i)$, for $1 \le i \le n$, is the label corresponding to the parent of the node labeled i in Min(A). Thus, using the encoding of Lemma 5.2 using $2n+o(n)$ bits, one can support the $\mathsf{PSV}_A(i)$ queries in constant time. A straightforward way to support $\mathsf{NSV}_A(i)$ is to construct the $2d$-Min heap structure for the reverse of the array A, and encode it using an additional $2n+o(n)$ bits. Therefore one can encode all answers of PSV_A and NSV_A using $4n+o(n)$ bits with constant query time. To reduce this size, Fischer proposed the *colored $2d$-Min heap* [25]. This



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	-∞	3	8	5	6	6	6	3	2	2	7	10	9

Figure 5.1 Colored $2d$-Min heap of A

has the same structure as normal $2d$-Min heap, and in addition, the vertices are colored either red or blue. Suppose there is a parent node x in the colored $2d$-Min heap with its children $x_1 \dots x_k$. Then for $1 < i \leq k$, node x_i is colored red if $A[x_i] < A[x_{i-1}]$, and all the other nodes are colored blue (see Figure 5.1). We define the operation $\mathsf{NRS}(x_i)$ which returns the leftmost red sibling to the right (i.e., next red sibling) of x_i.

The following lemma can be used to support $\mathsf{NSV}_A(i)$ efficiently using the colored $2d$-Min heap representation.

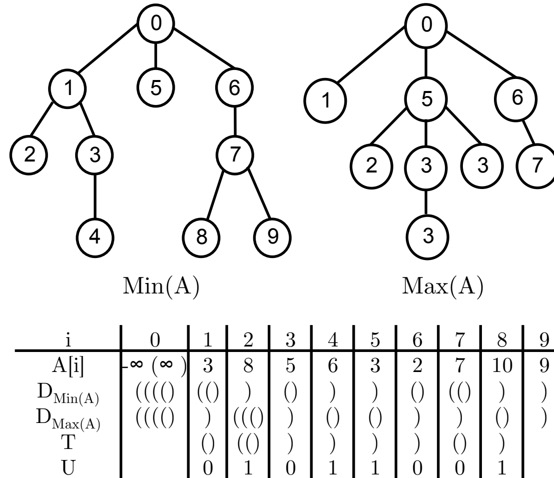
Lemma 5.4 ([25])**.** *Let CMin(A) be the colored $2d$-Min heap on A. Suppose there is a parent node x in CMin(A) with its children $x_1 \dots x_k$. Then for $1 \leq i \leq k$,*

$$\mathsf{NSV}_A(x_i) = \begin{cases} \mathsf{NRS}(x_i) & \text{if } \mathsf{NRS}(x_i) \text{ exists,} \\ x_k + \mathsf{subtree_size}(x_k) & \text{otherwise.} \end{cases}$$

If all the elements in A are distinct, then a $2n + o(n)$-bit encoding of $\mathsf{Min(A)}$ is enough to support RMinQ_A, PSV_A and NSV_A with constant query time. In the general case, Fischer proposed an optimal $2.54n + o(n)$-bit encoding of colored $2d$-Min heap on A using TC-encoding [22]. This encoding also supports two additional operations, namely *modified* $\mathsf{child}_{\mathsf{CMin(A)}}(x, i)$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}(x)$,

which answer the i-th red child of node x and the number of red siblings to the left of node x, respectively, in constant time. Using these operations, one can also support RLMinQ_A and RkMinQ_A in constant time.

5.2.2 Encoding range min-max queries



i	0	1	2	3	4	5	6	7	8	9
A[i]	-∞ (∞)	3	8	5	6	3	2	7	10	9
$D_{Min(A)}$	(((()	(()	)	()	)	)	()	(()	)	)
$D_{Max(A)}$	(((()	)	((()	)	()	()	)	)	()	)
T		()	(()	)	)	)	)	()	)	
U		0	1	0	1	1	0	0	1	

Figure 5.2 Encoding of $2d$-Min heap and $2d$-Max heap of A

One can support both RMinQ_A and RMaxQ_A in constant time by encoding both $\mathsf{Min(A)}$ and $\mathsf{Max(A)}$ separately using $4n + o(n)$ bits. Gawrychowski and Nicholson [30] described an alternate encoding that uses only $3n + o(n)$ bits while still supporting the queries in $O(1)$ time. There are two key observations which are used in obtaining this structure:

1. If we can access any $\lg n$-bit substring of $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ on $O(1)$ time, we can still support both queries in $O(1)$ time, using an additional $o(n)$ bits;

2. To generate $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ using Fischer and Heun's stack-based algorithm, in each step we push an element into both the min-stack and

max-stack, and pop a certain number of elements from exactly one of the stacks (assuming that $A[i] \neq A[i+1]$, for all i, where $1 \leq i < n$).

Now we describe the overall encoding in [30] briefly. The structure consists of two bit strings T and U along with various auxiliary structures. For $1 \leq i < n$, if k elements are popped from the min (max)-stack when we push $A[i] (1 \leq i < n)$ into both the stacks (from right to left), we prepend $(^{k-1}$) and 0(1) to the currently generated T and U respectively. Initially, when $i = n$, both min and max stacks push ')' so we do not prepend anything to both strings. But we can recover it easily because this is the last ')' in common. Finally, after pushing $A[1]$ into both the stacks, we pop the remaining elements from them, and store the number of these popped elements in min and max stack explicitly using $\lg n$ bits. One can show that the size of T is at most $2n$ bits, and that of U is $n - 1$ bits. Thus the total space usage is at most $3n$ bits. See Algorithm 1 for the pseudocode, and Figure 5.2 for an example.

To recover any $\lg n$-bit substring, $D_{\mathsf{Min(A)}}[d_1 \dots d_{\lg n}]$, in constant time we construct the following auxiliary structures. We first divide $D_{\mathsf{Min(A)}}$ into blocks of size $\lg n$, and for the starting position of each block, store its corresponding position in T. For this purpose, we construct a bit string B_{min} of length at most $2n$ such that $B_{min}[i] = 1$ if and only if $T[i]$ corresponds to the start position of the ith-block in $D_{\mathsf{Min(A)}}$. We encode B_{min} using the representation of Lemma 2.5 which takes $o(n)$ bits since the number of ones in B_{min} is $2n/\lg n$. Then if d_1 belongs to the i-th block, it is enough to recover the i-th and the $(i+1)$-st blocks in the worst case.

Now, to recover the i-th block of $D_{\mathsf{Min(A)}}$, we first compute the distance between i-th and $(i+1)$-st 1's in B_{min}. If this distance is less than $c \lg n$ for some fixed constant $c > 9$, we call it a *min-good block*, otherwise, we call it a *min-bad block*. We can recover a min-good block in $D_{\mathsf{Min(A)}}$ in $O(c)$ time using a $o(n)$-bit pre-computed table indexed by all possible strings of length $\lg n/4$ bits for T and U (we can find the position corresponding to the i-th block

Algorithm 1 Construction algorithm for T and U

```
1: Initialize T to ')', and U to ϵ.
2: Initialize Min-stack and Max-stack as empty stacks
3: Push A[n] into Min-stack and Max-stack.
4: for i := n − 1 to 1 do
5:     counter = 0
6:     if A[i] < A[i − 1] then
7:         Push A[i] into Max-stack
8:         while ((Min-stack is not empty) & (Top of Min-stack > A[i])) do
9:             Pop Min-stack
10:            counter = counter + 1
11:        end while
12:        Push A[i] into Min-stack
13:        Prepend (^(counter−1)) to T and 0 to U
14:    else // A[i] > A[i − 1]
15:        Push A[i] into Min-stack
16:        while ((Max-stack is not empty) & (Top of Max-stack < A[i])) do
17:            Pop Max-stack
18:            counter = counter + 1
19:        end while
20:        Push A[i] into Max-stack
21:        Prepend (^(counter−1)) to T and 1 to U
22:    end if
23: end for
```

in U in constant time), which stores the appropriate $O(\lg n)$ bits of $D_{\mathsf{Min(A)}}$ obtained from them (see [30] for details). For min-bad blocks, we store the answers explicitly. This takes $(2n/(c\lg n)) \cdot \lg n = 2n/c$ additional bits. To save this additional space, we store the min-bad blocks in compressed form using the property that any min-bad block in $D_{\mathsf{Min(A)}}$ and $D_{\mathsf{Max(A)}}$ cannot overlap more than $4\lg n$ bits in T, (since any $2\lg n$ consecutive bits in T consist of at least $\lg n$ bits from either $D_{\mathsf{Min(A)}}$ or $D_{\mathsf{Max(A)}}$). So, for $c > 9$ we can save more than $\lg n$ bits by compressing the remaining $(c-4)\lg n$ bits in T corresponding to each min-bad block in $D_{\mathsf{Min(A)}}$. Thus, we can reconstruct any $\lg n$-bit substring of $D_{\mathsf{Min(A)}}$ (and $D_{\mathsf{Max(A)}}$) in constant time, using a total of $3n + o(n)$ bits.

We first observe that if there is a position i, for $1 \leq i < n$ such that $A[i] = A[i+1]$, we cannot decode the '$)'$ in T which corresponds to $A[i]$ only using T and U since we do not pop any elements from both min and max stacks when we push $A[i]$ into both stacks. Gawrychowski and Nicholson [30] handle this case by defining an ordering between equal elements (for example, by breaking the ties based on their positions). But this ordering does not help us in supporting the PSV and PLV queries. We describe how to handle the case when there are repeated (consecutive) elements in A, to answer the PSV and PLV queries.

Gawrychowski and Nicholson [30] also show that any encoding that supports both RMinQ_A and RMaxQ_A cannot use less than $3n - \Theta(\lg n)$ bits for sufficiently large n (even if all elements in A are distinct).

5.3 Extended DFUDS for colored 2d-Min heap

In this section, we describe an encoding of colored $2d$-Min heap on A ($\mathsf{CMin(A)}$) using at most $3n + o(n)$ bits while supporting RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A and NSV_A in constant time. This is done by storing the color information of the nodes using a bit string of length at most n, in addition to the DFUDS representation of $\mathsf{CMin(A)}$. We can also encode the colored $2d$-Max

heap in a similar way. In the worst case, this representation uses more space than the colored $2d$-Min heap encoding of Fischer [25], but the advantage is that it separates the tree encoding from the color information. We later describe how to combine the tree encodings of the $2d$-Min heap and $2d$-Max heap, and (separately) also combine the color information of the two trees, to reduce the overall space.

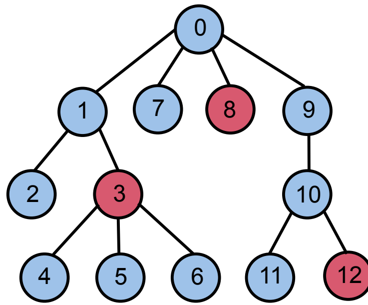
Now we describe the main encoding of $\mathsf{CMin(A)}$. The encoding consists of two parts: $D_{\mathsf{CMin(A)}}$ and V_{min}. The sequence $D_{\mathsf{CMin(A)}}$ is same as $D_{\mathsf{Min(A)}}$, the DFUDS representation of $\mathsf{CMin(A)}$, which takes $2n + o(n)$ bits and supports the operations in Lemma 5.2 in constant time.

The bit string V_{min} stores the color information of all nodes in $\mathsf{CMin(A)}$, except the nodes which are the leftmost children of their parents (the color of these nodes is always blue), as follows. Suppose there are p nodes in $\mathsf{CMin(A)}$, for $1 \leq p \leq n$, which are the leftmost children of their parents. Then we define the bit string $V_{min}[0 \ldots n-p]$ as follows. For $1 \leq i \leq n-p$, $V_{min}[i]$ stores 0 if the color of the node

$$\mathsf{node}_{V_{min}}(i) = \mathsf{pre_rank}_{\mathsf{CMin(A)}}(\mathsf{findclose}_{D_{\mathsf{CMin(A)}}}(\mathsf{select}_{((}(D_{\mathsf{CMin(A)}}, i+1)) + 1)$$

in $\mathsf{CMin(A)}$ is red, and 1 otherwise. This follows from the observation that if there is an $i, 1 \leq i < 2n-1$ such that $D_{\mathsf{CMin(A)}}[i] =$ '$($' and $D_{\mathsf{CMin(A)}}[i+1] =$ '$)$', then $D_{\mathsf{CMin(A)}}[i+2]$ corresponds to the node which is the leftmost child of the node $\mathsf{pre_rank}_{\mathsf{CMin(A)}}(D_{\mathsf{CMin(A)}}[i])$, so we skip these nodes by counting the pattern '$(\,($' in $D_{\mathsf{CMin(A)}}$. Also, we set $V_{min}[0] = 1$, which corresponds to the first open parenthesis in $D_{\mathsf{CMin(A)}}$. For example, for $\mathsf{CMin(A)}$ in Figure 5.1, we store the node 3's color in $V_{min}[4]$. This is becuase $\mathsf{select}_{((}(D_{\mathsf{CMin(A)}}, 5) = 7$, $\mathsf{findclose}_{D_{\mathsf{CMin(A)}}}(7) + 1 = 11$ and $\mathsf{pre_rank}_{\mathsf{CMin(A)}}(11) = 3$ (see Figure 5.3). We define the bit string V_{max} in a similar way.

The following lemma shows that encoding $\mathsf{Min(A)}$ and V_{min} separately, using at most $3n + o(n)$ bits, has the same functionality as the $\mathsf{CMin(A)}$ encoding of



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	$-\infty$	3	8	5	6	6	6	3	2	2	7	10	9

$D_{\mathsf{CMin(A)}}$	(	(	(	(	(	)	(	(	)	)	(	(	(	)	)	)	)	)	)	(	)	(	(	)	)	)
$\mathsf{pre_rank}_{\mathsf{CMin(A)}}$	0	0	0	0	0	0	1	1	1	2	3	3	3	3	4	5	6	7	8	9	9	10	10	10	11	12

V_{min}	1	1	0	1	0	1	1	0
$\mathsf{node}_{V_{min}}$	-	9	8	7	3	6	5	12

$\mathsf{pre_select}_{\mathsf{CMin(A)}}$	1	7	10	11	15	16	17	18	19	20	22	25	26
$\mathsf{node_color}_{\mathsf{CMin(A)}}$	-	-	-	4	-	6	5	3	2	1	-	-	7

Figure 5.3 $D_{\mathsf{CMin(A)}}$, $\mathsf{pre_rank}_{\mathsf{CMin(A)}}$, $V_{min}[i]$, $\mathsf{node}_{V_{min}}$, $\mathsf{pre_select}_{\mathsf{CMin(A)}}$ and $\mathsf{node_color}_{\mathsf{CMin(A)}}$ for colored $2d$-Min heap

Fischer [25], which only takes $2.54n + o(n)$ bits.

Lemma 5.5. *For an array $A[1 \dots n]$ of length n, there is an encoding for A which takes at most $3n + o(n)$ bits and supports RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A and NSV_A in constant time.*

Proof. The encoding consists of the $2n + o(n)$-bit encoding of $\mathsf{Min(A)}$ encoded using structure of Lemma 5.2, together with the bit string V_{min} that stores the color information of the nodes in $\mathsf{CMin(A)}$. We use a $o(n)$-bit auxiliary structure to support the $\mathsf{rank}/\mathsf{select}$ queries on V_{min} in constant time. Since the size of V_{min} is at most n bits, the total space of the encoding is at most $3n + o(n)$ bits.

To define the correspondence between the nodes in $\mathsf{CMin(A)}$ and the positions in the bit string V_{min}, we define the following operation. For $0 \leq i \leq n$, we define $\mathsf{node_color}_{\mathsf{CMin(A)}}(i)$ as the position of V_{min} that stores the color of the node i in $\mathsf{CMin(A)}$. This can be computed in constant time, using $o(n)$ bits, by

$$\mathsf{node_color}_{\mathsf{CMin(A)}}(i) = \begin{cases} \text{undefined} & \text{if } \mathsf{child_rank}_{\mathsf{CMin(A)}}(i) = 0 \\ \mathsf{rank}_{((}(D_{\mathsf{CMin(A)}}, c) - 1 & \text{otherwise} \end{cases}$$

where $c = \mathsf{findopen}_{D_{\mathsf{CMin(A)}}}(\mathsf{pre_select}_{\mathsf{CMin(A)}}(i) - 1)$ (note that $\mathsf{node_color}_{\mathsf{CMin(A)}}$ is the inverse operation of $\mathsf{node}_{V_{min}}$).

Now we describe how to support the queries in constant time. Fischer and Heun [27] showed that $\mathsf{RMinQ}_A(i, j)$ can be answered in constant time using $D_{\mathsf{CMin(A)}}$. In fact, they return the position $\mathsf{RRMinQ}_A(i, j)$ as the answer to $\mathsf{RMinQ}_A(i, j)$. Also, as mentioned earlier, $\mathsf{PSV}_A(i) = \mathsf{parent}_{\mathsf{CMin(A)}}(i)$, and hence can be answered in constant time. Therefore, it is enough to describe how to find $\mathsf{RLMinQ}_A(i, j)$, $\mathsf{RkMinQ}_A(i, j)$ and $\mathsf{NSV}_A(i)$ in constant time.

$\mathsf{RLMinQ}_A(i, j)$: As shown by Fischer and Huen [27], all corresponding values of left siblings of the node $\mathsf{RRMinQ}_A(i, j)$ in A are at least $A[\mathsf{RRMinQ}_A(i, j)]$ (i.e., the values of the siblings are in the non-increasing order, from left to right). Also,

for a child node m of any of the left siblings of the node $\mathsf{RRMinQ}_A(i,j)$, $A[m] > A[\mathsf{RRMinQ}_A(i,j)]$. Therefore, the position $\mathsf{RLMinQ}_A(i,j)$ corresponds to one of the left siblings of the node whose position corresponds to $\mathsf{RRMinQ}_A(i,j)$.

We first check whether the color of the node $\mathsf{RRMinQ}_A(i,j)$ is red or not using V_{min}. If $V_{min}[\mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j))] = 0$ then $\mathsf{RLMinQ}_A(i,j)$ is equal to $\mathsf{RRMinQ}_A(i,j)$. If not, we find the node $\mathsf{leftmost}(i,j)$ which is the leftmost sibling of the node $\mathsf{RRMinQ}_A(i,j)$ between the nodes in $[i \dots j]$. $\mathsf{leftmost}(i,j)$ can be found in constant time by computing the depth of node i and comparing this value with d_{right}, the depth of the node $\mathsf{RRMinQ}_A(i,j)$. More specifically,

$$\mathsf{leftmost}(i,j) = \begin{cases} i & \text{if } \mathsf{depth}_{\mathsf{CMin(A)}}(i) = d_{right}. \\ \mathsf{next_sibling}_{\mathsf{CMin(A)}}(\mathsf{la}_{\mathsf{CMin(A)}}(i, d_{right})) & \text{otherwise.} \end{cases}$$

In the next step, find the leftmost blue sibling n_v such that there is no red sibling between n_v and $\mathsf{RRMinQ}_A(i,j)$. This can be found in constant time by first finding the index v using the equation

$$v = \mathsf{select}_0(V_{min}, \mathsf{rank}_0(V_{min}, \mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j))) + 1) - 1$$

and then finding the node n_v using $n_v = \mathsf{node}_{V_{min}}(v)$. If $\mathsf{child_rank}_{\mathsf{CMin(A)}}(n_v) \leq \mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{leftmost}(i,j))$ or $\mathsf{child_rank}_{\mathsf{CMin(A)}}(n_v) = 1$ (this is the case that $\mathsf{leftmost}(i,j)$ can be the the lestmost sibling), then $\mathsf{RLMinQ}_A(i,j) = \mathsf{leftmost}(i,j)$. Otherwise, $\mathsf{RLMinQ}_A(i,j) = n_v$.

$\mathsf{RkMinQ}_A(i,j)$: This query can be answered in constant time by returning the k-th sibling (in the left-to-right order) of $\mathsf{RLMinQ}_A(i,j)$, if it exists. More formally, if $\mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j)) - \mathsf{child_rank}_{\mathsf{CMin(A)}}(\mathsf{RLMinQ}_A(i,j))$ is at least $k-1$, then $\mathsf{RkMinQ}_A(i,j)$ exists; and in this case, $\mathsf{RkMinQ}_A(i,j)$ can be computed in constant time by computing

$$\mathsf{child}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(\mathsf{RRMinQ}_A(i,j)), \mathsf{RLMinQ}_A(i,j) + k - 1).$$

$\mathsf{NSV}_A(i)$: By Lemma 5.4, it is enough to show how to support $\mathsf{NRS}(i)$ in

constant time (note that we can support $\mathsf{subtree_size}$ in constant time using Lemma 5.2). If node i is the rightmost sibling, then $\mathsf{NRS}(i)$ does not exist. Otherwise we define v' as $\mathsf{select}_0(V_{min}, \mathsf{rank}_0(V_{min}, \mathsf{node_color}_{\mathsf{CMin(A)}}(\mathsf{next_sibling}(i))))$. Let $n_{v'} = \mathsf{node}_{V_{min}}(v')$. If the parent of $n_{v'}$ is same as the parent of i, then $\mathsf{NRS}(i) = n_{v'}$; otherwise $\mathsf{NRS}(i)$ does not exist. Finally, if $\mathsf{NRS}(i)$ does not exist, we compute the node r which is the rightmost sibling of the node i can be found by

$$\mathsf{child}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(i), \mathsf{degree}_{\mathsf{CMin(A)}}(\mathsf{parent}_{\mathsf{CMin(A)}}(i)) - 1).$$

Then $\mathsf{NSV}_A(i) = r + \mathsf{subtree_size}_{\mathsf{CMin(A)}}(r)$. All these operations can be done in constant time. □

5.4 Encoding colored 2d-Min and 2d-Max heaps

In this section, we describe our encodings for supporting various subsets of operations, proving the results stated in Theorem 5.1. As mentioned in Section 5.2.1, the TC-encoding of the colored $2d$-Min heap of Fischer [25] can answer RMinQ_A, RRMinQ_A, PSV_A and NSV_A queries in $O(1)$ time, using $2.54n + o(n)$ bits. The following lemma shows that we can also support the queries RLMinQ_A and RkMinQ_A using the same structure.

Lemma 5.6. *For an array $A[1 \dots n]$ of length n, RLMinQ_A, RkMinQ_A can be answered in constant time by the TC-encoding of colored $2d$-Min heap.*

Proof. Fischer [25] defined two operations, which are modifications of the child and $\mathsf{child_rank}$, as follows:

- $\mathsf{mchild}_{\mathsf{CMin(A)}}(x, i)$ - returns the i-th red child of node x in $\mathsf{CMin(A)}$, and
- $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}(x)$ - returns the number of red siblings to the left of node x in $\mathsf{CMin(A)}$.

He showed that the TC-encoding of the colored $2d$-Min heap can support $\mathsf{mchild}_{\mathsf{CMin(A)}}(x, i)$ and $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}(x)$ in constant time. Also, since the TC-encoding supports $\mathsf{depth}_{\mathsf{CMin(A)}}$, $\mathsf{next_sibling}_{\mathsf{CMin(A)}}$, $\mathsf{la}_{\mathsf{CMin(A)}}$, $\mathsf{child}_{\mathsf{CMin(A)}}$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}$ in constant time on ordinal trees [41], we can support $\mathsf{leftmost}(i, j)$ (defined in the proof of the Lemma 5.5) in constant time. For answering $\mathsf{RLMinQ}_A(i, j)$, we first find the previous red sibling l of $\mathsf{RRMinQ}_A(i, j)$ using $\mathsf{mchild}_{\mathsf{CMin(A)}}$ and $\mathsf{mchild_rank}_{\mathsf{CMin(A)}}$. If such a node l exists, we compare the child ranks of $\mathsf{next_sibling}_{\mathsf{CMin(A)}}(l)$ and $\mathsf{leftmost}(i, j)$, and return the node with the larger rank value as the answer. $\mathsf{RkMinQ}_A(i, j)$ can be answered by returning the k-th sibling (in the left-to-right order) of $\mathsf{RLMinQ}_A(i, j)$ using $\mathsf{child}_{\mathsf{CMin(A)}}$ and $\mathsf{child_rank}_{\mathsf{CMin(A)}}$, if it exists. □

By storing a similar TC-encoding of colored $2d$-Max heap, in addition to the structure of Lemma 5.6, we can support all the operations mentioned in Theorem 5.1(c) in $O(1)$ time. This uses a total space of $5.08n + o(n)$ bits. We now describe alternative encodings to reduce the overall space usage.

More specifically, we show that a combined encoding of $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$, using at most $3.17n + o(n)$ bits, can be used to answer RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A, and PLV_A queries (Theorem 5.1(a)). To support the queries in constant time, we use a less space-efficient data structure that encodes the same structures, using at most $3.322n+o(n)$ bits (Theorem 5.1(b)). Similarly, a combined encoding of $D_{\mathsf{CMin(A)}}$, $D_{\mathsf{CMax(A)}}$, V_{min} and V_{max} using at most $4.088n + o(n)$ bits can be used to answer RLMinQ_A, RkMinQ_A, NSV_A, RLMaxQ_A, RkMaxQ_A, and NLV_A queries in addition (Theorem 5.1(c)). Again, to support the queries in constant time, we design a less space-efficient data structure using at most $4.58n + o(n)$ bits (Theorem 5.1(d)).

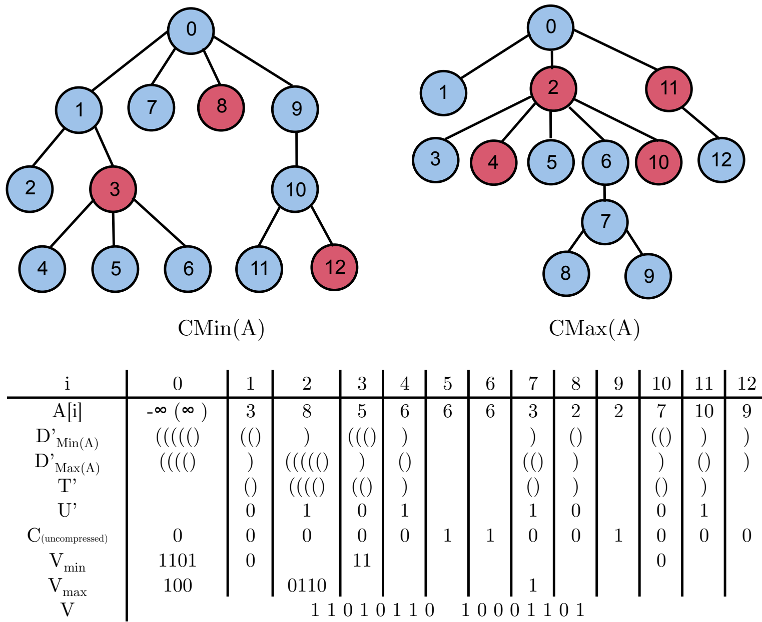
In the following, we first describe the data structure of Theorem 5.1(b) followed by the structure for Theorem 5.1(d). Next we describe the encodings of Theorem 5.1(a) and Theorem 5.1(c).

5.4.1 Combined data structure for $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$

As mentioned in Section 5.2.2, the encoding of Gawrychowski and Nicholson [30] consists of two bit strings U and T of total length at most $3n$, along with the encodings of B_{min}, B_{max} and a few additional auxiliary structures of total size $o(n)$ bits. In this section, we denote this encoding by E. To encode the DFUDS sequences of $\mathsf{CMin(A)}$ and $\mathsf{CMax(A)}$ in a single structure, we use E with some modifications, which we denote by E'. As described in Section 5.2.2, encoding scheme of Gawrychowski and Nicholson cannot be used (as it is) to support the PSV and PLV queries if there is a position i, for $1 \leq i < n$ such that $A[i] = A[i+1]$. To support these queries, we define an additional bit string $C[1 \dots n]$ such that $C[1] = 0$, and for $1 < i \leq n$, $C[i] = 1$ iff $A[i-1] = A[i]$. If the bit string C has k ones in it, then we represent C using $\lg \binom{n}{k} + o(n)$ bits while supporting rank, select queries and decoding any $\lg n$ consecutive bits in C in constant time, using Lemma 2.5. We also define a new array $A'[0 \dots n-k]$ by setting $A'[0] = A[0]$, and for $0 < i \leq n-k$, $A'[i] = A[\mathsf{select}_0(C, i)]$. (Note that A' has no consecutive repeated elements.) In addition, we define another sequence $D'_{\mathsf{CMin(A)}}$ of size $2n-k$ as follows. Suppose $D_{\mathsf{CMin(A')}} = (^{\delta_1}\ ) \dots (^{\delta_{n-k}}\ )$, for some $0 \leq \delta_1 \dots \delta_n \leq n-k$, then we set $D'_{\mathsf{CMin(A)}} = (^{\delta_1+\epsilon_1}\ ) \dots (^{\delta_{n-k}+\epsilon_{n-k}}\ )$, where $\delta_i + \epsilon_i$ is the number of elements popped when $A[i]$ is pushed into the min-stack of A, for $1 \leq i \leq n-k$. (Analogously, we define $D'_{\mathsf{CMax(A)}}$.)

The encoding E' defined on A consists of two bit strings U' and T', along with C, B'_{min}, B'_{max} and additional auxiliary structures (as in E). Let U and T be the bit strings in E defined on A'. Then U' is same as U in E, and size of U' is $n-k-1$ bits. To obtain T', we add some additional open parentheses to T as follows. Suppose $T = (^{\delta_1}\ )(^{\delta_2}\ ) \dots (^{\delta_{n-k}}\ )$, where $0 \leq \delta_i \leq n-k$ for $1 \leq i \leq n-k$. Then $T' = (^{\delta_1+\epsilon_1}\ ) \dots (^{\delta_{n-k}+\epsilon_{n-k}}\ )$, where $\delta_i + \epsilon_i$ is the number of elements are popped when $A[i]$ is pushed into the min or max stack of A, for $1 \leq i \leq n-k$ (see Figure 5.4 for an example). Since the length of T is at most

$2(n-k)$, and $|T'|-|T| = \sum_{i=1}^{n-k} \epsilon_i \leq k$, the size of T' is at most $2n-k$ bits. The encodings of B'_{min} and B'_{max} are defined on $D'_{\mathsf{CMin(A)}}$, $D'_{\mathsf{CMax(A)}}$ and T', similar to B_{min} and B_{max} in E. The total size of the encodings of the modified B'_{min} and B'_{max} is $o(n)$ bits. All the other auxiliary structures use $o(n)$ bits. Although we use E' instead of E, we can use the decoding algorithm in E without any modifications because all the properties used in the algorithm still hold even though T' has additional open parentheses compared to T. Therefore from E' we can reconstruct any $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ or $D'_{\mathsf{CMax(A)}}$ in constant time, and thus we can support rank and select on these strings in constant time with $o(n)$ additional structures by Lemma 2.5.



i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	-∞ (∞)	3	8	5	6	6	6	3	2	2	7	10	9
$D'_{Min(A)}$	((((()	(()	)	((()	)			)	()		(()	)	)
$D'_{Max(A)}$	(((()	)	((((()	)	()			(()	)		)	()	)
T'		()	(((()	(()	)			()	)		()	)	
U'		0	1	0	1			1	0		0	1	
$C_{(uncompressed)}$	0	0	0	0	0	1	1	0	0	1	0	0	0
V_{min}	1101	0		11							0		
V_{max}	100		0110					1					
V			1 1 0 1 0 1 1 0 1 0 0 0 1 1 0 1										

Figure 5.4 Data structure combining the colored $2d$-Min heap and colored $2d$-Max heap of A. C is represented in uncompressed form.

Decoding $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$

We use the following auxiliary structures to decode $D_{\mathsf{CMin(A)}}$ from $D'_{\mathsf{CMin(A)}}$ and C. For this, we first define a correspondence between $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$ as follows. Note that both $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$ have the same number of open parentheses, but $D'_{\mathsf{CMin(A)}}$ has fewer close parentheses than $D_{\mathsf{CMin(A)}}$. The ith open parenthesis in $D_{\mathsf{CMin(A)}}$ corresponds to the ith open parenthesis in $D'_{\mathsf{CMin(A)}}$. Suppose there are ℓ and ℓ' $(\leq \ell)$ close parentheses between the ith and the $(i+1)$st open parentheses in $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$, respectively. Then the last ℓ' close parentheses in $D_{\mathsf{CMin(A)}}$ correspond, in that order, to the ℓ' close parentheses in $D'_{\mathsf{CMin(A)}}$; the remaining close parentheses in $D_{\mathsf{CMin(A)}}$ do not have a corresponding position in $D'_{\mathsf{CMin(A)}}$.

We construct three bit strings P_{min}, Q_{min} and R_{min} of lengths $2n - k$, $\lceil 2n/\lg n \rceil$ and $\lceil 2n/\lg n \rceil$, respectively, as follows. For $1 \leq i \leq \lceil 2n/\lg n \rceil$, if the position $i \lg n$ in $D_{\mathsf{CMin(A)}}$ has its corresponding position j in $D'_{\mathsf{CMin(A)}}$, then we set $P_{min}[j] = 1$, $Q_{min}[i] = 0$ and $R_{min}[i] = 0$. If position $i \lg n$ in $D_{\mathsf{CMin(A)}}$ has no corresponding position in $D'_{\mathsf{CMin(A)}}$ but for some k_i where $1 \leq k_i < \lg n$, suppose there is a leftmost position $q = i \lg n + k_i$ which has its corresponding position j in $D'_{\mathsf{CMin(A)}}$. Then we set $P_{min}[j] = 1$, $Q_{min}[i] = 1$ and $R_{min}[i] = 0$. Finally, if all positions between $i \lg n$ and $(i+1) \lg n$ in $D_{\mathsf{CMin(A)}}$ have no corresponding position in $D'_{\mathsf{CMin(A)}}$, then we set $Q_{min}[i] = 1$ and $R_{min}[i] = 1$. In remaining positions for P_{min}, Q_{min} and R_{min}, we set their values as 0. We also store the values, k_i explicitly, for $1 \leq i \leq \lceil 2n/\lg n \rceil$, whenever they are defined (as in the second case). Since $k_i < \lg n$, we can store all the k_i values explicitly using at most $2n \lg \lg n / \lg n = o(n)$ bits.

Since the bit strings P_{min}, Q_{min} and R_{min} have at most $2n/\lg n$ 1's each, they can be represented using the structure of Lemma 2.5, taking $o(n)$ bits while supporting rank and select queries in constant time. We define P_{max}, Q_{max}, R_{max} in the same way, and represent them analogously.

In addition to these bit strings, we construct two pre-computed tables. In the rest of this section, we refer to the parenthesis strings (such as $D_{\mathsf{CMin(A)}}$ and $D'_{\mathsf{CMin(A)}}$) also as bit strings. To describe these tables, we first define two functions f and f', each of which takes two bit strings s and c as parameters, and returns a bit string of length at most $|s| + |c|$, as follows.

$$\begin{cases} f(s, \epsilon) = s \\ f(\epsilon, c) = \epsilon \\ f(s, 1 \cdot c_1) =) \cdot f(s, c_1) \\ f((^{\delta}\ ) \cdot s_1, 0 \cdot c_1) = (^{\delta} \cdot) \cdot f(s_1, c_1) \end{cases} \quad \begin{cases} f'(s, \epsilon) = s \\ f'(\epsilon, c) = \epsilon \\ f'(s, c_1 \cdot 1) = f'(s, c_1) \cdot) \\ f'(s_1 \cdot (^{\delta}\ ), c_1 \cdot 0) = f'(s_1, c_1) \cdot (^{\delta} \cdot) \end{cases}$$

One can easily show that if s is a substring of $D'_{\mathsf{CMin(A)}}$ and c is a substring of C whose starting (ending) position corresponds to the starting (ending) position in s, then $f(s, c)$ ($f'(s, c)$) returns the substring of $D_{\mathsf{CMin(A)}}$ whose starting (ending) position corresponds to the starting (ending) position in s,

We construct a pre-computed table T_f that, for each possible choice of bit strings s and c of length $(1/4) \lg n$, stores the bit string $f(s, c)$. These pre-computed tables can be used to decode a substring of $D_{\mathsf{CMin(A)}}$ given a substring of $D'_{\mathsf{CMin(A)}}$ (denoted s) and a substring of C whose bits correspond to s. The total space usage of T_f is $2^{(1/4) \lg n} \cdot 2^{(1/4) \lg n} \cdot ((1/2) \lg n) = o(n)$ bits. We can also construct $T_{f'}$ defined analogous to T_f using $o(n)$ bits.

Now we describe how to decode $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ in constant time. (We can decode $\lg n$ consecutive bits of $D_{\mathsf{CMax(A)}}$ in a similar way.) Suppose we divide $D_{\mathsf{CMin(A)}}$ into blocks of size $\lg n$. As described in Section 5.2.2, it is enough to show that for $1 \leq i \leq \lceil 2n/\lg n \rceil$, we can decode i-th block of $D_{\mathsf{CMin(A)}}$ in constant time. First, we check the value of the $R_{min}[i]$. If $R_{min}[i] = 1$, then the i-th block in $D_{\mathsf{CMin(A)}}$ consists of a sequence of $\lg n$ consecutive close parentheses. Otherwise, there are two cases depending on the value of $Q_{min}[i]$. We compute the position p which is a position in $D'_{\mathsf{CMin(A)}}$ (it's exact correspondence in $D_{\mathsf{CMin(A)}}$ depends on the value of the bit $Q_{min}[i]$), and

then compute the position c_p in C which corresponds to p in $D'_{\mathsf{CMin(A)}}$, using the following equations:

$$p = \mathsf{select}_1(P_{min}, i - \mathsf{rank}_1(R_{min}, i))$$

$$c_p = \begin{cases} \mathsf{select}_0(C, \mathsf{rank}_{)}(D'_{\mathsf{CMin(A)}}, p)) & \text{if } D'_{\mathsf{CMin(A)}}[p] =')' \\ \mathsf{select}_0(C, \mathsf{rank}_{)}(D'_{\mathsf{CMin(A)}}, p) + 1) & \text{otherwise} \end{cases}$$

Case (1) $\boldsymbol{Q_{min}[i] = 0}$. In this case, we take the $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ starting from p, and the $\lg n$ consecutive bits of C starting from the position c_p (padding at the end with zeros if necessary). Using these bit strings, we can decode the i-block in $D_{\mathsf{CMin(A)}}$ by looking up T_f with these substrings (a constant number of times, until the pre-computed table generates the required $\lg n$ bits). Since the position p corresponds to the starting position of the i-th block in $D_{\mathsf{CMin(A)}}$ in this case, we can decode the i-th block of $D_{\mathsf{CMin(A)}}$ in constant time.

Case (2) $\boldsymbol{Q_{min}[i] = 1}$. First we decode $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ whose starting position corresponds to the position p using the same procedure as in Case (1). Let S_1 be this bit string. Next, we take the $\lg n$ consecutive bits of $D'_{\mathsf{CMin(A)}}$ ending with position p, and the $\lg n$ consecutive bits of C ending with position c_p (padding at the beginning with zeros if necessary). Then we can decode the $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ whose ending position corresponds to the p by looking up $T_{f'}$ (a constant number of times) with these substrings. Let S_2 be this bit string. By concatenating S_1 and S_2, we obtain a $2\lg n$-bit substring of $D_{\mathsf{CMin(A)}}$ which contains the starting position of the i-th block of $D_{\mathsf{CMin(A)}}$ (since the starting position of the i-th block in $D_{\mathsf{CMin(A)}}$, and the position which corresponds to p differ by at most $\lg n$). Finally, we can obtain the i-th block in $D_{\mathsf{CMin(A)}}$ by skipping the first $\lg n - k_i$ bits in $S_1 \cdot S_2$, and taking $\lg n$ consecutive bits from there.

From the encoding described above, we can decode any $\lg n$ consecutive bits of $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$ in constant time. Therefore by Lemma 5.5, we can answer all queries supported by CMin(A) and CMax(A) (without using the color information) in constant time. If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, then the size of C is $\lg \binom{n}{k} + o(n)$ bits, and the size of E' on A is $3n - 2k + o(n)$ bits. All other auxiliary bit strings and tables take $o(n)$ bits. Therefore, by the Lemma 5.3, we can encode A using $3n - 2k + \lg \binom{n}{k} + o(n) \leq ((1+\lg 5)n + o(n) < 3.322n + o(n)$ bits. Also, this encoding supports the queries in Theorem 5.1(b) (namely RMinQ_A, RMaxQ_A, RRMinQ_A, RRMaxQ_A, PSV_A and PLV_A, which do not need the color information) in constant time. This proves Theorem 5.1(b).

Note that if $k = 0$ (i.e, there are no consecutive equal elements), E' on A is same as E on A. Therefore, we can support all the queries in Theorem 5.1(b) using $3n + o(n)$ bits with constant query time.

Encoding V_{min} and V_{max}

We simply concatenate V_{max} and V_{min} on A and store it as bitstring V, and store the length of V_{min} using $\lg n$ bits (see V in Figure 5.4). If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, Fischer and Heun's stack based algorithm [27] does not pop any elements from both stacks when these k elements and $A[n]$ are pushed into them. Before pushing any of the remaining elements into the min- and max-stacks, we pop some elements from exactly one of the stacks. Also after pushing $A[1]$ into both the stacks, we pop the remaining elements from the stacks in the final step. Suppose the n elements are popped from the min-stack during p runs of pop operations. Then, it is easy show that the elements are popped from the max-stack during $n-k-p$ runs of pop operations. Also, p $(n-k-p)$ is the number of leftmost children in CMin(A) (CMax(A)) since each run of pop operations generates exactly one open parenthesis whose matched closing parenthesis corresponds to the leftmost child

in $\mathsf{CMin(A)}$ ($\mathsf{CMax(A)}$). As described in Section 5.3, the size of V_{min} is $n-p+1$ bits, and that of V_{max} is $p+k+1$ bits. Thus, the total size of V is $n+k+2$ bits.

Therefore, we can decode any $\lg n$-bit substring of V_{min} or V_{max} in constant time using V and the length of V_{min}. By combining these structures with the encoding of Theorem 5.1(b), we can support the queries in Theorem 5.1(d) (namely, the queries RMinQ_A, RRMinQ_A, RLMinQ_A, RkMinQ_A, PSV_A, NSV_A RMaxQ_A, RRMaxQ_A, RLMaxQ_A, RkMaxQ_A, PLV_A and NLV_A) in constant time. By Lemma 5.3, the total space of these structures is $4n - k + \lg \binom{n}{k} + o(n) \leq ((3 + \lg 3)n + o(n) < 4.585n + o(n)$ bits. This proves Theorem 5.1(d).

Note that if $k = 0$ (i.e, there are no consecutive equal elements), E' on A is same as E on A, and the size of V is $n+2$ bits. Therefore we can support all the queries in Theorem 5.1(d) using $4n + o(n)$ bits with constant query time.

5.4.2 Encoding colored $2d$-Min and $2d$-Max heaps using less space

In this section, we give new encodings that prove Theorem 5.1(a) and Theorem 5.1(c), which use less space but take more query time than the previous encodings. To prove Theorem 5.1(a), we maintain the encoding E' on A, with the modification that instead of T' (which takes at most $2n - k$ bits), we store the bit string T (which takes at most $2(n - k)$ bits) which is obtained by constructing the encoding E on A'. Note that $f(s, c)$ is well-defined when s and c are substrings of $D_{\mathsf{CMin(A')}}$ and C, respectively. If there are k elements such that $A[i-1] = A[i]$ for $1 \leq i \leq n$, then the total size of the encoding is at most $3(n-k) + \lg \binom{n}{k} + o(n) \leq n \lg 9 + o(n) < 3.17n + o(n)$ bits. If we can reconstruct the sequences $D_{\mathsf{CMin(A)}}$ and $D_{\mathsf{CMax(A)}}$, by Lemma 5.5, we can support all the required queries. We now describe how to decode the entire $D_{\mathsf{CMin(A)}}$ using this encoding. (Decoding $D_{\mathsf{CMax(A)}}$ can be done analogously.)

Once we decode the sequence $D_{\mathsf{CMin(A')}}$, we reconstruct the sequence $D_{\mathsf{CMin(A)}}$

by scanning the sequences $D_{\mathsf{CMin(A')}}$ and C from left to right, and using the lookup table T_f. Note that $f(D_{\mathsf{CMin(A')}}, C)$ looses some open parentheses in $D_{\mathsf{CMin(A)}}$ whose matched close parentheses are not in $D_{\mathsf{CMin(A')}}$ but in $f(D_{\mathsf{CMin(A')}}, C)$. So when we add m consecutive close parentheses from the r-th position in $D_{\mathsf{CMin(A')}}$ in decoding with T_f, we add m more open parentheses before the position $pos = \mathsf{findopen}_{D_{\mathsf{Min(A')}}}(r-1)$. This proves Theorem 5.1(a).

To prove Theorem 5.1(c), we combine the concatenated sequence of V_{min} and V_{max} on A' whose total size is $n-k+2$ bits to the above encoding. Then we can reconstruct V_{min} on A by adding m extra 1's before $V_{min}[\mathsf{rank}_{((}(D_{\mathsf{Min(A')}}, pos)]$ when m consecutive close parentheses are added from the r-th position in $D_{\mathsf{CMin(A')}}$ while decoding with T_f. (Reconstructing V_{max} on A can be done in a similar way.) The space usage of this encoding is $4(n-k) + \lg \binom{n}{k} + o(n) \leq n \lg 17 + o(n) < 4.088n + o(n)$ bits. This proves Theorem 5.1(c).

5.5 Open problems

In this chpater, we obtained space-efficient encodings that support a large set of range and previous/next smaller/larger value queries.

We suggest the following open problems for future works.

- Can we support the queries in the Theorem 5.1(c) in $O(1)$ time using at most $4.088n + o(n)$ bits?
- As described in Section 5.2, Gawrychowski and Nicholson [30] show that any encoding that supports both RMinQ_A and RMaxQ_A requires at least $3n - \Theta(\lg n)$ bits. Can we obtain an improved lower bound in the case when we need to support the queries in Theorem 5.1(a)?
- Can we prove a lower bound that is strictly more than $3n$ bits for any encoding that supports the queries in Theorem 5.1(c)?

Chapter 6

Encoding Two-dimensional range Top-k queries

6.1 Introduction

Given a one-dimensional (1D) array $A[1 \dots n]$ from a total order and $1 \leq k \leq n$, the *Range Top-k query on A* ($\mathsf{Top}\text{-}k(i, j, A), 1 \leq i, j \leq n$) returns the positions of k largest values in $A[i \dots j]$. We can extend this query to the two-dimensional (2D) array case. Given a 2D array $A[1 \dots m][1 \dots n]$, from a total order and $1 \leq k \leq mn$, the *Top-k query on A* ($\mathsf{Top}\text{-}k(i, j, a, b, A), 1 \leq i, j \leq m, 1 \leq a, b \leq n$) returns the positions of k largest values in $A[i \dots j][a \dots b]$. Without loss of generality, we assume that all elements in A are distinct by ordering equal elements in the lexicographic order of their positions. Also, if the k positions of a $\mathsf{Top}\text{-}k$ query are reported in sorted order of the corresponding values, we refer to the query as *sorted* $\mathsf{Top}\text{-}k$ query; and refer to it as *unsorted* $\mathsf{Top}\text{-}k$ query, otherwise. For $1 \leq i, j \leq m$ and $1 \leq a, b \leq n$, we can also classify $\mathsf{Top}\text{-}k$ queries on 2D array by its range as follows.

- 1-sided query : The query range is $[1 \dots m][1 \dots b]$.

- 4-sided query : The query range is $[i \dots j][a \dots b]$.

We can also consider 2-sided and 3-sided queries which correspond to the ranges $[1 \dots j][1 \dots a]$ and $[1 \dots j][a \dots b]$ respectively. We consider how to support the $\mathsf{Top}\text{-}k$ queries in the encoding model.

In the rest of the chapter, we assume that for $\mathsf{Top}\text{-}k$ encodings, k is at most the size of the array (either 1D or 2D). Also, unless otherwise mentioned, we assume that all $\mathsf{Top}\text{-}k$ queries are sorted $\mathsf{Top}\text{-}k$ queries.

Previous Work. Encoding $\mathsf{Top}\text{-}k$ queries on 1D array has been widely studied in the recent years. For a 1D array $A[\dots n]$, Chan and Wilkinson [12] proposed a data structure that uses $\Theta(n)$ words and answers selection queries (i.e., selecting the k-th largest element) in $O(\lg k/ \lg\lg n)$ time. Grossi et al. [39] considered the $\mathsf{Top}\text{-}k$ encoding problem, and obtained an $O(n \lg \kappa)$-bit encoding which can answer the $\mathsf{Top}\text{-}k$ queries for any $k \leq \kappa$ in $O(\kappa)$ time or alternately, using $O(n \lg^2 \kappa)$ bits with $O(k)$ query time. (They also considered one-sided $\mathsf{Top}\text{-}k$ query, they proposed $n \lg k + O(n)$-bit encoding with $O(k)$ query time.) The space usage of this encoding was improved to $O(n \lg \kappa)$ bits, maintaining the $O(k)$ query time, by Navarro et al. [60]. Recently, Gawrychowski and Nicholson [31] proposed an $(k+1)nH_0(1/(k+1))+o(n)$-bit[1] encoding for $\mathsf{Top}\text{-}k$ queries and showed that at least $(k+1)nH_0(1/(k+1))(1-o(1))$ bits are required to encode $\mathsf{Top}\text{-}k$ queries.

To the best of our knowledge, there are no results for range $\mathsf{Top}\text{-}k$ queries for 2D array with general k. For $k = 1$, the $\mathsf{Top}\text{-}k$ query is same as the *Range Maximum Query (RMaxQ)*, which has been well-studied for 1D as well as for 2D arrays. For a 2D $m \times n$ array, Brodal et al. [10] proposed an $O(nm \min(m, \lg n))$-bit encoding which answers RMaxQ queries in $O(1)$ time. Brodal et al. [8] improved the space bound to the optimal $O(nm \lg m)$ bits, although this encoding does not support the queries efficiently.

[1] $H_0(x) = x \lg(1/x) + (1-x) \lg(1/(1-x))$

Array size	Query range	Space	Query time
$m \times n$	one-sided	$n \lceil \lg T \rceil$ bits	-
$m \times n$	four-sided	$O(mn \lg n)$ bits	$O(k)$
$m \times n$	four-sided	$m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ bits	-

Table 6.1 The summary of our results for Top-k queries on $m \times n$ 2D array. $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$

Our results. For an $m \times n$ 2D array A, we first obtain an $n \lceil \lg T \rceil$-bit encoding for answering one-sided Top-k queries, where $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$. We then show that any encoding that supports Top-k queries on A must use at least $n \lg T$ bits.

Next, we observe that there exists an $O(mn \lg n)$-bit data structure which answers 4-sided Top-k queries on A in $O(k)$ time by combining the results of [11] and [10]. We then propose an $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$-bit encoding for 4-sided Top-k queries on A, by extending the Top-k encoding of Gawrychowski and Nicholson for 1D arrays [31]. The summary of our results are in Table 6.1.

This chapter organized as follows. Section 6.2 gives an lower and upper bound for encoding one-sided Top-k queries on 2D array. In Section 6.3 we propose two encodings for answering 4-sided Top-k queries on 2D array. Finally, in Section 6.4, we give some open problems.

6.2 Encoding one-sided range Top-k queries on 2D array

In this section, we consider the encoding of one-sided Top-k queries on a 2D array $A[1 \dots m][1 \dots n]$. We first introduce the encoding by simply extending the encoding of one-sided Top-k queries for 1D array proposed by Grossi et al. [39]. Next we propose an optimal encoding for one-sided Top-k queries on A.

For a 1D array $A'[1 \dots n]$, one can define another 1D array $X[1 \dots n]$ such

as $X[i] = i$ for $1 \le i \le k$ and for $k < i \le n$, $X[i] = X[i']$ if there exist a position $i' < i$ such that $A'[i]$ is larger than $A'[i']$ which is the k-th largest value in $A'[1 \ldots i-1]$, and $X[i] = k+1$ otherwise. One can answer the $\mathsf{Top}\text{-}k(1, i, A')$ by finding the rightmost occurrence of every element $1 \ldots k$ in $X[1 \ldots i]$. By representing X (along with some additional auxiliary structures) using $n \lg k + O(n)$ bits, Grossi et al. [39] obtained an encoding which supports 1-sided $\mathsf{Top}\text{-}k$ queries on A' in $O(k)$ time.

For a 2D array A, one can encode A to support one-sided $\mathsf{Top}\text{-}k$ queries by writing down the values of A in column-major order into a 1D array, and using the encoding described above – resulting in the following encoding.

Proposition 6.1. *A 2D array $A[1 \ldots m][1 \ldots n]$ can be encoded using $mn \lg k + O(n)$ bits to support one-sided $\mathsf{Top}\text{-}k$ queries in $O(k)$ time.*

Now we describe an optimal encoding of A which supports one-sided $\mathsf{Top}\text{-}k$ queries. For 1D array $A'[1 \ldots n]$, we can define another 1D array $B'[1 \ldots n]$ such that for $1 \le i \le n$, $B'[i] = l$ if $A'[i]$ is the l-th largest element in $A'[1 \ldots i]$ with $l \le k$, and $B'[i] = k+1$ otherwise. Then we answer the $\mathsf{Top}\text{-}k(1, i, A')$ query as follows. We first find the rightmost position $p_1 \le i$ such that $B'[p_1] \le k$. Then we find the positions $p_2 > p_3 \cdots > p_k$ such that for $2 \le j \le k$, p_j is the rightmost position in $A'[1 \ldots p_{j-1}-1]$ with $B'[p_j] \le k-j+1$. Finally, we return the positions $p_1, p_2, \ldots, p_k$. Therefore by storing B' using $n \lceil \lg (k+1) \rceil$ bits, we can answer the one-sided $\mathsf{Top}\text{-}k$ queries on A'. Also we can sort $A'[p_1], \ldots, A'[p_k]$ using the property that for $1 \le b < a \le k$, $A'[p_a] < A'[p_b]$ if and only if one of the following two conditions hold: (i) $B'[p_a] \ge B'[p_b]$, or (ii) $B'[p_a] < B'[p_b]$ and there exist $q = B'[p_b] - B'[p_a]$ positions $j_1, j_2, \ldots, j_q$ such that $p_a < j_1 < \cdots < j_q < p_b$ and $B'[j_r] \le B'[p_a]$ for $1 \le r \le q$.

We can extend this encoding for the one-sided $\mathsf{Top}\text{-}k$ queries on a 2D array A. For $1 \le j \le n$, we first define the elements of j-th column in A as $a_{1j} \ldots a_{mj}$. Then we define the sequence $S_j = s_{1j} \ldots s_{mj}$ such that for $1 \le i \le m$, $s_{ij} = l$

if a_{ij} is the l-th largest element in $A[1 \ldots m][1 \ldots j]$ with $l \leq k$ and $s_{ij} = k+1$ otherwise. Since there exist $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$ possible S_i sequences, we can store $S^A = S_1 \ldots S_n$ using $n \lceil \lg T \rceil$ bits and we can answer the one-sided $\textsf{Top-}k(1, m, 1, j)$ queries on A by the following procedure.

1. Find the rightmost column q, for some $q \leq j$, such that S_q has $\ell > 0$ elements $s_{p_1 q}, \ldots, s_{p_\ell q}$ where $s_{p_1 q} < \cdots < s_{p_\ell q} < k+1$. If $\ell > k$, we return the positions of $A[p_1][q] \ldots A[p_k][q]$ as the answers of the query, and stop. If $\ell \leq k$, we return the positions of $A[p_1][q] \ldots A[p_\ell][q]$.

2. Repeat Step 1 by setting k to $k - \ell$, and j to $q - 1$.

We can return the positions in the sorted order of their corresponding values similar to the 1D array case. This encoding takes less space than the encoding in the Proposition 6.1 since $mn \lg k = n \lg \sum_{i=0}^{m} \binom{m}{i}(k-1)^i \geq n \lg T$. The following theorem shows that the space usage of this encoding is essentially optimal for answering one-sided $\textsf{Top-}k$ queries on A.

Theorem 6.1. *Any encoding of a 2D array $A[1 \ldots m][1 \ldots n]$ that supports one-sided $\textsf{Top-}k$ queries requires $n \lg T$ bits, where $T = \sum_{i=0}^{\min(m,k)} \binom{m}{i}(k!/(k-i)!)$.*

Proof. Suppose there are two distinct sequences $S^A = S_1 \ldots S_i$ and $S^{A'} = S'_1 \ldots S'_i$ which give one-sided $\textsf{Top-}k$ encodings of 2D arrays A and A', respectively. For $1 \leq b \leq n$, if $S_b \neq S'_b$ then $\textsf{Top-}k(1, m, 1, b, A) \neq \textsf{Top-}k(1, m, 1, b, A')$ by the definition of S^A and $S^{A'}$. Since for an $m \times n$ array, there are T^n distinct sequences $S^{A_1} \ldots S^{A_{T^n}}$, it is enough to prove that for $1 \leq q \leq T^n$, each $S^{A_q} = S^q_1 \ldots S^q_n$ has an array A such that $S^A = S^{A_q}$.

Without loss of generality, suppose that all elements in A come from the set $L = \{1, \ldots, mn\}$. Then we can reconstruct A from the rightmost column using S^{A_q} as follows. If $s^q_{jn} \leq k$, for $1 \leq j \leq m$, we assign the s^q_{jn}-th largest element in L to $A[j][n]$. After we assign all values in the rightmost column with $s^q_{jn} \leq k$, we discard all assigned values from L, move to $(n-1)$-th column

and repeat the procedure. After we assign all values in A whose corresponding values in S^{A_q} are smaller than $k+1$, we assign the remaining values in L to remaining positions in A_q which are not assigned yet. Thus for any $1 \le b \le n$, if S^q_b has $\ell > 0$ elements $s_{p_1 b}, \dots, s_{p_\ell b}$ where $s_{p_1 b} < \cdots < s_{p_\ell b} < k+1$, then the b-th column in A contains ℓ-largest elements in $A[1 \dots m][1 \dots b]$ by the above procedure. This shows that $S^A = S^{A_q}$. □

6.3 Encoding general range Top-k queries on 2D array

In this section, we give an encoding which supports general Top-k queries on 2D array. For an $m \times n$ 2D array, we first introduce an $O(mn \lg n)$-bit encoding which supports Top-k query in $O(k)$ time by using the RMaxQ encoding of Brodal et al. [8].

Proposition 6.2. *A 2D array $A[1 \dots m][1 \dots n]$ can be encoded using $O(mn \lg n)$ bits to support unsorted Top-$k(i,j,a,b,A)$ in $O(k)$ time for $1 \le a, b \le m$ and $1 \le i, j \le n$.*

Proof. We use a data structure similar to the one outlined in [11] (based on Frederikson's heap selection algorithm [29]) for answering unsorted Top-k queries in 1D array[2]. First encode A using $O(mn \lg n)$ bits to support RMaxQ (range maximum) queries in constant time for the any rectangular range in A. This encoding also supports finding the rank (i.e., the position in sorted order) of any element in A in $O(1)$ time [10]. Next, let $x = A[x_1][x_2]$ be the maximum value in $A[i \dots j][a \dots b]$, which can be found using an RMaxQ query on A. Then consider the 4-ary heap obtained by the following procedure. The root of the heap is x, and its four subtrees are formed by recursively constructing the 4-ary heap on the sub-arrays $A[i \dots x_1-1][a \dots b]$, $A[x_1+1 \dots j][a \dots b]$, $A[x_1][a \dots x_2-1]$ and $A[x_1][x_2+1 \dots b]$, respectively. Now, we can find the k largest elements in

[2] Brodal et al. [11] also give another structure to answer sorted Top-k queries, with the same time and space bounds.

the above 4-ary heap in $O(k)$ time using the algorithm proposed by Frederickson [29] (note that this algorithm only builds a heap with $O(k)$ nodes which is a connected subgraph of the above 4-ary heap). $\square$

We now introduce another encoding to support Top-k queries on an $m \times n$ 2D array A. This encoding extends the optimal Top-k encoding of Gawrychowski and Nicholson [31] for a 1D array. This encoding does not support the queries efficiently. Compared to the encoding of Proposition 6.2, this encoding uses less space when $n = \Omega(k^m)$. We first review the Gawrychowski and Nicholson [31]'s optimal Top-k encoding for 1D array, and show how to extend this encoding to the 2D array case.

For a given 1D array $A'[1 \dots n]$, we define the sequence of arrays $S^{A'} = S_1^{A'} \dots S_n^{A'}$, where for $1 \leq j \leq n$ and $1 \leq i \leq j$, $S_j^{A'}$ is an array of size j defined as follows.

$$S_j^{A'}[i] = \begin{cases} p & \text{if there are } p \ (< k) \text{ elements larger than } A'[i] \text{ in } A'[i+1 \dots j] \\ k & \text{otherwise} \end{cases}$$

See Figure 6.1 for an example.

If $S_j^{A'}[i] < k$, we call $A[i]$ in $A[1 \dots j]$ as *active*, otherwise $A[i]$ is *inactive* in $A[1 \dots j]$.

Gawrychowski and Nicholson [31] show that for $1 \leq i, j \leq n$, Top-$k(i, j, A')$ can be answered using $S_j^{A'}[i \dots j]$. They obtained a $\lg \binom{(k+1)n}{n} + o(n)$-bit encoding of $S^{A'}$ by representing $\delta_1^{A'} \dots \delta_{n-1}^{A'}$ (where $\delta_i^{A'} = \sum_{l=1}^{i+1} S_{i+1}^{A'}[l] - \sum_{l=1}^{i} S_i^{A'}[l]$) in unary, and compressing the sequence using the Lemma 2.1. Since $\sum_{i-1}^{n-1} \delta_i^{A'} \leq kn$, the unary sequence has kn zeros and n ones. The following lemma states their result for 1D arrays.

Lemma 6.1 ([31])**.** *Given an 1D array $A[1 \dots n]$, there is an encoding of A using $\lg \binom{(k+1)n}{n} + o(n)$ bits which supports Top-k queries.*

We now describe how to extend this encoding to a 2D $m \times n$ array A. For $1 \leq i \leq m$, let $A_i[1 \dots n]$ be the array of the i-th row in A. Then we first

A_1	3	7	8	2	6	4
A_2	6	4	10	3	5	2

$S_1^{A_1}$	0					
$S_2^{A_1}$	1	0				
$S_3^{A_1}$	2	1	0			
$S_4^{A_1}$	2	1	0	0		
$S_5^{A_1}$	2	2	0	1	0	
$S_6^{A_1}$	2	2	0	2	0	0

$S_1^{A_2}$	0					
$S_2^{A_2}$	0	0				
$S_3^{A_2}$	1	1	0			
$S_4^{A_2}$	1	1	0	0		
$S_5^{A_2}$	1	2	0	1	0	
$S_6^{A_2}$	1	2	0	1	0	0

$I_1^{(1,2)}$	1					
$I_2^{(1,2)}$	2	0				
$I_3^{(1,2)}$	2	1	1			
$I_4^{(1,2)}$	2	1	1	1		
$I_5^{(1,2)}$	2	1	1	2	0	
$I_6^{(1,2)}$	2	1	1	2	0	0

$I_1^{(2,1)}$	0					
$I_1^{(2,1)}$	1	0				
$I_1^{(2,1)}$	2	1	0			
$I_1^{(2,1)}$	2	1	0	0		
$I_1^{(2,1)}$	2	2	0	1	0	
$I_1^{(2,1)}$	2	2	0	2	0	0

Figure 6.1 $\mathsf{Top}\text{-}k$ encoding of the 2D array A when $k = 2$

maintain the $\mathsf{Top}\text{-}k$ encoding of $A_1 \dots A_m$ using Lemma 6.1, and this takes $m \lg \binom{(k+1)n}{n} + o(n)$ bits. In addition, for every $1 \le i \ne j \le m$, we define the sequence of arrays, $I^{(i,j)} = I_1^{(i,j)} \dots I_n^{(i,j)}$. For $1 \le r \le n$, $I_r^{(i,j)}$ is an array of size r defined as follows.

$$I_r^{(i,j)}[s] = \begin{cases} p & \text{if } i > j \text{ and there are } p\ (< k) \text{ elements which are} \\ & \text{larger than } A_i[s] \text{ in } A_j[s+1 \dots r] \\ q & \text{if } i < j \text{ and there are } q\ (< k) \text{ elements which are} \\ & \text{larger than } A_i[s] \text{ in } A_j[s \dots r] \\ k & \text{otherwise (if there are} \ge k \text{ elements, in the above two cases)} \end{cases}$$

See Figure 6.1 for an example.

We can answer the $\mathsf{Top}\text{-}k(i, j, a, b, A)$ queries as follows. We first define the 1D array $B[1 \dots b(j-i+1)]$ by writing down the values of $A[i \dots j][1 \dots b]$ in column-major order. Then we observe that $\mathsf{Top}\text{-}k(i, j, a, b, A)$ can be answered using $S_{b(j-i+1)}^{B}[a(j-i+1)+1 \dots b(j-i+1)]$.

The following lemma shows that we can compute the values in $S_{b(j-i+1)}^{B}$ using $S^{A_1} \dots S^{A_m}$ and all the arrays $I_b^{(c,d)}$, for $1 \le c \ne d \le m$.

Lemma 6.2. *Given a 2D array $A[1 \dots m][1 \dots n]$, for $1 \le i \le j \le m$ and $1 \le b \le n$, let $B[1 \dots q]$ be the 1D array of size $q = (j-i+1)b$ obtained by writing the elements of $A[i \dots j][1 \dots b]$ in column-major order. Also, for any $1 \le s \le q$, let (s_{row}, s_{col}) be the position corresponding $B[s]$ in A (which can be computed using $s_{col} = \lceil s/(j-i+1) \rceil$ and $s_{row} = s-(s_{col}-1)*(j-i+1)+(i-1)$). Then*

$$S_q^B[s] = \max(k, (S_b^{A_{s_{row}}}[s_{col}] + \sum_{i \le \ell \le j, \ell \ne s_{row}} I_b^{(s_{row},\ell)}[s_{col}])).$$

Proof. It is enough to count the number of elements in B (i.e., in $A[i \dots j][a \dots b]$) which are larger than $B[s]$ (i.e., $A[s_{row}][s_{col}]$) in $B[s+1 \dots q]$ (i.e., the corresponding elements in A). Let L be the set of these elements. If $|L| \ge k$, then $S_q^B[s] = k$. In the following, we describe how to compute $S_q^B[s]$ when $|L| < k$.

From the definition of $S_b^{A_{s_{row}}}$, it follows that the number of elements in L which are in row s_{row} is $S_b^{A_{s_{row}}}[s_{col}]$.. Also, for any row $\ell \ne s_{row}$, $I_b^{(s_{row},\ell)}[s_{col}]$ is the number of elements in L that belong to row ℓ. From all these values, we can compute $|L|$. □

By Lemma 6.2, we can answer the Top-k queries on A using the Top-k encodings of all the rows $A_1, \dots, A_m$, together with all the arrays $I^{(i,j)}$, for all $1 \le i \ne j \le m$. Since we can recover the order of all active elements in the prefix of i-th row using S^{A_i} [31], we can decode $I_p^{(i,j)}$ using $I_{p-1}^{(i,j)}$ and $\gamma_p^{ij} = \sum_{l=1}^{p} I_p^{(i,j)}[l] - \sum_{l=1}^{p-1} I_{p-1}^{(i,j)}[l]$ by the following procedure, for $p > 1$.

1. Append 0 to $I_{p-1}^{(i,j)}$. Let this array be $J_{p-1}^{(i,j)}$.
2. Find the positions of $\gamma_{p-1}^{(i,j)}$ smallest active values in $A_i[1 \dots p]$ using S^{A_i}, and increase the values of $J_{p-1}^{(i,j)}$ in these positions by 1.

Therefore, using $I_1^{(i,j)}$, and $\gamma_2^{(i,j)}, \dots, \gamma_n^{(i,j)}$, we can decode $I^{(i,j)}$. Since the sum $\sum_{\ell=2}^{\ell=n} \gamma_\ell^{(i,j)}$ is at most kn, we can encode all the arrays $I^{(i,j)}$ (for all possible $i \ne j$) using $m(m-1)\lg\binom{(k+1)n}{n} + o(n)$ bits (by converting $\gamma_\ell^{(i,j)}$'s into unary, as in the encoding of Lemma 6.1). Also, to encode $I_1^{(i,j)}$ for $i < j$ (note that if

$i > j$, $I_1^{(i,j)}$ is always 0), we need to store the ordering of all elements in the first column, which takes $m \lg m$ bits. This gives a proof of the following theorem.

Theorem 6.2. *Given a 2D array $A[1 \ldots m][1 \ldots n]$, there is an encoding of A using $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ bits which can answer the Top-k queries.*

6.4 Open problems

In this chapter, we obtained encodings which answer Top-k query on 2D array. We suggest the following open problems for future works.

- Can we support efficient query time on our proposed encodings of Theorem 6.1 and Theorem 6.2?

- For 2 and 3-sided query, can we obtain an encoding which uses less space than the 4-sided Top-k queries on 2D array?

- Is the effective entropy of unsorted Top-k queries smaller than the effective entropy of sorted Top-k queries on 2D arrays?

Chapter 7

Conculsion

In this thesis, we proposed various space-efficient data structures that answer rank and select on bit strings, NLN queries, range queries and next/previous larger/smaller values simultaneously, and Range Top-k queries on two-dimensional array. Most of our data structures not only require less space than existing data structures, but also support queries efficiently .

In Chapter 3, we are aware, carefully investigated V2F compressors as a basis for bitvectors. We have shown how V2F bitvectors can lead to simple bitvectors with low redundancy. Empirical testing of an implementation, which albeit differs considerably from the theoretical proposals, shows that low memory usage and good, robust speed performance can be obtained via V2F compressors.

In Chapter 4, we proposed data structures for NLV in one-dimensional arrays in indexing model, and NLN in two-dimensional arrays, in the encoding models. For two-dimensional arrays we obtained a data structure that uses asymptotically optimal space and supports NLN queries in constant time.

In Chapter 5, we obtained space-efficient encodings that support a large set of range and previous/next smaller/larger value queries. The encodings that

support the queries in constant time take more space than the ones that do not support the queries in constant time.

Finally, in Chapter 6, we obtained encodings which answer $\mathsf{Top}\text{-}k$ query on two-dimensional array. In particular, for $m \times n$ two-dimensional array, we proposed an optimal encoding when the query is one-sided.

In addition to the open problems in each chapter, we give the following general open problems which can be considered for all problems in this thesis.

- In this thesis, all data structures give exact answers for queries. Different from exact queries, for $\epsilon > 0$, ϵ-*approximate query* returns the answer between T and ϵT where T is an exact answer of the query. For example, $(1+\epsilon)$-approximate $\mathsf{NLN}(i)$ on $A[1 \dots n]$ returns the position j such that $A[j] > A[i]$ and $|j - i| \leq (1+\epsilon)|\mathsf{NLN}(i) - i|$. There are several studies about approximate queries such as nearest neighbor problem [2] and range minima in the middle [26]. Can data structures in this thesis use less space or query time for answering the approximate queries instead of exact queries?

- All data structures proposed in this thesis only works on a one or two-dimensional array. Extending these data structures for general n-dimensional array can be considered as an open problem. For example, Yuan and Atallah [75], and Davoodi et al. [15] obtained encodings which support RMinQ queries on n-dimensional arrays.

  More interesting open problem is that we can generalize an n-dimensional array into an n-*dimensional grid*. For example, we can consider the two-dimensional array as a special case of two-dimensional grid such that all grid points have an assigned value. Navarro et al. proposed data structures for various range queries on two-dimensional grid [57], but they did not considered the problems in terms of *density*, i.e, the ratio between the total number of grid points and the grid points which have an assigned

value. In many practical cases, the grid is *sparse*, that is, only few grid points have an assigned value. Can we design efficient data structures where input data is an n-dimensional grid and obtain better space or query time when the grid is sparse?

Bibliography

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97, 2010.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] T. Asano, S. Bereg, and D. G. Kirkpatrick. Finding nearest larger neighbors. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2009.

[4] T. Asano and D. G. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *WADS*, volume 8037 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2013.

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000, Proceedings*, pages 88–94, 2000.

[6] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[7] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 14(3):344–370, 1993.

[8] G. S. Brodal, A. Brodnik, and P. Davoodi. The encoding complexity of two dimensional range minimum data structures. In *ESA 2013, 2013. Proceedings*, pages 229–240, 2013.

[9] G. S. Brodal, P. Davoodi, M. Lewenstein, R. Raman, and S. S. Rao. Two dimensional range minimum queries and Fibonacci lattices. In *ESA*, pages 217–228, 2012.

[10] G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.

[11] G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *ISAAC 2009, Proceedings*, pages 173–182, 2009.

[12] T. M. Chan and B. T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013,2013*, pages 241–251, 2013.

[13] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996.

[14] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[15] P. Davoodi, J. Iacono, G. M. Landau, and M. Lewenstein. Range minimum query indexes in higher dimensions. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, pages 149–159, 2015.

[16] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *COCOON 2012, Proceedings*, pages 396–407, 2012.

[17] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *WEA*, volume 4007 of *LNCS*, pages 134–145. Springer, 2006.

[18] O. Delpratt, N. Rahman, and R. Raman. Compressed prefix sums. In *SOFSEM (1)*, volume 4362 of *LNCS*, pages 235–247. Springer, 2007.

[19] E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[20] M. Drmota, Y. A. Reznik, and W. Szpankowski. Tunstall code, khodak variations, and random walks. *IEEE Transactions on Information Theory*, 56(6):2928–2937, 2010.

[21] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

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

[23] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 269–278, 2001.

[24] P. Ferragina and G. Navarro. Pizza&chili corpus. `http://pizzachili.dcc.uchile.cl/texts.html`.

[25] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

[26] J. Fischer and V. Heun. Finding range minima in the middle: Approximations and applications. *Mathematics in Computer Science*, 3(1):17–30, 2010.

[27] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[28] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[29] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

[30] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range min-max and top-k. *CoRR*, abs/1411.6581, 2014.

[31] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range top-k k , selection, and min-max. In *ICALP 2015, Proceedings, Part I*, pages 593–604, 2015.

[32] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[33] B. Goethals and M. Zaki. Fimi: Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/`, 2004.

[34] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. *CoRR*, abs/1311.1249, 2013.

[35] M. J. Golin, J. Iacono, D. Krizanc, R. Raman, and S. S. Rao. Encoding 2d range maximum queries. In *ISAAC*, volume 7074 of *LNCS*, pages 180–189. Springer, 2011.

[36] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *ESA*, volume 4698 of *LNCS*, pages 371–382. Springer, 2007.

[37] A. Golynski, A. Orlandi, R. Raman, and S. S. Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.

[38] R. Gonzáalez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[39] R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. R. Satti. Encodings for range selection and top-k queries. In *Algorithms - ESA 2013*, volume 8125 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2013.

[40] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.

[41] M. He, J. I. Munro, and S. R. Satti. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms*, 8(4):42, 2012.

[42] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989.

[43] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

[44] V. Jayapaul, S. Jo, V. Raman, and S. R. Satti. Space efficient data structures for nearest larger neighbor. In *Combinatorial Algorithms - 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014*, pages 176–187, 2014.

[45] S. Jo, S. Joannou, D. Okanohara, R. Raman, and S. R. Satti. Compressed bit vectors based on variable-to-fixed encodings. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, page 409, 2014.

[46] S. Jo, R. Raman, and S. R. Satti. Compact encodings and indexes for the nearest larger neighbor problem. In *WALCOM 2015*, pages 53–64, 2015.

[47] S. Jo and S. R. Satti. Simultaneous encodings for range and next/previous larger/smaller value queries. In *Computing and Combinatorics - 21st International Conference, COCOON 2015, Beijing, China, August 4-6, 2015, Proceedings*, pages 648–660, 2015.

[48] T. Jurkiewicz and K. Mehlhorn. The cost of address translation. In P. Sanders and N. Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 148–162. SIAM, 2013.

[49] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 302–311, 2014.

[50] G. L. Khodak. Connection between redundancy and average delay of fixed-length coding. In *All-Union Conf. Problems of Theoretical Cybernetics*, 1969. (in Russian).

[51] D. K. Kim, J. C. Na, J. E. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. In *WEA*, volume 3503 of *LNCS*, pages 315–327. Springer, 2005.

[52] M. Lewenstein, J. I. Munro, and V. Raman. Succinct data structures for representing equivalence classes. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Proceedings*, pages 502–512, 2013.

[53] P. B. Miltersen. Cell probe complexity - a survey. *FSTTCS*, 1999.

[54] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[55] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[56] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[57] G. Navarro, Y. Nekrich, and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Theor. Comput. Sci.*, 482:60–72, 2013.

[58] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *SEA*, volume 7276 of *LNCS*, pages 295–306. Springer, 2012.

[59] G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *SEA*, volume 6630 of *LNCS*, pages 193–205. Springer, 2011.

[60] G. Navarro, R. Raman, and S. R. Satti. Asymptotically optimal encodings for range selection. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014,2014*, pages 291–301, 2014.

[61] D. Okanohara. Compressed rank select dictionary. `https://code.google.com/p/rsdic/`, 2012. [Online; accessed 1-March-2013].

[62] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*, 2007.

[63] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007.

[64] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[65] M. Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008.

[66] M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC*, pages 232–240. ACM, 2006.

[67] M. Patrascu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *SODA*, pages 117–122. SIAM, 2010.

[68] R. Raman. Encoding data structures. In *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, pages 1–7, 2015.

[69] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007.

[70] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 25pp, 2007.

[71] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[72] Y. Tabei. fmindex-plus-plus. `https://code.google.com/p/fmindex-plus-plus/`.

[73] B. P. Tunstall. *Synthesis of noiseless compression codes.* PhD thesis, Georgia Tech, 1967.

[74] S. Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *LNCS*, pages 154–168. Springer, 2008.

[75] H. Yuan and M. 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*, pages 150–160, 2010.

[76] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

요약

본 논문에서는 다양한 범위 질의 및 관련문제들을 해결하는 공간 효율적인 자료 구조들을 디자인 및 구현하였다. 본 논문에서 제안한 대부분의 자료 구조들은 정보 엔트로피에 가까운 적은 공간 만을 차지하면서도 효과적인 질의 시간을 지원한다. 세부적으로 본 논문에서는 다음 네 가지 문제 ((i) 압축된 비트 문자열 상에서 rank 와 select 질의를 지원하는 문제, (ii) nearest larger neighbor 질의 문제, (iii) 여러 범위 질의 및, next/previous larger/smaller value 질의 문제, (iv) 이차원 배열 상에서의 Top-k 질의 문제) 을 해결하는 공간 효율적인 자료 구조에 대해 연구하였다.

본 논문에서는 우선 압축된 비트 벡터를 실질적으로 구현하였다 [45]. 비트 벡터는 비트 문자열 상에서 rank 와 select 질의를 지원하는 자료 구조를 뜻한다. 본 논문에서는 이전까지 체계적으로 연구 되지 않았던 *V2F(variable-to-fixed)* 압축 알고리즘으로 압축되어진 비트 문자열 상에서 비트 벡터를 구현하는 방법에 대해 연구하였다. 본 논문은 이러한 접근 방식이 실제 상황에서 빠른 질의를 지원함과 동시에 적은 여분 공간 (압축 된 비트 문자열을 제외한 비트 벡터의 공간) 을 차지한다는 것을 보였다. 본 논문에서 제안한 비트 벡터는 다양한 방법으로 압축된 비트 문자열 상에서 rank 와 select 질의를 지원하는 효과적이면서도 실용적인 방안을 제공한다.

이어서 본 논문에서는 nearest larger neighbor 문제를 해결하는 공간 효율적인 자료 구조에 대해 연구하였다 [44, 46]. 전순서가 주어진 n개의 원소를 가지는 일차원 배열이 있을 때, nearest larger neighbor (NLN) 질의는 배열 상의 어느 한 위치가 질의로 주어졌을 때, 질의와 가장 가까운 곳에 위치하면서 질의보다 큰 값을 가진 배열 상의 원소의 위치를 반환한다. 배열상에 모든 원소들의 NLN 질의에 답하는 문제는 괄호 매칭 이나 계산 기하학 관련 문제들에 활용될 수 있기에 큰 주목을 받고 있다 [3, 4, 7]. 본 논문에서는 이러한 NLN 질의를 빠른 시간 안에 풀 수 있는 공간 효율적인 자료 구조들에 대해 연구하였다. 우선 본

논문은 인덱싱 모델 하에서 일차원 배열에서 질의 시간과 사용 공간 사이에 tradeoff 를 가지는 자료 구조를 제안하였으며 인코딩 모델 하에서 이차원 배열에서 최적에 가까운 공간을 사용하면서 상수 시간 안에 NLN 질의에 답할 수 있는 자료 구조를 제안하였다.

또한 본 논문에서는 다양한 범위 질의들(범위 최소 질의, 범위 최대 질의 및 이들에 대한 확장 질의) 과 함께 next/previous larger/smaller value 질의를 답할 수 있는 공간 효율적인 자료 구조에 대해 연구하였다 [47]. 전순서가 주어진 n개의 원소를 가지는 일차원 배열이 있을 때, 본 논문에서는 $4.088n + o(n)$ 비트의 공간을 사용하면서 위에 주어진 모든 질의들에 답할 수 있는 자료 구조를 제안하였다. 또한 본 논문에서는 $4.585n + o(n)$ 비트의 공간을 사용하면서 위에 주어진 모든 질의들을 상수 시간 안에 답할 수 있는 자료 구조를 제안하였다. 본 논문이 제안한 자료 구조는 기존의 Fischer 에 의해 연구 된 $5.08n + o(n)$ 비트의 공간을 사용하는 자료구조에 비해 적은 공간을 차지한다 [25]. 본 논문에서는 우선 색칠 된 $2d$-Min heap 과 색칠 된 $2d$-Max heap 를 인코딩 하기 위해 기존의 DFUDS [6] 인코딩 기법을 확장한 다음, Gawrychowski 와 Nicholson 이 $2d$-Min heap 과 $2d$-Max heap 을 동시에 인코딩 하기 위해 제안한 자료 구조를 수정하여 색칠 된 $2d$-Min heap 과 색칠 된 $2d$-Max heap 을 동시에 인코딩 할 수 있음을 보였다. 본 논문은 또한 위의 질의들 중 일부를 지원하면서 $4.088n + o(n)$ 비트보다 더 적은 공간을 사용하는 자료 구조를 제안하였다.

마지막으로 본 논문에서는 전순서가 주어진 이차원 배열 상에서 Top-k 질의에 답할 수 있는 다양한 자료 구조들에 대해 연구하였다. $m \times n$ 이차원 배열에서 본 논문은 질의 범위가 $[1 \ldots m][1 \ldots a]$ $(1 \leq a \leq n)$ 로 제한 되었을 때 최적의 공간을 사용하는 자료 구조를 제안하였다. 또한 본 논문은 Gawrychowski 와 Nicholson 이 제안한 일차원 배열상에서 Top-k 질의를 지원하는 자료 구조를 확장하여 $m^2 \lg \binom{(k+1)n}{n} + m \lg m + o(n)$ 비트의 공간을 사용하면서 일반적인 Top-k 질의를 지원하는 자료 구조를 제안하였다.

주요어: 공간 효율적인 자료구조, 간결한 자료구조, 인코딩 모델, 인덱싱 모델, 비트벡터, rank 질의, select 질의, nearest larger neighbor 문제, 범위 질의,

next/previous larger 질의, 범의 Top-k 질의

학번: 2011-30257