Master's thesis

Master's Programme in Computer Science

Postings List Compression with Run-length and Zombit Encodings

Ossi Siipola

December 12, 2022

Faculty of Science

University of Helsinki

Supervisor(s)

Assoc. Prof. Simon J. Puglisi

Examiner(s)

Assoc. Prof. Simon J. Puglisi,
Dr. Leena Salmela

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science
Tekijä — Författare — Author Ossi Siipola	
Työn nimi — Arbetets titel — Title Postings List Compression with Run-length and Zombit Encodings	
Ohjaajat — Handledare — Supervisors Assoc. Prof. Simon J. Puglisi	

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year December 12, 2022	Sivumäärä — Sidoantal — Number of pages 53 pages, 6 appendix pages

Tiivistelmä — Referat — Abstract

Inverted indices is a core index structure for different low-level structures, like search engines and databases. It stores a mapping from terms, numbers etc. to list of location in document, set of documents, database, table etc. and allows efficient full-text searches on indexed structure. Mapping location in the inverted indicies is usually called a *postings list*. In real life applications, scale of the inverted indicies size can grow huge. Therefore efficient representation of it is needed, but at the same time, efficient queries must be supported.

This thesis explores ways to represent postings lists efficiently, while allowing efficient NextGEQ queries on the set. Efficient NextGEQ queries is needed to implement inverted indicies. First we convert postings lists into one bitvector, which concatenates each postings list's characteristic bitvector. Then representing an integer set efficiently converts to representing this bitvector efficiently, which is expected to have long runs of 0s and 1s. Run-length encoding of bitvector have recently led to promising results. Therefore in this thesis we experiment two encoding methods (Top-k Hybrid coder, RLZ) that encode postings lists via run-length encodes of the bitvector. We also investigate another new bitvector compression method (Zombit-vector), which encodes bitvectors by finding redundancies of runs of 0/1s. We compare all encoding to current state-of-the-art *Partitioned Elisa-Fano* (PEF) coding.

Compression results on all encodings were more efficient than the current state-of-the-art PEF encoding. Zombit-vector NextGEQ query results were slighty more efficient than PEF's, which make it more attractive with bitvectors that have long runs of 0s and 1s. More work is needed with Top-k Hybrid coder and RLZ, so that those encodings NextGEQ can be compared to Zombit-vector and PEF.

ACM Computing Classification System (CCS)
Data → Data Structures → Theory of computation
Design and analysis of algorithms → Data structures design and analysis → Data compression

Avainsanat — Nyckelord — Keywords
algorithms, data compression, succint data structures, postings list, inverted index, RLZ, Zombit

Säilytyspaikka — Förvaringsställe — Where deposited
Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information
Algorithms study track

Contents

1 Introduction

Efficiently representing a static set of integers drawn from a finite universe is one of the fundamental tasks in data structures, and remains still an interesting and valid problem. Today, because of massive digitalization of information the sizes of the integer sets in current applications are large. Compression is therefore important, but at the same time, efficient queries must be supported.

One of the interesting applications of static integer set, is *inverted indices*. Inverted indices are the core data structure of many low level and useful applications like search engines, relational database systems or social media graph database index systems. For example, with search engines, inverted indices hold a list of documents for each term indexed by the search engine. Each posting list is sorted static integer set. When a user types a query to a search engine, each postings lists for each term in the query are intersected and the user gets a list of documents that contain all query terms.

To achieve intersection efficiently, lists are processed two lists at a time with pointers to the first list value $v1$ and second list value $v2$. If $v1$ is smaller than $v2$, we know that we can skip all values between $[v1, v2)$ in the first list and therefore find the next value greater or equal (NextGEQ) to $v2$ in first list. After two lists are intersected, the third list is intersected with the intersection of the first two lists and so on. To achieve intercetion efficiently, sets need to be able to compute NextGEQ for a given integer quickly. Some other operations of interest on static sets are access, rank and select.

Often integer sets are represented as a characteristic bitvector where a bit i is set on when index value i is in the set. Then compressing sets converts to compressing bitvectors. Compressing bitvectors is an interesting problem because it is such a widely used structure, and so achieving new performance results on space or speed have an impact on solving many other applications and problems.Querying NextGEQ can be done efficiently on bitvectors by rank and select queries.

The current state–of–the–art in integer set compression is *Partitioned Elias–Fano coding* (PEF) [36]. PEF has a good compression/speed tradeoff and it beats earlier compression techniques like Binary Interpolative coding [30], PForDelta [44], Variant-G8IU [42], a SIMD-optimized Variable Byte coding [42]. PEF parses the bitvector into blocks and encodes each block differently depending its properties.

Arroyuelo and Raman [1] presented new worst case entropy lower bound measures $\mathcal{B}_1(r, n, m)$ and $\mathcal{B}_2(k, r, n, m)$ for bitvector sets where $\mathcal{B}_2(k, r, n, m) \subseteq \mathcal{B}_1(r, n, m) \subseteq \mathcal{B}(n, m)$. They showed that if bitvector has some number of runs r and k number of runs that are bigger than 1, this set can be smaller than the set of bitvectors $\mathcal{B}(n, m)$. For these sets they provided constant time (or close) structures for RANK and SELECT. These new measures are closely related to GAP [4, 3] and RLE [13] measures. If bitvector is represented as runs of 0/1**s**, this could be thought of as a sequence of $2r \pm 1$ symbols, where r is the number of runs. Arroyuelo and Raman showed new zero-order compression measures $\mathcal{H}_0^{gap}$, $\mathcal{H}_0^{run}$ and $\mathcal{H}_0^{hyb}$, where $\mathcal{H}_0^{run}$ is instance of $\mathcal{B}_1(r, n, m)$ and $\mathcal{H}_0^{hyb}$ of $\mathcal{B}_2(k, r, n, m)$, when the sequence of runs is encoded using zero-order entropy of symbols. They also presented data structures that encoded sequences of runs close to these measures with constant time (or close) support to RANK and SELECT.

In this thesis we continue the work of Arroyuelo and Raman [1] by compressing bitvectors with gap and run techniques. We focus on representing a multiset S_M of static integers sets (eg. a posting lists) in small space with tradeoff having efficient NEXTGEQ operation on each set. If all set in the multiset are from universe $\mathcal{U}$ (postings lists formed from $\mathcal{D}$ documents), we deal the multiset as concatenation of each set S characteristic bitvector of length $|\mathcal{U}|$ with $|S|$ ones. The concatenated bitvector can then be thought of as one big set $\mathcal{S}$ of integers from universe $|S_M| \cdot |\mathcal{U}|$ with $\sum_{S \in S_M} |S|$ items in the set.

To compress $\mathcal{S}$ we build its characteristic bitvector as mentioned above and form a new sequence R of run-lengths of the bitvector. With our methods we try a few practical implementations of the encodings of R. We also test compression efficiency by transforming R into two sequences R_0 and R_1 which hold run-length values of 0**s** and 1**s** separately.

We first investigate the compressability of R, R_0 and R_1 by trying practical approaches. In our first method we implement *Top–k Hybrid coder* that splits the k most frequent symbols in R (R_0, R_1) and encodes these with variable-to-fixed Tunstall coding [43]. The rest are encoded with classical Huffman coding [18].

In the recent years a new variant of Lempel–Ziv77 dictionary based compression, named *Relative Lempel–Ziv* [7, 24] (RLZ), has aroused strong interest in bioinformatics and text compression research. Because we convert bitvector to runs of 0**s** and 1**s**, there could be possible repetition on run values and could be possibly separated to different clusters. Therefore we run a set of experiments to test how good the compression effect is with RLZ.

Secondly, in this thesis we take closer look at a new bitvector compression method called

Zombit-vectors [14]. A Zombit-vector does not try to encode runs of 0s and 1s directly but takes advantage of properties of runs if possible. In the original paper Zombit-vector provide only a narrow set of expreriments and comparison to previous implementations and the overall space usage is not obvious. We implement Zombit-vector and test our implementation with a few variations and configurations. Also for NextGEQ queries in this content we try to find an efficient practical solution.

We test our encodings with known sets of postings lists data *Gov2* and compare results with current state–of–the–art solution PEF.

Structure of this thesis is as following. First, in Chapter 2 we introduce core definitions, compression techniques and the current state–of–the–art PEF. Then in Chapter 3, we show in more detail how our methods work with sequences of runs. After methods have been introduced we present results of our experimental testing on Chapter 4. In these expreriments we test our implementations with different configurations and compare our results to PEF. We show results on compression efficiency and with Zombit-vector NextGEQ query efficiency. Finally we summarize our work and discuss future work in Chapter 5.

2 Background and Related Work

This chapter introduces the background information crucial to understand techniques, structures and tools used by methods in Chapter 3.

Data compression tries to store data using as few bits as possible. In data compression there usually are two phases: *encoding* and *decoding*, or compression and decompression. Encoding is the process of transforming the original information to a new compressed format. Sometimes the encoding method is called *code* or *coding*. Decoding is a reverse process of encoding: it transforms compressed data into the original format/coding. Compression is useful when the compressed data fits in cache or main memory and uncompressed data does not.Also data transmission comes quicker if data can be packed more tightly, less bits need to transfer via cable wires or radio signals. It is usually more efficient to compress data and send it through the network and then uncompress it, rather than send the data in original from. Data compression is perhaps still best known for storing files in a more compact form for long-term data storage like SSD or HDD.

An encoding $\mathcal{C}$ is defined as an injective function from source alphabet Σ to encoded form $\mathcal{C}(s)$, $\mathcal{C} : \Sigma \to \Gamma$. Notation $\mathcal{C}(s)$ for all $s \in \Sigma$ is called *codeword* and has length $l(s)$. Sometimes we use with $\mathcal{C}(S)$ to denote encoding of the whole set or sequences, such that it could be concatenation of each symbol encoding $\mathcal{C}(S[i])$ or concatenation of variable sizes parts of S. Usually encoding transforms the source alphabet to a binary representation and therefore encoding is called binary encoding. In computers, information can be only presented as bits, and therefore we focus only on binary encoding in this thesis.

Example 2.1 (encoding) *Let* $\Sigma = \{1, 2, 3\}$*, then encoding* $\mathcal{C}$ *could represent these number as their binary values:* $\mathcal{C}(1) = 1, \mathcal{C}(2) = 10, \mathcal{C}(3) = 11$.

The set of all encodings has one important subclass called *uniquely decodable*. If encoding is uniquely decodable then for all $S_1, S_2 \in \Sigma^n$ it holds that $\mathcal{C}(S_1) \neq \mathcal{C}(S_2)$ and $S_1 \neq S_2$. Encoding in Example 2.1 is not uniquely decodable because $\langle 1, 1, 1\rangle \neq \langle 3\rangle$ but $\mathcal{C}(\langle 1, 1, 1\rangle) = 111 = \mathcal{C}(\langle 3\rangle)$. The easiest way to make any encoding uniquely encodable is to set the same length to all codewords. We could modify $\mathcal{C}(1) = 00, \mathcal{C}(2) = 01, \mathcal{C}(3) = 10$ in example 2.1 to ensure that it is uniquely decodable.

One special encoding subclass guarantees to be always uniquely decodable: *binary prefix* *coding* or just *prefix coding.* Prefix coding ensures that any two symbols in the source are not prefixes of another.

Example 2.2 (prefix coding) *Let Σ be the same as in Example 2.1. Prefix coding for Σ could be example: $\mathcal{C}(1) = 1, \mathcal{C}(2) = 00, \mathcal{C}(3) = 01$.*

Kraft [22] and MacMillan [26] present a theorem stating that if encoding codeword lengths are $l(s_1), \ldots, l(s_\sigma)$ and the inequality

$$\sum_{s \in \Sigma} 2^{-l(s)} \leq 1$$

holds, there exist aprefix encoding $\mathcal{C}$ that can be assigned for symbols in Σ. If inequality becomes equality, $\mathcal{C}$ is said to be optimal. This means that there is no encoding $\mathcal{C}'$ that $|\mathcal{C}'(\Sigma)| < |\mathcal{C}(\Sigma)|$. Otherwise if $\sum_{s \in \Sigma} 2^{-l(s)} < 1$, encoding $\mathcal{C}$ is to be said *redundant.*

Data compression is typically measured by *compression ratio* or *bit rate.* In this thesis bit rate is used to express the level of compression in **bit per posting (bpp)**.

2.1 Entropy

Entropy is a measure of information content. The entropy of a data set is a lower bound on compression in the sense that information can be decompressed without extra assumptions in the decoding process. The concept of entropy was introduced by Shannon [41]. The entropy of a set $\mathcal{U}$ expresses the minimum number bits needed on average to uniquely encode an arbitrary element from the set. There are five types of entropy when dealing with informations systems: *worst case entropy*, *Shannon entropy*, *zeroth–order entropy*, *kth–order entropy* and *empirical entropy.*

With worst case entropy, all elements in the set are encoded with the same code length. To encode each element in set $\mathcal{U}$, there need to be at least $2^{|\mathcal{U}|}$ distinct codes. Then the worst case entropy of set $\mathcal{U}$ is

$$\mathcal{H}_{wc}(\mathcal{U}) = \log |\mathcal{U}| \text{ bits.}$$

So if set $\mathcal{U}$ elements are encoded with the same length, all need exactly $\lceil \mathcal{H}_{wc}(\mathcal{U}) \rceil$ bits. For example, let $\mathcal{U} = \{1, 2, 5, 7, 8\}$, then worst case entropy of $\mathcal{U}$ is $\mathcal{H}_{wc}(\mathcal{U}) = \log 5 \approx 2.322$ bits. Because a bit representation must use whole bits, all elements in set U need 3 bits for each. An example encoding of $\mathcal{U}$ is then $\mathcal{C}(\mathcal{U}) = \{000, 001, 011, 111, 110, 100\}$.

Using fixed-length code lengths for elements does not guarantee optimal encoding. In Shannon entropy [41] elements in $\mathcal{U}$ are given variable length codes, according to their probabilities in the set probability distribution. Shorter lengths are assigned to more probable codes. Then average lengths of the codes and Shannon entropy can be calculated with following equation:

$$\mathcal{H}(Pr) = -\sum_{u \in \mathcal{U}} Pr(u) \cdot \log\left(Pr(u)\right),$$

where Pr is a probability distribution, $Pr : \mathcal{U} \rightarrow [0,1]$. For example, if Pr for $\mathcal{U}$ is $\{0.6, 0.3, 0.05, 0.025, 0.025\}$ then the Shannon entropy of $\mathcal{U}$ is:

$$\begin{aligned}\mathcal{H}(\mathcal{U}) &= -(0.6 \cdot \log(0.6) + 0.3 \cdot \log(0.3) + 0.05 \cdot \log(0.05) + 0.025 \cdot \log(0.025) + 0.025 \cdot \log(0.25)) \\ &\approx 1.445 \text{ bits.}\end{aligned}$$

It can be seen that $\mathcal{H}(\mathcal{U}) < \mathcal{H}_{wc}(\mathcal{U})$ for this particular set. If the following codes are assigned $\mathcal{C}(\mathcal{U}) = \{1, 01, 001, 0001, 0000\}$ to the set, then the average code length is 1.55, which is larger than $\mathcal{H}$ but less than $\mathcal{H}_{wc}$.

Shannon entropy is maximized when all elements in the set have the same probability, in which case $\mathcal{H}(\mathcal{U}) = \mathcal{H}_{wc}(\mathcal{U})$. Shannon also showed in his paper that any optimal prefix code $\mathcal{C}$ from source alphabet Σ with model Pr has the following property:

$$\mathcal{H}(Pr) \leq \sum_{s \in \Sigma} Pr(s)|\mathcal{C}(s)| < \mathcal{H}(Pr) + 1.$$

This is known as the *Noiseless Coding Theorem.*

Shannon entropy can also be applied to sequences of symbols. This is called the *zeroth-order entropy* of the sequence. Entropy of sequence of length n that comes from source $\Sigma = [1, \sigma]$ can be computed as the following:

$$n\mathcal{H}(\langle p_1, \ldots, p_\sigma \rangle) = -n \sum_{1 \leq s \leq \sigma} p_s \log p_s,$$

where p_s is the probability of the symbol $s \in \Sigma$.

Zeroth-order entropy can be increased to higher orders. In some cases it can be useful to use knowledge of previous symbols in the sequence. After reading symbols $s_i, \ldots, s_{i+k-1}$ from the sequence, it might be that some symbols have higher probability to occur after symbols $s_i, \ldots, s_{i+k-1}$ have occured. This is called k^{th}*-order entropy* and it can be computed as follows:

$$\mathcal{H}(Pr) = \sum_{s_1, \ldots, s_{k-1}} Pr(s_1, \ldots, s_k)\mathcal{H}(\langle Pr(1|s_1, \ldots, s_{k-1}), \ldots, Pr(\sigma|s_1, \ldots, s_{k-1})\rangle)$$

Having a sequence of n symbols, it is not clear which source and probability model the sequence have been constructed. Then having encodings for symbols that do not occur in the sequence is unnecessary and increases code lengths of sequence symbols. Then it is good to build the source and the model from the sequence. The entropy is then called k^{th}-*order empirical entropy.* If $k = 0$ then it is the *zero-order empirical entropy.* Zero–order empirical entropy of sequence $\mathcal{S}$ length n over alphabet $\Sigma = [1, \sigma]$ is

$$n\mathcal{H}_0(\mathcal{S}) = n \sum_{1 \leq s \leq \sigma} \frac{n_s}{n} \log \frac{n}{n_s} \text{ bits.}$$

Empirical entropy can limit possible sequences from the original universe. Let $\mathcal{S} = \langle 0, 1, 2, 3, 0, 1, 3, 4, 1 \rangle$. Then $n = 9$ and $n_0 = 2, n_1 = 3, n_2 = 1, n_3 = 2, n_4 = 1$. Therefore zero-order empirical entropy would be:

$$n\mathcal{H}_0(S) = 9 \cdot \left(\frac{2}{9} \log \left(\frac{9}{2} \right) + \frac{3}{9} \log \left(\frac{9}{3} \right) + \frac{1}{9} \log \left(\frac{9}{1} \right) + \frac{2}{9} \log \left(\frac{9}{2} \right) + \frac{1}{9} \log \left(\frac{9}{1} \right) \right) \approx 19.774$$

bits. If $\mathcal{S}$ would been expected to come from source $\Sigma = [0, 5]$ with model $Pr = \{0.20, 0.25, 0.15, 0.10, 0.05, 0.25\}$, the entropy would have been

$$\begin{aligned} n\mathcal{H}(Pr) = -9 \cdot (&0.20 \log (0.20) + 0.25 \log (0.25) + 0.15 \log (0.15) + 0.10 \log (0.10) \\ &+ 0.05 \log (0.05) + 0.25 \log (0.25)) \approx 21.809 \text{ bits.} \end{aligned}$$

The k^{th}*–order empirical entropy* computed as:

$$\mathcal{H}_k(\mathcal{S}) = \sum_{\mathcal{S}_i = s_i \cdots s_{i+k} \in \mathcal{S}} \frac{k}{n} \cdot \mathcal{H}_0(\mathcal{S}_i).$$

It is easy to encode $\mathcal{U}$ in a number of bits that is to close entropy, but it is not trivial always to decode those bits back to the original input unless $\mathcal{U}$ is small and a lookup table for the codes $\mathcal{C}$ could be afforded. In the next section we are going to look at these kinds of structures that place additional structure on the compressed data so that fast decoding and other queries can be supported.

2.2 Succinct data structures

Succinct data structures (SDS) are data structures that try to represent data in space close to the entropy with additional sublinear structures for answering fast to queries.

There is no need to decompress data during querying data.Succinct data structures can be thought as compressed data plus a sublinear index structure. The most used additional structure in SDS, are bitvectors with constant time RANK and SELECT–queries. We will return bitvectors in Section 2.3. An overview of succinct data structures can be seen in Figure 2.1

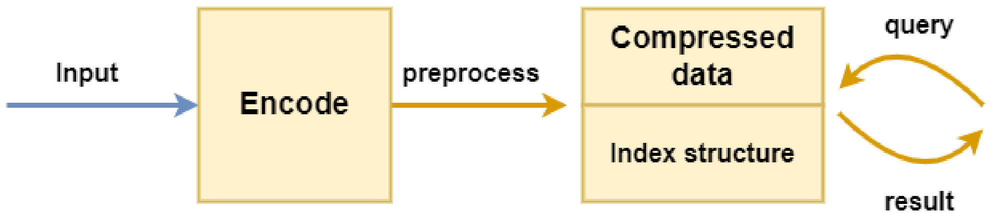


Figure 2.1: High-level illustration of succinct data structures. Succinct data structures consist of compressed data which is encoded using some data compression technique and an index structure for the compressed data. Then when information is needed from the compressed data, full decompression of the data is not needed and index structure can handle and answer the query.

SDS queries typically need more steps to answer, which can make them slower than classical solutions. However if SDS can be stored to lower memory hierarchy or in cache than classical solution, SDS could be much more efficient. The biggest difference in efficiency between SDS and classical solution is when classical solution cannot fit to main memory and needs fetching from secondary memory.

In the next section we introduce a fundamental component of succinct data structures: *bitvectors*.

2.3 Bitvectors

A bitvector is a string on the alphabet $\Sigma = \{0, 1\}$. An example bitvector can be seen in Figure 2.2. Usually we want to support the following operations: ACCESS, RANK and SELECT on bitvectors. ACCESS returns the ith bit value in the sequence.RANK and SELECT–functions are defined as follows.

Definition 2.1 (RANK$_x$) RANK$_x(B, i)$ *returns number of x–bits in first i bits in bitvector B.*

$$\text{RANK}_x(B, i) = |\{j \in [0 \ldots i] : B[j] = x\}|$$

Definition 2.2 ($\textsc{select}_x$) $\textsc{select}_x(B, j)$ *returns index of j^{th} x–bit.*

$$\textsc{select}_x(B, i) = max\{i \in [1 \ldots n] : \textsc{rank}_x(B, i) = j\}$$

Examples of the RANK and SELECT can be seen in Figure 2.3.

0	0	1	0	1	0	0	1	1	1	0	0	1	0	0

Figure 2.2: Example of bitvector.

There are many applications of bitvectors in succinct data structures. Bitvectors are also in an important role when compressing postings lists.

i	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
$B[i]$	0	0	1	0	1	0	0	1	1	1	0	0	1	0	0
$\textsc{rank}_1(B, i)$	0	0	1	1	2	2	2	3	4	5	5	5	6	6	6
$\textsc{rank}_0(B, i)$	1	2	2	3	3	4	5	5	5	5	6	7	7	8	9
$\textsc{select}_1(B, i)$	3	5	8	9	10	13									
$\textsc{select}_0(B, i)$	1	2	4	6	7	11	12	14	15						

Figure 2.3: Example of RANK and SELECT functions for bitvector $B = 001010011100100$. Figure shows function values for both 0s and 1s.

2.3.1 Constant time SELECT–function

To support constant time SELECT, the bitvector needs to have additional structure. Without additional structures the SELECT function needs to always scan the bitvector from the start and count 1 bits. It will be shown that an $o(n)$ space additional structure allows us to support the SELECT function in constant time.

Let B be bitvector of length n that has n_0 0s and n_1 1s. The simplest way to answer SELECT in constant time is to compute and store all SELECT values to an array $S[0, n_1)$. Then $\textsc{select}_1(B, i) = S[i]$. This approach adds $\mathcal{O}(n_1 \log n)$ bits of space. Sometimes this is a good solution for example, when the bitvector is very sparse and so has small number of 1s. More precisely when $n_1 = \mathcal{O}(n/(\log n)^2)$.

To achieve $o(n)$ space and constant query time B needs to be separated to n_1/b_1 variable size blocks $B_0, \ldots, B_{n_1/b_1}$. Each block B_k contains exactly $b_1 = (\log n)^2$ 1-bits. A SELECT query can be then answered as follows:

$$\textsc{select}_1(B, j) = \sum_{i \in [0,k)} |B_i| + \textsc{select}_1(B_k, j - ib_1),$$

where $k = \lfloor j/b_1 \rfloor$. Prefix sums $\sum_{i \in [0,k)} |B_i|$ for all $k \in [0, n_1/b_1)$ can be computed to a lookup table using $\mathcal{O}(n_1/\log n)$ bits.

Blocks B_k local SELECT query depends on size of the block. If $|B_k| \geq (\log n)^4$, this means B_k is sparse and all SELECT queries can be stored to array $S_k[0, b_1)$. Precomputed queries need $\mathcal{O}((\log n)^3)$ bits. Because there are no more than $\mathcal{O}(n/(\log n)^4)$ these kinds of blocks, the total space usage of sparse B_k blocks is $\mathcal{O}(n/\log)$ bits.

Otherwise we need to build another block level inside blocks smaller than $(\log n)^4$. These kinds of blocks are divided into smaller blocks, each containing $b_2 = \sqrt{\log n}$ 1–bits. These smaller blocks B'_k also need to have prefix sums stored, like bigger blocks. These prefix sums fit an array of $\mathcal{O}(n_1 \log\log n/\sqrt{\log n})$ bits. Blocks B'_k need to have their own local SELECT so that block B_k local SELECT is fast. These blocks local SELECT are also divided by the sizes of the blocks. If $|B'_k| < \log n$, local SELECT query values can be stored in lookup tables that need $\mathcal{O}(\sqrt{n}(\log n)^2)$ bits of space. Otherwise queries are stored in array $S'_k[0, b_2)$. Array S'_k needs $\mathcal{O}(b_2 \log\log n)$ bits, which means arrays S'_k uses in total $\mathcal{O}(n \log\log n/\sqrt{\log n})$ bits because there are no more than $n/\log n$ such blocks.

In summary, bitvector B can answer $\textsc{select}_1$ queries in constant time using an additional structure of size $\mathcal{O}((n \log\log)/\sqrt{\log n}) = o(n)$ bits. The same kind of structure can be built to implement the $\textsc{select}_0$ function if needed. Typically applications need only one of these SELECT functions to be answered in constant time. When the other SELECT function is working at constant time, the other can be answered in $\mathcal{O}(\log n)$ time by using the other SELECT function.

2.3.2 Compression

Without compression, a bitvector of length n takes n bits of space. Because so many succinct data structures use bitvectors, improving bitvectors space usage can have a huge impact on improving other structures space usage or efficiency (faster RANK/SELECT). Typically bitvectors can be compressed by dividing the bitvector into block or clusters, so that each block/cluster has some properties that allow it to be compressed. Compressing bitvecors is not interesting if its basic functions RANK and SELECT cannot be implemented efficiently. Next we present the compressibility of bitvectors from [33].

Let B be a bitvector of length n and let m be the number of 1**s** in B. In the worst case, representing any bitvector B needs n bits ($\mathcal{H}_{wc}(B) = n$). Although B has m 1**s**, which leads that class of bitvectors of length n with m 1**s** ($\mathcal{B}(n, m)$) is smaller than 2^n. Then we need $\mathcal{H}_{wc}(\mathcal{B}(n, m)) = \log \binom{n}{m}$ bits to present any bitvector in the set $\mathcal{B}(n, m)$, because there are $\binom{n}{m}$ bitvectors that have m 1**s**. If B if thought to come from a zero order source, it has zero-order empirical entropy of

$$\mathcal{H}_0(B) = \mathcal{H}\left(\frac{m}{n}\right) = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}$$

bits. Figure 2.4 presents the function of zero-order empirical entropy for all m and n. For $m = n$ and $m = 0$ function $\mathcal{H}(m/n)$ is undefined, but it approaches to 0 in analytically[*]. It is notable that $\mathcal{H}_{wc}(\mathcal{B}(n, m)) < n\mathcal{H}_0(B)$, where $B \in \mathcal{B}(n, m)$. This is because

$$\begin{aligned}\mathcal{H}_{wc}(\mathcal{B}(n, m)) &= \log\binom{n}{m} = n\log n - m\log m - (n-m)\log(n-m) - \mathcal{O}(\log n)\\ &= n\left(\frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}\right) - \mathcal{O}(\log n)\\ &= n\mathcal{H}_0(B) - \mathcal{O}(\log n)\end{aligned}$$

As seen in Figure 2.4, the entropy of bitvector is most 1, which happens when B has as many 0**s** and 1**s**. In terms of worst case and zero-order empirical entropy, B is incompressible. This is because the size of set $\mathcal{B}(n, n/2)$ is $\binom{n}{n/2} = \frac{2^n}{\sqrt{n}} \cdot \Theta(1)$, therefore $\mathcal{H}_{wc}(\mathcal{B}(n, n/2)) = n - \Theta(\log n)$ and $n\mathcal{H}_0(B) = n \cdot 1 = n$. These types of bitvectors can be still compressed with other methods, like run-length coding.

Arroyuelo and Raman [1] presented a new lower bound measure for specific bitvector sets $\mathcal{B}_1(r, n, m)$ and $\mathcal{B}_2(k, r, n, m)$, where r is the number of runs of 1**s** and k is the number of runs that are bigger than 1. Class $\mathcal{B}_1(r, n, m)$ contains bitvectors that are of length n with m 1**s** and r number of groups of contiguous 1**s**. Class $\mathcal{B}_2(k, r, n, m)$ is almost same as $\mathcal{B}_1(r, n, m)$ does not include bitvectors having less than k runs bigger than 1. They showed that these new classes are subclasses of set $\mathcal{B}(n, m)$ and it hold that $\mathcal{B}_2(k, r, n, m) \subseteq \mathcal{B}_1(r, n, m) \subseteq \mathcal{B}(n, m)$. In Section 3.1 we try a few practical approaches to achieve these bounds.

[*]$\lim_{x\to 0} \log \frac{1}{x} = 0$

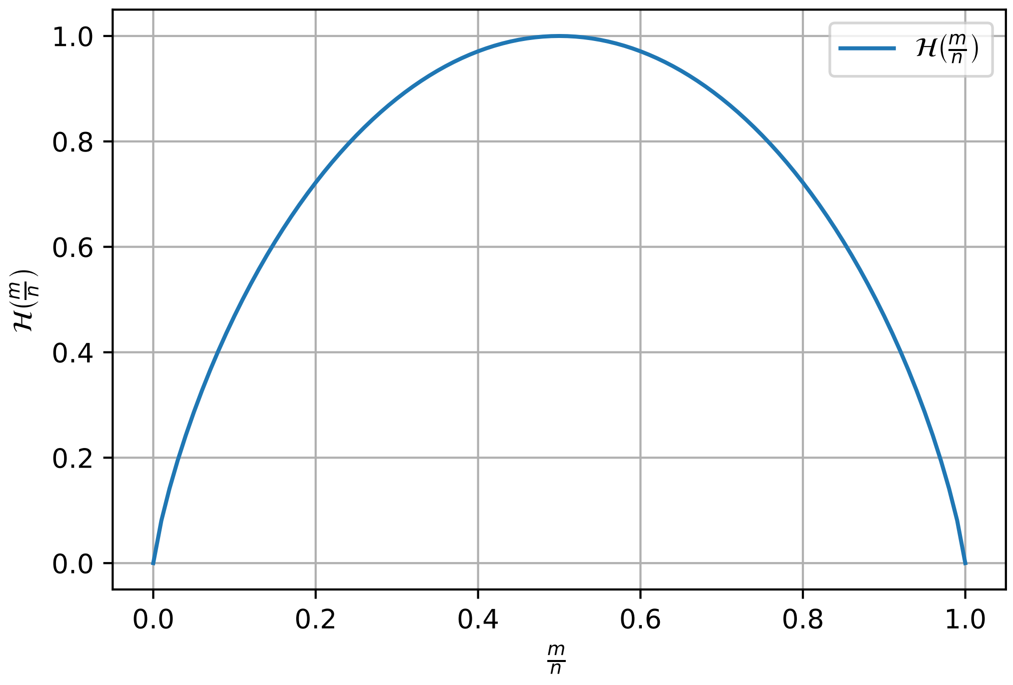


Figure 2.4: Graph shows binary entropy values changes according to probability of 1s (or 0s).

The succinct representation of bitvectors has been an interesting problem in research field of data compression. All current solutions are based on the earlier work from [2, 5, 15, 16, 19, 31, 37, 32, 39]. RRR-vector from Raman et. al [38] was the first representation of bitvectors that achieved $\mathcal{B}(n, m) + o(n)$ space usage to support constant time queries on RANK and SELECT. RRR-vector is still very usefull implementation, because it achieve usually best compression with random bitvectors. In recent years there have been interesting competitive bitvector compression implementations [20, 36, 35].

In Kärkkäinen et. al [20], the Hybrid scheme compresses bitvector with worst case upper bound $n + o(n)$ and support RANK in $\mathcal{O}(1)$ time and SELECT in $\mathcal{O}(\log n)$. In practice the space of the Hybrid bitvector is much less than $n + o(n)$ bits on compressible inputs. It was developed for fast count-queries for patterns in text T with FM–indices. The idea is to split bitvector to $\lceil n/b \rceil$ blocks, size of b. Each block is encoded with different coding method depending its properties: i) if the block has less runs of 1**s**, then it use run–length encoding; ii) if block is sparse (measured by minority bit), it is encoded with position of minority bits; iii) otherwise the block is encoded as a plain bitvector. Worst case space usage is obtained when all blocks have $b/\log b$ minority bits or the same amount of runs. However, in practice this would be very rare.

If $m \ll n$, then the bitvector is sparse and the overhead of $o(n)$ can be large. For these types of bitvectors, Okanohara and Sadakane [35] presented *sd-array* structure. Their structure needs overall $m \log{(n/m)} + \mathcal{O}(m)$ bits, and answers SELECT in constant time and RANK with $\mathcal{O}(\log{(n/m)})$ time. Sd-array is actually an implementation of plain Elias-Fano coding which we discusses more in Section 2.6 and show optimized version of Elias-Fano.

2.4 Huffman coding

One of the most known entropy compression methods is *Huffman coding* [18]. Huffman coding compute minimum-redundancy prefix codes from given alphabet $\Sigma = [1, \sigma]$ and probability model $Pr : \Sigma \to [0, 1]$ or weights $W : \Sigma \to \mathcal{N}$. Because Huffman coding builds optimal prefix coding of the source it is guaranteed to use on average below $\mathcal{H}(Pr) + 1$ bit per symbol. So any sequence $S \in \Sigma^n$ can be encoded using $n(\mathcal{H}_0(S) + 1)$ bits. Huffman coding can be constructed in $\mathcal{O}(\sigma)$ time if the model Pr and Σ are in sorted order, otherwise $\mathcal{O}(\sigma \log \sigma)$ time. Encoding can be done in $\mathcal{O}(1)$ and decoding near $\mathcal{O}(1)$ time per symbol. In this section we introduce simple way to construct Huffman coding and how to decode sequence of Huffman codes. Section A introduces a more efficient way to build and decode Huffman codes. We use this approach in our Huffman coding implementation with Hybrid coder.

The original solution [18] building Huffman coding is to form *Huffman tree* from the source Σ and simulate it to encode/decode process. The tree is formed by a greedy process by setting all symbols $s \in \Sigma$ as individual root nodes with weight value $Pr(s)$. Then pick up two lightest root nodes and form a tree with these nodes by adding parent node for them. The new root node gets a weight of the sum of its childs. Continue this until all nodes are in the same tree. Then all internal nodes have at least two children. Code for each s can be then assigned by traveling the tree from root to leaf labeled by s. Concatenate symbol 0 to the code $\mathcal{C}(s)$ if the left child was selected on the way down and otherwise symbol 1. Example of Huffman tree and constructing classical Huffman codes can be seen in figure 2.5. Algorithm 1 shows a detailed simple way to build Huffman codes from source Σ and model Pr. Example if we have source $\Sigma = [1, 2, 7, 22, 90, 131, 304, 501]$ with weights $\mathcal{W} = \{1, 2, 4, 4, 2, 1, 1, 9\}$ we would have following Huffman codes as in figure 2.5(b).

Encoders need to store model of encoded Huffman codes, or if the source is small actual codes can be stored directly. Huffman encoded sequence can be simply decoded by traveling Huffman tree bit by bit and echoing the symbol when leaf is reached and re-

Algorithm 1: BUILD-HUFFMAN-CODES builds Huffman codes $\mathcal{C}$ from the source Σ and the model Pr

```
Input: Alphabet $\Sigma = [1, \sigma]$, probability model $Pr$ over $\Sigma$
Output: Huffman codes $\mathcal{C}$
1 PQ = min–heap
2 for $s \in \Sigma$ do PQ.insert({ s }, Pr(s))
3 while |PQ| > 1 do
4     $A \leftarrow$ PQ.pop()
5     $B \leftarrow$ PQ.pop()
6     for $a \in A$ do $\mathcal{C}(a) \leftarrow 1\mathcal{C}(a)$
7     for $b \in B$ do $\mathcal{C}(b) \leftarrow 0\mathcal{C}(b)$
8     PQ.insert( $\{A \cup B\}, \sum_{a \in A}$ Pr($a$) + $\sum_{b \in B}$ Pr($b$))
9 return $\mathcal{C}$
```

turning back to the root node. Then the decoding symbol takes $\mathcal{O}(l)$ times, where l is length of the encoded symbol. So total size of Huffman coded sequence of length n is $n(\mathcal{H}(Pr) + 1) + \mathit{size}(Pr)$ bits.

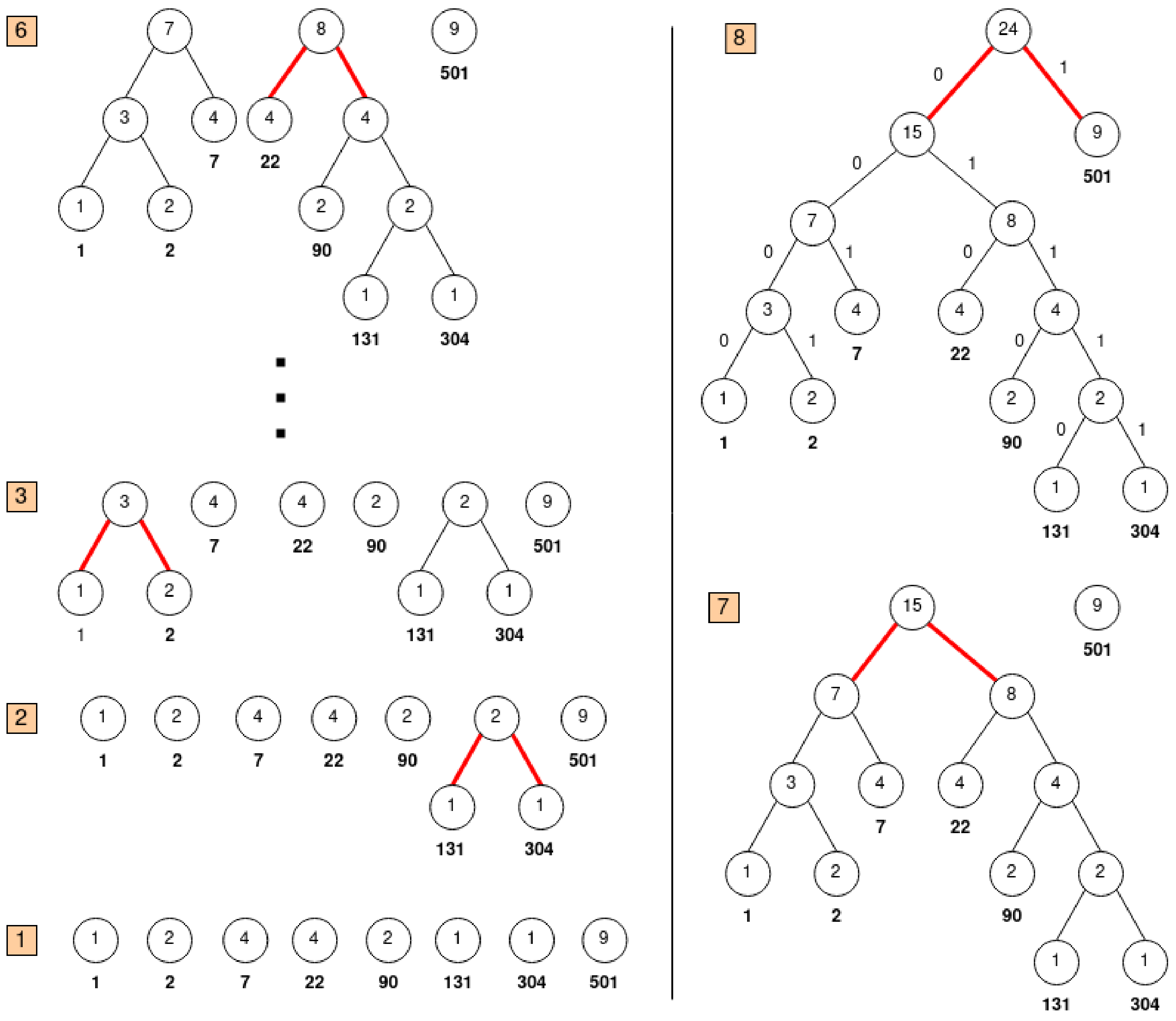
If for all $s_i, s_j \in \Sigma$ holds that $s_i < s_j$, $|\mathcal{C}(s_i)| = |\mathcal{C}(s_j)|$ and $\mathcal{C}(s_i) < \mathcal{C}(s_j)$ Huffman coding is called *canonical*. Canonical Huffman coding can be formed by computing first lengths of codes and then assigning codes for each symbol. Canonical Huffman coding is useful to construct because it gives more information about the code for the decoder.

In appendix A we show a more efficient way to build Huffman coding and how to decode sequence more quickly.

2.5 Tunstall coding

Tunstall coding is optimal variable-to-fixed length coding. It was first introduced by Tunstall [43]. Like in Huffman coding we tried to use as less bits for more frequent symbols, in Tunstall coding we try to assign fixed kth length codeword to as many symbols as we can. Tunstall coding can be thought of as dictionary based compression

When given a sequence $\mathcal{S}$ of symbols, Tunstall coding builds a prefix free dictionary $\mathcal{D}$ from the sequence or given model Pr. So for all $d_i, d_j \in \mathcal{D}$ is holding that d_i is not a prefix of d_j. Then sequence $\mathcal{S}$ is parsed into variable length parts $d_1, d_2, \ldots, d_m$, where



(a) Process of building Huffman codes.

Σ	1	2	7	22	90	131	304	501
$\mathcal{C}(s)$	0000	0001	001	010	0110	01110	01111	0

(b) Final Huffman codes.

Figure 2.5: Example of classical procedure constructing Huffman codes of source $\Sigma = \{1, 2, 7, 22, 90, 131, 304, 501\}$ with weights $W = \{1, 2, 4, 4, 2, 1, 1, 9\}$. Figure 2.5(a) shows one example for constructing Huffman codes, after phase 8 Huffman tree is finished. Figure 2.5(b) represents final Huffman codes for Σ and W.

each $d_i \in \mathcal{D}$. Then each part d_i is encoded with kth length encoding. Tunstall coding is to be said optimal because it can minimize

$$m + \sum_{d_i \in \mathcal{D}} |d_i|$$

according to the model Pr and variable k. Model Pr can be constructed from the input sequence.

Size of the dictionary $\mathcal{D}$ depends on variable k that is assigned before building $\mathcal{D}$. Because all $d_i \in \mathcal{D}$ have same length k, $2^k \geq |\mathcal{D}|$ must hold. If it would not, then some items in $\mathcal{D}$ would have same code. The dictionary can be seen as a tree structure, which is typically called *Tunstall tree*. Each path from root to leaf concatenating labels from an item in $\mathcal{D}$. Because all items d_i in $\mathcal{D}$ have same length codeword, it do not matter which codeword is assigned to each item d_i. Example codewords could be assigned in sorted order from 0 to 2^k. Algorithm 2 describe how k-Tunstall codes are builded from the source or model. Figure 2.6(a) illustrates process of algorithm 2 when building 3-Tunstall codes from model $Pr = \{2 : 0.6, 5 : 0.3, 7 : 0.1\}$. Example Tunstall coding for sequence $\mathcal{S} = \langle 2, 2, 5, 2, 7, 5, 5, 2, 5 \rangle$ using $\mathcal{D}$ from figure 2.6(b) would be 001100101101011

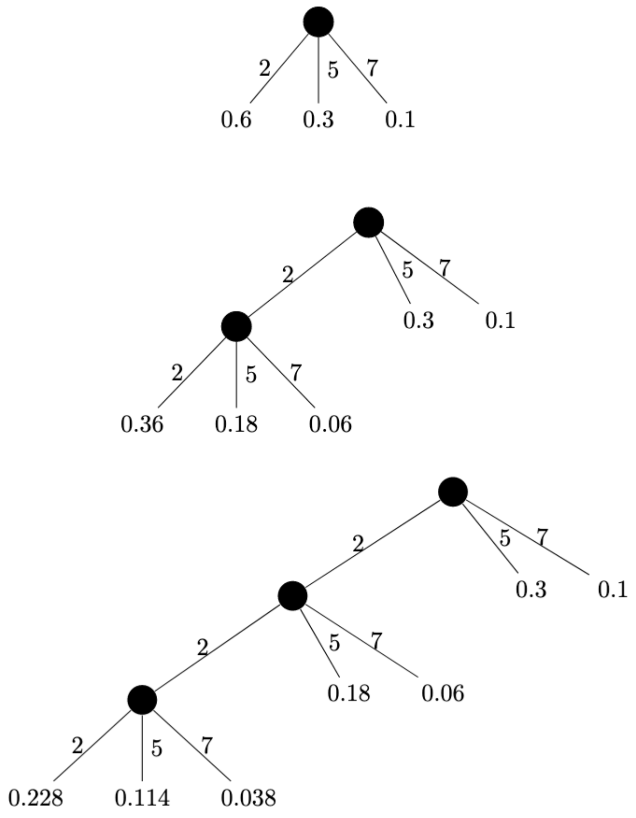
To decode Tunstall coding, the decoder needs the dictionary $\mathcal{D}$ or the model Pr to build $\mathcal{D}$. Then decoder can read k bits from the encoded sequence and convert k bits to j symbols of the original sequence $\mathcal{S}$. Example if we use same $\mathcal{D}$ as in figure 2.6(b), binary sequence 000010001011000 would decode as sequence $\mathcal{S} = 22222722525222$.

Algorithm 2: Builds Tunstall codings of length k from source Σ, model Pr.

```
Input: Alphabet $\Sigma = [1, \sigma]$, model $Pr$ and codeword length $k$
Output: Tunstall dictionary $\mathcal{D}$
1 Create root node and assign all symbols $s \in \Sigma$ its leaf node and give them value
    Pr($s$).
2 while $2^k - \sigma > |\mathcal{D}|$ do
3     Find leaf $l$ with highest probability value.
4     For all $s_i \in \Sigma$ add child with value value($l$) · Pr($s_i$) to node $l$, edges labeled with
        $s_i$
5 return $\mathcal{D}$
```

Overall Tunstall coded sequence takes km bits space, where m is number of dictionary items used. Average bitrate for sequence $\mathcal{S}$ of length n is

$$\frac{1}{n} \sum_{d_i \in \mathcal{D}}^{\sigma} Pr(d_i)|d_i|$$



(a) Tunstall tree

$\mathcal{D}$	
d_i	$\mathcal{C}(d_i)$
222	000
225	001
227	010
25	011
27	100
5	101
7	110

(b) Tunstall codes

Figure 2.6: Example of Tunstall coding when $Pr = \{2 : 0.6, 5 : 0.3, 7 : 0.1\}$. Figure 2.6(a) illustrates process of building the codes and shows the final Tunstall tree. Figure 2.6(b) shows the final Tunstall codes and dictionary $\mathcal{D}$.

bits. So the total size of Tunstall coding is $km + (size(\mathcal{D})$ or $size(Pr))$ bits.

2.6 Elias-Fano coding

Elias–Fano coding (EF) [9, 10] is compresses an increasing positive integer sequence $\mathcal{S} = \langle x_1, x_2, \ldots, x_n \rangle$ from universe $\mathcal{U} = [0, u]$. Usually $x_n = u$. EF coding gives efficient ACCESS and NEXTGEQ operations for the sequence. It can compress the sequence $\mathcal{S}$ down to $n\lceil \log u/n \rceil + 2n$ bits.

Let us assume that elements of $\mathcal{S}$ are represented as $\lceil \log u \rceil$ bits each. EF Coding separates the elements bits representation into two bitvectors L (lower) and U (upper). Vector L holds l "lower" bits of each $x_i \in S$ concatenated together. Integer l can be selected from range $[0, \lceil \log u \rceil]$, but the best space usage is achieved when $l = \lfloor \log u/n \rfloor$. The remaining $\lceil \log u \rceil - l$ bits from $\mathcal{S}$ items are set first as a temporary sequence T. Then T is converted as *delta–encoding* ($T[i+1] = T[i+1] - T[i]$ for all $i \in [1, n)$). Bitvector U is then constructed from T by concatenating the unary encodings of its elements. An example of Elias–Fano coding for sequence $\langle 3, 4, 8, 13, 24 \rangle$ can be seen in Figure 2.7.

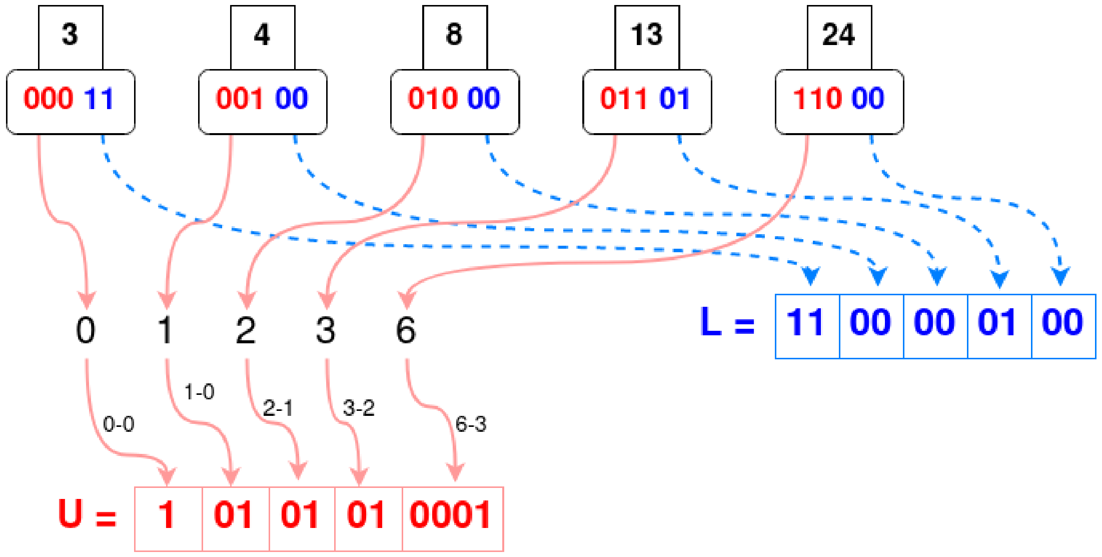


Figure 2.7: Example how to construct Elias–Fano encoding for sequence $\langle 3, 4, 8, 13, 24 \rangle$. Size of the lower bits are 2, which is optimal for total size of Elias–Fano encoding for the sequence ($\lfloor \log 24/5 \rfloor = 2$). Bits labelled red represents upper bits, and blue bits for lower bits. The final Elias–Fano encoding for the sequence is $L = 1100000100$ and $U = 10101010001$.

To support operations ACCESS and NEXTGEQ operations bitvector U needs to have an index structure to support SELECT$_1$ and SELECT$_0$ functions on U. To access $S[i]$ on S we have to locate and concatenate bits in L and U. Bits in L are easy to locate because each element has the same width and bits needed for $\mathcal{S}[i]$ are in range $[li \dots l(i+1))$ on L. Because U do not present upper bits explicitly, those can be computed with SELECT$_1(i, U) - i$. Then final answer can be computed as ACCESS$(i) = ((\text{SELECT}_1(i, U) - i) \ll l) | L[li, l(i+1))$.

We can notice that there are $p = \text{SELECT}_0(h_x, U) - h_x$ number of elements in $\mathcal{S}$ that have smaller upper bits value than x and $p' = \text{SELECT}_0(h_x + 1, U) - (h_x + 1)$ elements that have smaller or equal upper bits, where h_x is upper bits of x. Then to find NEXTGEQ(x), p elements can be skipped and L can be scanned starting from $p + 1$ to p' finding next greater or equal than l_x, where l_x are x lower bits. If $L[i]$ is first equal or greater value than l_x, then we can return $(h_x \ll l)|L[li, l(i+1))$. In the worst case we need to scan 2^l items. If $p = p'$, there is not elements in $\mathcal{S}$ that has same upper bits than x. Then we know that the result is in index $p+1$ in L. Because this element upper bits are unkown, we need to fetch those same way as in ACCESS operation, and the result is $((\text{SELECT}_1(p+1, U) - (p+1)) \ll l) | L[l(p+1), l(p+2))$.

EF coding achieves good compression when the universe $\mathcal{U}$ is small. On average each element needs $\lceil \log \mathcal{U}/n \rceil + 2$ bits. So more sparse sequence have more redundant bits. Amount $\lceil \log \mathcal{U}/n \rceil$ can be the average distance of sequence elements. Next we are going to look for a way to compress sequences knowing this information about EF coding.

2.6.1 Partitioned Elias-Fano coding

As described above, the size of Elias–Fano coding depends on the sequence length n and universe u. Ottaviano and Venturini [36] presented a two-level Elias–Fano coding representation of a positive increasing sequence $\mathcal{S}$ that tries to reduce the universe in Elias–Fano coding.The scheme they presented is called *Partitioned Elias–Fano coding.* As the name suggest, sequence is to be separated into variable sizes blocks and each block is encoded separately.

For simplicity let us first assume $\mathcal{S}$ is partitioned with same size blocks length of b, such that $\mathcal{S} = S_1, S_2, \ldots, S_{\lceil n/b \rceil}$ and all blocks are encoded with Elias–Fano coding (PEF-uniform). To reduce the universe of each block and to help implement operations ACCESS, NEXTGEQ, array L' is constructed from last elements of each block, so $L' = [S_1[b], S_2[b], \ldots, S_{\lceil n/b \rceil}[b]]$. Array L' is encoded with Elias–Fano coding. With array L' each block S_j can be encoded to universe $u' = L'[j] - L'[j-1] - 1$ and each item in S_j need to be reduced by $L'[j-1]+1$. Then S_j needs $\log(u'/b) + 2$ bits per element.

Operation ACCESS for $\mathcal{S}$ can be computed fast from the partitioned representation. To get the ith item in $\mathcal{S}$, we need to get block boundaries from the block S_j where item i lies and its previous block S_{j-1}, such that we can compute the universe size u' of block S_j. Let $j = \lfloor i/b \rfloor$ and $k = i \bmod b$, so item i is kth item in S_j and block S_j is encoded from universe $L'[j] - L'[j-1] - 1$. Then $\text{ACCESS}(i) = L'[j-1] + 1 + S_j[k]$, where $S_j[k]$ can be computed as a normal ACCESS operation on Elias–Fano coding.

To compute operation $\text{NEXTGEQ}(x)$ we need to first find the successor of x in L'. This can be done by a local NEXTGEQ query on L'. Let $L'[j]$ be largest value in L' that is equal or greater than x, then the successor of x is in block S_j and the next or equal item on S of x can be found from this block j by computing $\text{NEXTGEQ}(x - L'[j-1] - 1)$ from block S_j. As it can be seen block size b do not affect the performance of ACCESS and

NextGEQ operations.

If block is dense it can be better to encode without Elias-Fano. When the block size b approaches the universe size u', it can be seen that the encoding size of the block approaches then to $2u'$ bits. Then it is more efficient to represent the block as a plain bitvector using only u' bits. So when $b = u'$ block S_j can be encoded using 1s and the result of operations access(i) and NextGEQ(i) in the block is simply i. If $b \neq u'$ but $b > u'/4$ it is also effective to encode block as a plain bitvector. Operations access and NextGEQ are not so trivial as when $b = u'$ but can be answered efficiently by creating structures for rank and select operations and using these for the query.

Ottaviano and Venturini also presented an optimized version (PEF ϵ-optimal) of the fixed block size version. In the optimized version, block sizes are variable length size. They presented a linear time dynamic programming algorithm to find the partition of blocks withing cost $(1+\epsilon)$ factor to size of optimal partition, where $\epsilon \in (0, 1)$. The idea in dynamic programming algorighm is to build a weighted graph of all the possible partitions. Edges in the graph represent cost of next partition. Then optimal partion can be found by finding the lightest path from the graph. Because finding the optimal path would take $\mathcal{O}(n^2)$ time, an approximation algorithm is needed.

When the block are variable length, operation access needs additional information about location of the block containing item i, NextGEQ work as before because it is not needed to know the block location of search item explicitly. Ottaviano and Venturini's idea was to create array E to store position of each block endpoints. Then access operation can locate item i block with NextGEQ on E. Array E is encoded as plain Elias-Fano, so NextGEQ is easy and efficient to find.

In experimental testing in [36] PEF ϵ-optimal is about 43,9% smaller than plain Elias-Fano coding and 11,3% smaller than PEF-uniform. Quering PEF ϵ-optimal is still slightly slower than plain Elias-Fano and PEF-uniform.

3 Methods

The goal of the thesis is to find a way to efficiently represent a static interger set, such that the set can handle NextGEQ operations efficiently. Our goal is then to build a succinct data structure for a static integer set. To achieve this we focus on converting the set to the characteristic bitvector and then finding a way to compress it.

This chapter consists of two parts. First in Section 3.1 we continue Arroyuelo and Raman's work on compressing bitvector by its run-length of 0/1s [1]. Compressibility of runs is also considered when runs of 0s and 1s are separeted into different sequences. We experiment two practical methods to compress this sequence of runs. Then we focus overall on compressing a sequence of symbols from a large source with these methods. Compressing a sequence of symbols is not new task with textual data, and $\mathcal{H}_k$ would be natural lower bound for compressing a sequence of runs and the total space usage would be $n\mathcal{H}_k + \mathcal{O}(n + \sigma^{k+1} \log k)$ bits. However with textual data the alphabet is not usually large, for example using an ASCII alphabet we have only 255 symbols, but with integers the alphabet size can be much larger and the second term above is not negligible. Then $\mathcal{H}_0$ would be an interesting compressibility measure of integers sequences with a large alphabet. Our first method is an adaptive encoder, *Top-k Hybrid coder* which uses Tunstall and Huffman coding to encode the sequence(s). The second practical method we experiment with is *Relative Lempel-Ziv.*

In the second part of the chapter, Section 3.2 takes a closer look at the new bitvector compression method *Zombit-vector* [14]. For compressing a bitvector with k runs of 1s, a Zombit-vector needs $\mathcal{O}(\sqrt{rn}) + o(n)$ bits and can answer NextGEQ queries in $\mathcal{O}(1)$ time. Zombit-vectors do not try to compress runs of 0/1s explicitly but reduce space by finding runs implicitly by splitting the bitvector into blocks. If a block is full of 0s or 1s, the block can be encoded using only 1 bit. Our method on Zombit-vector is to implement it and take closer look of its compressibility and efficiency for NextGEQ queries on bitvectors with runs.

3.1 Approach with run–length encoding

As discussed in Section 2.3.2 Arroyuelo and Raman [1] showed a new lower bound for spesific bitvector sets and how bitvector B can be compressed by converting B into a sequence $R = \langle x_{1_0} x_{1_1} \dots x_{r_0} x_{r_1} \rangle$ of run-lengths in B. We also consider splitting runs of 0**s** and 1**s** into separate sequences $R_0 = \langle x_{1_0} x_{2_0} \dots x_{r_0} \rangle$ and $R_1 \langle x_{1_1} x_{2_1} \dots x_{r_1} \rangle$. Because the source of the run sequence is expected to be large, we are interested in compressing the sequence with zero-order entropy. To illustrates this, let $B = 1111110011000111100101$ which yields that $R = \langle 6, 2, 2, 3, 3, 2, 1, 1, 1 \rangle$, $R_0 = \langle 2, 3, 2, 1 \rangle$ and $R_1 = \langle 6, 2, 3, 1, 1 \rangle$. Next we are going to introduce our two methods to encode these sequence(s) of run-lengths.

3.1.1 Top–*k* Hybrid coder

Top–k Hybrid coder is an adaptive succinct data structure that encode sequence of positive integers using three different approaches: Tunstall coding $\mathcal{T}$, Huffman coding $\mathcal{HC}$ and bitvector B'. As the name suggests, the top k frequent symbols from the alphabet Σ are encoded with Tunstall coding and the rest with Huffman coding. Bitvector B' informs us of the encoding of symbol $R[i]$, such that $B[i] = 0$ iff symbol $R[i]$ is encoded with Tunstall coding and otherwise Huffman coding. Decoding is not possible without B'. A plain representation of bitvector B' have same length as sequence of runs R.

For example, let $R = \langle 1, 2, 2, 3, 2, 4, 2, 1, 2, 4, 2, 4, 7, 4 \rangle$ be runs of 0s and 1s (or only 0s or 1s). If $k = 2$, the k most frequent symbols in R would be 2 and 4. Then R is split into sequences $\mathcal{T} = \langle 2, 2, 2, 4, 2, 2, 4, 2, 4, 4 \rangle$, $\mathcal{HC} = \langle 1, 3, 1, 7 \rangle$ and bitvector $B' = 1001000100001$. When $\mathcal{T}$ and $\mathcal{HC}$ are encoded with their own $\mathcal{H}_0$ encodings. Codes for each symbol in $\mathcal{T}$ and $\mathcal{HC}$ can be seen in Figure 3.1. Using these codes, the sequence R is encoded as:

$$\begin{aligned} \mathcal{C}(\mathcal{T}) &= 010011100 \\ \mathcal{C}(\mathcal{HC}) &= 010011 \\ B' &= 1001000100001. \end{aligned}$$

Algorithm 3 gives us a detailed version of constructing Top-k Hybrid coder.

The idea with Top-k Hybrid coder is that run-length sequence is formed from source that there would be some small value k, such that the k most frequent symbols in the sequence would cover almost all sequence symbols. Then in our case, we encode these k most frequent symbols with Tunstall coding. If k is small, Tunstall coding could get good

$\mathcal{D}$	
d_i	$\mathcal{C}(d_i)$
2222	000
2224	001
224	010
242	011
244	100
422	101
424	110
44	111

(a) Tunstall codes for $\mathcal{T}$

Σ	$\mathcal{C}(s)$
1	0
3	10
7	11

(b) Huffman codes for $\mathcal{HC}$

Figure 3.1: Tunstall and Huffman encodings for Tunstall sequence $\mathcal{T} = \langle 2,2,2,4,2,2,4,2,4,4 \rangle$ and Huffman sequence $\mathcal{HC} = \langle 1,3,1,7 \rangle$.

Algorithm 3: BUILD–HYBRID–CODER, builds Top-k Hybrid coder from positive integer sequence R and positive integer k.

```
Input: integer sequence $R$, integer $k$
Output: Top–$k$ Hybrid coder HYB($\mathcal{T}, \mathcal{HC}, B$)
Function build-hybrid-coder($R, k$):
    freq_map ← compute frequencies of different symbols in $R$
    top_k_alphabet ← top $k$ most frequent symbols in freq_map
    tunstall_seq, huffman_seq ← {}; $B \leftarrow [1, |R|]$
    for $i \leftarrow 1$ to $|R|$ do
        if $R[i] \in$ top_k_alphabet then tunstall_seq.add($r$)
        else
            huffman_seq.add($r$)
            B[$i$] = 1
    tunstall_seq.encode()
    huffman_seq.encode()
    B.encode()
    HYB ← {tunstall_seq, huffman_seq, B}
    return HYB
```

compression results and it is fast for decoding, because all codes have the same length.

For the rest of the symbols we use an encoding that achieves close to $\mathcal{H}_0$ encoding. The decoding speed does not need to be efficient, because most of the time in decoding we would need to fetch symbols from the Tunstall sequence. We therefore selected Huffman coding because it has good compression efficiency.

For bitvector B', the value $B'[i]$ is 0 if $R[i]$ belongs to the Tunstall sequence. In our ideal case when Tunstall is used for encoding most of the symbols of the sequence R, the bitvector B' would be very sparse. Therefore, for example, Okanohara and Sadakane's sd-array [35] could be good practical choice for B'. Hybrid vector [20] also benefits if B' has a small number of runs, which could be true because in the ideal case the bitvector is sparse. Therefore we test both implementations for bitvector B' in our experimental testing.

3.1.2 Relative Lempel-Ziv

Relative Lempel-Ziv (RLZ) is a relative new dictionary based compression method [7, 24], that has recieved a lot of interest in research Ex. [6, 8, 21, 23, 25, 34]. RLZ is based on Lempel-Ziv 77 [45] (LZ77) compression method but it uses a static dictionary instead of an adaptive/window dictionary. The static dictionary is formed from the target sequence or using some known dictionary. Usually the RLZ dictionary is referred to as the *reference sequence*. RLZ was first developed for genome data, which have repetitive structure. When RLZ was used for genome data, one sequence in the collection of genomes is selected to be the static dictionary. RLZ was later found to also be a good compression method for other applications and it gets best compression on data if it has repetitive structure.

Let $\mathcal{S}$ be the target sequence and $\mathcal{R}$ its reference sequence. Relative Lempel-Ziv factorizes sequence $\mathcal{S}$ by $\mathcal{R}$ to parts $p_1 p_2 \dots p_m$, where each part p_i $(1 \leq i \leq m)$ is *i*) a symbol s that do not occur in $\mathcal{R}$; *ii*) Longest prefix of $\mathcal{S}[j, |\mathcal{S}|]$ that occurs as a substring in $\mathcal{R}$, where $j = |p_1 p_2 \dots p_{j-1}|$. If the prefix $\mathcal{S}[j, |\mathcal{S}|]$ starts in $\mathcal{R}$ at index k and has length of l, then p_i encoded as (k, l), which is called a factor. In ideal situation compressing sequence with RLZ is that $\mathcal{S}$ is factorized to $\mathcal{S} = \underbrace{\mathcal{R}\mathcal{R}\dots\mathcal{R}}_{m}$.

Example let $\mathcal{S} = 255257222225552225$ and $\mathcal{R} = 22525552$, then Relative Lempel-Ziv factorization of $\mathcal{S}$ be $RLZ(\mathcal{S}|\mathcal{R}) = (4,3)(2,2)7(8,3)(1,3)(6,6)$.

As described above, RLZ factorizes the sequence by a greedy process, reading the sequence

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
$\mathcal{S} =$	2	5	5	2	5	7	2	2	2	2	2	5	5	5	2	2	2	5
$\mathcal{R} =$	2	2	5	2	5	5	5	2										

Figure 3.2: Table of sequence $\mathcal{S} = 255257222225552225$ and its reference sequence $\mathcal{R} = 22525552$.

from left to right. It does not not try to find an optimal partition. Also it is still an open question how to construct an optimal reference sequence $\mathcal{R}$ from $\mathcal{S}$. However, Hoobin et al. [17] have showed that taking random samples from $\mathcal{S}$ and concatenating them to form $\mathcal{R}$ gives efficient factorization in practice. In their paper, they even obtained good results when $\mathcal{R}$ was only 0.1% the size of $\mathcal{S}$. Gagie et al. [11] provided a theoretical analysis of why randomly generated references lead to effective compression.

We are assuming that the bitvector B consists of a list of postings lists and each postings list is formed from different and unique terms. This means mostly postings lists are not the same kinds, which leads to characteristic bitvectors of each list having different kinds of structure, and finally that sequence of runs-lengths mostly do not consist of large similar structures. That is why we cannot choose some posting list or some subsequence as reference sequence like in DNA/RNA sequences. Then we are going to build a reference sequenece for randomly taking random sample blocks of length b from the input sequence.

3.2 Zombit-vector

Second part of this chapter focuses on the new bitvector compression method called a Zombit-vector introduced by Gómez-Brandón [14]. The Zombit-vector can represent a bitvector of length n using $\mathcal{O}(rb + \frac{n}{b}) + o(n)$ space, where r is the number of runs of 1s in B and b is block length. It can support both RANK, ACCESS and NEXTGEQ in $\mathcal{O}(1)$ time. Our method is to implement Zombit-vector and investigate its space usage better than in the original paper. We also for the first time implement the NEXTGEQ operation for Zombit-vector, benchmarking and compare it to PEF. Next we introduce Zombit-vector and how a NEXTGEQ query can be answered in constant time.

A Zombit-vector represents bitvector B with three bitvectors U, O and M. To form these bitvectors we first have to split B into blocks of size b. Bitvectors U and O have length of $\lceil n/b \rceil$ and M's length depends on B blocks properties. Each block is labelled as z, o or m depending on block values. If a block has only 0s, it is labelled z. If it is full of 1s, its label is o, and otherwise the label is m.

The bitvector U indicates union blocks. If a block i is labelled with z or o then $U[i] = 1$ ($i \in \lceil n/b \rceil$), otherwise $U[i] = 0$. The bitvector O determines if a block has any 1s in it and can be called *ones blocks*. Then $O[i] = 1$ if the block is labelled with o or m and otherwise $O[i] = 0$. The bitvector M is concatenation of all the blocks that have been labelled with m. A Zombit-vector can be represented also as a recursive schema. In recursive version, the bitvector M is represented recursively as a Zombit-vector until level c. Figure 3.3 illustrates Zombit-vector without recursion with block length of 2. Algorithm 4 give detailed instruction how to build Zombit-vector. For simplicity BUILD-ZOMBIT expect b mod $|B|$ to be 0. For example blocks B_1, B_3 and B_6 are only blocks labelled with m, it can be seen that $M = B_1 B_3 B_6$. Blocks B_1, B_3 and B_6 contain 0s and 1s, therefore $O[i] = 1$ but $U[i] = 0$ for all $i \in \{1, 3, 6\}$.

	B_0		B_1		B_2		B_3		B_4		B_5		B_6		B_7	
i:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
B:	1	1	0	1	1	1	1	0	0	0	0	0	1	0	1	1
labels:	o		m		o		m		z		z		m		o	

(a)

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
U:	1	0	1	0	1	1	0	1
O:	1	1	1	1	0	0	1	1
M:	0	1	1	0	1	0		

(b)

Figure 3.3: Example of Zombit-vector constructed from $B = 1101111000001011$ with block length 2. Figure 3.3(a) shows bitvector values and how blocks are labelled. Figure 3.3(b) represents the final Zombit-vector (U, O and M-vectors) for bitvector B combined according to the block labels and properties. For example three blocks are labelled with m, the M-vector is then of length 6 and it values are the concatenation of block B_1, B_3 and B_6. Also because B_0 is labelled to o, this means the block is full of ones and union blocks and then $U[0] = O[0] = 1$.

Total space usage of the Zombit-vector depends on the properties of the input bitvector B. Zombit-vector use $2 \cdot \lceil n/b \rceil + |M|$ bits to represent B. Gómez-Brandón [14] showed an upper bound for Zombit-vector to be $\mathcal{O}(rb + \frac{n}{b}) + o(n)$ bits. If b is selected to $\sqrt{n/r}$, then the space bound reduces to $\mathcal{O}(\sqrt{rn}) + o(n)$ bits. The size of the recursive representation

Algorithm 4: BUILD-ZOMBIT, builds Zombit-vector with block size b from bitvector B

```
Input: Bitvector $B$, block size $b$, recursion level c
Output: next or equal integer in $B$ than $x$
Function build-Zombit($B, b,$ c):
    U, O ← init $bitvector(\lceil b/n \rceil)$
    label ← $o$; m_blocks ← $0$
    for $i \leftarrow 1$ to $\lceil b/n \rceil$ do
        if $B[ib] = 0$ then label ← $z$
        else label ← $o$
        for $j \leftarrow 2$ to $b$ do
            if $B[ib + j] = 0$ and label $= z$ then continue
            if $B[ib + j] = 1$ and label $= o$ then continue
            if $B[ib + j] = 0$ and label $= o$ then label ← $m$; break
            if $B[ib + j] = 1$ and label $= z$ then label ← $m$; break
        if label $= z$ or label $= o$ then U[$i$] ← $1$
        else U[$i$] ← $0$
        if label $= z$ then O[$i$] ← $0$
        else O[$i$] ← $1$
        if label $= m$ then
            m_blocks ← m_blocks $+ 1$
            m_blocks_idx.add($i$)
    M ← init $bitvector(b \cdot$ m_blocks$)$
    $i \leftarrow 1$
    for $x$ in m_blocks_idx do
        for $k \leftarrow 1$ to $b$ do M[$i$] ← $B[xb + k]; i \leftarrow i + 1$
    if c $> 0$ then
        $b'$ ← block-model()
        M ← Zombit-Vector(M, $b'$, c $- 1$)
    return (U, O, M)
```

is not clear. In the original paper [14] it is claimed to be $\mathcal{O}(r^{1-\epsilon}n^{1-\epsilon})$, where $\epsilon = 1/2^c$. With a recursive representation the operations get a bit slower and can be computed in

$\mathcal{O}(\log n/r)$ time.

3.2.1 NextGEQ query

A non-recursive version of the Zombit-vector provides efficient constant time NextGEQ queries on B. Let x be an integer in the range $[0, |B|)$; beg_i be first index in block B_i; end_i be last index in B_i; and Δ_k be distance of beg_i and $beg_i + k$, such that $beg_i \leq k \leq end_i$. Let B_j be a block where x lies in B. There are three scenarios for finding successor of x: i) Block B_j is labelled with o, then $B[x] = 1$ and the successor of x is itself; ii) Block B_j is labelled with m and the successor of x is in the same block. iii) Block B_j is labelled with z or m and the successor of x is in the next block that contains 1s (either block labelled with o or m).

Case i) is the easiest, and we can return just x. To check that the case is i), we need to check that both $U[j] = O[j] = 1$. For example, if $x = 5$, then x is in block B_2, and $U[2] = O[2] = 1$, therefore the result is 5.

In case ii) because B_j is labelled with m, we need to find the location of block B_j in M. This can be found with $q = \text{RANK}_0(U, j)$ because if for some block index k, $U[k] = 0$ then block B_k is labelled with m. Then the final location of B_j in M is found with $beg_M(q)$. The successor for x in case ii) can be answered by quering the successor of $beg_M(q) + \Delta_x$ in M. If s is the result of the query, then the final answer is $beg_B(j) + \Delta_s$. For example if $x = 2$, x is in block B_1 $(2/2 = 1)$ and the block is labelled with m. Then $\delta_x = 0$ and $q = \text{RANK}_0(U, 2) = 1$, which tells us that B_1 is the first block labelled with m and its bits are located in $M[0, 1]$. To get the final result, we need to compute $\text{NextGEQ}(M, beg_M(1) + \Delta_2) = \text{NextGEQ}(M, 0 + 0)$, which is 1, and the final result is $beg_B(1) + \Delta_1 = 2 + 1 = 3$.

If $s > end_M(q)$ or B_j is labelled with z, searching for a successor for x falls in case iii). To find the successor of x in this case we need to find the next block $B_{j'}$ that contains at least one 1. bit. This is found with $\text{NextGEQ}(O, j + 1)$ query. If $B_{j'}$ is a union block, we can just return the starting index of $B_{j'}$. Otherwise $B_{j'}$ is a mixed block and we can find the next 1-bit the same way as in case ii).

Algorithm 5 gives details on how to find NextGEQ in a Zombit-vector.

Algorithm 5: ZOMBIT-VECTOR-NEXTGEQ, return first index i greater or equal than the index x, such that $B[i] = 1$, where B is the original input bitvector for Zombit.

```
Input: Bitvector $B$ and integer $x$
Output: next or equal integer in $B$ than $x$
1  Function nextGEQ($B, x$):
2      $j \leftarrow \lfloor x/b \rfloor$
3      $q \leftarrow \textsc{rank}_0(\mathsf{U}, j)$
4      if $\mathsf{U}[j]$ then
5          if $\mathsf{O}[j]$ then return $x$
6      else
7          $s \leftarrow \textsc{NextGEQ}(\mathsf{M}, (\mathsf{beg}_q + \Delta_x))$
8          if $s \leq \mathsf{end}_q$ then return $\mathsf{beg}_j + \Delta_s$
9      $j' \leftarrow \textsc{NextGEQ}(\mathsf{O}, j + 1)$
10     if $\mathsf{U}[j']$ then return $\mathsf{beg}_{j'}$
11     $q \leftarrow \mathsf{U}[j]\ ?\ q : q + 1$
12     $s' \leftarrow \textsc{NextGEQ}(M, \mathsf{beg}_q)$
13     return $\mathsf{beg}_{j'} + \Delta_{s'}$
```

3.2.2 Practical implementation

In the original Zombit article [14], it is not described how U, O and M-vectors are implemented. Also implementation of local NEXTGEQ operation for O and M-vector is not discussed in the article. Therefore we experiment in Section 4.2.2 with different practical implementations for U, O and M-vectors. We test of implementations *bitvector*, *sd-vector*, *hybrid vector*, *interleaved bitvector* and *RRR* from SDLS-library [12] for all Zombit bitvectors. We also try different block sizes, and test how it affects compression size. As Algorithm 5 use Δ-functions (modulo), it could be good choice to choose a block size that is a power of 2, which can give as benefits in practice.

Zombit-vector can be implemented recursively, as mentioned earlier. Therefore we want to test how much a recursive implementation of the M vector can affect total compression. An M-vector consists of mixed blocks, and then each block has a run of 0s and 1s. Depending on the block size and concatenations of the blocks, in each level the probability that the M-vector would have blocks full of 0s or 1s goes down rapidly. Then doing a deep recursion would not be efficient because the M structure is random (many small runs of 0s and 1) and could not compress further. The original article does not say anything about what block size should be used at the next recursion level. One choice would be to compute the optimal block size. In our experimental testing we try 3 different block models for the next recursive level, *div2, div4* and *div8*. As the names suggest, *divX* divides current level block length by X and continues to the next recursive level with this and keeps next block length as power of 2. In our experimental testing, recursion ends when the input recursive-level is zero or next recursive level block length is 1. When the block length is one, no further compression of bitvector M can be achieved.

Implementing local NextGEQ query

For NextGEQ queries for O, M-vectors we want to test two different options, which are described in Algorithms 6 and 7. Algorithm 6 computes the successor of x using rank and select functions and returns $\text{select}_1(B, \text{rank}_1(B, x)+1)$. If $\text{rank}_1(B, x) = \text{rank}_1(B, |B|)$ this means B do not have more 1s after x and the select_1 function is not valid for $\text{rank}_1(B, x)+1$. Therefore B does not have successor for x. The time for finding a successor with rank and select depends on the implementations of those. However, although the time for finding the successor with rank and select would be constant time, it can be "slow" in practice because of the complex index structure for the select function. Therefore the query can include many cache misses. Our other method for local NextGEQ tries to get around of that.

Algorithm 6: nextGEQ-rank-select, return first index i greater or equal than the index x, such that $B[i] = 1$.

```
Input: Bitvector $B$ and integer $x$
Output: next or equal integer in $B$ than $x$
1 Function nextGEQ-rank-select($B, x$):
2     $r \leftarrow \text{rank}_1(B, x) + 1$ if $r > \text{rank}_1(B, |B|)$ then
3         return $-1$
4     return $\text{select}_1(B, r)$
```

We assume that O, M-vectors have random structure or more dense 1s than 0s. Therefore, in practice the successor of x is located nearby x, and in practical implementation actually inside the same word or in the next one. Of course there might be cases where this is not true, specially with O-vector, where the original input bitvector holds a lot of blocks labelled with z (full of 0s). Therefore we want to use the same kind of super block structure to find the next 1 bit from x as practical RANK structures computes RANK queries in contant time. We actually use almost the same super block structures as *rank_support_v5* from [12]. A *rank_support_v5* stores super block prefix values of RANK queries for all $2048i$, $i \in [0, |B|/2048]$. The next super block that has a higher value than the previous super block, contains the answer for the x NEXTGEQ query. Algorithm 7 presents more detail how to find a local NEXTGEQ answer using super block scanning, and then word scanning inside the super block. At a high-level we check first the current super block for the next 1 bit (iterating super block words from location of x). If the result cannot be found, then we iterate the super blocks until super block i has a higher value than super block $i-1$ ($SB[i] > SB[i-1]$). After super block i, we iterate its word-level blocks. The successor of x is in the word that is greater than zero (i.e. has some 1 bits). After the word block is located, the first 1 bit can be found on it by a built-in hardware function like *___builtin_ctz*, which returns the number of zeros before the first 1 bit in the integer. This implementation is interesting only for plain bitvectors because it is hard to get word blocks of compressed bitvectors. For our implementation we assume w is a power of 2.

Algorithm 7: NEXTGEQ-SUPERBLOCK-SCAN, return first index i greater or equal than the index x, such that $B[i] = 1$ by finding first next super block that has a 1-bit, and then scanning that super block words and stopping after finding first 1-bit.

```
Input: Bitvector $B$ and integer $x$
Output: next or equal integer in $B$ than $x$
1  Function nextGEQ-superBlock-scan($A, x$):
2      $\mathsf{x\_w} \leftarrow A[\lfloor x/w \rfloor] \gg (x \,\&\, (w-1))$
3      if $\mathsf{x\_w} > 0$ then return $x +$ __builtin_ctz(x_w)
4      if $\lfloor x/64 \rfloor = \lfloor |B|/64 \rfloor$ then return $-1$
5      $\mathsf{sb\_i} \leftarrow x/2048$
6      if $|\mathsf{sb}| = 0$ or $(x/2048) \geq |S|$ then $j \leftarrow |B|/64$
7      else $j \leftarrow (\mathsf{sb\_i} + 1) \cdot 32 + 32$
       // Check first current super block
8      for $i = \lfloor x/w \rfloor + 1$ to $j$ do
9          if $A[i]$ then return __builtin_ctz($A[i]$) $+ iw$
       // Find next super block that have 1-bit
10     while $\mathsf{sb\_i} + 1 \leq |\mathsf{sb}|$ and $\mathsf{sb}[\mathsf{sb\_i}] = \mathsf{sb}[\mathsf{sb\_i} + 1]$ do $\mathsf{sb\_i} \leftarrow \mathsf{sb\_i} + 1$
11     if $x/2048 = \mathsf{sb\_i}$ then $j \leftarrow (x/64) + 1$
12     else $j \leftarrow (\mathsf{sb\_i} \cdot 32 + 32)$
       // Scan founded super block
13     for $i = j$ to $|A|$ do
14         if $A[i]$ then return __builtin_ctz($A[i]$) $+ iw$
15     return $-1$
```

4 Experiments

In this chapter we present experiments aimed at determining the compression effect and speed of NextGEQ gained with our methods. The efficiency of the NextGEQ function is only measured for Zombit-vector. We used the Gov2 dataset as an input for compression, which is widely used in the information retrieval community. Because the Gov2 dataset is composed of a list of postings lists, we also show how we converted that list into a bitvector for compression.

First we show compression results and charasteristics of structures from RLZ and the Top-k Hybrid coder, and then compression results of Zombit-vector. Finally, we show efficiency of the Zombit-vector's NextGEQ query.

sequences	n	size	int size	alphabet size
Gov2	5 055 078 461	38GB	64	NA
R	3 303 572 206	13GB	32	175 118
R_0	1 651 786 104	6.2G	32	175 000
R_1	1 651 786 104	6.2G	32	7 899

(a)

	n	size	1-bits	runs of 1s
B	196 433 251 848	23GB	5 055 078 461	1 651 786 104

(b)

Figure 4.1: Information on Gov2 dataset and run-length sequences R, R_0 and R_1. Run-length sequences are created from the bitvector in Figure 4.1(b), which is created using the Gov2 dataset as an input for Algorithm 8. The alphabet for the Gov2 sequence is NA because the file has been already preprocessed to one big integer set from lists of postings lists.

4.1 Setup

All tests were run on Cubbli/Ubuntu 20.04.5 LTS (GNU/Linux 5.11.0-40-generic x86_64) with Intel(R) Xeon(R) CPU E7-4830 v3 @ 2.10GHz CPU and 1,5MiB + 1,5MiB L1, L2 12MiB and L3 120MiB cache levels. The test server had 1.4TB size of RAM. All tests were executed with neglible usage on the server. All tested programs were implemented in C++ and compiled using the GNU Compiler Collection (GCC) version 9.4.0 with *-std=c++11 -03 -lsdsl -ldivsufsort -ldivsufsort64* compiler optimization and libraries. Implementations used help structures from SDSL-library [12].

The Gov2 dataset consits of 64-bit positive integers. To encode this sequence of integers we create a bitvector from it. The constructed bitvector can be thought of as a charasteristic bitvector of a large integer set. Figure 4.1 gives information on Gov2 dataset and the run-length sequences created from the bitvector.

To form the bitvector from Gov2, we used the following Algorithm 8 to preprocess it.

Algorithm 8: BUILD-BITVECTOR , builds bitvector from given sequence of positive integers.

```
Input: Intger sequence $\mathcal{S}$
Output: bitvector $B$
Function built-Bitvector($\mathcal{S}$):
    S_diff[1] ← $\mathcal{S}[1]$
    sum ← S_diff[1]
    for $i \leftarrow 2$ to $|\mathcal{S}|$ do
        if $\mathcal{S}[i] > \mathcal{S}[i-1]$ then S_diff[$i$] ← $\mathcal{S}[i] - \mathcal{S}[i-1]$
        else S_diff[$i$] ← $\mathcal{S}[i]$
        sum ← sum + S_diff[$i$]
    B ← init bitvector(sum + 1)
    for $i \leftarrow 2$ to $\sum$ do B[$i$] = 0
    sum ← S_diff[1]
    B[sum] ← 1
    for $i \leftarrow 2$ to $|\mathcal{S}|$ do
        sum ← sum + S_diff[$i$]
        B[sum] ← 1
    return B
```

4.2 Results

Next we present the results of our experiments. We observe all encodings beat PEF in terms of space. Figure 4.2 highlights the results of the top configuration for each encoding. All encodings were tested with different configurations to find the best compression result and to see how encodings behave with different settings.

Overall Top-k Hybrid encoded postings lists to the most compact from, using only 2.463 bit per posting which is clearly smaller than the current state-of-the-art, PEF. However, with this configuration Top-k Hybrid coder is not efficient at NextGEQ, because not enought symbols are not encoded with Tunstall coding. With the best settings for hypothetical speed/space tradeoff, Top-k Hybrid coder got compression result 2.705 bpp, which was still better than the other encoders. The speed for NextGEQ query for Top-k Hybrid might not be constant time, which makes it somewhat unattractive, especially in cases were speed is critical. Therefore RLZ and Zombit-vector results may give a better overall space/speed-tradeoff.

encoder	size (bpp)	size (GB)
Top-128 Hybrid coder(R_0)+ Top-8 Hybrid coder(R_1)	2.705	1.592
Top-64 Hybrid coder(R)	2.958	1.741
RLZ ($R_0 + R_1$)	3.193	1.879
RLZ (R)	3.156	1.857
Zombit-vector<RRR>-512-Rec1 SB scan	3.121	1.837
PEF	8.838	5.201

Figure 4.2: Figure highlights results of top configuration from each encoding tested in this thesis. RLZ and Top-k Hybrid coder are presented two times in the figure because both were tested with sequence of runs of 0s and 1s (R) and runs separated to separate sequences (R_0, R_1). Encoders are not in spesific order.

4.2.1 Run length sequence(s) compression

Next we look at RLZ and Top-k Hybrid coder compression results on run-length encoding of the bitvector of the Gov2 dataset. Both encoders obtain good overall compression results. Results for RLZ are just hypothetical and give us a good guideline for future research work. It is interesting to see that RLZ is more efficient with compressing runs of

0s and 1s together, whereas Hybrid coder is more efficient when runs are split into separate sequences. In this Section, R means sequence of run-lengths of 0s and 1s. Sequence R_0 and R_1 holds runs of 0s and 1s separately. Sequences R, R_0 both have a large alphabet approximately 175 000, but the 512 most frequent symbols cover about 99% of the symbols in the sequence. The alphabet for R_1 is much smaller, and to cover about 99% of the sequence symbols in R_1, only the 8 most frequent symbols in needed. This is what we expect from a run-length sequence of postings lists, but still when only 512 symbols are needed to cover 99% of symbols in the sequence, Tunstall coding lacks the efficiency to adequately compress R and R_0 in Hybrid coder.

Relative Lempel-Ziv results

For Relative Lempel-Ziv we experimented with different sizes of reference seqeunce. We used sizes from 0.1% to 0.4% of the input sequence. Results for RLZ are promising and better than we expected. It seems that RLZ compresses R and R_1 much better than R_0. Sequence R beats sequence R_0 and R_1 in space when they are combined to together. Our space usage for RLZ is hypothetical and its computed as follows:

$$\lceil \log_2 |\Sigma_{\mathcal{R}}| \rceil \cdot |\mathcal{R}| + 32 \cdot \#factors.$$

Therefore, our space usage of RLZ consists of encoding of the reference sequence $\mathcal{R}$, where we could encode each symbol in the reference sequence using $\log_2 |\Sigma_{\mathcal{R}}|$ bits, where $\Sigma_{\mathcal{R}}$ is alphabet of $\mathcal{R}$. And then we could encode the factors using 32 bits for each factor. For aswering queries like NextGEQ, the RLZ encoding would need, for example, some kind of succinct structure for prefix sum queries. Therefore our results are not that accurate but represent a good lower bound. All RLZ results can be seen in Figure 4.3. The figure presents information of RLZ properties, such as the amount of factors, factors of length one, and the size of reference sequence alphabet.

Although R_0 and R_1 have same length, there is big difference in compression results. One explanation is that R_1's alphabet is much smaller than R_0's, where R_1's alphabet size is 4% of R_0's alphabet size. Then less and longer factors are needed to encode R_1. This can be seen also in the results in Figure 4.3, where in the worst case R_1 have approximately 93 million factors, where R_0 has in best case approximately 360 million. Also the average factor length is over 4 times bigger with R_1, and there are 20 times less factors of length 1.

Increasing the size of the reference sequence gives better compression results. With all sequences, the amount of factors and length-1 factors decrease, and average/maximum factor lengths also increase. However, encoding the refence sequence gives more overhead to overal space usage than compression effect obtained with increasing reference seqeunce size.

Although with R, R_0, the reference sequence alphabet covers only on average 30% of the input seqeunce symbols, we are going to see with Top-k Hybrid coder that with only 512 symbols, 99% of the whole sequence is covered.With RLZ we are not sure if the alphabet includes all those most frequent symbols, which could lead to an inefficiency of compression with R and R_0. With R_1 the 2 most frequent symbols cover 88% of the sequency and the 4 most frequent 96% of the symbols. This might be the reason why RLZ is efficient with R_1: there is simply a higher probability that these 4 symbols are inlcuded with reference sequence alphabet.

sequence	Reference seq size (%) from org seq	Reference seq $\|\Sigma\|$	factors	AVG factor length	MAX factor length	unit factors	**size (bpp)**
R_0	0.1	20 084	410 312 448	4.03	774	28 684 491	2.656
	0.2	28 434	385 822 680	4.28	770	21 711 299	2.560
	0.3	34 565	371 055 902	4.45	1031	18 388 518	2.604
	0.4	39 934	360 135 025	4.59	1068	16 294 817	2.620
R_1	0.1	1 609	93 257 441	17.71	1034	870 562	0.633
	0.2	2 267	87 600 981	18.86	1302	604 589	0.639
	0.3	2 577	84 137 491	19.63	1649	445 458	0.787
	0.4	2 845	81 628 776	20.24	2057	375 865.	0.857
R	0.1	20 479	497 280 867	6.64	817	2 734 109	3.246
	0.2	28 572	467 585 765	7.07	1024	1 976 057	3.156
	0.3	34 862	449 614 623	7.35	1254	1 623 287	3.160
	0.4	40 332	436 338 017	7.57	1792	1 396 758	3.180

Figure 4.3: Results of RLZ compression on run-length sequences. Three different sequences were tested, R, R_0 and R_1. Sequence R holds run-length values of 0s and 1s from bitvector formed from Gov2 dataset. Sequences R_0 and R_1 holds also run-length values from bitvector formed from Gov2 dataset, but R_0 have only run-length values for 0s and R_1 for 1s. Table results factorize compression results of RLZ. Each sequence is tested with 4 different sizes reference sequence, 0.1-0.4% from original sequence.

Top-k	R-Tunstall bpp	R-Huffman bpp	R-B bpp	R% symbols in tunstall	R total bpp
2	0.381	1.602	1.107	0.601	3.09
4	0.738	1.316	0.889	0.703	2.944
8	1.129	1.038	0.714	0.786	2.88
16	1.535	0.779	0.539	0.853	2.853
32	1.924	0.555	0.398	0.904	2.876
64	2.312	0.374	0.271	0.94	2.958
128	2.706	0.238	0.178	0.964	3.122
256	3.082	0.142	0.109	0.98	3.333
512	3.448	0.079	0.063	0.99	3.59
1024	3.791	0.041	0.033	0.995	3.866
2048	4.145	0.021	0.017	0.998	4.184
4096	4.473	0.011	0.009	0.999	4.493

Figure 4.4: Results of Top-k Hybrid coder on Gov2 dataset for run-length sequence of 0s and 1s (R). Hybrid coder results are presented separately by each substructure. Column *R% symbols in Tunstall* shows how usefull R would be in practise for NEXTGEQ, values higher $\sim$ 90% are interesting.

Top-k Hybrid coder results

Top-k Hybrid coder shows interesting results for compression and it obtains the best compression results compared to others encodings we present in this thesis. When runs of 0/1s are separated, compression results are more efficient than having a combined sequence. However, combining runs into same sequence R may be a much better choice even if it is bit larger because decoding is simpler and has better cache behavior. All Top-k Hybrid coder results are presented in Figures 4.5 and 4.4. In the figures, Top-k Hybrid coder strctures are broken down. We exprerimented with different k values from 2 to 4096. For bitvector B, we used the *sd-array* implementation from [12], and omit results were B is implemented with Hybrid bitvector.

The results show that it is not feasible to choose always the same value for k. This is explained from the sequences alphabet size and symbol frequencies. Therefore, choosing best value for k seems to have some sweet spot for all sequences. Tunstall coding seems to have better compressibility with smaller k values. This is explained by the fact that, when

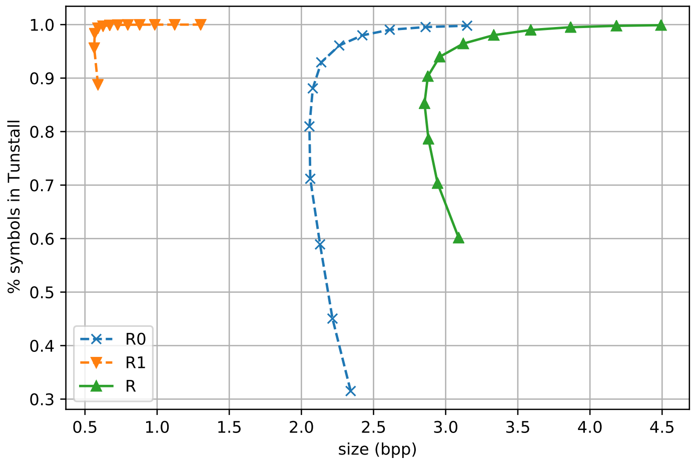
Top-k	R_0-Tunstall bpp	R_0-Huffman bpp	R_0-B bpp	R_0% symbols in Tunstall	R_0 total bpp	R_1-Tunstall bpp	R_1-Huffman bpp	R_1-B bpp	R_1% symbols in Tunstall	R_1 total bpp	total bpp (R_0 + R_1)
2	0.122	1.457	0.762	0.315	2.342	0.25	0.119	0.221	0.888	0.59	2.932
4	0.304	1.248	0.663	0.45	2.216	0.397	0.06	0.107	0.956	0.564	2.78
8	0.56	1.007	0.562	0.589	2.129	0.491	0.029	0.048	0.983	0.567	2.697
16	0.861	0.765	0.436	0.712	2.061	0.555	0.014	0.022	0.994	0.59	2.652
32	1.176	0.548	0.332	0.81	2.056	0.608	0.007	0.01	0.997	0.625	2.681
64	1.478	0.371	0.23	0.881	2.079	0.663	0.003	0.005	0.999	0.672	2.751
128	1.748	0.236	0.153	0.929	2.138	0.722	0.002	0.003	0.999	0.727	2.864
256	2.025	0.141	0.096	0.961	2.262	0.794	0.001	0.001	0.999	0.797	3.059
512	2.289	0.078	0.056	0.98	2.422	0.878	0.001	0.001	0.999	0.879	3.302
1024	2.542	0.041	0.03	0.99	2.613	0.983	0.00025	0.0003	0.999	0.984	3.597
2048	2.824	0.021	0.016	0.995	2.861	1.122	0.0001	0.0001	0.999	1.122	3.983
4096	3.13	0.011	0.008	0.998	3.149	1.302	0.00003	0.00003	0.999	1.302	4.451

Figure 4.5: Results of Top-k Hybrid coder on Gov2 dataset for run-length sequence of 0s (R_0) and 1s (R_1). Hybrid coder results are presented separately by each substructure. Column R_x% *symbols in Tunstall* shows how usefull R_x would be in practise for NEXTGEQ, values higher $\sim 90\%$ are interesting. Column *total bpp* combines results from columns R_0 *totall bpp* and R_1 *totall bpp* for same k value but in practise we use different k value for R_0 and R_1.

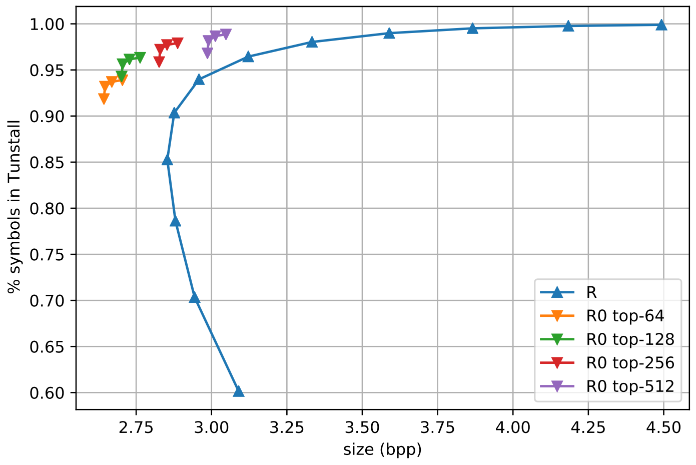
alphabet for Tunstall coding is bigger, then the Tunstall tree becomes more and more flat. This causes codewords to include smaller amounts of symbol, which leads to inefficiency of Tunstall coding.

Because Top-k Hybrid coder was build that some k symbols would take almost all percentage of the symbols, only results where %symbols in Tunstall coding is over 90% are interesting. This is because Tunstall coding provides more efficient decoding versus decoding in Huffman coding. Figures 4.6(a) and 4.6(b) conludes our results of Top-k Hybrid coder. In these figures results are better if they are closer to left top corner. The y-axis shows the percentage of symbols in Tunstall coding and the x-axis shows the size of the encoding. These figures illustrate the sweet spot for the Tunstall coding.

For the bitvector B in Hybrid coder, we tested *hybrid vector* and *sd-array* implementations from [12]. $B[i] = 0$ if i-th symbol in the sequence is encoded with Tunstall and otherwise with Huffman. For the results of the Top-k Hybrid coder, we only show results where bitvector B is implemented with *sd-array*, because it was clearly better with sparse bitvectors. Hybrid vector was only better where the percentage of symbols in Tunstall was low, which means the bitvector will have more 1s and be more random.



(a)



(b)

Figure 4.6: Figures presents Top-k Hybrid coder compression results on Gov2 dataset. On the x-axis is measured the size of the structure and y-axis shows how many symbols (%) from the sequence are encoded with Tunstall coding. Result is better if it is closer to left top corner. On Figure 4.6(a) all sequences R, R_0 and R_1 are showed in own lines, where each point is for different k value. Sequence R holds run-length values for both 0s and 1s, where R_0 and R_1 only for other (0s or 1s). Figure 4.6(b) combines results of R_0 and R_1. Values are combined that 4 best results from each R_0 (Top-k: 64, 128, 256, 512) and R_1 (Top-k: 4,8,16,32) added together, such that each line on the figure expect line for R varies R_1 values on each R_0 results separately.

4.2.2 Zombit-vector expriments

For Zombit-vector we tested it with different block sizes and different implementations for U, O and M-vectors. All bitvector implementations used were from the SDSL-library

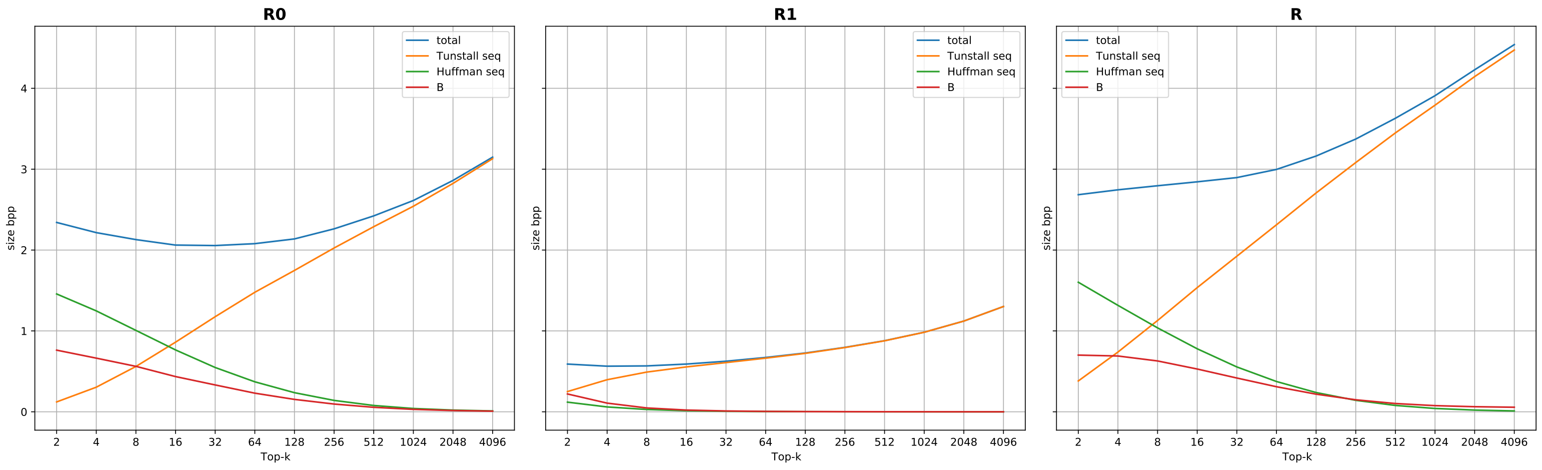


Figure 4.7: Figures presents Top-k Hybrid coder compression results breaked down to each substrucutre on Gov2 dataset for each run-length sequences separately and different k values. Left figure shows results on R_0 sequence, middle for R_1 and right for R. Sequence R holds run-length values for both 0s and 1s, where R_0 and R_1 only for other (0s or 1s).

[12]. In our experimental testing we used block sizes as powers of 2 from 16 to 16384 and the five bitvector implementations from SDSL: SD_ARRAY , RRR , HYBRID-VECTOR , PLAIN BITVECTOR and INTERLEAVED BITVECTOR . All block sizes were tested with the recursive scheme. Overall space usage of non-recursive Zombit-vector is interesting because it reduces space lower than PEF space usage. Also Zombit is faster with NEXTGEQ query, which gives it much better space/time tradeoff and makes it much more attractive for compressing postings lists. In our Zombit-vector results we show only the most important results and focus on Zombit-vectors where U and O-vectors are implemented with plain bitvector. For the M-vector we used plain bitvector and RRR implementations. Notation Zombit<X> means that implementation X is used for vector M.

Different block sizes effect how well Zombit-vector compress bitvector. Figure 4.8 plots non-recursive Zombit-vector space usage with different block sizes. On the graph there are four lines, two have plain bitvector implementation for M-vector and the other two M is implemented with RRR. The plot shows that there are tiny differences with space when super block scanning is used versus rank & select for local NEXTGEQ query. The figure shows that lower block sizes are more efficient. This is explained by the fact that with bigger block sizes, the probability for the block being a mixed one is higher. On the other hand, the smallest blocks are not most suitable because those increase sizes of U and O-vector. The curve for RRR is much smoother because it still gets some compression on M-vector. Figure 4.9 verifies why the plain bitvector implementation lacks of compression efficiency with bigger block sizes. Already with block size 512, bitvector implementation on M-vector is dominating Zombit-vector space usage. The figure also shows why we did not use other implementation than plain bitvector for U and O-vector. Using other implementation would have reduced size, but time and cache misses would be increased.

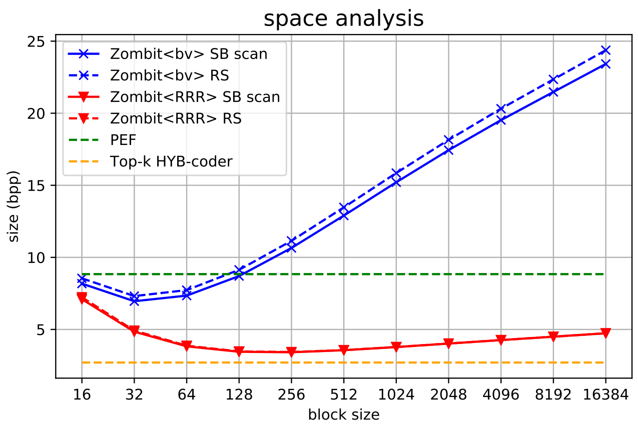


Figure 4.8: Zombit-vector space analysis without recursion model on Gov2 dataset for different block sizes and implementations on Zombit-vector. Blue lines represent Zombit-vector, where M-vector is plain bitvector. For red lines, M-vector is implemented with RRR. For Local NextGEQ query for O-vector (also for M-vector with Zombti<bv>), super block scanning is used for solid lines and dashed lines compute local NextGEQ with rank and select. Dashed orange horizontal line shows compression size of best Top-k Hybrid coder implementation (Top-128 R0, Top-8 R1), and green dashed line shows PEF compression size.

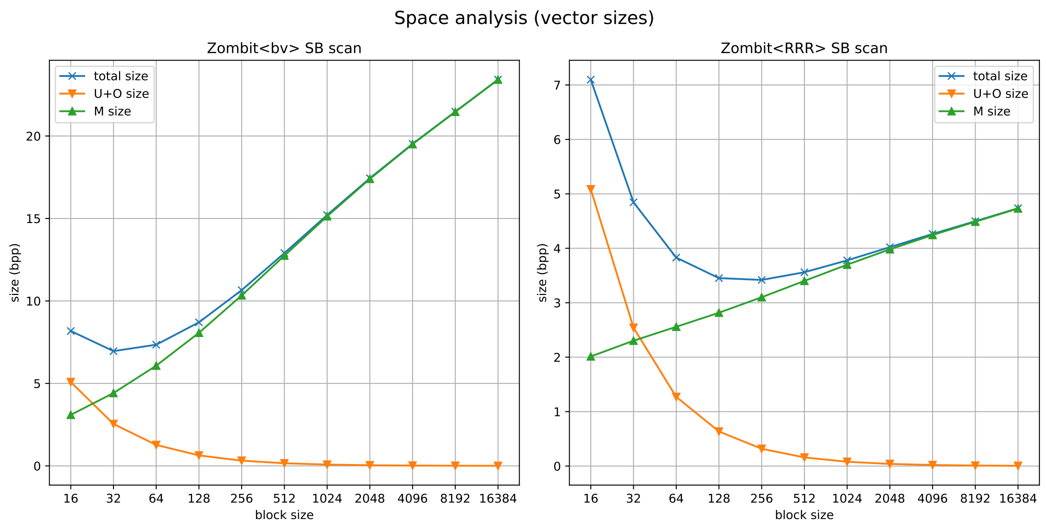


Figure 4.9: Analysis to show how different vectors space usage effect Zombit-vector total space usage on Gov2 dataset with different block sizes and implementations on Zombit-vector. Left figure represents a Zombit-vector, where the M-vector is plain bitvector, and for right figure, M-vector is implemented with RRR. For both figures, space usages are computed in the sense that local NextGEQ query for O-vector (also for M-vector with Zombit<bv>) is computed with super block scanning.

The recursive version of Zombit-vector reduces space usage. Figure 4.10 shows Zombit-vector space usage for each different block sizes on plain bitvector and RRR implementations for M. Especially Zombit-vector with plain bitvector achieve more competitive results with RRR version and reduce close to size of PEF. With the recursive version, increasing the number of blocks for Zombit<RRR> reduces space usage below PEF. For recusive Zombit we show only results for the $div8$ model to determine block length for next recursion level of Zombit. Other models got the same results but with deeper recursive usage. We continued with recurison levels until the next level Zombit-vector would have block length 1 or space usage $k+1$ recursive level would use more memory than the k level Zombit-vector. Going deeper and deeper with recursion is not good because overhead of nested U and O-vector structures grows inside each recursion level. In the figure this can be seen better with Zombit<RRR>.

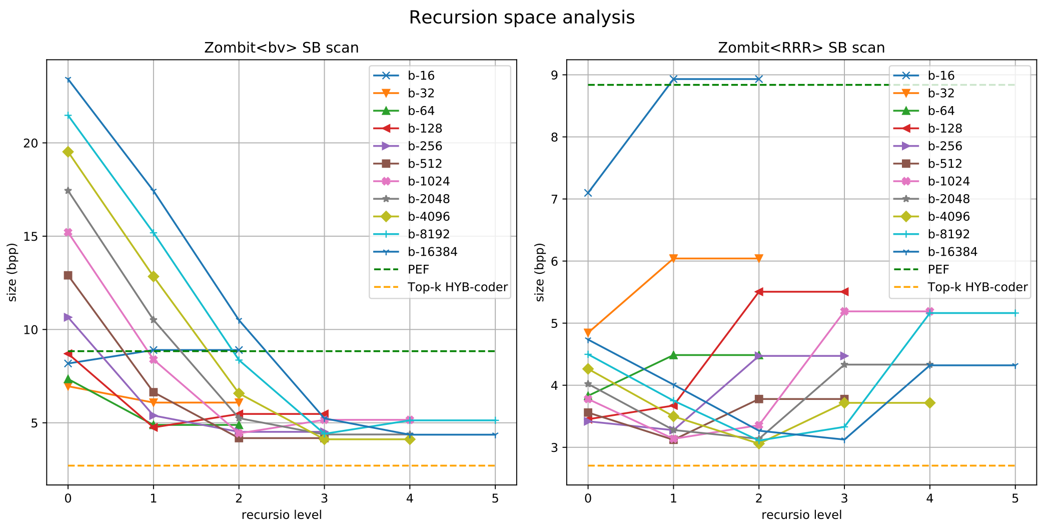


Figure 4.10: Recursive Zombit-vector space analysis on Gov2 dataset for different block sizes on Zombit-vector. Left figure shows space analysis for Zombit<bv> and right for Zombit<RRR>. Each line in the plots represent Zombit-vector implementation for specific block size (b-X), and shows how recursion encoding of M-vector effects on total space usage. For local NEXTGEQ query for O-vector (also for M-vector with Zombti<bv>), super block scanning is used in both figures.

For the NEXTGEQ query, as dicussed in Section 3.2, Zombit-vector was tested with two different solutions for local NEXTGEQ queries. For PEF we used implementation from https://github.com/hiipivahalko/PEF. Figure 4.11 show query results for Zombit<bv> and Zombit<RRR> without recursion. Both implementations were tested with super block scan technique (SB scan) and RANK&SELECT technique (RS). In the figure, solid lines are methods using the super block scan technique and dotted lines are for RANK&SELECT technique. As we expected, super block scanning is the more efficient solution, especially for plain bitvector implementation because we can use the solution for both O and M vectors local NEXTGEQ queries. Another interesting result is that all implementations beats PEF in speed. Figure 4.12 shows how the recursive version of the Zombit-vector effects qeury time. It can be seen that growth is linear, but the deeper recursive version is only usable if it gains enough compression, which would be case with big block size on Zombit<bv>.

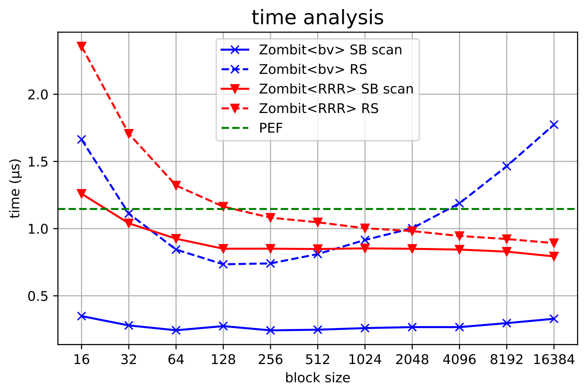


Figure 4.11: Zombit-vector time analysis without recursion model on Gov2 dataset for different block sizes and implementations on Zombit-vector. Blue lines represent Zombit-vector, where M-vector is plain bitvector. For red lines, M-vector is implemented with *RRR*. For Local NEXTGEQ query for O-vector (also for M-vector with Zombti<bv>), super block scanning is used for solid lines and dashed lines compute local NEXTGEQ with RANK and SELECT.

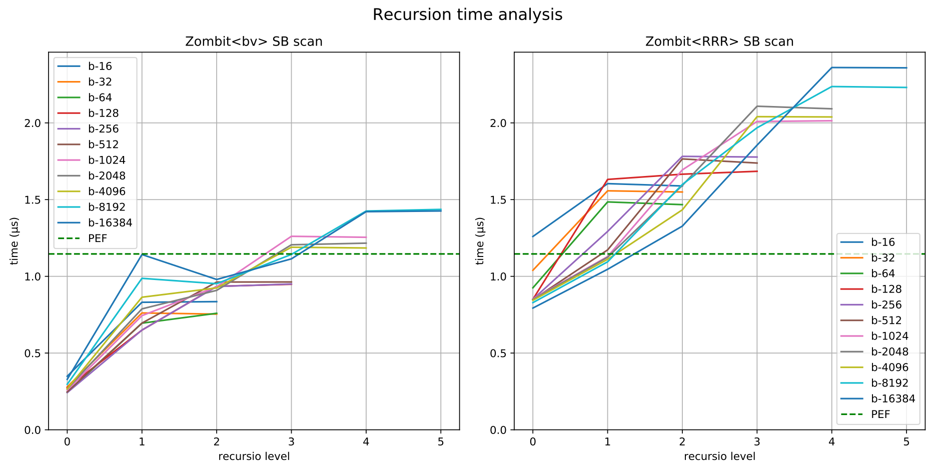


Figure 4.12: Zombit-vector recursion time analysis on Gov2 dataset for different block sizes on Zombit-vector. Left figure shows time analysis for Zombit<bv> and right for Zombit<RRR>. Each line in the plots represent Zombit-vector implementation for specific block size (b-X), and shows how recursion encoding of M-vector effects on average time usage of NEXTGEQ query. For local NEXTGEQ query for O-vector (also for M-vector with Zombti<bv>), super block scanning is used.

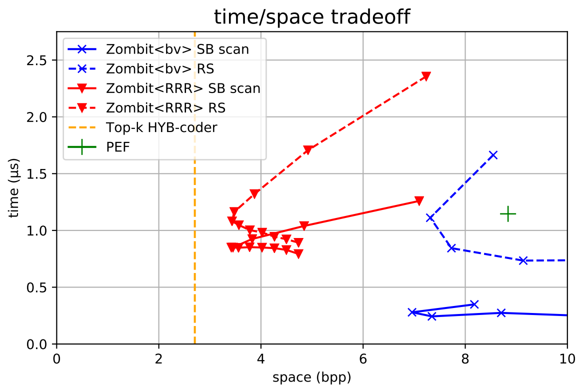


Figure 4.13: Zombit-vector space/time tradeoff on Gov2 dataset for 2 type of Zombit-vector implementations (Zombit<RRR>, Zombit<bv>) and 2 type of NEXTGEQ query for O-vector (SB scan, RS). Figure's x-axis measures space usage for the Zombti-vector and y-axis average time usage for NEXTGEQ query. Each point in the lines represents Zombit-vector implementation with different block size. Point on the figure is better is is more close to bottom left corner. Red lines illustrates Zombit-vector build using RRR-vector for M, and for blue lines Zombit-vector uses plain bitvector for M. For solid lines local NEXTGEQ query for O-vector is implemented with super block scanning (SB scan). Zombit<bv> use super block scanning also for M-vector. Dashed lines use RANK and SELECT (RS) to compute local NEXTGEQ query. Dashed orange vertical line shows compression size of best Top-k Hybrid coder implementation (Top-128 R_0, Top-8 R_1), and green "+" point shows PEF space/time tradeoff.

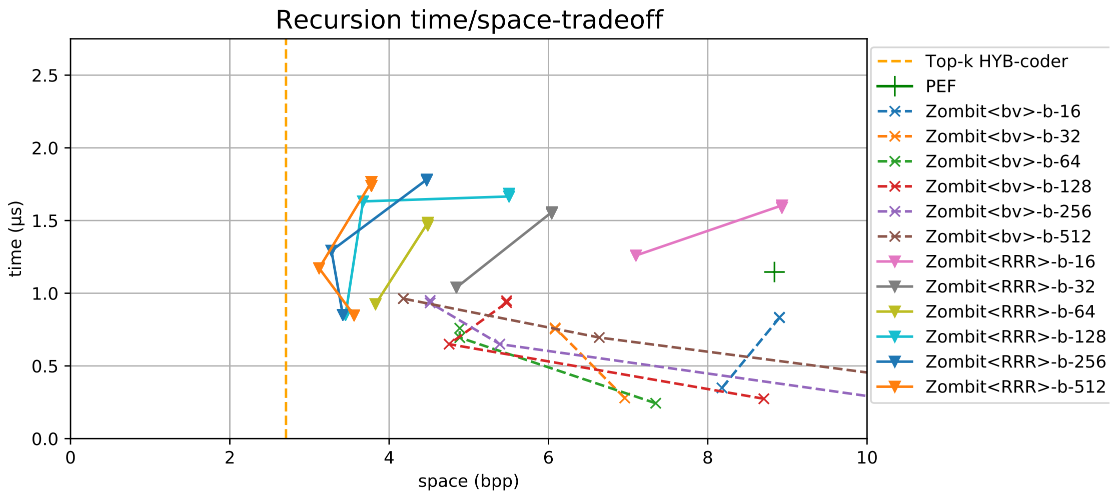


Figure 4.14: Zombit-vector recursion space/time tradeoff on Gov2 dataset for 2 type of Zombit-vector implementations (Zombit<RRR>, Zombit<bv>). Figure's x-axis measures space usage for the Zombit-vector and y-axis average time usage for NEXTGEQ query. Point on the figure is better is is more close to bottom left corner. Dashed lines are for Zombit<bv> and solid lines for Zombit<RRR> implementation. For Local NEXTGEQ query on O-vector (also for M-vector with Zombit<bv>), super block scanning is used. Label b-X represent block size of Zombit-vector in the zeroth recursion level. Dashed orange vertical line shows compression size of best Top-k Hybrid coder implementation (Top-128 R_0, Top-8 R_1), and green "+" point shows PEF space/time tradeoff.

Figures 4.13 and 4.14 conclude the Zombit-vector experimental results. The figures a shows time/space tradeoff, with the bottom left corner being better. Both figures are using same scale and space over 10 bpp are omitted, because they are less interesting. In Figure 4.13, all results are for the non-recursive Zombit-vector. Solid lined presents super block scan results and dotted lines RANK&SELECT results, where blue lines are Zombit<bv> implementation and Zombit<RRR> red ones. The figure shows that super block scan gets a much more better tradeoff, and that non-recursive Zombit<bv> cannot compete with Zombit<RRR>.

In Figure 4.14, only results for super block scan are shown, because it obtained much better results in its non-recursive implementation. Solid lines represents results for Zombit<RRR> and dotted lines for Zombit<bv>. As with the recursive version, Zombit<bv> gets better space results, but Zombit<RRR> is still more attractive choice to take. Also the non-recursive version of Zombit<RRR> is better than the recursive model, although it has lower space usage. Recursive model of Zombit-vector almost doubles the time for NextGEQ query versus non-recursive model.

5 Conclusions

This thesis investigated representations for static integer sets by application of list of postings list converted as charasteristic bitvector. Then representing static integer set converted to representing efficiently specific bitvector which is expected have property of long runs of 0/1s. The thesis was separated into two parts that tried find to redudancy in bitvector of runs of 0/1s. The first part further converted the bitvector to a sequence of run-lengths and tried to find redudancy explicitly. Two different methods: Top-k Hybrid coder and Relative Lempel-Ziv were used to encode sequences of runs. The second part of the thesis focused bitvector compression method called Zombit-vector [14], which is an interesting encoding because of its efficient NextGEQ query. All these encoding methods were introduced in detail and either implemented or used know efficient implementations.Practical implementations were tested with the Gov2-dataset.

The current state–of–the–art, Partitioned Elias-Fano (PEF) encoding, for postings lists characteristic bitvector was introduced by Ottaviano and Venturini [36]. Their solution finds an almost optimal partition of the bitvector, such that each partition can be encoded to smaller universe which improves plain Elias-Fano and plain bitvector encoders. Alternative encodings were presented in this thesis to compete with PEF. All encodings were tested with a real application dataset and tested with different configurations. All encodings introduced in this thesis beat PEF clearly in space, which was interesting result. Althought Top-k Hybrid coder and RLZ beat PEF, time usage for NextGEQ is not yet known and, especially for Top-k Hybrid coder, it is expected to be much slower than PEF. Therefore if speed is a more important feature than space usage, PEF is likely a better solution than Top-k Hybrid coder.

Although Top-k Hybrid coder and RLZ were not compared with PEF for speed, the Zombit-vector space/time-tradeoff results were really interesting, because it beat PEF in both dimensions for NextGEQ queries. However, Zombit-vector space result had biggest variation from 3 bpp to 20 bpp, which shows that the Zombit-vector needs to be configured carefully.

5.1 Future work

Although all methods compressed the bitvector/sequence more efficiently than PEF, there is room for impropment, especially with the Zombit-vector. As we saw earlier, Zombit-vector splits input vector blindly to blocks, and M-vector is the bottleneck in the compression. One way to improve the size of M could be to split input bitvector with two different block sizes. This way more blocks could be labelled with z or o, reducing the size of M. To find the optimal partioning, some preprocing would be needed. Splitting blocks to different sizes could make operations slower, but speed/space tradeoff may still be useful. Because blocks are different sizes, information about which block is which size needs to be stored. This could be done, for example, with bitvector B' of length of b_n/n, where b_n is the number of blocks. Bitvector B' would then have the same size as vectors U and O. As we saw in Section 4.2.2, combined space usage of U and O vectors is small compared to overall space usage, bitvector B' space would be small overhead to overall space usage.

Using different block sizes could also be taken further. It would be interesting to see if all block sizes would be variable size, like in PEF. Block partioning could be done in the same way as it is done with PEF, using a graph based approximation algorithm to determine shortest path in the partioning graph. This way more random bitvectors may also have better compression with Zombit-vector.

As Arroyuelo and Raman showed in [1], run-length encoding could give good compression results for bitvectors having a small number of runs. Although our Top-k Hybrid coder gave better compression results than PEF, it could be improved. For example, Tunstall coding could be replaced with some other more efficient and fast encoding because, as we saw it did not work efficiently when 90% or more of the symbols were encoded with Tunstall. Another way to improve Top-k Hybrid coder would be to make Huffman coding decoding faster, allowing us to descrease k so that more symbols would be encoded with Huffman. One way to improve Huffman coding decoding could be to build a lookup table, where each key size of t bits would tell: *i*) how many Huffman codes this t bits include (h_n); *ii*) the sum s of the included codes (s); *iii*) location t_{end} end of last code. Then in decoding we could read t bits from the Huffman encoding and continue decoding if needed from t_{end}. Lookup table would then cost 2^t bits plus key values $\{h_n, s, t_{end}\}$.

Bibliography

[1] D. Arroyuelo and R. Raman. “Adaptive Succinctness”. English. In: *Algorithmica* 84.3 (2022), pp. 694–718.

[2] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. “Representing trees of higher degree”. In: *Algorithmica* 43.4 (2005), pp. 275–292.

[3] D. K. Blandford and G. E. Blelloch. “Compact Dictionaries for Variable-Length Keys and Data with Applications”. In: *ACM Trans. Algorithms* 4.2 (May 2008). ISSN: 1549-6325. DOI: 10.1145/1361192.1361194. URL: https://doi-org.libproxy.helsinki.fi/10.1145/1361192.1361194.

[4] D. K. Blandford and G. E. Blelloch. “Dictionaries Using Variable-Length Keys and Data, with Applications”. In: SODA ’05. Vancouver, British Columbia: Society for Industrial and Applied Mathematics, 2005, pp. 1–10. ISBN: 0898715857.

[5] A. Brodnik and J. I. Munro. “Membership in constant time and almost-minimum space”. In: *SIAM Journal on computing* 28.5 (1999), pp. 1627–1640.

[6] A. J. Cox, A. Farruggia, T. Gagie, S. J. Puglisi, and J. Sirén. “RLZAP: Relative Lempel-Ziv with Adaptive Pointers”. In: *String Processing and Information Retrieval*. Ed. by S. Inenaga, K. Sadakane, and T. Sakai. Cham: Springer International Publishing, 2016, pp. 1–14. ISBN: 978-3-319-46049-9.

[7] S. Deorowicz and S. Grabowski. “Robust relative compression of genomes with random access”. In: *Bioinformatics* 27.21 (Sept. 2011), pp. 2979–2986. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr505. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/21/2979/16900033/btr505.pdf. URL: https://doi.org/10.1093/bioinformatics/btr505.

[8] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. “Fast relative Lempel–Ziv self-index for similar sequences”. In: *Theoretical computer science.* 532 (May 2014), pp. 14–30. ISSN: 0304-3975.

[9] P. Elias. “Efficient storage and retrieval by content and address of static files”. In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.

[10] R. M. Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[11] T. Gagie, S. J. Puglisi, and D. Valenzuela. “Analyzing Relative Lempel-Ziv Reference Construction”. In: *String Processing and Information Retrieval.* Ed. by S. Inenaga, K. Sadakane, and T. Sakai. Cham: Springer International Publishing, 2016, pp. 160–165. ISBN: 978-3-319-46049-9.

[12] S. Gog, T. Beller, A. Moffat, and M. Petri. “From Theory to Practice: Plug and Play with Succinct Data Structures”. In: *Experimental Algorithms.* Ed. by J. Gudmundsson and J. Katajainen. Cham: Springer International Publishing, 2014, pp. 326–337. ISBN: 978-3-319-07959-2.

[13] S. Golomb. “Run-length encodings (Corresp.)” In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401. DOI: 10.1109/TIT.1966.1053907.

[14] A. Gómez-Brandón. “Bitvectors with Runs and the Successor/Predecessor Problem”. In: *2020 Data Compression Conference (DCC).* 2020, pp. 133–142. DOI: 10.1109/DCC47342.2020.00021.

[15] R. Grossi and J. S. Vitter. “Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching”. In: *SIAM Journal on Computing* 35.2 (2005), pp. 378–407. DOI: 10.1137/S0097539702402354. eprint: https://doi.org/10.1137/S0097539702402354. URL: https://doi.org/10.1137/S0097539702402354.

[16] T. Hagerup and T. Tholey. “Efficient minimal perfect hashing in nearly minimal space”. In: *Annual Symposium on Theoretical Aspects of Computer Science.* Springer. 2001, pp. 317–326.

[17] C. Hoobin, S. J. Puglisi, and J. Zobel. “Relative Lempel-Ziv Factorization for Efficient Storage and Retrieval of Web Collections”. In: *Proc. VLDB Endow.* 5.3 (Nov. 2011), pp. 265–273. ISSN: 2150-8097. DOI: 10.14778/2078331.2078341. URL: https://doi.org/10.14778/2078331.2078341.

[18] D. A. Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[19] G. J. Jacobson. “Succinct static data structures”. PhD thesis. Carnegie Mellon University, 1989.

[20] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. “Hybrid Compression of Bitvectors for the FM-Index”. In: *2014 Data Compression Conference.* 2014, pp. 302–311. DOI: 10.1109/DCC.2014.87.

[21] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. “Lazy Lempel-Ziv Factorization Algorithms”. In: 21 (Oct. 2016). ISSN: 1084-6654. DOI: 10.1145/2699876. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2699876.

[22] L. G. Kraft. “A device for quantizing, grouping, and coding amplitude-modulated pulses”. PhD thesis. Massachusetts Institute of Technology, 1949.

[23] S. Kuruppu, S. J. Puglisi, and J. Zobel. “Optimized Relative Lempel-Ziv Compression of Genomes”. In: ACSC ’11. Perth, Australia: Australian Computer Society, Inc., 2011, pp. 91–98. ISBN: 9781920682934.

[24] S. Kuruppu, S. J. Puglisi, and J. Zobel. “Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval”. In: *String Processing and Information Retrieval*. Ed. by E. Chavez and S. Lonardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 201–206. ISBN: 978-3-642-16321-0.

[25] K. Liao, M. Petri, A. Moffat, and A. Wirth. “Effective Construction of Relative Lempel-Ziv Dictionaries”. In: *Proceedings of the 25th International Conference on World Wide Web*. WWW ’16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 807–816. ISBN: 9781450341431. DOI: 10.1145/2872427.2883042. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2872427.2883042.

[26] B. McMillan. “Two inequalities implied by unique decipherability”. In: *IRE Transactions on Information Theory* 2.4 (1956), pp. 115–116.

[27] A. Moffat and A. Turpin. “On the implementation of minimum redundancy prefix codes”. In: *IEEE Transactions on Communications* 45.10 (1997), pp. 1200–1207. DOI: 10.1109/26.634683.

[28] A. Moffat. “Huffman coding”. In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–35.

[29] A. Moffat and J. Katajainen. “In-place calculation of minimum-redundancy codes”. In: *Workshop on Algorithms and Data Structures*. Springer. 1995, pp. 393–402.

[30] A. Moffat and L. Stuiver. “Binary interpolative coding for effective index compression”. In: *Information Retrieval* 3.1 (2000), pp. 25–47.

[31] J. I. Munro, V. Raman, and S. S. Rao. “Space efficient suffix trees”. In: *Journal of Algorithms* 39.2 (2001), pp. 205–222.

[32] J. I. Munro and V. Raman. “Succinct Representation of Balanced Parentheses and Static Trees”. In: *SIAM Journal on Computing* 31.3 (2001), pp. 762–776. DOI: 10.1137/S0097539799364092. eprint: https://doi.org/10.1137/S0097539799364092. URL: https://doi.org/10.1137/S0097539799364092.

[33] G. Navarro. *Compact data structures : a practical approach*. eng. New York: University of Cambridge, 2016. ISBN: 978-1-107-15238-0.

[34] G. Navarro and V. Sepúlveda. “Practical Indexing of Repetitive Collections Using Relative Lempel-Ziv”. In: *2019 Data Compression Conference (DCC)*. 2019, pp. 201–210. DOI: 10.1109/DCC.2019.00028.

[35] D. Okanohara and K. Sadakane. “Practical entropy-compressed rank/select dictionary”. In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 60–70.

[36] G. Ottaviano and R. Venturini. “Partitioned Elias-Fano Indexes”. In: *Proceedings of the 37th International ACM SIGIR Conference on Research &amp; Development in Information Retrieval*. SIGIR ’14. Gold Coast, Queensland, Australia: Association for Computing Machinery, 2014, pp. 273–282. ISBN: 9781450322577. DOI: 10.1145/2600428.2609615. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2600428.2609615.

[37] R. Pagh. “Low redundancy in static dictionaries with constant query time”. In: *SIAM Journal on Computing* 31.2 (2001), pp. 353–363.

[38] R. Raman, V. Raman, and S. R. Satti. “Succinct Indexable Dictionaries with Applications to Encoding K-Ary Trees, Prefix Sums and Multisets”. In: *ACM Trans. Algorithms* 3.4 (Nov. 2007), 43–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290680. URL: https://doi-org.libproxy.helsinki.fi/10.1145/1290672.1290680.

[39] J. P. Schmidt and A. Siegel. “The spatial complexity of oblivious k-probe hash functions”. In: *SIAM Journal on Computing* 19.5 (1990), pp. 775–786.

[40] E. S. Schwartz and B. Kallick. “Generating a canonical prefix encoding”. In: *Communications of the ACM* 7.3 (1964), pp. 166–169.

[41] C. E. Shannon. “A mathematical theory of communication”. In: 27 (1948), pp. 398–403.

[42] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. "SIMD-Based Decoding of Posting Lists". In: CIKM '11. Glasgow, Scotland, UK: Association for Computing Machinery, 2011, pp. 317–326. ISBN: 9781450307178. DOI: 10.1145/2063576.2063627. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2063576.2063627.

[43] B. P. Tunstall. "Synthesis of noiseless compression codes". PhD thesis. Georgia Institute of Technology, 1967.

[44] H. Yan, S. Ding, and T. Suel. "Inverted Index Compression and Query Processing with Optimized Document Ordering". In: *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. Madrid, Spain: Association for Computing Machinery, 2009, pp. 401–410. ISBN: 9781605584874. DOI: 10.1145/1526709.1526764.

[45] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

Appendix A Efficient way to build Huffman coding

In the previous section 2.4 we presented a simple way to encode/decode sequence with Huffman coding. In this section we introduce [27, 29] a fast way to construct Huffman coding, and how to encode/decode sequence of symbols fast. To build Huffman coding efficiently, explicit tree structure is not used. Moffat and Katajainen [29] showed a in place way to compute codeword lengths in linear $\mathcal{O}(\sigma)$ time if the alphabet and the model Pr are sorted order. If the model and alphabet are not sorted in order, those can be sorted efficiently for example with a quicksort, so linear time changes to $\mathcal{O}(\sigma \log \sigma)$.

In this Section we assume that source and probability/weight model are in sorted order. Figure A.1 illustrates this with the same source and weights as in Figure 2.5.

s_i	1	2	7	22	90	131	304	501
w_i	1	2	4	4	2	1	1	9

(a) Input source with weights.

r_i	0	1	2	3	4	5	6	7
s_i	501	7	22	2	90	1	131	304

(b) Source mapping after sort.

Figure A.1: Example of sorting source model by weights and creating new mapping for the source. Input alphabet for the sorting and mapping is $\Sigma = \{1, 2, 7, 22, 90, 131, 304, 501\}$ with weights $W = \{1, 2, 4, 4, 2, 1, 1, 9\}$. Figure A.1(a) presents the input source and its weights, where row s_i shows the source and row w_i weights for each symbol. Figure A.1(b) shows a new source on row r_i after sorting and mapping Σ and W. In row s_i are shown the original symbols for each new source symbol in r_i.

Algorithm 9 from [28] shows Moffat and Katajainen idea to compute Huffman codeword lengths. The algorithm consists of three phases. The first phase computes weight values of internal nodes and root of Huffman tree. After the first phase $W[1]$ contains sum of the initial weight values. If W initially consist of values from the model Pr, then $W[1] = 1$. Other $\sigma - 2$ internal nodes $W[i]$ have an offset value from its parent node. Second phase traverses array W from the root and computes each internal node depth. Lastly in the final phase, the array W is looped a third time, where internal node depths are converted as depths of leaf nodes. Figure A.2 shows array W at end of each phase to describe how codeword lengths are computed.

With codeword lengths, codeword values are easy to assign to non-decreasing order [40]. Let L be maximum and L_{min} minimum value in $W = \langle l_1, \ldots, l_\sigma \rangle$ after algorithm 9. Lets assign $\mathcal{C}_L(1) = 0$ for shortest symbol 1, then symbol $i+1$ can be computed as $\mathcal{C}_L(i+1) = \mathcal{C}_L(i) + 2^{L-l_i}$, where $\mathcal{C}_L(i)$ is L–bit codeword for symbol i. Then canonical Huffman code

Algorithm 9: COMPHUFFLEN, Moffat [28] implementation on [29] to compute Huffman codeword lenths

```
Input: Model array W, alphabet size $\sigma$
Output: Huffman code lengths in array W
1  Function CompHuffLen(W, $\sigma$):
       // Phase one
2      leaf $\leftarrow \sigma - 1$, root $\leftarrow \sigma - 1$
3      for next $\leftarrow \sigma - 1$ downto 1 do
           // use internal node
4          if leaf $< 0$ or (root $>$ next and W[root] $<$ W[leaf]) then W[next] $\leftarrow$ W[root]
           // use leaf node
5          else W[next] $\leftarrow$ W[leaf], leaf $\leftarrow$ leaf $- 1$
           // use internal node
6          if leaf $< 0$ or (root $>$ next and W[root] $<$ W[leaf]) then
7              W[next] $\leftarrow$ W[next] $+$ W[root]
           // use leaf node
8          else W[leaf] $\leftarrow$ W[leaf] $+$ W[leaf], leaf $\leftarrow$ leaf $- 1$
9      W[1] $\leftarrow 0$
       // Phase two
10     for next $\leftarrow 2$ to $n - 1$ do W[next] $\leftarrow$ W[W[next]] $+ 1$
       // Phase three
11     $avail \leftarrow 1$, $used \leftarrow 0$, $depth \leftarrow 0$, root $\leftarrow 1$, next $\leftarrow 0$
12     while avail $> 0$ do
           // count internal nodes used at depth depth
13         while root $< n$ and W[root] $=$ depth do
14             $used \leftarrow used + 1$, root $\leftarrow$ root $+ 1$
           // assign as leaves any nodes that are not internal
15         while avail $>$ used do
16             W[next] $\leftarrow depth$, next $\leftarrow$ next $+ 1$, $avail \leftarrow avail - 1$
           // move to next depth
17         $avail \leftarrow 2 \cdot used$, $depth \leftarrow depth + 1$, $used \leftarrow 0$
18     return W, where W[$i$] now contains the length $l_i$ of the $i$th codeword
```

	W							
i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
phase 1:	9	24	1	2	2	3	4	5
phase 2:	9	0	1	2	2	3	3	4
phase 3:	1	3	3	4	4	4	5	5

Figure A.2: Example values for array W after each phase in Algorithm 9, where $W = \{9, 4, 4, 2, 2, 1, 1, 1\}$ is W init values.

for symbol i is the l_i most significant bits in $\mathcal{C}_L(i)$ $(\mathcal{C}(i) = \mathcal{C}_L(i) \gg (L - l_i))$. Figure A.3 illustrates this idea.

i	w_i	l_i	$\mathcal{C}(i)$	l_i-bit integer	$\mathcal{C}_L(i)$	L-bit integer
0	9	1	0	0	00000	0
1	4	3	100	4	10000	16
2	4	3	101	5	10100	20
3	2	4	1100	12	11000	24
4	2	4	1101	13	11010	26
5	1	4	1110	14	11100	28
6	1	5	11110	30	11110	30
7	1	5	11111	31	11111	31
8	-	-	-	32	100000	32

Figure A.3: Example of canonical Huffman codes for input source $\Sigma = \{1, 2, 7, 22, 90, 131, 304, 501\}$ with weights $W = \{1, 2, 4, 4, 2, 1, 1, 9\}$. Lengths of each code are computed using Algorithm 9. Column $\mathcal{C}(i)$ shows final codes for each symbol i, where i is a symbol from new source created from Σ as in Figure A.1.

Let assume that all values $i \in [1, \sigma]$ in $\mathcal{C}(i)$ and $\mathcal{C}_L(i)$ are stored into arrays *l-bit* and *L-bit*. Moffat and Turpin [27] showed that only the first elements (minimum code) for each lengths in arrays *l-bit* and *L-bit* are needed to encode/decode symbols in Σ. This can be done because all columns are in increasing order, so the arrays have known properties. The arrays can have only L items stored, so these arrays are cheap to store because $L \ll \sigma$. Let store these first values into arrays *first_symbol*, *first_code_r* and *first_code_l*, where *first_symbol* has symbol values of each first item in length j, *first_code_r* has first values from array *l-bit* and *first_code_l* from array *L-bit*. Figure A.4 shows an example using values from figure A.3.

Algorithm 10 shows how to encode each symbol $s \in \Sigma$ with arrays *first_symbol* and

l	*first_symbol*$[l]$	*first_code_r*$[l]$	*first_code_l*$[l]$
0	0	0	0
1	0	0	0
2	1	2	16
3	1	4	16
4	3	12	24
5	6	30	30

Figure A.4: Example of *first_symbol*, *first_code_r* and *first_code_l* values constructed from codewords in Figure A.3

first_code_r. To encode symbols s, an array to determine codeword lengths is needed. The algorithms use in this purpose array *code_len*. If space is concern in encoding, array *code_len* uses σ codewords. Then the length l of symbol s could be linear or binary searched from array *first_code_r*, but usually symbol s is mapped from original source Σ', where mapping uses σ words of space. Then the space safe if not magnitude and speed of decoding suffers.

Algorithm 10: CANONICAL–ENCODING

```
Input: symbol s ∈ Σ
Output: minimum prefix code of symbol s
1 Function canonical-encoding(s):
2     $l$ ← code_len[s]
3     offset ← s − first_symbol[$l$]
4     return (first_code_r[$l$] + offset, $l$)
```

For decoding only arrays *fisrt_symbol* and *first_code_l* are needed. Process of decoding is the same kind as encoding. Let *buffer* by next L-bits from the encoded input stream. First we need to determine the length l of the next codeword. This cannot be found from table because it would need at least array of 2^L entries. So we compute length l by linear searching from *first_code_l* until *buffer* $<$ *first_code_l*$[l+1]$. Then we need only to find the *offset* from *first_symbol*$[l]$ and then decoded symbol is *first_symbol*$[l]$ + *offset*. Algorithm 11 shows a detailed version of the decoding symbol s.

Bottleneck at decoding symbol in CANONICAL-DECODING is to find length of the next codeword in the *buffer*. This could be speed up with a direct lookup table size of 2^L. Size

Algorithm 11: CANONICAL–DECODING

```
Input: buffer of $L$-bits
Output: decoded symbol s
1 Function canonical-decoding(buffer):
2     $l \leftarrow L_{min}$
3     while first_code_l[$l$] $\leq$ buffer $<$ first_code_l[$l+1$] do $l \leftarrow l+1$
4     offset $\leftarrow$ (buffer $-$ first_code_l[$l$]) $\gg (L-l)$
5     return first_symbol[$l$] + offset
```

v	$search\_start_2[v]$	$search\_start_3[v]$
0	1	1
1	1	1
2	3	1
3	4^*	1
4		3
5		3
6		4
7		4^*

Figure A.5: Two examples for $search\_start_t$ lookup table ($t = 2$ and $t = 3$) to speed up CANONICAL–DECODING . Values are computed using codewords in Figure A.3.

2^L could be much larger than σ so it could dominate the total size. Moffat and Turpin [27] showed that linear search and table lookup for finding l can be merged with small extra space. In their idea, table *search_start* would be constructed with 2^t entries, where $L_{min} \leq t \leq L$. Each value in *search_start* would tell the shortest length value l for the prefix of t bits in the buffer. So *search_start* table could not possibly store all l values for *buffer*, but could decrease the amount of time spended in the loop. Then we could change line 2 in CANONICAL-DECODING following:

$$\textbf{2:}\ l \leftarrow search\_start[v]$$

Figure A.5 continues our example and shows an example of *search_start* table with values $t = 2$ and $t = 3$.

So in total decoding sequence $S \in \Sigma^n$ would need at least

$$n(\mathcal{H}_0(S) + 1) + 2(L+1)w + 2^t \cdot 8 + \sigma\lceil \log \sigma \rceil$$

bits, where w is wordsize. If $L \leq 32$ then w could be 32 otherwise 64 or larger but usually $L \leq 32$. We needed to also add $\sigma\lceil\log\sigma\rceil$ bits because typically $Pr(S)$ and Σ need to be sorted and mapping for canonical codes need to be available in encoding and decoding.