Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Engineering compressed bit vectors

Master's Thesis

2022

Bc. Andrej Korman

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Engineering compressed bit vectors

Master's Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	Mgr. Jakub Kováč, PhD.

Bratislava, 2022
Bc. Andrej Korman

68975404

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Andrej Korman
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Engineering Compressed Bit Vectors
Efektívna implementácia komprimovaných bitových polí

Anotácia: Úsporné dátové štruktúry zaberajú len o málo viac miesta ako je informačno-teoretické minimum. Poznáme efektívne reprezentácie mnohých dátových štruktúr od reťazcov cez stromy, grafy, až po mriežky a textové indexy. Väčšina z týchto zložitejších štruktúr pritom pri svojej reprezentácii využíva bitové polia (s operáciami access, rank a select). Zlepšovanie bitových polí je preto zaujímavé nielen z hľadiska návrhu fundamentálnych dátových štruktúr, ale má potenciál prakticky zlepšiť aj ostatné úsporné dátové štruktúry.

Cieľom tejto práce je skúmať reprezentácie komprimovaných bitových polí, vyvinúť ich efektívnu implementáciu a porovnať ju s existujúcimi riešeniami.

Vedúci:	Mgr. Jakub Kováč, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	20.08.2021
Dátum schválenia:	21.08.2021

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....
študent

....
vedúci práce

68975404

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Andrej Korman
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Engineering Compressed Bit Vectors

Annotation: Succinct data structures take only very little space on top of the information-theoretic lower bound. Efficient representations are known for various data structures ranging from strings, trees and graphs to grids and text indices. Most of these more complicated data structures use succinct bit vector (with operations access, rank and select) as a basic building block. Thus, improving bit vectors is interesting not only from the point of view of designing fundamental data structures - it has potential practical benefits for other succinct data structures as well. The goal of this thesis is to explore representations of compressed bit vector, develop an efficient implementation, and compare its performance with existing solutions.

Supervisor:	Mgr. Jakub Kováč, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	20.08.2021
Approved:	21.08.2021

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgments: I want to express my deepest gratitude to my supervisor Kuko for the time, knowledge and effort he put into this work. I am very thankful for the opportunity to work under his guidance. I am also very grateful for the enormous support I receive from my family and my partner Kika every day.

Abstrakt

Úsporné dátové štruktúry sú užitočné v prípadoch, keď pracujeme s veľkým množstvom dát a klasické dátové štruktúry nás limitujú svojími pamäťovými nárokmi. Bitové pole s operáciami *access*, *rank* a *select* je stavebným kameňom mnohých prakticky užitočných úsporných dátových štruktúr. V našej práci sa venujeme implementácii bitového poľa pomocou metódy *RRR*, ktorá delí postupnosť bitov na bloky, ktoré sú ďalej jednotlivo komprimované. V tejto práci sme predstavili nový algoritmus na kompresiu a dekompresiu blokov ale aj novú hybridnú metódu, ktorá výmenou za nekódovanie niektorých blokov šetrí priestor na reprezentácii ostatných. Obidve myšlienky sme naimplementovali a experimentálne otestovali v umelých ale aj reálnych podmienkach. Naša dekódovacia stratégia sa ukázala veľmi kompetitívnou a v testoch jasne porazila predchádzajúce implementácie v rýchlosti dekódovania dlhších blokov. Zrýchlenie sa prenieslo aj na implementáciu dátovej štruktúry FM-index, ktorá s našou verziou bitového poľa dosahovala pre dlhšie bloky zrýchlenie na úrovni cez 10%. Jednostranná verzia hybridnej implementácie sa ukázala zaujímavou pre riedke bitové postupnosti avšak druhý, obojstranný variant, nepriniesol zaujímavé praktické výsledky.

Kľúčové slová: bitové pole, úsporné dátové štruktúry

Abstract

Succinct data structures are interesting in scenarios, when we work with huge amount of data and ordinary data structures could limit us because of their memory requirements. Bit vector with operations *access*, *rank* and *select* is a building block of many practically useful succinct data structures. We are mainly concerned with implementation of bit vector based on the RRR representation which divides bit sequence into individually compressed blocks. In our work, we introduced a new algorithm for block encoding and decoding and also a hybrid implementation which leaves some blocks uncompressed in exchange for space savings on all other compressed blocks. We provided implementation for both of these methods and then experimentally evaluated their performance on artificial and real data. Our new decoding algorithm is very competitive in practice as it beats the existing decoding implementations on longer blocks. We also have been able to measure the speedup of data structure FM-index, which with our version of bit vector achieved over 10% speedup for longer blocks. The one-sided variant of hybrid implementation turned out to be interesting for sparse sequences. The two-sided variant, however, did not bring interesting practical results.

Keywords: bit vector, succinct data structures

Contents

Chapter 1

Introduction to succinct data structures

1.1 Motivation

In many applications, people work with so large amounts of data that the choice of the data structures is heavily influenced by their space usage. The field of *succinct data structures* focuses on representing data using as little space as possible while trying to minimize the time and performance penalty on methods that these structures support. Many succinct data structures for varied problems have been devised such as succinct dictionaries (Raman et al., 2007), graph representations (Farzan and Munro, 2013), grid representations (Chazelle, 1988), text collections (Ferragina and Manzini, 2000) and many more, nicely summarized by Navarro (2016). While many of the succinct data structures come with solid theoretical bounds on the space they use, others look into real-world space usage and performance.

Succinct data structures are very helpful in scenarios where we work with an immense amount of data. In these scenarios, using the ordinary data structures may force us to place the entire representation of data structure onto the slower type of memory storage. This reduces the usability of data structure and often completely ruins runtime due to the high amount of slow *I/O operations*. Even if a succinct version of the data structure may help us to store the data in a faster type of memory (e.g. fast RAM instead of the slower disk), we pay some price for using it. The price mostly comes in the form of more complex implementation.

In succinct data structures, it is common to distinguish between the space efficiency of representations more strictly. Suppose that I is the number of bits that are needed to store the data. We call the data representation

- *compact* – if it uses $\mathcal{O}(I)$ bits of space.
- *succinct* – if it uses $I + o(I)$ bits of space,

- *implicit* – if it uses $I + \mathcal{O}(1)$ bits of space,

In this work, we are mainly concerned with *bit vector*, one of the simplest data structures that represents a sequence S of zeroes and ones while supporting the methods:

- $access(S, i)$ – returning the i-th symbol of S,
- $rank_c(S, j)$ – returning the number of occurrences of symbol c in S before j-th index,
- $select_c(S, k)$ – returning the index of k-th occurrence of symbol c in S.

Values of these methods are defined only for integers i, j, k such that $0 \leq i < |S|$, $0 \leq j \leq |S|$ and $1 \leq k \leq \#_c(S)$ where $\#_c(S)$ denotes the number of occurrences of symbol c in S. If it is clear on what sequence we are doing the operation, we use the variants $access(i)$, $rank_c(j)$ and $select_c(k)$.

The reason behind our focus on bit vector and particularly its compressed version is the fact that it is one of the main building blocks of many succinct data structures.

In the following sections, we introduce some interesting and useful applications of bit vectors.

Throughout our work, we use a unit-cost RAM model with word size of $\Theta(\log n)$ bits. In this model, arithmetic and logic operations on and between memory words take constant time.

1.2 Application 1: Sparse array

Let A be an array of N elements, each taking k bits of space. We would like to support accessing the i-th element of A. For simplicity, we assume that the elements of the array do not change after the initial construction. A straightforward representation of this array takes $N \cdot k$ bits of space and in general, it is not possible to make it better. However, imagine a scenario where a significant portion of the array is empty and just a handful of elements are present. Take for example a sparse vector of numbers, where most of the elements are 0. Then we could use a more space-efficient approach. Let us assume that only n out of N elements are present and also that $n \ll N$.

One approach considering the sparseness of an array is to store only the non-default elements as (position, value) pairs, which takes $n \cdot (k + \lg N)$ bits of space, where by $\lg x$ we denote $\lceil \log_2 x \rceil$. If we just store these pairs sorted by the position, accessing the i-th element takes time $\mathcal{O}(\log n)$. We may also use a hash table to obtain a constant time solution but this comes with additional memory overhead.

An alternative approach, using the bit vector, is to store non-default elements in a packed array P of length n taking $n \cdot k$ bits of space. Alongside P, we store a bit vector B of length N where

$$B[i] = \begin{cases} 1, & \text{if } A[i] \text{ is occupied} \\ 0, & \text{if } A[i] \text{ is empty/default value.} \end{cases}$$

Using this representation of A, if we want to access the i-th element, we first check for the value of $B[i]$. If it is zero, we return the default value. Otherwise, we need to find how many ones are preceding this particular one in B to identify the location of the requested element in array P. This is where we find the rank_1 method useful. Similarly, to obtain the position of i-th non-empty value in A, we can use the select_1 method.

The total space used by this representation is $N + n \cdot k + R$ where R is the space required to support an efficient rank_1 query over B. As we shall show in Section 2.1.1, *rank* can be implemented succinctly in constant time, i.e., with sublinear space overhead $R = o(N)$. So if we are provided with bit vector implementation along with *access* and *rank* methods, we are able to reduce the total space used from $k \cdot N$ to $N + n \cdot k + o(N)$ bits. Note that in practice, for a really small number of non-default values, the hashtable representation takes up less space. On the other hand, bit vector provides us with a solution that is usable for scenarios when A is mediumly filled in, e.g., $n/N > 0.1$.

1.3 Application 2: Storing elements of non-uniform length

Let us consider another problem of representing an array of elements of variable length. Elements with variable lengths can often arise in succinct data structures. Even though we can store these elements one after another in memory, with the variable-length elements, we do not have an easy and fast way to tell where is the i-th element located.

Let us assume that we want to represent n elements of variable length. The first solution is to allocate the array of length $n \cdot k_{MAX}$ bits where k_{MAX} is the number of bits used for the element with the longest bit representation. This approach enables constant time access but wastes a lot of space.

A second possible solution is to allocate a bit array R where the elements are stored one after another in their raw bit representation. To locate the i-th element, we add a helper array P, such that $P[i]$ is the position where the i-th element begins in R. The helper array P takes roughly $\Theta(n \log |R|)$ bits of space as each entry contains an index into the array R.

Raw binary representation R:	1	1	0	1	1	1	1	0	1	0
Beginnings of elements	**1**	0	0	0	**1**	0	**1**	0	0	**1**

Figure 1.1: Raw binary representation of elements 1101, 11, 101 and 0 stored one after another. Note that the helper bit array is of the same size as R, with ones on the positions where a new element begins.

The helper array P can be replaced by a bit vector of length $|R|$, storing ones at positions where some element in R begins (see an example in Fig. 1.1). Identifying the beginning of the i-th element now comes down to efficiently locating the i-th one in the helper bit vector, which can be answered using the $select_1$ method. In Chapter 2, we shall see how this method can be implemented efficiently.

1.4 Application 3: FM-index

Let us now consider a practically interesting application within a more complex data structure. As we shall see, succinct data structure used to solve this problem also uses bit vector and other building blocks commonly used in succinct data structures.

Let us consider a text T. After some initial preprocessing, we would like to quickly answer questions such as "how many times is some pattern P contained in T" and also "where in T are the occurrences of P located". This problem is generally called a *text indexing* problem and is particularly useful in bioinformatics, where we often have a very long sequence of DNA and we are interested in searching for some subsequences in it, e.g., the problem of *read alignment* (Langmead et al. (2009) and Li and Durbin (2010)).

One of the solutions that can be used for shorter texts is based on a *suffix array* of T. This is a data structure which stores information about the lexicographical order of suffixes of T. More precisely, the i-th position of suffix array S, stores the starting position of the suffix that is i-th in the lexicographical order. For simplicity, in this section, we assume that every text T contains at the end a special symbol $ and this symbol is also lexicographically smaller than any other symbol contained in T. Searching for pattern P in suffix array of T uses the fact that if P is contained in T, it is located at the beginning of some suffixes. Since these are lexicographically sorted, the result forms a consecutive subsequence of S. The suffix array consumes $\mathcal{O}(n \log n)$ memory asymptotically and in practice uses about $5n$ bytes of space if the text symbol can be encoded using 1 byte and offset of the suffix can be stored in the 32-bit integer. In practice, it is possible to get to much lower number of bits, if we use succinct data structures.

FM-index, proposed by Ferragina and Manzini (2000), is a succinct data structure that is in some aspects similar to suffix array. FM-index can find the pattern in the preprocessed text in time complexity close to linear in $|P|$. Particular space usage depends on the compressibility of the text but the resulting space used by FM-index is in many cases smaller than the space used for the original text T. For instance, FM-index over a DNA sequence can take just 30–40% of the space taken by the original text T as was observed by Ferragina and Manzini (2001).

In the rest of this section, we will explain how we construct the FM-index, how we search in it and how/why FM-index uses bit vector with method *rank*. This is particularly interesting for us as later, in Chapter 4, we use FM-index to benchmark our new implementation of bit vector.

Burrows-Wheeler transform *Burrows-Wheeler transform*(BWT; Burrows and Wheeler (1994)) is a key part of the FM-index. BWT of a text T gives us a sequence T_{BWT} of the same length. Furthermore, this operation is reversible in a sense that we are able to reconstruct the original text T only using T_{BWT}. This transformation is used as a preprocessing step of compression algorithms such as *bzip2* (Seward, 1996) and was studied more by Manzini (2001), since T_{BWT} is oftentimes easier to compress than the original text. Let us first explain the construction and then provide an intuition why it is easier to compress.

Consider a sequence T of symbols over an arbitrary alphabet Σ. Take all the cyclical rotations $T_1, T_2, \ldots, T_n$ of T, sort them lexicographically and form a (conceptual) table M (see an example in Fig. 1.2). Note that each column is a permutation of T. The first column, F, consists of all the characters of T, sorted. The last column, L, is the Burrows-Wheeler transform of T.

FM-index stores just columns F and L from matrix M. As the first column of M consists of runs of sorted symbols, it can be represented in FM-index using the helper array *Count* where *Count*$[c]$ is the number of occurrences of symbols *preceding* c. Note that the run of symbol c in F starts at the index *Count*$[c]$.

BWT is usually more compressible than the original text because it frequently contains runs of the same symbol. This can be better explained on an example. Consider us having BWT of a text containing a lot of mentions of the word `house`. All the rotations prefixed with `ouse` will form a consecutive subsequence of rows of M. Some of these rows will end with symbol `m` for word `mouse`, some of them with `p` for `spouse` but many of these lines contain `h` at the end as this is very common symbol preceding `ouse` in the text.

Generally, it is common for (natural) texts that symbols can be predicted quite well from the context following them. If we sort these contexts (beginnings of rows of M), then the last row will often contain runs of the same symbol but also subsequences where just a handful of symbols occur.

	F						L	sorted suffixes
0	`$`	`b`	`a`	`n`	`a`	`n`	`a`	`$`
1	`a`	`$`	`b`	`a`	`n`	`a`	`n`	`a$`
2	`a`	`n`	`a`	`$`	`b`	`a`	`n`	`ana$`
3	`a`	`n`	`a`	`n`	`a`	`$`	`b`	`anana$`
4	`b`	`a`	`n`	`a`	`n`	`a`	`$`	`banana$`
5	`n`	`a`	`$`	`b`	`a`	`n`	`a`	`na$`
6	`n`	`a`	`n`	`a`	`$`	`b`	`a`	`nana$`

Figure 1.2: On the left, we may observe the matrix M filled with cyclic rotations of sequence $T =$ `banana$`. On the right, we may observe the sorted suffixes of T(content of suffix array). The Burrows-Wheeler transform of T is string $L =$ `annb$aa` – the last column of M. Note that in practice, we do not need to construct the whole table as more efficient algorithms exist. It is also notable that matrix M includes very similar information compared to suffix array as rows of M basically start with individual suffixes. FM-index stores only representation of columns F and L, the grey area is not stored.

Searching in an FM-index Searching in an FM-index is based on two important properties of matrix M:

1. Rotations starting with prefix w form a consecutive subsequence of rows in M.

2. The i-th occurence of symbol c in F corresponds to the i-th occurence of c in L.

The first property also enabled us to search for a pattern in a suffix array. The second property is less trivial to observe. Let us take two rows in M, namely T_i and T_j such that $i < j$. Let T_i be of the form cA and T_j of the form cB where c is a symbol from the text and A and B are sequences of symbols. Since $i < j$, it follows that $A < B$ and this means that rotated rows Ac and Bc are in the same relative order as the original rows. From this observation, it follows that the relative ordering of the same symbol is preserved between F and L.

In the next part, we describe how we search for some arbitrary pattern $P = p_0 p_1 \dots p_{n-1}$ in FM-index. Let us denote suffix of P starting at i-th element $P_{i\dots}$. The result of the search for P is a range of M's rows that have P as their prefix. The search for these rows proceeds iteratively from the end of P to its beginning. At first, we find the range of rows starting with $P_{n-1\dots}$, then gradually continue by finding rows that start with $P_{n-2\dots}, P_{n-3\dots}$ and so on up to $P_0 = P$. In every step, these rows form a consecutive subsequence of rows of M so we will maintain just the beginning and the end of the interval. This follows from the observation 1.

First, let us show the process on an example (see Fig. 1.3). Assume we are searching for the word `house` and we already found the range of rows in M that start with `ouse` (rows 200–204). Symbols in the last column of this range correspond to the symbols preceding `ouse` in the text. We may observe different symbols in arbitrary order there such as `m` for the word `mouse` or `h` for `house`.

To search for the range starting with `house`, we look into the range of all rows beginning with the symbol `h`. Locating this range is easy as it starts at position *Count*[`h`] and ends before *Count*[`i`]. Rows in this range are sorted according to the second symbol so there will be for example some lines continuing with symbol `a` if the text contained word `hashtag` or symbol `e` if the word `head` was present in the text. Among them, lines starting with `house` are located, but we do not store anything except for *Count* and L. However, we already found the location of suffixes starting with `ouse`. All the symbols `h` in the last column correspond to left rotations of rows 100–105 by one. Note that rotations of row 103 and 104 are rows 201 and 204. Rotations of `hashtag`/`head`/`hind` start with `ash`/`ead`/`ind`, so they come *before* row 200. On the other hand, rotation of row 105, starts with `uge`, which comes *after* `ouse`. To locate the offset of `house` along the lines starting with `h`, we need to count the number of occurences of `h` in L before the occurrences of `ouse`.

In general, to count the number of occurrences of pattern P, we first find the subsequence of rows of M beginning with $P_{n-1\dots}$. As this is just one symbol, p_{n-1}, the initial subsequence is a run of symbol p_{n-1} in F given by

$$\begin{aligned} b_{n-1} &= \mathit{Count}[p_{n-1}] \\ e_{n-1} &= \mathit{Count}[p_{n-1}+1]. \end{aligned}$$

The next step is to find the subsequence of rows of M, given by b_{n-2} and e_{n-2}, which has $P_{n-2\dots}$ as a prefix. As we already located rows beginning with $P_{n-1\dots}$, we can use this information. In a range from b_{n-1} to e_{n-1} some rows end with symbol p_{n-2}. These are rows which after being rotated, create subsequence we are looking for. Subsequence we are looking for is also subsequence of a run of symbol p_{n-2} in F, starting from $\mathit{Count}[p_{n-2}]$ spanning rows up to $\mathit{Count}[p_{n-2}+1]$. To find the beginning and end of our subsequence inside of this run, we compute the number of occurrences of symbol p_{n-2} in L up to b_{n-1}, the start of previous subsequence of rows and also the number of occurrences of symbol p_{n-2} in L up to e_{n-1} giving us its end. This works thanks to the property 2. Thus giving us the new subsequence

$$\begin{aligned} b_{n-2} &= \mathit{Count}[p_{n-2}] + \mathit{rank}_{p_{n-2}}(L, b_{n-1}) \\ e_{n-2} &= \mathit{Count}[p_{n-2}] + \mathit{rank}_{p_{n-2}}(L, e_{n-1}). \end{aligned}$$

	Row n.	F							L
	0								
	...								
$Count[h]$	100	h	a	s	h	t	...		
	101	h	e	a	d				
	102	h	i	n	d			b	e
b_0	103	h	o	u	s	e			
	104	h	o	u	s	e			
e_0	105	h	u	g	e				
	...						...		
b_1	200	o	u	s	e				m
	201	o	u	s	e				h
	202	o	u	s	e				m
	203	o	u	s	e		...	s	p
	204	o	u	s	e				h
e_1	205								
	...								

Figure 1.3: We are counting occurences of `house` and we already found the range of rows starting with `ouse` given by b_1 and e_1. To find the range of rows starting with `house`, we first look at the position where rows starting with `h` begin in F – $Count[h]$. However, not all of the rows starting with `h` continue with `ouse`. To find this subinterval, we use the information from previous iteration. We count the number of occurences of `h` in L before position b_1 and use this as offset into run of symbol `h`. In this particular case, $b_0 = Count[h] + rank_h(L, b_1) = 100 + 3 = 103$ and $e_0 = Count[h] + rank_h(L, e_1) = 100 + 5 = 105$.

We continue and repeat the previous step until we compute b_0 and e_0 or until $b_i = e_i$ for some i (in this case, the searched word is not present in the text). The possible pseudocode for counting the number of occurrences of P in T is in listing 1.

Algorithm 1: Count number of occurrences of pattern P in an FM-index

```
Data: $P \in \Sigma^n$
$b \leftarrow Count[P[n-1]]$;
$e \leftarrow Count[P[n-1]+1]$;
for $i = n-2$ downto $0$ do
    if $b = e$ then
        break;
    end
    $b \leftarrow Count[P[i]] + rank_{P[i]}(L, b)$;
    $e \leftarrow Count[P[i]] + rank_{P[i]}(L, e)$;
end
return $e - b$
```

As we can see, the FM-index requires the method *rank* on a general string. In the next paragraph, we show that we can use bit vectors to provide a reasonable implementation of the *rank*/*select* methods on a sequence over a general alphabet.

Wavelet tree Let us assume for a moment that we have a bit vector implementation supporting methods *access*, *rank* and *select*. We have a sequence S of length n over an arbitrary alphabet Σ. Our goal is to build vector over this sequence supporting the methods *access*, *rank* and *select*.

A straightforward approach that uses bit vector is to have one bit vector B_c for every symbol c from the alphabet Σ, storing ones at positions where c occurs in S

$$B_c[j] = \begin{cases} 1, & \text{if } S[j] = c \\ 0, & \text{otherwise.} \end{cases}$$

This is very fast because each *rank* and *select* operation can be answered using only a single binary *rank* or *select*. However, we use roughly $|\Sigma|$ times more space than the single bit vector over this sequence would use. Or in other words, the space usage is growing linearly with the alphabet size.

Wavelet tree data structure proposed by Grossi et al. (2003) uses a divide-and-conquer approach to solve this problem. It takes the alphabet Σ of size σ and recursively splits the alphabet into two subsets creating a hierarchical partitioning of an alphabet. In the root node of the tree, the alphabet Σ is split into two subsets Σ_0 and Σ_1 of

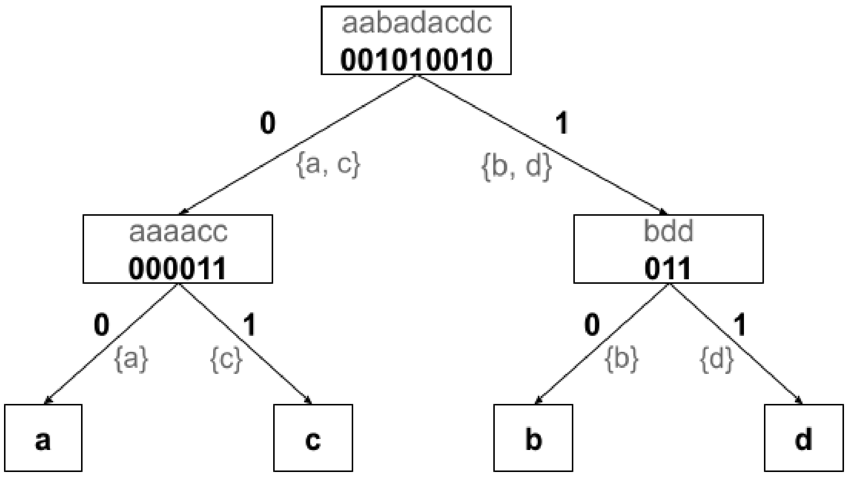

Figure 1.4: Wavelet tree representation of text $S = aabadacdc$. We can see how the recursive partitioning of the alphabet works. In every node, we also show the subsequence represented (grey text) in the subtree of the node. Note that we include the grey parts in the visualisation to help in understanding but they are not stored by the data structure.

roughly equal size. A bit vector B of size n is stored in this node such that

$$B[i] = \begin{cases} 0, & \text{if } S[i] \in \Sigma_0 \\ 1, & \text{otherwise.} \end{cases}$$

Then two strings S_0 and S_1 are created from S by taking just symbols from Σ_0 and Σ_1, respectively. The left and right child of the root node are then built by recursively applying the same idea on subsequences S_0 and S_1 until we end up with a trivial unary alphabet at the leaves. An example of this partitioning is shown in Fig. 1.4. It follows that the depth of the wavelet tree is $\mathcal{O}(\log \sigma)$.

In a wavelet tree, both $rank_c$ and $select_c$ methods on the original sequence can be implemented using *rank*/*select* methods applied on bit sequences that are stored in nodes along the path from the root to the leaf containing symbol c. Thus, the number of *rank* and *select* queries on individual bit vectors depends on the depth of a leaf containing queried symbol c. Regarding the space usage of wavelet tree; as on every level of the wavelet tree, we store roughly a bit vector of length n, it is possible for this representation to use just $n \lg \sigma + o(n)$ bits of space.

Even if this version of wavelet tree can be used inside of the FM-index, it is possible to make a solution that is faster and more space efficient in some scenarios. Time complexity of both methods, $rank_c$ and $select_c$, depends on the time complexity of the bit vector implementation used in the nodes, but also on the depth of symbol c in the tree. This depth is now for every single symbol $\mathcal{O}(\log \sigma)$.

To obtain a better solution, let us give another perspective on the wavelet tree and assign a single bit to every edge – `0` for left and `1` for right (see Fig. 1.4). Now, we can think of the wavelet tree as an assignment of a binary code to every alphabet symbol. The code of a symbol c is obtained by concatenating bits along the path from the root to the leaf for symbol c.

The idea proposed by Mäkinen and Navarro (2005) is to shape the wavelet tree in such a way that a code of every symbol is equal to its Huffman code. The first advantage is that the space used decreases from $\mathcal{O}(n \log \sigma)$ to $\mathcal{O}(nH_0(S))$. The second advantage is that if we query for each symbol according to its frequency, then on average, we need to visit $\mathcal{O}(H_0)$ nodes rather than $\mathcal{O}(\log \sigma)$, where H_0 is the empirical zero-order entropy, defined as

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c},$$

where n_c denotes number of occurrences of symbol c in sequence S of length n. The value of zero-order entropy is a lower bound for the number of bits that S can be compressed to by compressor that considers only frequencies of symbols and $H_0(S) \leq \lg \sigma$. In the worst case, we need $\lg \sigma$ bits to store every symbol. In some cases, however, it is the case that frequencies of symbols are not even and thus $H_0(S) \ll \lg \sigma$.

The maximum depth of a Huffman shaped wavelet tree may be bigger compared to the original, however, Grabowski et al. (2004) showed that we can enforce the maximum depth to be $\mathcal{O}(\log \sigma)$ with the average depth limited by $H_0 + 2$. This is, however, as they claimed, not very practical. The important fact is that with Huffman shaped wavelet tree, it is possible to decrease the space usage in some scenarios but also average number of nodes accessed and thus decrease the average query time of *rank* and *select*.

It was shown by Grossi et al. (2003) that it is also possible for the wavelet tree of original shape to achieve space usage of $nH_0(S) + o(n \log \sigma)$ bits, if the bit vector implementation that takes space close to the zero-order entropy is used inside of the tree nodes.

In practice, it is beneficial to combine these two ideas and use the Huffman shaped wavelet tree with compressed bit vector inside.

Space usage of FM-index We already showed how FM-index is constructed and how the searching in FM-index works and can be supported should we have a data structure that supports *rank* and *select* on sequences over the general alphabet. We also showed how wavelet tree, one such data structure works.

Kärkkäinen and Puglisi (2011) showed that combining Huffman shaped wavelet tree and compressed bit vector taking space close to zero-order entropy brings down the total space used by FM-index to $nH_k(T) + o(n) \log \sigma$ for the text T of length n over alphabet of size σ where H_k is k-th order empirical entropy. This quantity is defined as

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| H_0(w_S)$$

where w_S is the string consisting of concatenation of symbols following w in S. This measure gives us the lower bound for number of bits that S can be compressed to by compressor that considers context of length k when encoding each symbol of S.

In this chapter, we presented some useful applications of the bit vector. We looked more closely on FM-index, the data structure used for the problem of text indexing. There, we encountered the problem of answering *rank* and *select* over the general alphabet and described wavelet tree, one possible data structure that can be used to solve it, assuming we have an implementation of bit vector supporting *access*, *rank* and *select* methods. We also showed that compressed bit vector that takes space close to zero-order entropy can be used to obtain succinct representation of a wavelet tree but also very space efficient version of FM-index.

1.5 Outline of the Thesis

The goal of this thesis is to describe the current state of the bit vector implementations supporting *access*, *rank* and *select* methods and come up with improvements that make bit vectors more usable in practice. This can be either by speeding up the current implementations, saving some additional space but also by combination of these two as obtaining some new trade-offs of query time and space used by the implementation can also open doors to new applications.

In the second chapter, we describe the state-of-the-art implementations of *rank* and *select* methods over bit sequence. We discuss what are the theoretically optimal solutions but also what are their practical drawbacks. We then proceed to describe one of the widely used implementations of a compressed bit vector called *RRR*.

In the third chapter, we propose our own modifications to the implementation of a compressed bit vector based on RRR and we discuss theoretical aspects of these modifications.

In the fourth chapter, we show our proposed implementation of previously devised methods and experimentally test and evaluate them. We measure the performance of our solution on artificial as well as real-world data. Finally, we demonstrate the capabilities of our new bit vector inside of the FM-index.

Chapter 2

Binary sequence representation

As we have shown in the previous section, bit vector can be used inside of the data structures that solve various practical problems. This chapter is dedicated to outlining the current state of the bit vector implementations supporting methods *access*, *rank* and *select*. We start with a succinct representation of bit vector and present ideas to support *rank* and *select* in sublinear extra space. Then we look at the compressed representation of bit vector introduced by Raman et al. (2007).

2.1 Bit vector implementation

In this section, we introduce practical but also optimal constant time implementations of *rank* and *select* methods with sublinear space overhead.

2.1.1 Rank

Regarding *rank* in bit vector, we are concerned with two different methods, namely $rank_0(i)$ and $rank_1(i)$. In every binary sequence, it holds that

$$rank_0(i) = i - rank_1(i).$$

Thus, it is common to only provide the implementation for one of them and answer the other one using the formula above. Thus, from now on, we consider only $rank_1(i)$ method and denote it *rank* to simplify notation. There are two straightforward solutions we can begin with to support *rank* on a bit vector B.

The first solution does not use any precomputation at all. Every time we want to compute $rank(i)$, we go through all the bits preceding i-th and count all the ones. This solution is not practical for long bit vectors but it does not require any additional space and precomputation.

The second approach is to precompute *rank* of every bit, which enables us to answer *rank* query in constant time, using a single table lookup. However, the space needed

B	0 0 1 0 1 1 0 1	1 1 1 0 1 1 1 1	0 0 0 0 0 1 1 0	1 1 0 0 1

precomputed ranks	0	4	11	13

Figure 2.1: Example of dividing bit sequence B into superblocks of length 8 and precomputing $rank_1$ for the beginning of every superblock. Note that last superblock may not contain full number of 8 bits.

to support this solution is $\mathcal{O}(n \cdot \log n)$ bits, where n is the length of the bit sequence.

Presented solutions can be combined to obtain an idea of a practically interesting solution where the pre-computed *rank* values are stored only for some bits. At first, we choose a constant k and split bit vector into non-overlapping subsequences of k bits called *superblocks*. We then precompute *rank* only for the beginning of every superblock. Example of this representation is presented in Fig. 2.1. This representation enables answering *rank* in time $\mathcal{O}(k)$ as only the precomputed value is accessed and then bits from the start of the superblock up to the queried position are accessed. This solution uses $\mathcal{O}(\lceil n/k \rceil \log n)$ bits of memory as there are $\lceil n/k \rceil$ superblocks and every rank stored is at most n, thus taking $\mathcal{O}(\log n)$ bits. This version of the implementation allows us to balance between speed and space usage using the parameter k. Increasing parameter k saves space, on the other hand, smaller k requires less computation to be done inside of the superblock.

In practice, the time of answering the query for smaller k may be dominated by cache miss that often occurs when accessing *rank* precomputed for the superblock. Subsequent linear scan through superblock is very cache-friendly and thus very fast.

Constant time rank with sublinear space overhead The previous solution works well and is commonly used in practice. However, it is possible to answer *rank* query in constant time with only sublinear space overhead. The constant time solution is based on work of Jacobson (1988). As in the previous solution, we start by splitting the bit sequence into superblocks. This time we set the length of a superblock to $\mathcal{O}(\log^2 n)$ and again precompute *rank*s for the beginning of every superblock. Then, every superblock is further subdivided into blocks of length $\lceil (\log_2 n)/2 \rceil$. Similarly to superblocks, for all of these smaller blocks, we precompute *rank*. However, to save space, we only precompute these *rank*s from the beginning of the corresponding superblock.

Now, when queried for *rank* of some position i, we combine:

1. *rank* precomputed for the beginning of its superblock, plus

2. *rank* precomputed for the beginning of its block, plus

3. *rank* inside of its block

The third step can be done also in constant time on unit-cost RAM model with word size $\Theta(\log n)$ that we use in most of our work. In weaker models, we can still use a precomputed table. This table stores for every possible block and *rank* query its result.

The additional space used by this solution can be broken down into three parts:

1. Precomputed *rank* for every superblock: The number of superblocks is $\mathcal{O}(n/\log^2 n)$ and we need $\mathcal{O}(\log n)$ bits to store every single *rank* value so the total amount is
$$\mathcal{O}\left(\frac{n}{\log^2 n} \cdot \log n\right) = o(n). \tag{2.1}$$
2. Precomputed *rank* for every block: The number of smaller blocks is $\mathcal{O}(n/\log n)$. For every block, precomputed *rank* from the beginning of a superblock is stored. This value is at most $\log^2 n$, thus we need $\mathcal{O}(\log\log^2 n) = \mathcal{O}(\log\log n)$ bits to store it. The total amount of space is
$$\mathcal{O}\left(\frac{n}{\log n} \cdot \log\log n\right) = o(n). \tag{2.2}$$
3. Precomputed table storing result for every possible *rank* query over every possible block: There are only $\mathcal{O}(2^{(\log n)/2}) = \mathcal{O}(\sqrt{n})$ blocks of length $\mathcal{O}((\log n)/2)$. The number of possible *rank* queries over a block is equal to its length. For every element stored in the table we need at most $\mathcal{O}(\log\log n)$ bits of space so the total amount of space is
$$\mathcal{O}(\sqrt{n} \cdot \log n \cdot \log\log n) = o(n). \tag{2.3}$$

The total space used for this solution of *rank* is therefore a sum of 2.1, 2.2 and 2.3, which is sublinear in n.

Even if optimal in theory, this solution is not often used in practice as it involves a quite complex implementation and produces 3 cache misses per query: One for accessing the precomputed rank up to the start of the superblock, then another one for the precomputed value of rank to the beginning of the block and in the end also one for accessing the precomputed *rank* value of the block.

2.1.2 Select

In case of a *select* over bit vector, we are again interested in two methods $select_0$ and $select_1$. Even if there is not a simple way how to convert the result of one to another just like with *rank*, we shall be interested mainly in $select_1$ version as the other one can be implemented using the same ideas. The important property of the *select* method is that it works much like an inverse to *rank*. This is given by the fact that

$$rank_c(select_c(i)) = i.$$

Thanks to *rank* being a nondecreasing function, it is possible to binary search for the result of $select_c(i)$ if we have an efficient implementation of *rank*. This can be combined with the solution from the previous section. At first, we binary search for the solution in the samples of *rank* stored in superblocks. After identifying the correct superblock of length k bits, we linearly scan for the result. This solution does not require any additional memory on top of the space used for *rank*. The answer is computed in time $\mathcal{O}(\log(n/k) + k)$. Even though this solution is not optimal, it works very well in practice, as observed by González et al. (2005) on bit vectors of length up to $2^{20} \approx 10^6$.

Constant time select Clark (1998) proposed solution for *select* in constant time and sublinear space overhead. The solution is, similarly to the constant time *rank* solution, based on a division into blocks and superblocks.

We begin by precomputing $select(i)$ for every i being multiple of $t_1 = \log n \cdot \log\log n$. These precomputed values take $\mathcal{O}(n/\log\log n)$ bits of space. Results of these precomputed *select* queries split the bit sequence into superblocks of possibly variable length such that each superblock contains exactly t_1 ones (except possibly for the last superblock).

There are now two categories of superblocks. The ones called *long* that are longer than t_1^2. In long superblocks, we can store the positions of all ones as they are sparse and there are not many of these blocks. The total space used is

$$\mathcal{O}((n/t_1^2) \cdot t_1 \cdot \log n) = \mathcal{O}(n/\log\log n),$$

which is still $o(n)$.

Dealing with *short* superblocks is harder. On short superblocks, we apply the idea that we already used for the original bit sequence. Inside of every short superblock, we precompute the *select* from the beginning of the superblock for every multiple of $t_2 = (\log\log n)^2$. These precomputed values are small as they only represent positions from the beginning of a short superblock. Each of these values takes $\mathcal{O}(\log\log n)$ of space so the total amount of space used by these values is $\mathcal{O}(n/t_2 \cdot \log\log n) = \mathcal{O}(n/\log\log n)$.

This procedure breaks short superblocks into blocks. Again, each of these blocks, possibly except for the last one contains t_2 ones. For blocks longer than t_2^2, we store the positions of all ones as there are not many of these blocks. We may use the same reasoning as in the previous part to conclude that this takes $\mathcal{O}(n/t_2^2 \cdot t_2 \cdot \log\log n) = \mathcal{O}(n/\log\log n)$ bits.

To deal with the blocks smaller than t_2^2, we can again precompute a table of all the possible ways how the block may look and all the possible *select* queries over it with their result. The number of possible blocks of length t_2^2 is equal to $\mathcal{O}(2^{t_2^2})$, the number of possible queries over the block is at most its length t_2^2 and to store the results we need roughly $\mathcal{O}(\log t_2)$ bits. The table size is thus

$$\mathcal{O}(2^{t_2^2} \cdot t_2^2 \cdot \log t_2) = o(n).$$

When answering *select*$_1(i)$, we first find the location of the right superblock. If this is a long superblock, we just look at the precomputed positions of ones in the superblock. The matter is more complicated if we are dealing with a short superblock. In this case, we find the location of the correct block. If this is a long block, we can once again just look at the position of one we are interested in. If it is a short block, we use the precomputed table.

2.2 Compressed representation

In the previous section, we showed how *rank* and *select* queries can be answered in constant time with just sublinear space overhead. In this section, we show that it is possible to compress the whole bit vector close to the zeroth order entropy while still keeping the constant time *rank* and *select*.

Up to now, we have been working with a straightforward representation of bit vector which consists of all the bits, stored one after another. This is the best we can do in general, but there are scenarios where there is a room for improvement. One example is if the zeroth order entropy H_0 of a bit sequence is small. This occurs in sequences that have a very skewed frequency of zeroes or ones. If sequence of length n has m ones in it, we can store it as before using n bits. There is, however, only $\binom{n}{m}$ such sequences and it may be more beneficial for small/big m to store rather the sequence number which one of these $\binom{n}{m}$ sequences we are working with. It is possible to prove for sequence S of length n with m ones that

$$\lg \binom{n}{m} = nH_0(S) - \mathcal{O}(\log n).$$

This means that using this representation may be beneficial in scenarios when $nH_0(S) \ll n$. This is an idea that RRR is based on.

RRR is a data structure based partly on the work of Pagh (2001) and proposed by Raman et al. (2007). We split the bit sequence into blocks of length b and then represent a block with c ones using only $\lg \binom{b}{c}$ bits as there are $\binom{b}{c}$ combinations for positions of ones. The whole block can then be uniquely represented as a pair (c, o) where c is the number of ones in the block, called *class* and o is an *offset* of this block in a sequence of all the $\binom{b}{c}$ blocks in this particular class. Even if the ordering of the blocks with the same class can be arbitrary, only lexicographical ordering is heavily used in practice.

Now, for this structure to work, we need to find a way how to convert between the bit representation of a block and its compressed form (c, o). The process of obtaining

c and o from the raw representation of block is called *encoding*. The opposite process is called *decoding*. We would like both processes to be fast, however, in most of the applications where bit vector is used, we do the encoding only once at the initial construction of the bit vector. Decoding, on the other hand, is done every time we are accessing a particular bit in B. Thus, in practice, it is more important to optimise the speed of decoding.

Block encoding/decoding For shorter block lengths, such as $b \leq 15$, it is reasonable to generate two helper tables E and D. Table E used for the encoding, maps the block to its offset. The other table D is two dimensional and stores the bit representation of the block that is associated with pair (c, o) on position $D[c][o]$. Both these tables E and D can be precomputed by generating all the possible blocks in lexicographical order. After this precomputation, the encoding and decoding of a block takes constant time.

For longer blocks, it is impractical or even impossible to store huge helper tables. On the other hand, longer blocks yield better compression rates because of smaller per block overhead. Navarro and Providel (2012) developed method we shall call *on-the-fly decoding*, that does not require these big helper tables. This method relies on a bit by bit encoding and decoding of the block, taking $\mathcal{O}(b)$ time. While decoding, we can compute on every position, how many blocks precede a given prefix using simple combinatorics. Based on that number, we decide whether next bit should be `0` or `1`. Pseudocode of this idea computing binary representation of block is in listing 2.

On-the-fly decoding does not use helper tables E and D but to achieve the best possible time complexity, it requires the precomputation of Binomial coefficients it uses. These, however, use together less than $\mathcal{O}(b^3)$ of bits.

Arrangement of encoded blocks In the previous paragraph, we showed how a single block can be decoded. The next question is how the encoded pairs representing blocks are arranged in memory. This arrangement should consider space efficiency but also allow easy access to individual blocks of B.

The main part of this representation comprises of two arrays C and O storing the classes and offsets of the blocks, respectively. The array C is an array of elements that are of fixed length. Every element takes $\lg(b + 1)$ bits of space as the number of ones in the block can be anywhere between 0 and b. The array O, on the other hand, is an array of elements of variable length where the i-th element is $\lg \binom{b}{C[i]}$ bits long where $\binom{b}{C[i]}$ is the number of blocks along the class $C[i]$. Note that accessing i-th element in array C is easier than doing the same in array O.

Algorithm 2: On-the-fly decoding

```
Data: $c, o$
$block \leftarrow 0$;
for $i = 0$ to $b - 1$ do
    // $p$ is the number of blocks with the same prefix and zero on
       $i$-th position
    $p \leftarrow \binom{b-i-1}{c}$;
    if $o < p$ then
        // add zero at the end of the block
        $block \leftarrow 2 \cdot block$;
    end
    else
        // add one at the end of the block
        $block \leftarrow 2 \cdot block + 1$;
        $c \leftarrow c - 1$;
        $o \leftarrow o - p$;
    end
end
return $block$
```

Accessing bits in RRR Accessing x-th bit of the original sequence B consists of three steps. The first step is to obtain the compressed representation of a block where x-th bit is located. The second step is to decode this block and the third is to access the particular bit of interest in the decoded block. The x-th bit is contained in the i-th block where $i = \lfloor x/b \rfloor$. Its compressed representation is $(C[i], O[i])$ where $C[i]$ stands for class and $O[i]$ for the offset along the all possible blocks with class $C[i]$.

Obtaining $C[i]$ is trivial as it is at the fixed memory offset from the beginning of array C. Getting the value of $O[i]$ is harder as it is not at a known memory offset but this memory offset can be expressed as

$$\sum_{j=0}^{i-1} \lg \binom{b}{C[j]}$$

, where we basically sum up the lengths of all the elements preceding $O[i]$. This can not be, however, computed in constant time without any precomputed information and we need to basically one by one skip over elements that come before $O[i]$. To access the i-th block without precomputed information, we need in the worst case to look at all the elements of C and this takes $\mathcal{O}(n/b)$ time. For now, let us analyze space usage of this solution.

Space usage of RRR Our current representation needs to store the arrays C, O. Let us now analyze the space used by these structures. Array of classes C is an array of $\lceil n/b \rceil$ elements of fixed length $\lg(b+1)$. For offset array O we argue that its size is bounded by

$$\begin{aligned}\sum_{i=1}^{n/b} \lg \binom{b}{c_i} &\leq \sum_{i=1}^{\lceil n/b \rceil} \log_2 \binom{b}{c_i} + \lceil n/b \rceil \\ &= \log_2 \prod_{i=1}^{\lceil n/b \rceil} \binom{b}{c_i} + \lceil n/b \rceil \\ &\leq \log_2 \binom{n}{\#_1(B)} + \lceil n/b \rceil \qquad \leq nH_0(B) + \lceil n/b \rceil\end{aligned}$$

where $\#_1(B)$ denotes the total number of ones in B. The second inequality was obtained using the observation that $\binom{n}{k}\binom{m}{\ell} \leq \binom{n+m}{k+\ell}$. This can be seen when we interpret the left side as the number of ways we can choose k elements from n elements and ℓ elements from another m elements. Any such choice is included in the right side, which represents choices of $k+\ell$ elements from $n+m$. To understand the last inequality, consider all binary sequences of length n with m ones. There are $\binom{n}{m}$ such sequences, thus in the worst case, the space we need to represent a single one of these sequences is equal to

$$\begin{aligned}\lg \binom{n}{m} &= n \log_2 n - m \log_2 m - (n-m) \log_2 (n-m) - \mathcal{O}(\log n) \\ &= m \log_2 n + (n-m) \log_2 n - m \log_2 m - (n-m) \log_2 (n-m) - \mathcal{O}(\log n) \\ &= n \left(\frac{m}{n} \log_2 \frac{n}{m} + \frac{n-m}{n} \log_2 \frac{n}{n-m} \right) - \mathcal{O}(\log n) \\ &= nH_0(B) - \mathcal{O}(\log n).\end{aligned}$$

The first equation is obtained by using Stirling's approximation.

If we choose the decoding method with a helper table, D is storing 2^b entries and each entry takes b bits of storage. To summarize, the total space used is then

$$nH_0(B) + \lceil n/b \rceil + \lceil n/b \rceil \cdot \lg(b+1) + b2^b.$$

Block access speed up To speed up the process of accessing blocks, we can store pointers to every k-th element of O. This again creates some bigger superblocks as can be observed in Fig. 2.2. This representation speeds up the process of locating offset $O[i]$. Now, we first find the nearest pointer leading to superblock where i is located and then we just skip through the superblock at most k times to locate $O[i]$. This additional structure of $\lfloor n/(bk) \rfloor$ integers uses $\mathcal{O}(\log(n) \cdot \frac{n}{bk})$ bits of space.

When setting the block length to $\log(n)/2$, we obtain interesting practical results as the total space used by our representation is equal to $nH_0(B) + o(n)$ bits. This

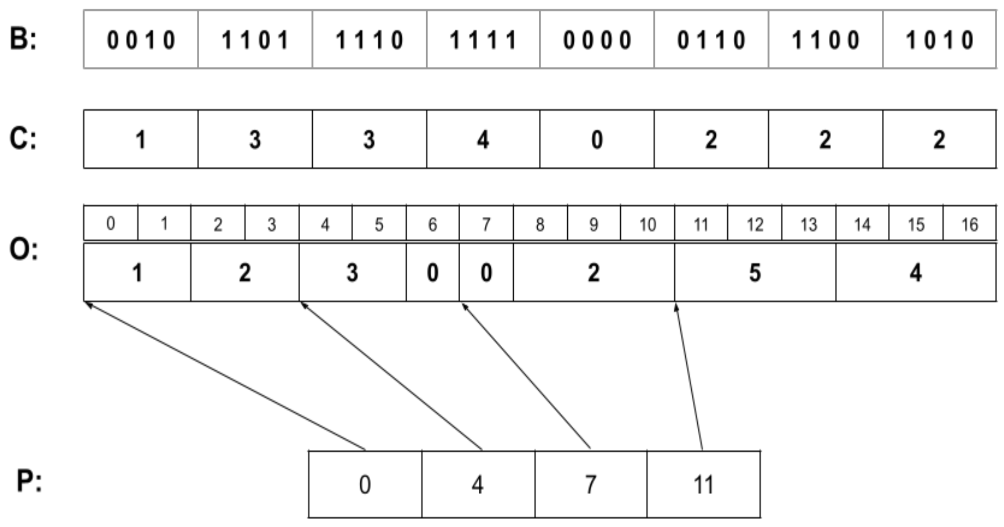

Figure 2.2: RRR implementation. B shows the original bit sequence cut to blocks. C stores the class which is in this case number of ones in the block. O uses variable number of bits per entry, in general, i-th entry uses $\lg \binom{b}{C[i]}$ bits and stores the lexicographical order of this block in the class $C[i]$. For $k = 2$, we can see a helper array P storing bit offsets into every k-th element namely $0, 2, 4 \ldots$

means that we are storing only a sublinear amount of data on top of the zeroth order empirical entropy.

When we are interested in obtaining the best practical results, block length becomes one of the most important parameters of RRR implementation. As we shall show in the Chapter 4, longer blocks yield lower per bit overhead. Also, not all block lengths are used in practice. Very often, we are interested in block lengths of the form $2^k - 1$. This is because the number of ones in block of length $2^k - 1$ can be number between 0 and $2^k - 1$ making this in total 2^k possibilities. Storing this number in the fixed bucket of size k bits makes use of all the available space. This is why most commonly used bucket sizes in practice are 4, 5, 6 and 7 for block lengths 15, 31, 63 and 127.

For block length of 15, the encoding and decoding table each occupies roughly 64kB of space. This is because each table consists of 2^{15} entries each taking 2 bytes of storage. Unfortunately, for block length of 31, these two tables would consume roughly $2^{31} \cdot 4$ bytes of storage. That amounts to roughly 8.5GB of space and makes this approach unusable in practice. This problem forces us to use the on-the-fly decoding for block lengths bigger than 15. The disadvantage of this approach is that it takes $\mathcal{O}(b)$ steps to decode the block. Furthermore, on-the-fly decoding contains branches and it is hard to parallelize its steps in some meaningful way. Overall, this makes the block length a parameter that we can adjust to balance between better space efficiency and faster runtime performance.

Chapter 3

Our contributions

In this chapter, we propose various ideas that we hope lead to a better practical implementation of RRR. We first propose a new general block encoding and decoding routine. Then, we show how we can exploit the assumptions about the represented bit sequence to come up with implementation that is better tailored for the concrete bit sequence in terms of the query time and space usage.

3.1 Block encoding

As we discussed in Section 2.2, there are to the best of our knowledge two widely used methods to encode and decode the blocks in RRR. Disadvantage of the table decoding method is its inability to reasonably support longer blocks in practice because of the huge size of helper tables. On the other hand, the on-the-fly decoding method may be used to support longer blocks with the downside being longer encoding and decoding times. In this section, we propose a new method for block encoding and decoding. The main objective of the new method is to enable use of longer blocks while not hurting the runtime so significantly.

The main idea of our proposed solution is to use a divide-and-conquer approach to break the problem of finding the order of the block B in class c into finding orders of several smaller sub-blocks of B. The potential advantage of this solution is that it may enable us to use the table method to solve the smaller subproblems. To facilitate our solution, we altered the respective order of the blocks along the class and we do not use the lexicographical ordering anymore. Note that we continue to use the number of ones to identify the class of a block. In our solution, every block B will be thought of as a concatenation of two smaller *sub-blocks*, B_1 and B_2. Primarily, the blocks are ordered according to the value of their *class pair* (c_1, c_2) where c_1 and c_2 are the classes of the smaller sub-blocks B_1 and B_2, respectively. Only then, the tie is broken by order of B_1 and B_2 in their respective classes. An example of the new ordering for blocks of

Offset	Block	(c_1, c_2)
0	`000 011`	
1	`000 101`	$(0, 2)$
2	`000 110`	
3	`001 001`	
4	`001 010`	
5	`001 100`	$(1, 1)$
6	`010 001`	
7	`010 010`	

Offset	Block	(c_1, c_2)
8	`010 100`	
9	`100 001`	
10	`100 010`	$(1, 1)$
11	`100 100`	
12	`011 000`	
13	`101 000`	$(2, 0)$
14	`110 000`	

Figure 3.1: An example of the new ordering for block length $b = 6$ and class $c = 2$. Every block is divided into two sub-blocks of length 3. Note the differences to the lexicographical ordering. Block `011 000` at offset 12 is preceded by lexicographically greater blocks `100 001`, `100 010` and `100 100` since it has higher number of ones in the first sub-block.

length 6 is shown in Fig. 3.1.

Formalizing this, we shall write $B \prec_X B'$ if blocks B and B' are of the same length, class and at the same time B precedes B' in ordering X. In this situation, we often refer to block B as being smaller than B'. We write $B \prec_{Lex} B'$ if B precedes B' in lexicographical ordering. Using this notation, our new proposed ordering P can be formalized as:

$$\begin{aligned} B \prec_P B' \iff & [\#_1(B_1) < \#_1(B'_1)] \\ & \vee [(\#_1(B_1) = \#_1(B'_1)) \wedge B_1 \prec_X B'_1] \\ & \vee [B_1 = B'_1 \wedge B_2 \prec_Y B'_2] \end{aligned}$$

where $B = B_1 \cdot B_2$, $B' = B'_1 \cdot B'_2$ and X, Y are arbitrary orderings. Note that as X and Y, we can use lexicographic ordering or we can use the same idea recursively.

In the rest of this section, we are going to explain how encoding and decoding works with our proposed ordering.

Encoding Before providing the general encoding routine, let us demonstrate the process on a simple example and then generalize the ideas behind the process. Imagine encoding block `100010` of length 6 using sub-block length 3. We can see this block in Fig. 3.1 at offset 10.

The class of this block is 2 as there are two ones in the whole block. Obtaining the offset is more complicated; we calculate it by counting the number of blocks preceding `100 010` in class $c = 2$. We divide these blocks into 3 categories:

- Blocks with a smaller number of ones in the first sub-block. (There are 3 such blocks – those beginning with `000`.)

- Blocks with the same number of ones in the first sub-block, but smaller first sub-block. (There are 6 blocks with this property – those beginning with `010` or `001`.)
- Blocks with the same first sub-block, but smaller second sub-block. (There is 1 such block, namely `100 001`.)

Summing up, we get that there are 10 blocks preceding `100 010`. Together with its class, we encode this block as a pair of numbers $(2, 10)$.

Let us generalize the process of encoding block B of length b with sub-blocks of length b_1 and b_2. We also need two orderings X and Y that can be used to order sub-blocks B_1 and B_2, respectively. The first step is to count the number of ones to obtain classes of individual sub-blocks $c_1 = \#_1(B_1)$ and $c_2 = \#_1(B_2)$; thus $c = \#_1(B) = c_1+c_2$. To compute the offset of B we first obtain pairs (c_1, o_1) and (c_2, o_2) by recursively encoding the sub-blocks B_1 and B_2. Then, we obtain the offset of B by counting the number of blocks B' preceding B. These can be divided into 3 categories:

1. B' such that $\#_1(B'_1) < \#_1(B_1)$,
2. B' such that $\#_1(B'_1) = \#_1(B_1), B'_1 \prec_X B_1$, and
3. B' such that $B'_1 = B_1, B'_2 \prec_Y B_2$.

One can observe that these categories closely resemble the definition of our ordering P above.

The number of blocks in the first category is equal to

$$\sum_{i=0}^{c_1-1} \binom{b_1}{i}\binom{b_2}{c-i}.$$

For fixed i, $\binom{b_1}{i}$ counts the number of sub-blocks with i ones and $\binom{b_2}{c-i}$ counts the number of sub-blocks with the remaining $c - i$ ones.

The number of blocks in the second category is equal to

$$o_1 \times \binom{b_2}{c_2}.$$

These are all the blocks with first sub-block smaller than B_1 and any second sub-block with c_2 ones.

The number of blocks in the third category is equal to o_2. This is the number of blocks that have B_1 as the first sub-block and the second sub-block is smaller than B_2.

The resulting offset of block B is given by the sum of these 3 quantities.

Decoding We now propose a general decoding scheme for block B of length b combined from the sub-blocks of lengths b_1 and b_2. Decoding is a process of obtaining bit representation of block B from its encoded representation (c, o). When our new encoding scheme is used, the main part of the decoding process is to obtain encoded representations (c_1, o_1) and (c_2, o_2) of sub-blocks B_1 and B_2. These can then be fed into the decoding subroutines to obtain sub-blocks B_1 and B_2. Concatenating these two representations gives us the decoded block B.

Our first step when decoding is to find c_1 and c_2 using o. Assume that our block has an unknown class pair (c_1, c_2), such that $c_1 + c_2 = c$. All the blocks with class c are primarily sorted by class pair so blocks with the same class pair form a consecutive range. Range of class pair (c_1, c_2) contains $\binom{b_1}{c1} \times \binom{b_2}{c2}$ blocks. To find the class pair of block with offset o we precompute the beginnings of each range $Z_0, \ldots, Z_c$[1] where Z_i is the beginning offset of block having class pair $(i, c-i)$ and then binary search to find the correct range. For example in Fig. 3.1 we basically need to decide, whether the offset belongs to the range 0–2 with class pair $(0, 2)$, or the range 3-11 with class pair $(1, 1)$, or the range 12–14 with class pair $(2, 0)$.

Assume that our block has the class pair equal to (c_1, c_2). The second step is to find o_1 and o_2. Let

$$o' = o - Z_{c_1}$$

be the offset from the beginning of the range of blocks with class pair (c_1, c_2).

There are $\binom{b_1}{c_1}$ and $\binom{b_2}{c_2}$ ways how the first and second sub-block may look, respectively. As all the combinations of the first and the second sub-block are possible, we want to identify the o'-th block among the range of $\binom{b_1}{c_1} \cdot \binom{b_2}{c_2}$ blocks. The blocks are sorted primarily by the first sub-block and then by the second sub-block. This means that the ordered sequence of blocks within the class pair (c_1, c_2) will be just a cyclically repeating sequence of all the possible $\binom{b_2}{c_2}$ second sub-blocks, every time with a different, yet increasing first sub-block. Furthermore, this cycle of $\binom{b_2}{c_2}$ sub-blocks will repeat $\binom{b_1}{c_1}$ times (once for each possible first sub-block). Using o' we can easily find the order of the first and the second sub-block. We just need to know how many cycles of length $\binom{b_2}{c_2}$ fit into o' and what is the offset of the second sub-block at which the last cycle is positioned when reaching offset o'. This is straightforward to compute as:

$$o_1 = \left\lfloor o' \middle/ \binom{b_2}{c_2} \right\rfloor \qquad o_2 = o' \bmod \binom{b_2}{c_2}$$

Now we have obtained (c_1, o_1) and (c_2, o_2) so we can reuse the decoding subroutines for the block lengths b_1 and b_2.

In this section, we proposed new block ordering and showed that it can be encoded and decoded efficiently. Our method for block length b_1 and b_2 yields to a block

[1]more precisely $Z_{\max(c-b_2,0)}, \ldots, Z_{\max(c-b_1,0)}$

length $b_1 + b_2$. This gives us an opportunity to obtain block length b using different combinations of b_1 and b_2 and even combine different orderings.

3.2 Hybrid encoding

In this section, we would like to focus more on the space-saving aspect of RRR and use some assumptions about the input bit sequence to change how blocks are represented and by doing it, save some additional space. Storing block of length b, in a raw bit representation always takes exactly b bits of space. The number of bits used by compressed form, however, depends heavily on the blocks class. If the block has class c, then the compressed representation uses $\lg(b+1)$ bits to store its class and $\lg \binom{b}{c}$ bits to store the offset along the class. In Fig. 3.2, we show how the space saved per block depends on the block's class. At the same time, we see that RRR saves most of the memory on very sparse and very dense blocks. We would like to exploit situations when the frequency of these blocks is very high.

Sequences with fixed densities First scenario, that is easy to study and analyze, is devising the best possible RRR implementation for sequence that contains fixed percentage of ones that are randomly distributed. In order for this scenario to be interesting, taking into considerations the practical capabilities of RRR, we shall be most interested in cases when percentage of ones is between 5–20% as was also studied by Navarro and Providel (2012). Let us consider a randomly generated bit sequence B of length n containing $p\%$ of ones. Let us analyze what is the probability distribution of the block classes in a sequence containing this fixed percentage of ones distributed randomly.

For a block length b, the random variable X denoting number of ones in a single block follows a binomial distribution

$$X \sim Bin(b, p).$$

As we may observe in Fig. 3.3, the probability that block contains a lot of ones decreases exponentially in a sequence containing only 5% of ones. Even in very long sequences, these blocks occur very rarely. This led us to the idea of an encoding method we call *hybrid encoding.*

The hybrid encoding uses a property that in some sequences blocks with high number of ones are very rare. Encoding these blocks is not that beneficial as their compressed representation may take even more bits than the original raw representation. Storing the whole block may waste space if the number of ones is big but there are not many possibilities how the block may look. Although we waste some space by this approach from time to time (for every block that is densely packed with ones), we can

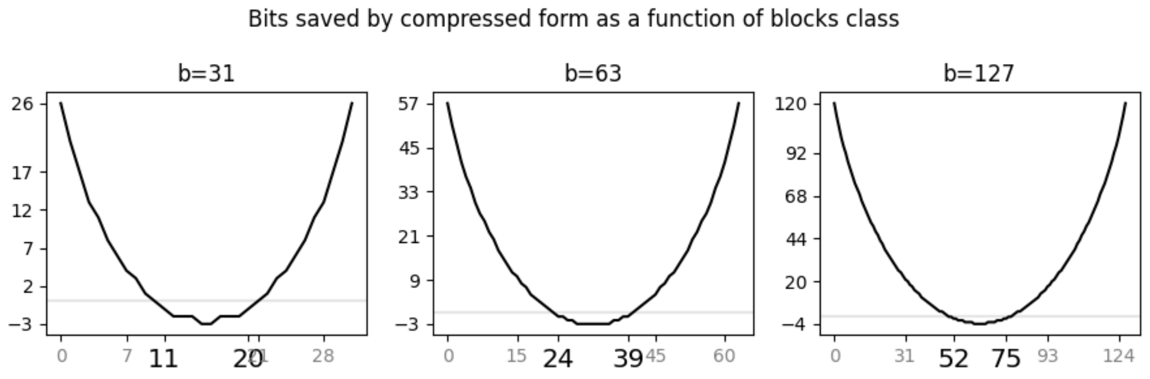

Figure 3.2: Individual graphs present for different block lengths (31, 63, 127), the space saved by using the compressed block representation. We can observe how the blocks class influences number of saved bits. Black numbers on the x-axis denote start and end of the interval where compressed version is even worse in space used (negative number of bits saved).

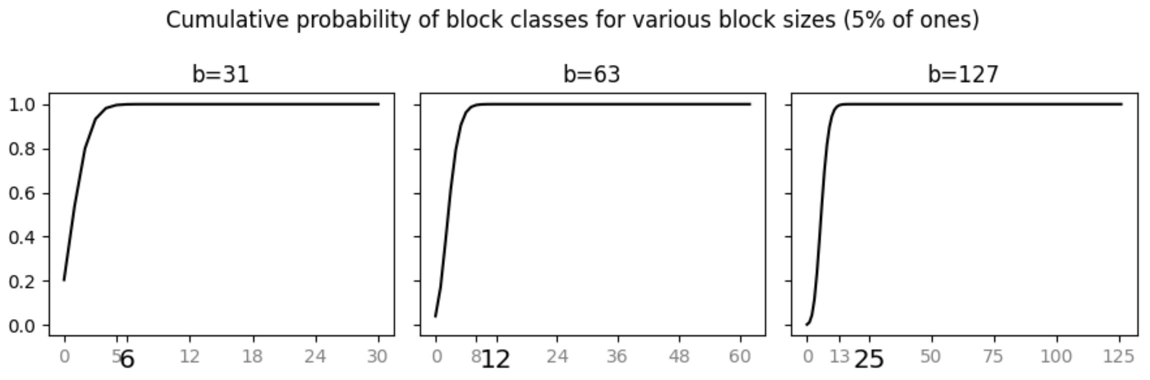

Figure 3.3: On these 3 graphs, we can see the cumulative distribution of blocks classes for the block lengths 31, 63, 127. The frequency of ones is fixed to 5%. Note the marked classes on the x-axis. Numbers 6, 12 and 25 mark the place up to which 99% of probability distribution lies for the block lengths 31, 63 and 127 respectively.

save a small amount of space by decreasing the number of bits used to store the classes. The main idea of hybrid encoding is that every block with a class bigger than some threshold has its class set to this threshold and this means that the block is stored using full b bits instead of $\lg \binom{b}{c}$ bits. Let c_k be a cutoff value. Blocks with class bigger than c_k will not be encoded but just copied with the class set to c_k no matter what is the number of ones in them.

$$\text{stored class} = \begin{cases} \#_1(B), & \text{if } \#_1(B) < c_k \\ c_k, & \text{otherwise} \end{cases}$$

Possible value for blocks class is a number from 0 up to b. When the cuttoff c_k is used, the class of block is a number from 0 up to c_k. To possibly save some space on the bit representation of the blocks class, we need to choose c_k such that

$$\lg(c_k + 1) < \lg(b + 1).$$

Let us now compare these two representations and the theoretical space savings that can be obtained.

Let n be a length of the sequence, b the block length and C_i the number of blocks with class i. To simplify the calculations in this section, we assume that n is divisible by b.

The first representation takes

$$(n/b)\lg(b+1) + \sum_{i=0}^{b} C_i \lg \binom{b}{i}$$

bits of space with the first and second term being the number of bits that is used by the classes and offsets respectively. The hybrid representation with cutoff c_k takes

$$(n/b)\lg(c_k+1) + \sum_{i=0}^{c_k-1} C_i \lg \binom{b}{i} + \sum_{i=c_k}^{b} C_i b$$

bits of space. We would like to find out what is the expected space that we save using the hybrid encoding. We start by simplifying the expected value of the difference of these representations:

$$\begin{aligned} E[\Delta\text{space}] &= E\left[(n/b)(\lg(b+1) - \lg(c_k+1)) + \sum_{i=c_k}^{i \le b} C_i \cdot \left(\lg \binom{b}{i} - b\right)\right] \\ &= (n/b)(\lg(b+1) - \lg(c_k+1)) + \sum_{i=c_k}^{i \le b} E\left[C_i \cdot \left(\lg \binom{b}{i} - b\right)\right] \\ &= (n/b)(\lg(b+1) - \lg(c_k+1)) + \sum_{i=c_k}^{i \le b} E\left[C_i\right] \cdot \left(\lg \binom{b}{i} - b\right) \end{aligned}$$

$$E[C_i] = (n/b) \cdot \binom{b}{i} \cdot p^i \cdot (1-p)^{b-i} \tag{3.1}$$

p	b	c_k	used space
	31	7	0.83
	31	15	0.91
	63	15	0.91
5	63	31	0.95
	127	7	1.86
	127	15	0.93
	127	63	0.98

p	b	c_k	used space
	31	7	0.90
	31	15	0.94
	63	15	0.94
10	63	31	0.97
	127	7	2.03
	127	15	1.22
	127	63	0.99

Figure 3.4: We may observe the space used by the hybrid implementation to space used by our standard RRR implementation for various percentage of ones in the text – p as well as for different block lengths – b and cutoff value – c_k.

Here we mainly used the linearity of the expected value. To compute the expected value of C_i we used the fact, that number of blocks of a certain class follows binomial distribution. We know that in the bit sequence, there is $p\%$ of ones. There are n/b blocks and each has probability $\binom{b}{i}p^i(1-p)^{b-i}$ to be of class i. This probability is independent from the previous blocks and as we are interested in total number of blocks with this class, it follows that C_i follows binomial distribution. As

$$C_i \sim Bin\left(n/b, \binom{b}{i}p^i(1-p)^{b-i}\right)$$

it follows that the expected value is given by formula (3.1).

We present the expected space used by hybrid implementation for some chosen values of p, b and c_k in Fig. 3.4. We may observe, that for some choices of cutoff, we may expect to save roughly 10–17% of space compared to the original method. On top of this, hybrid encoding approach has a potential to be of the same or even better performance as it is easier to decode the blocks saved in the raw form. However, we should note that hybrid encoding is based on our expectations that the number of these blocks is low, so the possible speedup may not be very significant.

Two-sided cut-off The previous version of hybrid encoding may work well for blocks with low density of ones. However, we would also like to use this idea as a viable solution for sequences that contain similar frequencies of zeroes and ones but still exhibit behaviour that some block classes are underrepresented. This may, for example, occur when sequence has a property, that it is globally balanced, but is very unbalanced in some local parts. Then it may be the case, that blocks with low count of zeroes or ones are more frequent than blocks with balanced counts of zeroes and ones.

Let us say that we want to only store r classes instead of all $b+1$ of them. To make

Class	0	1	2	3	4	5	6	7
Hybrid mapping	0	1	2	2	2	2	2	7

Figure 3.5: Encoding of blocks class. Mapping class values in closed range $[cut\_from, cut\_to]$ to single value for $cut\_from = 2, cut\_to = 6$

Class	0	1	2	3	4	5	6	7
Hybrid mapping	0	1	3	3	3	3	3	2

Figure 3.6: Example of hybrid encoding for balanced sequences. $[cut\_from, cut\_to]$ to single value for $cut\_from = 2, cut\_to = 6$. All possible values can be stored using 2 bits as there are 4 possible values. Hybridly encoded pairs have as class value the biggest possible value we can store using 2 bits.

some space saving possible, it must hold that

$$\lg r < \lg(b+1).$$

We create a range of classes given by a closed interval $[cut\_from, cut\_to]$. If the block's class is outside of this range, it is encoded in a standard way. On the other hand, all the values inside of this interval are mapped to one particular class number and the corresponding block is stored in a raw form instead of its offset. The special block class value could be without loss of generality the lowest value cut_from. An example of this idea can be observed in Fig. 3.6. In this example, the number of different possible class values r, is equal to 4. However, using this representation, we would not save any space as the maximum value of class is still 7 and we need 3 bits to store it. This would consume the same number of bits to store as original representation. It is easy to see, however, that these values can be mapped into the range $[0, 3]$, thus only consuming two bits per class value.

Space used by this representation is equal to

$$(n/b)\lg(c_k+1) + \sum_{i=0}^{cut\_from-1} C_i \lg\binom{b}{i} + \sum_{i=cut\_from}^{cut\_to} C_i b + \sum_{i=cut\_to+1}^{b} C_i \lg\binom{b}{i}.$$

To summarize, we proposed two different ideas to enhance current implementations of RRR. The first idea is related to the way blocks are encoded and decoded. We devised a new ordering that can be nicely combined with divide-and-conquer approach and reuse table decoding subroutine for smaller sub-problems to which we divide the original problem.

The second idea, targeted mainly on sequences with special properties, was to just store blocks that are rare in their original bit representation. By using their original representation, we lose some space on these blocks but save space on every single class value, as we do not need to distinguish between blocks that have certain classes. We devised two versions of hybrid encoding, one more suitable for bit vectors that are very sparse and contain small percentage of ones (5–10%). We shall call this *one-sided version of hybrid encoding* as it encodes in a standard way just blocks with smaller number of ones. The second version, is mainly targeted on bit sequences that have balanced number of zeroes and ones but contain many blocks with either small or high number of ones. We refer to this one as *two-sided version of hybrid encoding.*

Chapter 4

Implementation and results

In this chapter, we propose the implementation of ideas we have developed in the previous chapter and show experimental results obtained when measuring their performance. This chapter is split into two sections according to the main topics discussed in the previous chapter, namely our new general encoding and decoding routine and hybrid encoding. In both of these sections, we first describe the important parts of our implementation and then describe how we benchmarked our implementation and present the results we measured.

The source code of our implementation written in `C++` along with all benchmarks can be found in the electronic attachment to this work but also on `Github` (see Appendix A for further information on how to reproduce our results). All the results presented in this chapter were obtained on a machine with 8-core AMD Ryzen 7 2700X with 16 MB of cache, running at 3.7 GHz with 16 GB of RAM. The running operating system was Ubuntu 20.04.3 LTS. We used both `GCC` and `Clang` compilers with versions 9.4.0 and 10.0.0, respectively. All available optimizations were turned on most of the time. There are no special hardware requirements, but to obtain the best possible results, our implementation requires a processor with support for `SSE2` instruction set.

SDSL library We decided to make our solution a part of the `SDSL` library (Gog et al., 2014). This is one of the most mature and versatile libraries implementing succinct data structures. It is written in `C++` and with almost 2 000 commits from more than 30 contributors, `SDSL` is a heavily tested library, offering various implementations of succinct structures such as bit vector, integer vector, wavelet tree, FM-index, suffix array and many more. It allows easy use of different building blocks to implement more complex data structures, e.g., using different bit vector implementations inside of a wavelet tree. On top of this, we also took advantage of thorough tests and benchmarks that were implemented alongside the main functionality.

4.1 New decoding method

4.1.1 Implementation

RRR in SDSL The RRR implementation of bit vector in `SDSL` is provided by templated class `rrr_vector` that enables use of block length from 3 up to 256. To support longer blocks, `SDSL` implements 128-bit and 256-bit integers. The implementation generally uses the on-the-fly decoding. The table decoding method is provided for block length 15 by template specialization of `rrr_vector`.

To support *access*, `rrr_vector` supports single bit access operator `[]` as well as `get_int` method. To facilitate *access*, `rrr_vector` uses the implementation presented in Fig. 2.2 that consists of the array C of fixed sized elements, storing classes, array O of elements of variable length that stores offsets of blocks and third array P, that stores pointers to the array O, more precisely to the beginning of every superblock.

`SDSL` uses *rank* implementation, where the result of *rank* is precomputed for the beginning of every superblock. To answer the query, we first access the precomputed value for the superblock. Then we do a linear search for the final result inside of the superblock. The first part of answering *select* is to binary search between precomputed values of *rank*, then to do the *select* inside of the superblock. This demonstrates, how the size of superblock can be used to balance the ratio of space used and the speed of the *access*, *rank* and *select* methods. Even if the size of a superblock is one of the parameters of `rrr_vector`, we did not make any changes to the default number of blocks per superblock that is set in `SDSL` to 32.

We decided to use the specialization for block length 15 as an underlying solution for the encoding and decoding of sub-blocks. We provided specialized implementations for block lengths 31, 63 and 127 that are most used in practical scenarios. We based our implementation on the general implementation of `rrr_vector` and tried to keep the number of changes as small as possible to easily observe the effects of our new decoding method. Thanks to the modularity of `SDSL`, we have been able to implement our changes and at the same time alter only two methods that the `SDSL` uses for encoding and decoding namely `bin_to_nr(bin)` and `nr_to_bin(k, nr)`. As we mentioned, the encoding is less performance-critical in most of the applications as it is done only once when constructing the bit vector. This is why we focus more on the decoding part of the implementation.

Division of decoding problem into sub-problems The decoding routine can be divided into three steps. The first is to divide problem into subproblems, second is to use subroutines to decode subproblems and third is to combine the results obtained to the final results. When implementing our decoding routine, we put most of our focus

Z_k	class pair	value	offsets mapped
Z_0	$(0, 2)$	0	0–2
Z_1	$(1, 1)$	3	3–11
Z_2	$(2, 0)$	12	12–14

Figure 4.1: The table shows all the class pairs with their corresponding value Z for $b = 6, c = 2$.

on the process of dividing a problem into sub-problems of smaller size. This is the part, where we obtain pairs (c_1, o_1) and (c_2, o_2) from the encoded pair (c, o).

There are various reasons why we focus on this part of the algorithm. The first reason is that for smaller blocks solving the sub-problems is done using the helper table which is quite fast and can hardly be made any faster as it consists of only one table lookup. The second reason is that the dividing of the problem is blocking us from solving the sub-problems and even though sub-problems may be potentially solved somehow in parallel, e.g., by instruction-level parallelism, this part is harder to parallelize.

Let us now break down the process of dividing the original problem into sub-problems into the following 3 steps:

1. Finding the class pair (c_1, c_2).

2. Counting the number of possibilities for first and second sub-block $\text{pos}_1, \text{pos}_2$.

3. Computing offsets of sub-blocks o_1 and o_2.

The most trivial part is the third step as it only consists of a number of arithmetic operations. Then, second step, that is a computation of pos_1 and pos_2. We know that pos_1 and pos_2 is equal to $\binom{b}{c_1}$ and $\binom{b}{c_2}$, respectively. These two numbers can be computed beforehand as there are only roughly b^2 combinations of possible pairs of c and c_1. Roughly speaking, we can do the last two steps at the price of two cache misses.

The approach we use is to first precompute for every possible class c, numbers $Z_0, \ldots, Z_c$[1] where Z_i is the offset of first block with class pair $(i, c - i)$. We map the offset of the block to the number Z_i and thus identify (c_i, c_j). As Z-numbers form an increasing sequence, we can binary search for "group" that contains our offset. Example of these values can be observed in Fig. 4.1.

Despite having good time complexity, binary search may not be the fastest solution in practice as the number of buckets where our offset may land is bounded by $b+1$ and thus quite small. As we found out and later demonstrate, linear search outperforms

[1]more precisely $Z_{\max(c-b_2,0)}, \ldots, Z_{\max(c-b_1,0)}$

binary search most of the time. Another idea that we tested was to speed up the linear search using SIMD instructions. These can be used to do 4 comparisons at a time using a special 128-bit register. In practice, we found that sequential search using SIMD instructions leads to the best results for smaller block lengths. Examples for block length 30 of possible implementations that we tested, follow in listings 4.1, 4.2, 4.3. The naming convention adheres to SDSL naming of k for class, nr for the offset. To simplify naming, if block B was encoded using division to sub-blocks B_1 and B_2 such that $B = B_1 \cdot B_2$ then we call B_1 the *left sub-block* and consequently its class left_k while we call B_2 the *right sub-block*.

Listing 4.1: Linear search for classes of sub-blocks (b=30)

```
uint32_t get_left_class(uint8_t k, uint32_t nr) {
  int left_k_from = std::max(k - 15, 0);
  int left_k_to = std::min(k, 15);
  int left_k = left_k_from;
  for (; left_k < left_k_to; ++left_k) {
    uint32_t curr_index = Z[k][left_k+1];
    if (curr_index >= nr) {
      if (curr_index == nr)
        ++left_k;
      break;
    }
  }
  return left_k;
}
```

Listing 4.2: SIMD enhanced linear search for classes of sub-blocks (b=30)

```
uint32_t get_left_class(uint8_t k, uint32_t nr) {
  int left_k_from = std::max(k - 15, 0);
  int left_k_to = std::min(k, 15);
  __m128i keys = _mm_set1_epi32(nr);
  __m128i vec1 =
    _mm_loadu_si128(reinterpret_cast<__m128i*>(&Z[k][0]));
  __m128i vec2 =
    _mm_loadu_si128(reinterpret_cast<__m128i*>(&Z[k][4]));
  __m128i vec3 =
    _mm_loadu_si128(reinterpret_cast<__m128i*>(&Z[k][8]));
  __m128i vec4 =
    _mm_loadu_si128(reinterpret_cast<__m128i*>(&Z[k][12]));

  __m128i cmp1 = _mm_cmpgt_epi32(vec1, keys);
  __m128i cmp2 = _mm_cmpgt_epi32(vec2, keys);
  __m128i cmp3 = _mm_cmpgt_epi32(vec3, keys);
  __m128i cmp4 = _mm_cmpgt_epi32(vec4, keys);

  __m128i tmp1 = _mm_packs_epi32(cmp1, cmp2);
  __m128i tmp2 = _mm_packs_epi32(cmp3, cmp4);
  uint32_t mask1 = _mm_movemask_epi8(tmp1);
  uint32_t mask2 = _mm_movemask_epi8(tmp2);

  uint32_t mask = (mask2 << 16) | mask1;

  int left_k = left_k_to;

  if (mask != 0)
  {
    left_k = (1 + __builtin_ctz(mask)) / 2;

    if (Z[k][left_k] > nr)
      --left_k;
  }
  return left_k;
}
```

Listing 4.3: Binary search for classes of sub-blocks (b=30)

```
uint32_t get_left_class(uint8_t k, uint32_t nr) {
  int left_k_from = std::max(k - 15, 0);
  int left_k_to = std::min(k, 15);
  // std::upper_bound(a, b, val) returns pointer to first elem. greater than
      val in <a;b)
  auto it = std::upper_bound(Z[k].begin() + left_k_from,
                                   Z[k].begin() + left_k_to, nr);
  // returns index of it in array Z[k]
    int left_k = std::distance(Z[k].begin(), it);
    if (Z[k][left_k] > nr)
      --left_k;
  return left_k;
}
```

Choosing the right sub-problem size In the previous chapter, we showed that our method can take encoding and decoding subroutines for block lengths b_1 and b_2 and combine them together to obtain routines for encoding and decoding of block length

$b_1 + b_2$. It is up to us, how we obtain the decoding for block lengths bigger than 15. A minor inconvenience is, that the most interesting block lengths are of the form $2^i - 1$. Combining two block lengths of this form, however, leads only to a block length of the form $2^{i+1} - 2$ so to obtain a usable block length again, we need to "extend" this solution by one bit.

We based our solutions on table lookup for block length 15 and combined two 15-bit blocks to obtain 30-bit block decoding. To obtain the next interesting block length 31, we then extended this 30-bit solution by one bit. To obtain block length 63, there are, however, multiple possible combinations that look reasonable. We may take 62-bit solution that is built from two 31-bit blocks and then again extend it. On the other hand, it may be more beneficial, to do 63-bit block length by splitting it into 3 and 60 bit block. Thus, before choosing what solution we want to fully include into `SDSL`, we explored and tried several promising combinations and microbenchmarked them.

4.1.2 Experimental results

In this subsection, we show how we measured the performance and practical usability of our solution and also show the results of our experiments.

To measure the performance of our implementation, we mainly relied on two sorts of benchmarks. The first type used a `Google Benchmark` library. This is one of the standard libraries used for microbenchmarking of code. A code snippet that is being evaluated is run many times until stable results are obtained. This makes the results reliable even if the measured time is very small and also limits interference of other running processes on the results. The limitation of microbenchmarks is that they are more artificial in nature and do not give the best possible sense of how the solution may behave on real data.

The second type of benchmarks we used are the ones included in `SDSL`. These are testing bit vector on real data and also as a part of the more advanced data structure. They use data from the `Pizza&Chili` datasets (Ferragina and Navarro, 2005). These include many types of data such as DNA sequences, `C` and `Java` source codes from `Linux` and `GCC` projects as well as English texts from the Gutenberg project. More information about this data such as statistics about the compressibility can be found in the work of Ferragina et al. (2009), Section 4.2. We only provide information necessary to explain the measured phenomena later when exploring the results of our benchmarks.

Microbenchmarks of block decoding These benchmarks were used in the early stage of the development to measure a potential gain from our new method of encoding and decoding. We mainly focused on measuring different block lengths in combination with different methods used for dividing a block into subproblems and with different approaches to obtain particular block length. The tested block lengths were 15, 30, 31, 62, 63 and 127. Lengths 15, 31, 63 and 127 are most useful in practice, however, they can be obtained using different combinations of block length 30 and 62, e.g., 31 can be combined as 1 bit and 30-bit solution, 63 can be divided to 1 and 62 as well as to 3 and two sub-problems of length 30. Even if these tests were not made on practically useful data, they were mainly used as an indication which implementations are interesting and should be the best candidates for further evaluation.

We tested 3 different techniques to divide a problem into sub-problems. These were a linear scan, binary search and linear scan enhanced by `SIMD` instructions. In order to implement the solution for 127-bit block length, we used the 128 bit type `__uint128_t` provided by both `GCC` and `Clang` compilers. This type is implemented on platform `x86_64` as a combination of two 64-bit integers. We compared these results with the table decoding specialization in `SDSL` and with on-the-fly decoding. We ran the deconding algorithm on randomly generated data. We generated 1 000 pairs of class and offset by first randomly picking class of the block and then the offset along this class. To make the comparison fair, for the on-the-fly decoding, we measured the time to decode the middle bit as this decoding method can stop once it obtained the queried bit. The results we measured are shown in Tab. 4.2.

As we can see, our implementation is faster than on-the-fly decoding provided by `SDSL` on every block length that was tested. Linear search outperforms binary search on every block length tested but the binary search is slowly catching up with increasing block length. For 30-bit blocks, the `SIMD` version dominates both linear and binary searhc approaches. This version is, however, harder to use for longer blocks as it requires support of special instructions. Version obtaining 63-bit block using one 3-bit and two 30-bit blocks seems to be performing slightly better than the other variant that extends block length 62 by one bit. Important observation is that our 63-bit version was measured to be even faster than on-the-fly decoding of a 31-bit block.

Bit vector on real data After benchmarking and finding the potential for speeding up the bit vector using our new method, we decided to implement and benchmark our implementation on data that are closer to the real-life usage of bit vectors.

We used a benchmarking part of `SDSL` targeted on `rrr_vector` class. This test measures on different types of data how query time for *access*, *rank* and *select* depends on the block length used. The first type of sequence is a randomly generated bit sequence with density of ones equal to 50%. The origin of second type of data is closely described in Gog and Petri (2014). It consists of numerous bit vectors, stored in files such as `WT-DNA-16MB` or `WT-WEB-1GB`. The structured name describes how the bit vector was created. For instance, `WT-DNA-1GB` has been created by:

- Taking 1GB prefix of a file containing DNA sequence.

Block length	Method	ns per query
15	SDSL_Table	2
	SDSL_ON_THE_FLY	18
30	OUR_LINEAR_15_15	9
	OUR_SIMD_15_15	8
	OUR_BINARY_15_15	15
	SDSL_ON_THE_FLY	38
31	OUR_LINEAR_1_30	11
	SDSL_ON_THE_FLY	37
62	OUR_LINEAR_31_31	36
	OUR_BINARY_31_31	42
	SDSL_ON_THE_FLY	50
63	OUR_LINEAR_1_62	33
	OUR_LINEAR_3_30_30	32
	SDSL_ON_THE_FLY	48
127	OUR_LINEAR_1_63_63	88
	OUR_BINARY_1_63_63	91

Figure 4.2: Results of microbenchmarking various types of decoding implementations. The numbers in method name denote to what block lengths the problem was broken. The name also reflects what method was used to break the problem into subproblems.

- Creating BWT of this text.
- Building Huffman shaped wavelet tree over it.
- Concatenating all the bit vectors in the wavelet tree.

Other versions of test data have been created similarly. Queries into the underlying bit vector were generated randomly.

As a baseline, we choose the 15-bit specialized implementation already implemented in `SDSL` that uses the table decoding method. We present the measured results of our and the original implementation in Fig. 4.3.

There are several interesting things to observe in these results: Our new implementation beats the older implementation on almost all block lengths and all types of data. Note that the difference is more visible on random and DNA data. On web data, however, the difference is less noticeable. We attribute this behaviour to the fact, that was observed by creators of this dataset – the number of uniform blocks (full of either zeroes or ones) is much bigger in `WT-WEB` than in `WT-DNA`. For block length 63, they observed that `WT-WEB-1GB` contains 84% of uniform blocks compared to 28% in `WT-DNA`. As the uniform blocks are decoded trivially in both implementations, this makes less opportunities for our implementation to save time.

Another visible pattern is that with increasing block length, the query time generally goes up as decoding takes more and more time. On the other hand, on *select* in `WT-WEB` data, we can see that at the beginning the query time decreases with increasing block length. Similar behaviour on this data was observed by Gog and Petri (2014) and attributed to the faster binary search in precomputed *rank* values offsetting the growing decoding time.

Overall, we saved almost 50% of time on *access*, *rank* and *select* over randomly generated data, and close to 50% of time on `WT-DNA` data when compared to the baseline implementation. Our results are positive also on `WT-WEB`, particularly for block length 127. This block length is particularly faster in our implementation and we argue this could be also due to the inefficiencies in `SDSL`s implementation of 128-bit integer but also due to the fact, that after dividing the problem into subproblems, we continue to work with 64-bit integers which are much faster on current machines.

Bit vector in FM-index Even if the previous benchmark was done on realistic data, we wanted to test our implementation in a practically useful application. We benchmarked our bit vector as a part of Huffman shaped wavelet tree used inside of the FM-index. Implementation of FM-index in `SDSL`, as most of the other implementations, provides mainly 3 methods:

- *count*(P) – returns the number of occurrences of P in text T,

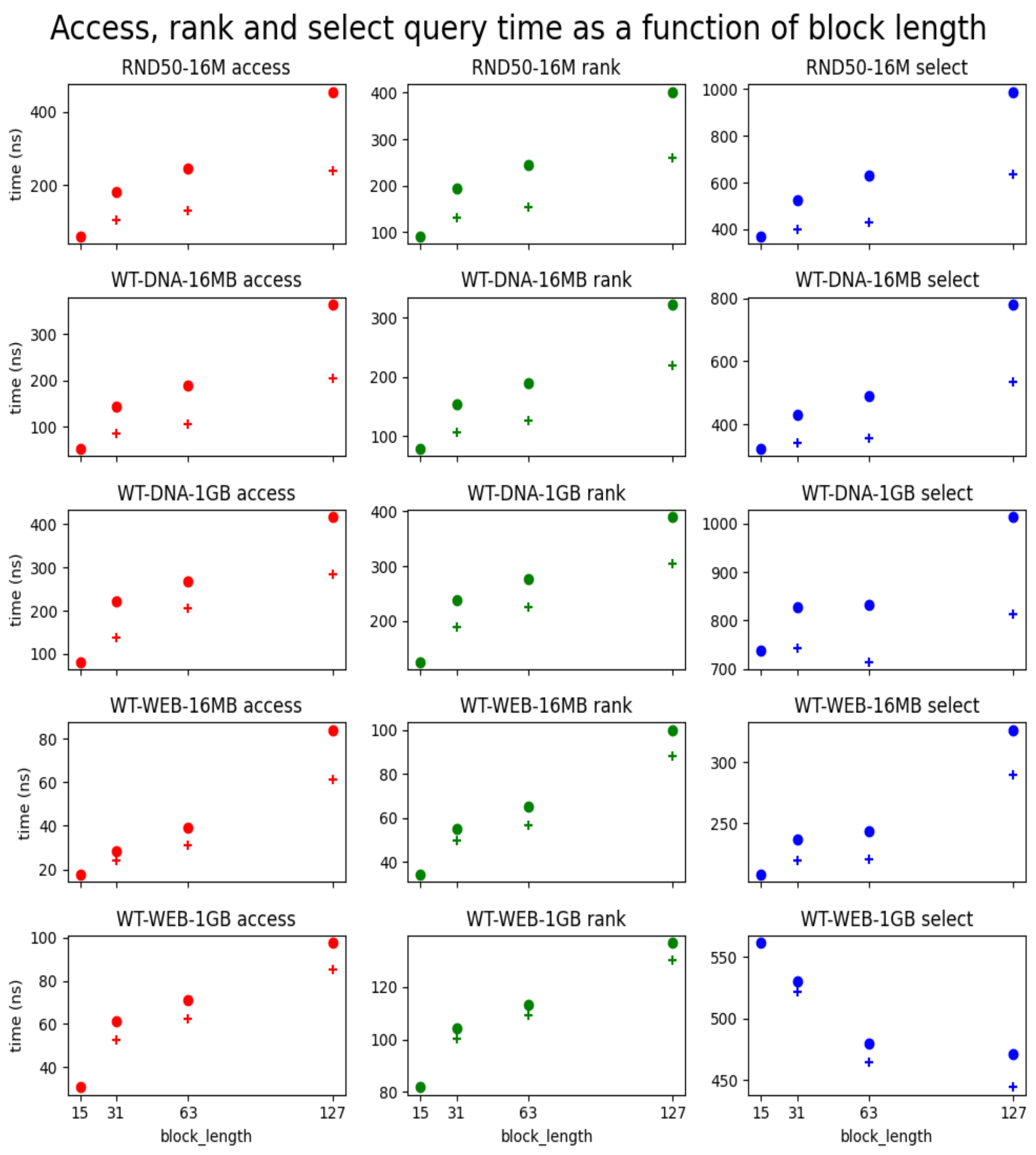

Figure 4.3: `SDSL` benchmark measuring the query time of *access*, *rank* and *select* and its dependence on the block length across various data types. Our implementations are marked by cross, the `SDSL` implementations by dot.

- *locate*(P) – returns all positions of pattern P in text T,
- *extract*(i, j) – returns the subsequence of T starting on i-th and ending on j-th index.

The reason that the *extract* method is useful and non-trivial is that FM-index does not store the original sequence T – at least not in an easily readable form.

We again used benchmarks provided by `SDSL` library and tested how the performance of methods *count*, *locate* and *extract* changes when our bit vector is used. These benchmarks are run on the data from the `Pizza&Chili` dataset and use mainly methodology proposed by Ferragina et al. (2009). The data we show here are the English texts from Gutenberg project, source codes and then DNA and protein sequences. In these benchmarks, we used as a baseline the FM-index version based on block length 15. To provide more context, we also included a version based on the uncompressed bit vector.

To measure performance of *count*, the index over the text is built. Then, random patterns of various lengths are extracted from the text and subsequently used for benchmarking. Code that generates these patterns in `SDSL` is a slight modification of a version provided by `Pizza&Chili` project. We measure the trade-off between space used for the index and the speed normalized by the total number of symbols contained in all searched patterns. We present the results in Fig. 4.4.

Note that the speed gain we have measured in the previous benchmarks also translated into a speedup of FM-index operation *count*. On the English dataset, block lengths 127 and 63 have been able to obtain the speed of the block lengths 63 and 31 provided by `SDSL`. The biggest speed up can be observed for 127-bit block – on English data the time saved is close to 20%, on DNA it is roughly 30% speedup.

On graphs where we can observe the tradeoff between space and time it is important not only to focus on fastest and most succinct implementations but also on those that are *Pareto optimal*. This means that certain implementation may not be the fastest or space optimal but to obtain faster implementation leads to increase in space usage and vice versa.

For benchmarking of *extract*, `SDSL` uses a methodology proposed by Ferragina et al. (2009) in Section 5.4. The queries used for benchmarking consist of numerous substrings of length 512 starting at random positions in text. The additional dimension of FM-index that is tested in this benchmark is sampling rate of suffix array and inverse suffix array. This is a parameter that can be used to trade-off between the speed of FM-index and its memory usage. Options for this sampling ratio are by default the powers of 2 from 4 up to 256. We only show the results for a sampling rate 16 in Fig. 4.5, however, note that the trends that can be observed in these results are also visible for every sampling rate.

We may observe similar speedup as for *count* method. The visible difference is

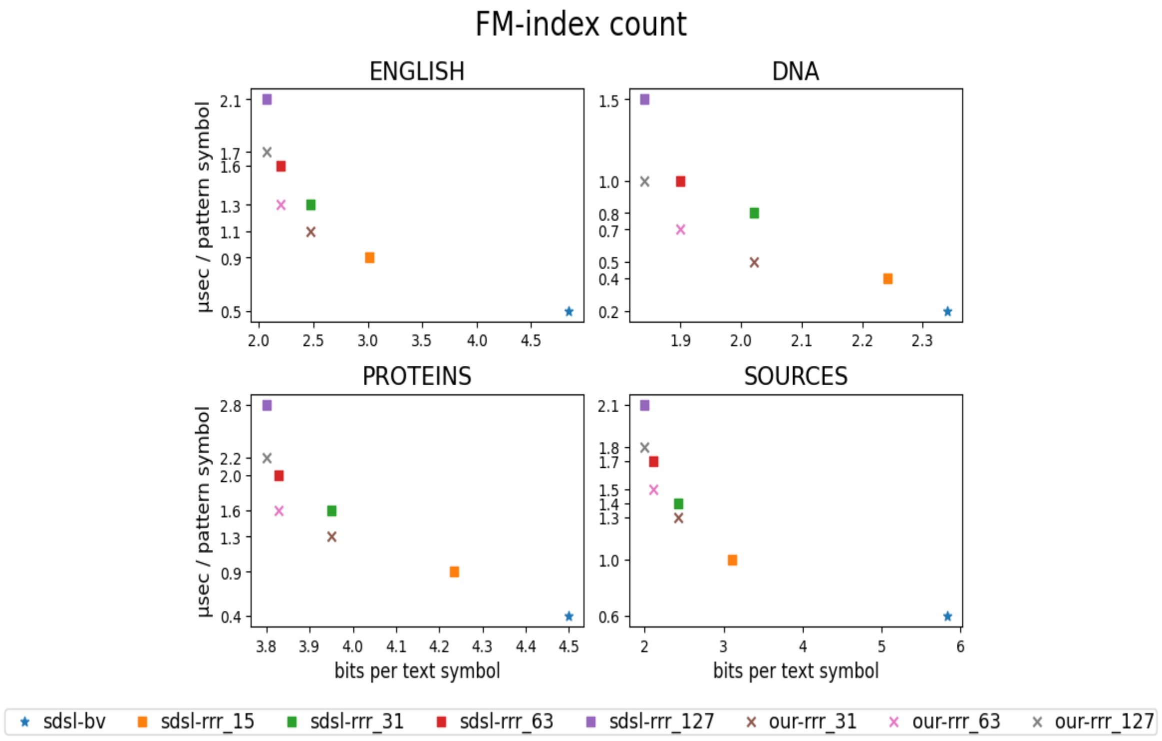

Figure 4.4: Counting occurrences of patterns in various different texts. Displaying the query time per pattern symbol on y-axis and the size of the index over the text on x-axis. Uncompressed bit vector (sdsl-bv) is included for reference.

slightly bigger speedup on Sources for block length 31. On the DNA data, the speedup is hitting again 30%.

Benchmarking of operation *locate* again uses a methodology proposed by Ferragina et al. (2009) in Section 5.3. This consists of locating random patterns of length 5 in the text such that in total 2–3 millions of occurrences are found in the text. We present the results obtained in Fig. 4.6.

The results are very strong for DNA, Protein and Sources datasets but even on English, where our method performed the worst we can observe 10–20% speed up on every block length.

Correctness On top of making sure that our solution is as fast as possible we also wanted to make sure it is correct. Mainly, we used two types of tests for this purpose. The first are the tests of *access*, *rank* and *select* functionality in `SDSL` that run more on smaller bit sequences but cover special cases such as bit vector full of zeroes/ones and certain special patterns that are not encountered often in practice. The second type of tests used was the benchmark running on wavelet tree built over the `WT-DNA` and `WT-WEB` data. Alongside the individual timings, this benchmark produces a checksum of all the results of the *access*, *rank* and *select* queries. These checksums can be compared

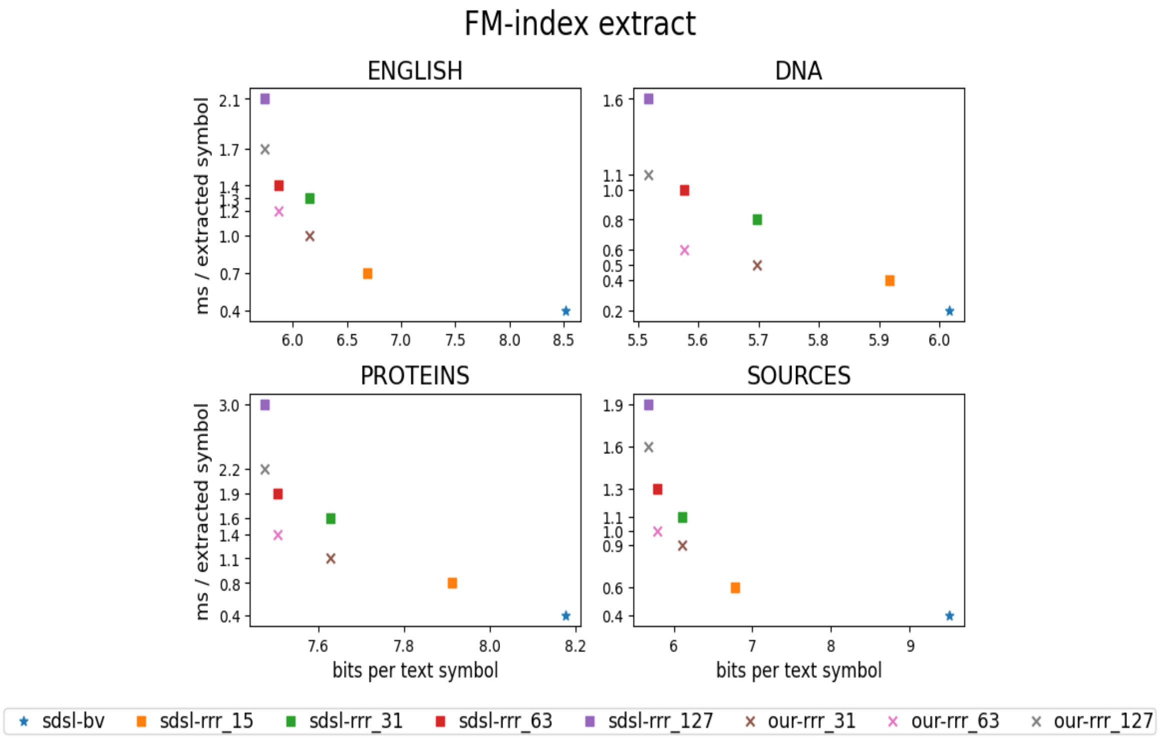

Figure 4.5: Extracting parts of the represented text. Displaying the query time per extracted symbol on y-axis and the size of index over the text on x-axis. Uncompressed bit vector (sdsl-bv) is included for reference.

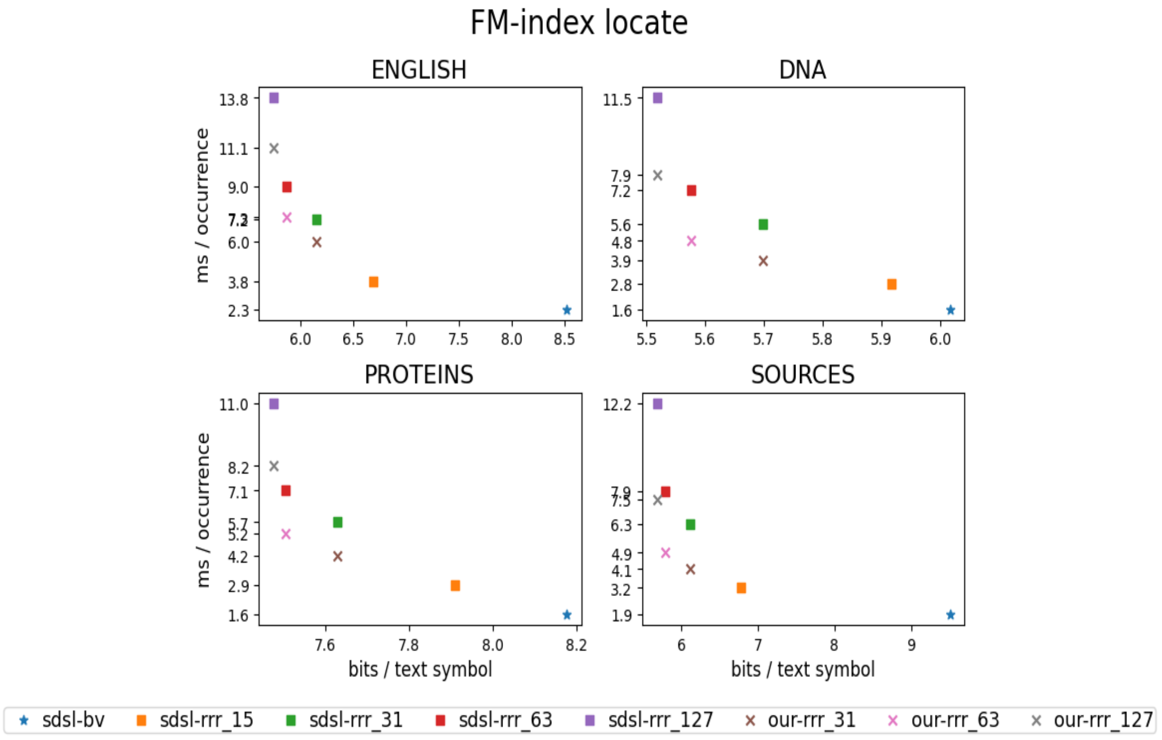

Figure 4.6: Locating occurrences of patterns in the represented text. Displaying the query time per occurrence for various block lengths and the size of index over the text.

between our and original implementation to make sure that our implementation is giving the correct results.

4.2 Hybrid encoding

Let us now present our implementation of the hybrid encoding. In the previous chapter, we proposed two variants of hybrid encoding – one-sided and two-sided version. Both of these were based on the idea not to encode blocks that are rare. This in turn reduced the number of classes and we have been able to save space on bits used for blocks classes. On the other hand, these two versions differed in decision, which blocks are not encoded but just saved in ther original representation. The first solution treated differently just blocks with higher number of ones. The second, was focused on balanced sequences and treated differently blocks that have roughly the same number of zeroes and ones.

4.2.1 Implementation

Implementation of the hybrid encoding required changes to more than only encoding and decoding routines. The first necessary change is addition of hybrid cutoff parameter c_k to the `rrr_vector` class. The second is reimplementation of function `space_for_bt(c)` that is used in `SDSL` to get the number of bits that are needed to store offset of block with class c. The third change, most impactful on a runtime of `rrr_vector`, relates to the way how we work with superblocks.

When computing *rank* of a bit in particular block, we can still binary search for the superblock where the i-th one is located. Then, to linearly search for a result inside of the superblock, we previously needed only information from the array of classes C. This is because C stores the number of ones in the blocks and we can linearly search for the answer using its successive entries. We skip over the blocks inside of the superblock until we find the block where the result is located. This block needs to be decoded and only then we find answer to *rank* inside of a single block. To locate the beginning of this smaller block, we take the memory offset of the superblock in O and add the memory offset of this block from the beginning of superblock. This can be counted along with the linear search for *rank* result. The simplified implementation of the *rank* in `SDSL` is in Listing 4.4.

With hybrid implementation, however, we are not able to linearly scan through the superblock only using information contained in C. This is because now, for some classes, C does not store the number of ones in the block. Thus when we are linearly searching for the result of *rank* query along the superblock, we need to also from time to time access O to count number of ones for some block. This creates additional memory accesses that may slow down the hybrid implementation. The new version of implementation, adapted to the hybrid encoding is in Listing 4.5.

In both versions of hybrid implementation, we introduced cutoff parameter c_k to the class `rrr_vector`. Its meaning is that exactly c_k classes are represented as before and the rest of them is represented using a single value. In the first version, these are classes bigger or equal than c_k. In the second version, these are classes from range `cut_from` up to and including `cut_to`. For certain c_k, we need then $\lg(c_k + 1)$ bits of space to represent a single class.

Listing 4.4: Rank query, SDSL implementation

```
int rank(int i) {
  int block_idx = i/BLOCK_SIZE;
  int superblock_index = block_idx/BLOCKS_PER_SUPERBLOCK;
  int offset = P[superblock_idx]; // precomputed offset into O
  int rank = R[superblock_idx]; // precomputed rank for superblock
  for (int j = superblock_idx*BLOCKS_PER_SUPERBLOCK; j < block_idx; ++j) {
    uint16_t c = C[j];
    rank += c;
    offset += rrr_helper::space_for_class(c);
  }
  uint16_t off = i % BLOCK_SIZE;
  if (!off) {
    return rank;
  }
  uint16_t c = C[block_idx];

  uint16_t block_length = rrr_helper::space_for_class(c);
  uint32_t o = rrr_helper::get_blocks_offset(O, offset, block_length);
  uint16_t popcnt = __popcount(rrr_helper::nr_to_bin(c, o) << (32-off));
  return rank + popcnt;
}
```

Listing 4.5: Rank query, hybrid implementation (one-sided)

```
int rank(int i) {
  int block_idx = i/BLOCK_LENGTH;
  int superblock_index = block_idx/BLOCKS_PER_SUPERBLOCK;
  int offset = P[superblock_idx];
  int rank = R[superblock_idx]; // precomputed rank for superblock
  for (int j = superblock_idx*BLOCKS_PER_SUPERBLOCK; j < block_idx; ++j) {
    uint16_t c = C[j];
    if (c >= c_k) {
      uint32_t o = rrr_helper::get_blocks_offset(O, offset, BLOCK_LENGTH);
```

```
   rank += __popcount(btnr);
  }
  else {
   rank += c;
  }
  offset += rrr_helper::space_for_class(r);
 }
 uint16_t off = i % BLOCK_LENGTH;
 if (!off) {
  return rank;
 }
 uint16_t c = C[block_idx];

 uint16_t block_length = rrr_helper::space_for_class(c);
 uint32_t o = rrr_helper::get_blocks_offset(0, offset, block_length);
 uint16_t popcnt = __popcount(rrr_helper::nr_to_bin(c, o) << (32-off));
 return rank + popcnt;
}
```

Two sided version The biggest notable difference from the previous version of hybrid encoding are the two helper functions that are used for mapping or as we call it compressing and decompressing the blocks class before any of its use. We present these methods in listing 4.6, and 4.7. To choose for particular c_k, which c_k classes we are going to represent in a standard way, we set

$$\text{cut\_from} = \lfloor (c_k + 1)/2 \rfloor \qquad \text{cut\_to} = b - \text{cut\_from} + 1.$$

Listing 4.6: Compressing blocks class

```
int compress_class(int k) {
 if (!is_hybrid_impl) return k;
 if (k < cut_from) {
  return k;
 }
 else if (k <= cut_to) {
  return cutoff;
 }
 else {
  return k-(cut_to-cut_from+1);
 }
}
```

Listing 4.7: Decompressing blocks class

```
int decompress_class(int k) {
  if (!is_hybrid_impl) return k;
  if (k < cut_from) {
    return k;
  }
  else if (k == cutoff) {
    return cut_to;
  }
  else {
    return k+(cut_to-cut_from+1);
  }
}
```

4.2.2 Experimental results

The goal of the first benchmark was to observe, how the one-sided version of hybrid encoding behaves on randomly generated sequences with fixed percentage of ones. The reason behind this test is that bit vector implementations based on RRR do not perform very well on this type of sequences and are often dominated by sparse bit vector implementations.

We generated a random sequence of bits containing 5% of ones and compared hybrid implementation with our original RRR implementation and also sparse bit vector provided by SDSL. We present the results in Fig. 4.7. We may observe that our implementation was already competitive with sparse array for *access* and *rank*. Hybrid encoding only amplified these differences. However, on *select* where our implementation was dominated by sparse array, these differences were not completely overcome by hybrid implementation. We can conclude that hybrid version of 63 and 127-bit block created new usable alternatives, most importantly for *select* method and 127-bit block where our hybrid encoding moved the representation very close to optimal 0.29 bits per one bit in original sequence.

We tested the two-sided version of hybrid encoding on the previously used datasets WT-DNA and WT-WEB. The graph comparing results for classic bit vectors and hybrid bit vectors can be observed in Fig. 4.8. The results of our hybrid implementation are not very promising on these data, except for the *access*. The reason why the results are better for *access* is that hybrid implementation creates additional memory accesses when answering *rank* and *select* as we described in the previous subsection.

To test if our hybrid implementation could be used in the practice, we benchmarked FM-index implementation of method *count*, with hybrid bit vector used inside of the

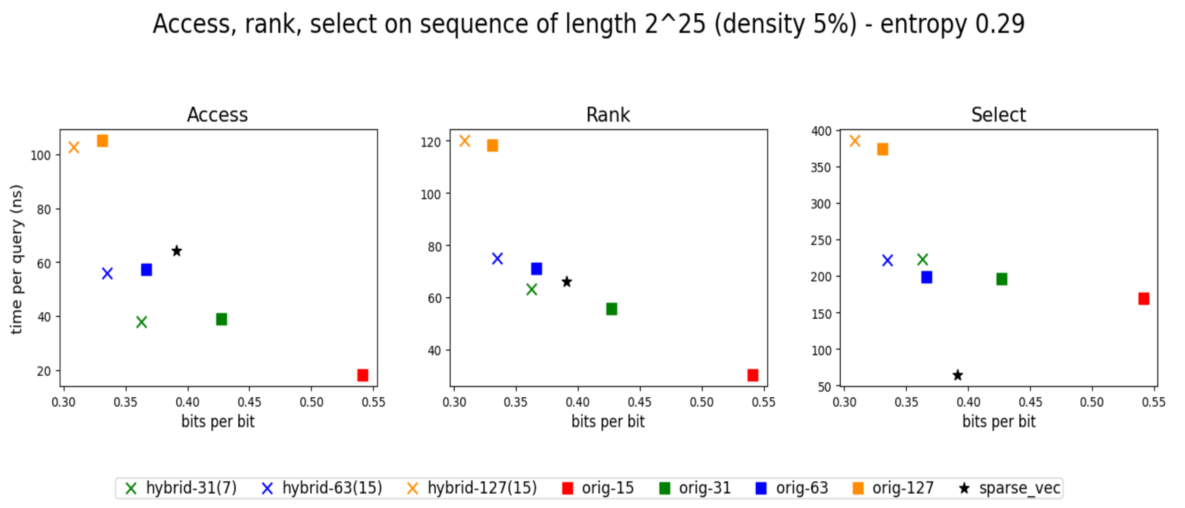

Figure 4.7: Timing of *access*, *rank* and *select* methods over the randomly generated sequence with 5% of ones in it. Comparison of standard implementation and our hybrid encoding marked with cross. For reference, we included sparse vector implementation. We may observe how implementations get closer to the entropy lower bound (0.29) with increasing block length. Name `hybrid-`x`(`y`)` denotes implementation of block length x with cutoff y.

Huffman shaped wavelet tree. The results are shown in Fig 4.9. We may observe that hybrid implementation did not create any viable alternative and our original implementation has a faster or more succinct alternative for every cutoff on the graph. This could be expected as *rank* and *select* are way more used inside of the wavelet tree.

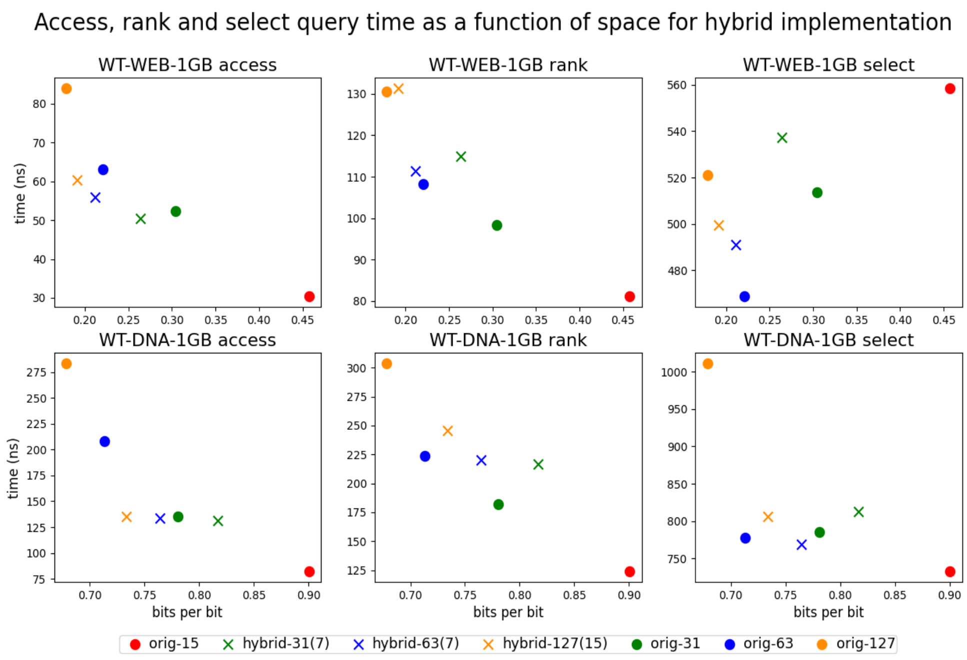

Figure 4.8: This graph is showing the results of benchmark of *access*, *rank* and *select* queries over bit vectors from wavelet tree over DNA sequence data `WT-DNA` and XML bibliographic data `WT-WEB`. On x-axis is the space usage of the particular representation and on y-axis the average query time for the respective operation. Original implementation is denoted by dots. These are our implementations except for the 15-bit version based on the table lookup provided by `SDSL`. Every cross symbols hybrid implementation with particular cutoff. The same block length is denoted by a single color. Name `hybrid-`x`(`y`)` denotes implementation of block length x with cutoff y.

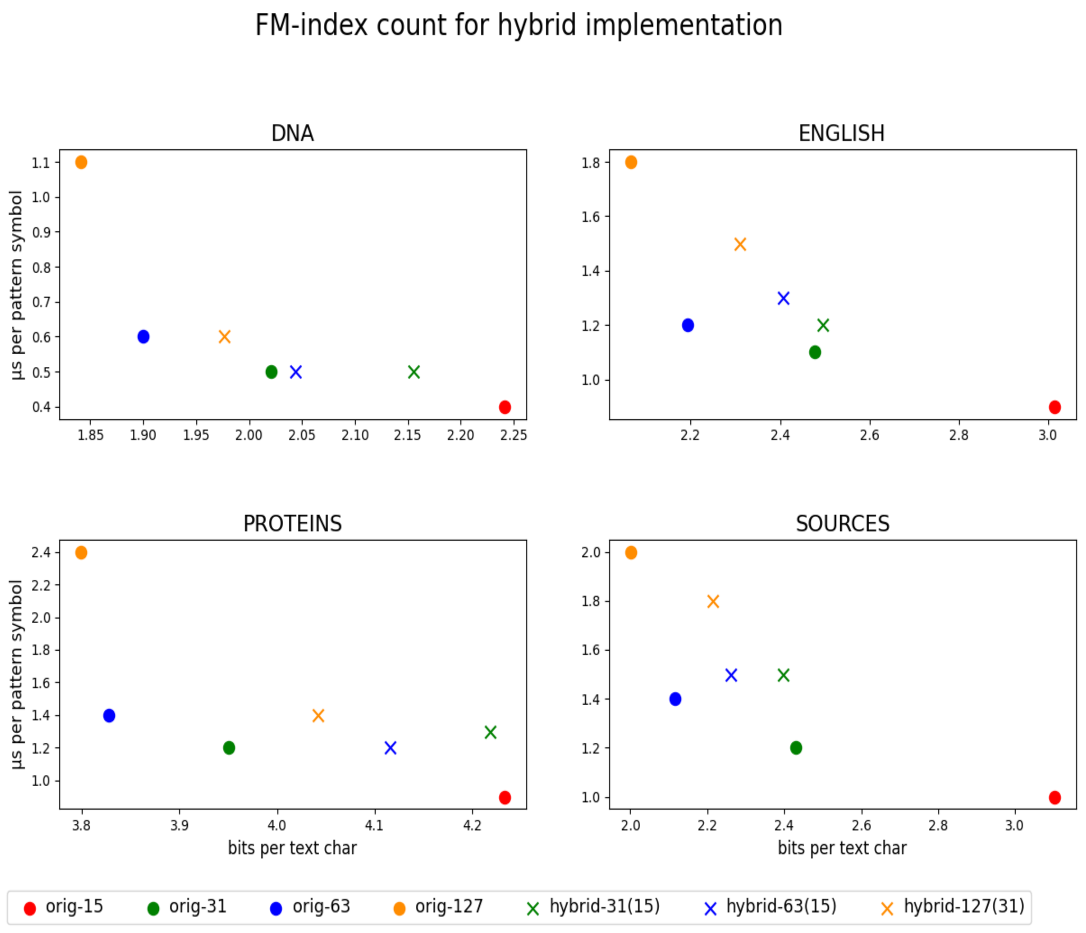

Figure 4.9: Results of benchmark of *count* operation on FM-index.

Conclusion

The goal of this thesis was to study existing implementations of compressed bit vector, explore their shortcomings and develop new implementation of bit vector that mitigates some of the existing problems. We picked up RRR – one particular representation of bit vector and identified places where this representation and algorithms working with it can be improved.

The biggest shortcoming was that there were only two widely used techniques to encode and decode blocks and both have significant disadvantages. The table lookup method provides very fast encoding and decoding but uses too much space for longer blocks. On the other hand, on-the-fly decoding supports longer blocks but trades-off quite a lot of performance for this. We addressed this problem by proposing and implementing new decoding method based on the divide-and-conquer approach that can be used to divide the process of decoding of the block to more sub-problems that can be solved recursively or using one of the existing decoding methods.

We implemented this idea and tested it as a part of `SDSL` library that is regarded as one of the most popular libraries implementing succinct data structures. The new method was very successful in artificial but also practical testing. The most important result presented in this work is the relative speedup of FM-index when `SDSL` bit vector is replaced by our implementation. We measured about 20–30% speedup of *count*, *extract* and *locate* methods thanks to our changes. This was observed on various data such as DNA sequences, source codes or protein sequences.

The second idea was of hybrid encoding. The idea is not to encode some of the possibly rare blocks. By doing this we waste some space on blocks that are not compressed but gain on every other block by decreasing number of bits used for class of the block.

We developed two versions of hybrid encoding. The one-sided version is better suited for sparse sequences with roughly 5% of ones where RRR is often dominated by sparse bit vector implementations. This version indeed worked quite well and when compared to our ordinary version of bit vector it was able to deliver the same speed with roughly 5% lower space usage. The second, two-sided version of hybrid encoding was found not to be practically competitive in its current version due to additional memory accesses that are needed to answer *rank* and *select* queries.

With our new decoding scheme, future work could lead to discovering what tradeoffs between space usage and speed can be achieved with longer blocks, e.g., 255-bit block. Working with longer blocks is way slower because computers natively support 64-bit integers. Our new method, however, enables us to quickly decompose problem into smaller subproblems that could fit into a 64-bit integer.

Future work on hybrid encoding could lead to more experiments on other types of data and finding the ways how to limit the memory accesses that are created because of hybrid encoding. Another practically interesting idea would be to develop a method that automatically adapts the cutoff to the underlying data to get the best possible results.

Bibliography

Burrows, M. and D. Wheeler (1994). A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer.

Chazelle, B. (1988). A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing 17*(3), 427–462.

Clark, D. R. (1998). *Compact PAT Trees.* Ph. D. thesis, CAN.

Farzan, A. and J. I. Munro (2013). Succinct encoding of arbitrary graphs. *Theoretical Computer Science 513*, 38–52.

Ferragina, P., R. González, G. Navarro, and R. Venturini (2009). Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA) 13*, 1–12.

Ferragina, P. and G. Manzini (2000). Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pp. 390–398. IEEE.

Ferragina, P. and G. Manzini (2001). An experimental study of a compressed index. *Information Sciences 135*(1-2), 13–28.

Ferragina, P. and G. Navarro (2005). Pizza&chili corpus—compressed indexes and their testbeds. *September*.

Gog, S., T. Beller, A. Moffat, and M. Petri (2014). From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, pp. 326–337. Springer.

Gog, S. and M. Petri (2014). Optimized succinct data structures for massive data. *Software: Practice and Experience 44*(11), 1287–1314.

González, R., S. Grabowski, V. Mäkinen, and G. Navarro (2005). Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pp. 27–38. CTI Press and Ellinika Grammata Greece.

Grabowski, S., V. Mäkinen, and G. Navarro (2004). First Huffman, then Burrows-Wheeler: A simple alphabet-independent FM-index. In *International Symposium on String Processing and Information Retrieval*, pp. 210–211. Springer.

Grossi, R., A. Gupta, and J. S. Vitter (2003). High-order entropy-compressed text indexes.

Jacobson, G. J. (1988). *Succinct Static Data Structures.* Ph. D. thesis, USA.

Kärkkäinen, J. and S. J. Puglisi (2011). Fixed block compression boosting in FM-indexes. In *International Symposium on String Processing and Information Retrieval*, pp. 174–184. Springer.

Langmead, B., C. Trapnell, M. Pop, and S. L. Salzberg (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology 10*(3), 1–10.

Li, H. and R. Durbin (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics 26*(5), 589–595.

Mäkinen, V. and G. Navarro (2005). Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pp. 45–56. Springer.

Manzini, G. (2001). An analysis of the Burrows-Wheeler transform. *Journal of the ACM (JACM) 48*(3), 407–430.

Navarro, G. (2016). *Compact data structures: A practical approach.* Cambridge University Press.

Navarro, G. and E. Providel (2012). Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pp. 295–306. Springer.

Pagh, R. (2001). Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing 31*(2), 353–363.

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

Seward, J. (1996). bzip2 and libbzip2. *avaliable at http://www.bzip.org*.

Appendix A: Electronic Attachment

All the relevant source code is available in the electronic attachment to this work. The code is divided into two parts. Benchmarking part is contained in the `master-thesis` folder and the fork of `SDSL` library can be found in `sdsl-lite` folder. Both of these can be also accessed at `https://github.com/Aj0SK/master-thesis` and `https://github.com/Aj0SK/sdsl-lite`.

The first repository contains 5 experiments in directory `master-thesis/code`:

- `experiment1` – microbenchmarking of various RRR implementations
- `experiment2` – `SDSL` and our benchmark of the new decoding method
- `experiment3` – our benchmark for hybrid implementation
- `experiment4` – `SDSL` benchmark of our new decoding method using FM-index
- `experiment5` – `SDSL` benchmark of two-sided hybrid encoding using FM-index

In all these folders, there are scripts to run the benchmarks. We used these also to obtain all the presented results in this work.

The second repository is a fork of the `SDSL` library. The individual features are implemented on separate branches:

- `master` – unmodified version of the master branch
- `benchmark_original` – master branch containing minor changes to limit the number of benchmarked blocks and make the tests pseudorandom but reproducible
- `rrr_vector_31_spec` – branch with minimum changes, implementing 31 bit specialization
- `rrr_vector_63_127_spec` – branch with 63 and 127 bit specializations (branched from `rrr_vector_31_spec`)
- `rrr_vector_hybrid` – hybrid implementation for block lengths 31, 63, 127

- `rrr_vector_hybrid_twoside` – special hybrid implementation for sequences with balanced zeroes and ones
- `rrr_benchmark_our` – branch with 31, 63 and 127 bit specializations and benchmarks
- `rrr_benchmark_our_hybrid` – branch with hybrid 31, 63 and 127 bit specializations and benchmarks