EPIIC: A NOVEL ENCODING PLUGGABLE LOSSLESS DATA COMPRESSION ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TAYLAN İSMAİL DOĞAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2018

Approval of the thesis:

EPIIC: A NOVEL ENCODING PLUGGABLE LOSSLESS DATA COMPRESSION ALGORITHM

submitted by **Taylan İsmail Doğan** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Yusuf Sahillioğlu
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering Dept., METU

Assoc. Prof. Dr. Yusuf Sahillioğlu
Computer Engineering Dept., METU

Assist. Prof. Dr. Mehmet Tan
Computer Engineering Dept.,
TOBB University of Economics and Technology

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name:	Taylan İsmail Doğan
Signature :	

ABSTRACT

EPIIC: A NOVEL ENCODING PLUGGABLE LOSSLESS DATA COMPRESSION ALGORITHM

Doğan, Taylan İsmail
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Yusuf Sahillioğlu

July 2018, 86 pages

Encoding pluggable inverted index compression (EPIIC) is a novel lossless data compression algorithm that applies a pipeline of conventional compression techniques on files that are transformed into a structure similar to inverted indexes. What makes this study novel is the idea of compressing the positions or indexes of hexadecimals that make up a file, instead of focusing on compressing the original file. By leveraging the inverted index structure underneath, we have been able to avoid storing the positional data of the most frequent hexadecimal in a file. Moreover, a slightly different variation of run length encoding is used to make the data even more compressible. As a result, it is observed that this new notion of compression performs on a par with widely known algorithms like LZMA and bzip2, especially when used on text and XML files.

Keywords: Data compression, Lossless, Inverted index, Burrows wheeler transform, Encoding pluggable

ÖZ

EPIIC: KODLAMA EKLENEBİLİR YENİ BİR KAYIPSIZ DATA SIKIŞTIRMA ALGORİTMASI

Doğan, Taylan İsmail
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Yusuf Sahillioğlu

Temmuz 2018 , 86 sayfa

Kodlama eklenebilir ters indeks sıkıştırması (EPIIC), ters indeks benzeri bir yapıya dönüştürülmüş dosyalar üstünde alışılagelmiş sıkıştırma tekniklerini uygulayan yeni bir kayıpsız veri sıkıştırma algoritmasıdır. Bu çalışmayı orijinal yapan, asıl dosyayı sıkıştırmaya odaklanmak yerine dosyayı oluşturan heksadesimallerin pozisyonlarını ya da indekslerini sıkıştırmaktır. Ters indeks yapısından faydalanarak dosyada en sık görülen heksadesimalin pozisyonel verisini saklamaktan kaçınıldı. Dahası, kısmen farklı bir run length encoding kullanarak veri daha da sıkıştırılabilir bir hale getirildi. Sonuç olarak, bu yeni sıkıştırma fikrinin yaygın bilinen LZMA ve bzip2 gibi algoritmalar ile özellikle metin dosyalarında ve XML uzantılı dosyalarda benzer performans sergilediği gözlendi.

Anahtar Kelimeler: Veri sıkıştırma, Kayıpsız, Ters indeks, Burrows wheeler dönüşümü, Kodlama eklenebilir

I am dedicating this thesis to my parents, who were always there supporting me through everything, to my brother Kaan whom I love so much and to my adorable friends who kept believing in me..

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisors Prof. Dr. Nihan Kesim Çiçekli and Assoc. Prof. Dr. Yusuf Sahillioğlu for their continuous guidance, support and encouragement. Not only did they appreciate and embrace my ideas, but they also believed in me and my skills from the beginning to the end.

I would like to extend my gratitude to my dear friends for providing the emotional support, constant motivation, and great suggestions throughout my study.

TABLE OF CONTENTS

CHAPTERS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

XML	Extensible Markup Language
KL Divergence	Kullback–Leibler Divergence
CERN	Conseil Europèen pour la Recherche Nuclèaire
IR	Information Retrieval
BWT	Burrows–Wheeler Transform
RLE	Run-Length Encoding
EPIIC	Encoding Pluggable Inverted Index Compression
DNA	Deoxyribonucleic acid
JPEG	Joint Photographic Experts Group
MPEG	Moving Picture Experts Group
AVC	Advanced Video Coding
CPU	Central Processing Unit
MTF	Move to Front Transform
LZ	Lempel–Ziv
LZW	Lempel–Ziv–Welch
LZMA	Lempel–Ziv–Markov Chain Algorithm
LZHAM	LZ, Huffman, Arithmetic, Markov
ZSTD	Z-Standard
UNIX	Uniplexed Information and Computing Service
PNG	Portable Network Graphics
EOF	End of File
ANS	Asymmetric Numeral Systems
BASC	Binary Adaptive Sequential Encoding
VB	Variable Byte

PPM	Prediction by Partial Matching
S9	Simple9
PFOR	Patched Frame of Reference
FOR	Frame of Reference
SIMD	Single Instruction Multiple Data
IPC	Binary Interpolative Coding
RBUC	Recursive Bottom Up Coding
HTML	Hypertext Markup Language
TLV	Type-Length-Value
TLVG	Type-Length-Value-Guard
ERLE	Encoded RLE Length
EILV	Encoded Inverted List Vector Length
TCS	Total Compressed Size of Encoded Inverted Lists
KS	Kärkkäinen and Sanders
RAM	Random Access Memory
API	Application Programming Interface
LZO	Lempel–Ziv–Oberhumer
SSE	Streaming SIMD Extensions
PIE	Positional Information Encoder

CHAPTER 1

INTRODUCTION

Data compression is a problem that is nearly as old as the modern computer science discipline itself. The fundamentals of it are deeply rooted in Shannon's source coding theorem [1] in which Claude Shannon formalizes this decades-old problem and establishes a limit on compression of an information source, the Shannon entropy. In his studies, Shannon measured the amount of average information required to identify random samples from a given discrete probability distribution. Later, this concept is extended to continuous probability distributions with differential entropy. Yet, due to the shortcomings of differential entropy, relative entropy, also known as KL divergence, has followed. Ultimately these advancements led to quantum information theory, which is still in its infancy.

The compression algorithms, also known as encodings, are actually a series of transformations that are applied to some data such that the output of the algorithm uses fewer bits than the original. They are commonly divided into two major categories, namely lossless and lossy compression. Lossy compressions are irreversible transformations due to the information loss during the process. On the other hand, as the name suggests, we do not lose any information while using lossless encodings. Hence, the lossless compression algorithms basically represent the same information using less bits. The concepts of lossy and lossless compression are discussed verbosely in Chapter 2.2.

With the emergence of the Internet and increase in the use of computers in households and Internet of Things applications, we, the earthlings, use and produce much more data than ever. Thus, considering the bandwidth and storage constraints, the demand for efficient compression algorithms is strongly present. Additionally, in recent years, big data technologies and applications have become a standard for businesses since these methods offer a competitive advantage to companies. Also, research centers such as Conseil Europèen pour la Recherche Nuclèaire (CERN) generate huge amounts of data everyday. Thus, data storage is definitely an important problem and the problem is faced in many different applications.

The compression techniques that are already available may be specifically tailored for some kind of data like genomic, textual data or they may be general purpose algorithms. However, what remains the same is a trade-off between compression speed and compression ratio. Lately, apparently decompression speed has also become an important criteria when comparing compression algorithms due to the rightful notion of "compress once, decompress many times", since many files like fonts, game assets, Linux distribution files, etc. fit into this category. Our solution is also a general purpose algorithm and it uses already available compression techniques. What makes our approach novel is that we use an inverted index and try to compress the positional data of hexadecimals that make up the file.

Inverted indices are essential data structures for information retrieval (IR) systems like search engines. They keep information related to the *terms* in a document collection in a simple and structured way. These *terms* may be words or phrases like *"Middle East Technical University"* depending on the application. As described by Yan et al. 2009 [2]; an *inverted index I* is actually a collection of a set of *inverted lists* $I_{t_0}, I_{t_1}, .., I_{t_{m-1}}$ where I_t contains a *posting* for each document containing the term *t*. Each *posting* is composed of document id, term frequency and the position of occurrences of the term within the document. Thus, they can be denoted as $(d_i, f_i, p_{i,0}, p_{i,1}, .., p_{i,f_i-1})$ [2], where $p_{i,j}$ = *k* if *t* is the *k*-th word in document d_i. Since response times of IR systems are crucial, it is preferable to keep the inverted index compressed especially due to the size of their posting lists. Compressing these inverted indices is an active research area, and the methods and algorithms that are employed in this regard are explained at length in Section 3.2.

In this study, we present a new lossless data compression algorithm with the aims of compressing the inverted index that keeps the position information of hexadecimals. In order to get high compression ratios, the common data transforms Burrows-

Wheeler (BWT) [3] and Run Length Encoding (RLE) [4] are applied in the preprocessing step, since these transforms make the data more compressible. Then, after building an inverted index we proceed with employing the already available compression algorithms, namely Delta encoding and Binary Adaptive Sequential Coding (BASC) [5]. As a result, we have observed that our technique is competent with commonly used lossless data compression algorithms and tools like gzip, 7zip and bzip2.

Our algorithm, Encoding Pluggable Inverted Index Compression (EPIIC) introduces a couple of novel techniques to the field of lossless data compression. Most importantly, to the best of our knowledge, there are no prior research that attempts to encode the positional information of hexadecimals of a file. Hence, this notion of ***positional information encoding*** is indeed novel. Another novelty of the algorithm is that it is possible to use different integer encoding algorithms with minimal effort, which makes EPIIC *"encoding pluggable"*. That is, after converting the input file into an inverted index, we are left with positional data which can be compressed by any integer encoding algorithm. Due to this convenience, EPIIC can leverage the progress in integer encoders and increase its effectiveness just by replacing the integer encoding algorithm.

Another approach that can be considered new is that BWT is applied on hexadecimals that comprises the file directly, deforming the actual data. BWT or block sorting algorithm is widely known and it is typically used on characters that make up the data. However, in our case, the input file is read as a sequence of hexadecimals and BWT is immediately applied on this sequence. That is, given a hexadecimal sequence such as *"48 65 6c 6c 6f 20 57 6f 72 6c 64 21 $"* (The spaces between hexadecimals are added for readability, and *'$'* denotes the special character used in BWT.), BWT converts it into *"12 2f 47 6 $ 6 0c 82 5c c7 f5 46 66 66"*. Note that, the initial hexadecimal sequence is completely altered and we actually disrupted the characters of the input file. This relatively new approach that employs BWT is explained in Section 4.2.1.

Also, a new variation of RLE is introduced in Section 4.2.2. In this alternative version of RLE, counts of the recurrences of characters and the remainder of the actual data are basically encoded separately. Although RLE is an extremely common technique used in lossless and lossy compression algorithms, to the best of our knowledge, such an approach is unprecedented.

Lastly, after building the inverted index, we replace all of the positional data concerning the most frequent hexadecimal that occurs in the file with a single byte. We refer to this method as *fill value trick* and it allows us to gain huge savings. We explain how the inverted data structure and the notion of positional information encoding enable us to perform such an action in Section 4.2.4.

Even though EPIIC is simply an application of well-known compression techniques and data structures, we believe that the novel contributions introduced in this study may come in handy in other studies as well. Furthermore, as stated before, we suggest a new paradigm of ***positional information encoding*** which may prove itself in some use cases in the future.

CHAPTER 2

A DATA COMPRESSION PRIMER

In this chapter, we introduce and explore the foundations of data compression. After a brief explanation of data compression and encodings, we elaborate on entropy and lossy/lossless compression and then we examine the conventional, established compression algorithms.

The most basic description of data compression may be given as follows: Eliminating the redundancy in the data. From an information theoretical point of view, it may be defined as a process that increases the entropy of given data. Redundancies exist in many different forms and types like recurring characters or complex patterns in data. Compression algorithms simply try to capture these redundancies and replace them with a succinct and condensed representation of data, or remove them altogether in case of lossy compression algorithms.

Data compression is also referred to as coding or encoding, which is nothing but a specialized format of data. American Standard Code for Information Interchange, ASCII is a well known encoding that represents texts in computers. Also, braille and Morse codes can be given as different examples of encodings. Encodings can also be interpreted as mappings from source alphabets to code alphabets. For example, English alphabet, that is comprised of 26 letters from *'a'* to *'z'*, is mapped to *97* to *122* in ASCII. Thus, codes or encodings are basically a set of correspondences between source messages and *codewords* (words of the code alphabet).

A code is said to be *distinct*, if the mapping between source messages and codewords is one-to-one. Another property besides *distinctness* is *uniquely decodability*. To be able to encode and decode an information source without any ambiguities, a code is required to be both *distinct* and *uniquely decodable*.

Suppose that we want to encode a given DNA sequence, *S* = *ACTGG*. Our source alphabet is *L* = *{A, C, T, G}* and we have a code *C*, a mapping from *L* to a code alphabet *D* = *{0, 1}*. And the mappings that comprises the code, *C* are as follows:

- *C(A) = 1*
- *C(C) = 11*
- *C(T) = 0*
- *C(G) = 01*

When the DNA sequence, *S* is encoded with *C*, the output is *11100101*. However, as we attempt to decode this encoded sequence, we realize that it is decodable in a couple of different ways. That is, *11100101* can be decoded into one of the following sequences: *AAATTAG*, *ACTTAG*, *ACTGG*, *ACTTAG*, *AAATGG*. As a result, we have no way of deducing the original sequence, since all of these decodings are valid according to our code, *C*. Even though our code is distinct, it is not uniquely decodable.

A simple way to ensure that a code is uniquely decodable, is the *prefix property*. Prefix property dictates that no codeword is a prefix of any other codeword. Hence, prefix property is actually a requirement and the codes that fulfill it are uniquely decodable. Consider that in the given example, encoding of *A* is *1*, which is a prefix of the encoding of *C*, *11*.

A uniquely decodable code, *C'* that has the prefix property, is as follows:

- *C(A) = 00*
- *C(C) = 01*
- *C(T) = 10*
- *C(G) = 11*

Our DNA sequence, *S* becomes *0001101111* when encoded with *C'*, and it is uniquely decodable. Nevertheless, it is possible to create uniquely decodable codes even if they do not have the prefix property. However, for the sake of clarity, we deem a further discussion of codes irrelevant and we proceed with the explanation of the concept of entropy.

2.1 Entropy

Entropy is rather a complicated, intricate term that may bear different meanings depending on the context. While entropy is a completely distinct property in thermodynamics and quantum mechanics, it is also used as a measure of order/disorder of a system, or irreversibility. In this paper, the term entropy is obviously used in an information theoretical context, defined by Claude Shannon in his prominent study, *A Mathematical Theory of Communication* [1]. With his studies, Shannon not only established a subfield of computer science, but also unveiled a technique to precisely quantify information.

Information theoretical data compression algorithms are concerned with entropy encoders. Since the goal of compression algorithms is to represent the given data with less bits, entropy encoders try to achieve this by mapping the most frequent characters in source message to the shortest codeword possible. For example, in Morse code letter *'e'* is mapped to a single dot, since *'e'* is the most occurring character in English. Because it is more likely to encounter with an *'e'*, it should be encoded with least possible number of bits. At this point, we are concerned with likelihoods of characters. Thus, entropy is not a measure that is only limited to recurrences of characters. We can extend it to actual events, given their probability distributions.

Entropy of a set of possible events whose probabilities of occurrence are $p_1, p_2, .., p_n$, is defined as follows:

$$H = -K \sum_{i=1}^{n} p_i \log p_i \tag{2.1}$$

In Equation 2.1, entropy is denoted by H and K is just a positive coefficient that determines the unit of measure and it is often neglected. H is a measure of the amount of information generated by the source per symbol. When the logarithmic base is 2, entropies are said to represent bits per symbol.

Assume that, the probability distributions of DNA bases in human genome and cat genome are given as follows:

- $p_{human}(A)$ = *0.3*
- $p_{human}(C)$ = *0.2*
- $p_{human}(T)$ = *0.3*
- $p_{human}(G)$ = *0.2*
- $p_{cat}(A)$ = 0.1
- $p_{cat}(C)$ = 0.4
- $p_{cat}(T)$ = 0.1
- $p_{cat}(G)$ = 0.4

The respective entropies of genomes can be calculated by employing Equation 2.1. Thus, entropy of the genomes are given by:

$$\begin{aligned} H_{human} &= -\sum_{i=1}^{4} p_{i(human)} \log p_{i(human)} \\ &= -(0.3\log(0.3) + 0.2\log(0.2) + 0.3\log(0.3) + 0.2\log(0.2)) \\ &= -(-0.521 - 0.464 - 0.521 - 0.464) \\ &= 1.97\, bits/symbol \end{aligned}$$

$$\begin{aligned} H_{cat} &= -\sum_{i=1}^{4} p_{i(cat)} \log p_{i(cat)} \\ &= -(0.1\log(0.1) + 0.4\log(0.4) + 0.1\log(0.1) + 0.4\log(0.4)) \\ &= -(-0.332 - 0.528 - 0.332 - 0.528) \\ &= 1.72\, bits/symbol \end{aligned}$$

That is, in order to encode a given human / cat DNA sequence properly, at least 1.97 bits / 1.72 bits per symbol are required respectively. Since human genome has a higher entropy according to the given probability distributions, it is considered richer in information. In other words, the degree of uncertainty is higher and we expect more surprises while inspecting the human genome. On the other hand, we are much more sure about the cat genome since it mostly comprised of *C* and *G* bases.

By expanding this intuition it is trivial to figure out the maximum and minimum entropy in this example. Maximum entropy occurs when all signals or symbols are equally likely. In genomic sequences, the maximum entropy is achieved in the case where the likelihood of each DNA base is equal. That is, each base is equally likely to be observed with a probability of 0.25.

$$\begin{aligned}
H_{maximum} &= -\sum_{i=1}^{4} p_i \log p_i \\
&= -(0.25\log(0.25) + 0.25\log(0.25) + 0.25\log(0.25) + 0.25\log(0.25)) \\
&= -(-0.5 - 0.5 - 0.5 - 0.5) \\
&= 2\ bits/symbol
\end{aligned}$$

We can formalize maximum entropy as follows:

$$\begin{aligned}
H_{maximum} &= -\sum_{i=1}^{n} p_i \log p_i \\
&= -(\frac{1}{n}\log\frac{1}{n} + \frac{1}{n}\log\frac{1}{n} + \cdots + \frac{1}{n}\log\frac{1}{n}) \\
&= -(n \times \frac{1}{n}\log\frac{1}{n}) \\
&= \log n\ bits/symbol
\end{aligned}$$

Note that, maximum entropy depends on the number of all possible outcomes of an event. On the other hand, minimum entropy is reached when a DNA sequence consists of just a single base. Although such a case is impossible in real life, we consider its entropy for the sake of the example. Minimum entropy occurs when a symbol is certain with a probability of 1 and other symbols are impossible. In such cases, there is no uncertainty and there are no surprises. And the entropy is simply zero. Hence, information theory and entropy provides us a lower and an upper bound of achievable bits per symbol for events or the messages generated from a source. Given *n*, the number of all possible outcomes of an event, the range of attainable entropies for that event is bounded by:

$$0 \leq entropy \leq \log n$$

Entropy is also considered a measure of complexity. Pincus suggested approximate entropy as a measure of system complexity [6]. Bandt and Pompe introduced permutation entropy to quantify the complexity of time series in their study [7]. There are many different complexity measures such as Kolmogorov complexity [8], however these measures are out of the scope of this study and they are not discussed.

2.2 Lossless vs. Lossy Compression

Compression algorithms are essentially divided into two categories, namely lossless and lossy compression. We do not lose any information during the execution of lossless data compression algorithms, which is not the case in lossy algorithms. While it is affordable to get rid of some information in some data formats, other data types like texts simply render lossy compression techniques useless since we cannot tolerate any loss of information in such cases.

Lossy compression algorithms are typically used to compress images, audio or video data. The very reason why lossy compression is applicable in these areas is basically the imperfections of human perception. A lossless sound and a compact, compressed version of the very same sound are mostly indistinguishable to our ears. Due to this, it is feasible to lose some information in the case of images, sounds and videos, in order to achieve a higher compression ratio.

JPEG [9] is a common lossy image compression algorithm that provides different levels of image quality and compression ratio by utilizing discrete cosine transform. H.264/AVC [10] for video compression and MPEG-1 [11] (commonly referred to as MP3) for audio compression may be given as examples of other well-known lossy encodings. While it is a matter of quality versus compressed size in lossy algorithms, this trade-off presents itself in lossless algorithms as compression ratio versus CPU/memory utilization.

On the other hand, lossless compression algorithms allow reconstruction of the original data that is subject to encoding. By contrast, lossy algorithms permit us to only obtain an approximation of the original input data. Due to the pigeonhole principle, no lossless data compressor can efficiently compress all kinds of data. Hence, it is common to see that lossless encodings are generally tailored for some data type specifically.

Because a lossless data compression algorithm is suggested in this study, some of the fundamental lossless compression algorithms are explained in Section 2.3.

2.3 Fundamental Compression Algorithms

2.3.1 Run Length Encoding

Run length encoding or RLE [4] can be argued to be the most common lossless encoding. RLE accompanies many compression algorithms in both preprocessing and late stages of the algorithm, and it is rarely used by itself. Simply put, it represents repeated values as a count and a symbol. That is, the sequence *'CCCAAAAAAAHKWWW'* becomes *'C3A7H1K1W3'*. Although it is a trivial technique, it removes a considerable amount of redundancy in a very effortless way.

Note that, for non-recurring symbols in the sequence, RLE elongates the input sequence instead of shrinking it. Nevertheless, this is the default version of RLE and there are many versions of the algorithm, possessing distinct strategies.

We also introduce a novel version of RLE in this thesis. In the proposed version of the algorithm, we basically consider the recurrences of symbols as runs when a symbol is at least repeated *twice*. While we iterate over the given data sequence, we remove the excess recurrences of symbols and keep the counts in a separate place. In the end, we are left with separate counts of runs and the remaining data sequence as different bundles of data. Later, these two bundles may be encoded using different lossless compression algorithms. This new proposed variant of RLE is described in detail in Section 4.2.2.

2.3.2 Move-To-Front

Introduced by Bentley et al. [12], Move-To-Front (MTF) is a locally adaptive encoding or transformation that generally improves the performance of other compression algorithms. Due to this fact, it is commonly employed as a processing step in algorithms such as bzip2. Nevertheless, the performance gains ultimately depend on the data sequence subject to MTF. Furthermore, it is perfectly possible for this transformation to backfire, when used on some type of data source.

MTF simply maintains a symbol list of the alphabet throughout the encoding and decoding. While iterating over the data sequence, it encodes symbols with their position in the symbol list, and then modifies the list by moving the most recent symbol to the front of the list. The very same process is carried out during the decompression phase as well and the original data sequence is recovered.

There are many unofficial versions of the algorithm employed in the wild. For example, a variation does not initialize the symbol list with the alphabet. Instead, it constructs its own list during the encoding. Such a strategy may further improve the performance in terms of both compression ratio and compression/decompression speed, since a manually constructed list may end up shorter than the original alphabet. Although there are many distinct alterations of MTF such as Mohanty and Tripathy's improved version [13], to the best of our knowledge, there is no survey regarding this subject. Hence, it is currently infeasible to compare the variations of this encoding.

Given an alphabet $L = \{c_1, c_2, c_3, c_4, c_5\}$, and a data sequence $\{c_2c_2c_3c_4c_1c_4c_3c_4c_4c_2\}$, both encoding and decoding phases are performed step by step, as follows:

Encoding Phase

Data Sequence	**Symbol List**
$\{c_2c_2c_3c_4c_1c_4c_3c_4c_4c_2\}$	$[c_1, c_2, c_3, c_4, c_5]$
$\{1c_2c_3c_4c_1c_4c_3c_4c_4c_2\}$	$[c_2, c_1, c_3, c_4, c_5]$
$\{10c_3c_4c_1c_4c_3c_4c_4c_2\}$	$[c_2, c_1, c_3, c_4, c_5]$
$\{102c_4c_1c_4c_3c_4c_4c_2\}$	$[c_3, c_2, c_1, c_4, c_5]$
$\{1023c_1c_4c_3c_4c_4c_2\}$	$[c_4, c_3, c_2, c_1, c_5]$
$\{10233c_4c_3c_4c_4c_2\}$	$[c_1, c_4, c_3, c_2, c_5]$
$\{102331c_3c_4c_4c_2\}$	$[c_4, c_1, c_3, c_2, c_5]$
$\{1023312c_4c_4c_2\}$	$[c_3, c_4, c_1, c_2, c_5]$
$\{10233121c_4c_2\}$	$[c_4, c_3, c_1, c_2, c_5]$
$\{102331210c_2\}$	$[c_4, c_3, c_1, c_2, c_5]$
$\{1023312103\}$	$[c_2, c_4, c_3, c_1, c_5]$

Decoding Phase

Data Sequence	**Symbol List**
$\{1023312103\}$	$[c_1, c_2, c_3, c_4, c_5]$
$\{c_2023312103\}$	$[c_2, c_1, c_3, c_4, c_5]$
$\{c_2c_223312103\}$	$[c_2, c_1, c_3, c_4, c_5]$
$\{c_2c_2c_33312103\}$	$[c_3, c_2, c_1, c_4, c_5]$
$\{c_2c_2c_3c_4312103\}$	$[c_4, c_3, c_2, c_1, c_5]$
$\{c_2c_2c_3c_4c_112103\}$	$[c_1, c_4, c_3, c_2, c_5]$
$\{c_2c_2c_3c_4c_1c_42103\}$	$[c_4, c_1, c_3, c_2, c_5]$
$\{c_2c_2c_3c_4c_1c_4c_3103\}$	$[c_3, c_4, c_1, c_2, c_5]$
$\{c_2c_2c_3c_4c_1c_4c_3c_403\}$	$[c_4, c_3, c_1, c_2, c_5]$
$\{c_2c_2c_3c_4c_1c_4c_3c_4c_43\}$	$[c_4, c_3, c_1, c_2, c_5]$
$\{c_2c_2c_3c_4c_1c_4c_3c_4c_4c_2\}$	$[c_2, c_4, c_3, c_1, c_5]$

As illustrated in the example, input and output of the encoding phase are of the same length. Since characters are simply replaced by their indices, MTF does not alter the size of a given input, instead, it reduces the entropy of a given data sequence by exploiting the redundancies locally. Also note that, MTF does not require any extra space except for the alphabet, which can be hard coded into the algorithm.

2.3.3 Huffman Encoding

Proposed by David A. Huffman in 1952, Huffman coding [14] is an entropy encoder that assigns the minimum possible bit representation to a symbol in the input data sequence, by considering their frequency. Given its creation date, it is still among the most widely known, commonly used lossless data compression algorithms. Since it is a prefix code, it provides an unambiguous encoding.

Like other information theoretical encoders, more common symbols are represented with shorter strings of bits and the symbols that occur infrequently are encoded as longer strings. Although, probability distributions of occurrences of characters in particular languages are quite known and accurate, the frequencies of symbols are dissimilar across different messages. Thus, a Huffman encoder is ought to be generated during the encoding for each message, since it may not work on a data sequence as well as the previous case.

Huffman encoding achieves its efficiency by using a binary tree and a priority queue. First, it constructs a priority queue of all of the alphabet characters in descending order of their frequencies. Then it constructs a binary tree from the bottom up such that each character in the alphabet becomes a leaf node. In each iteration, two least occurring symbols are added to the tree and removed from the queue. Since this process is well known, we do not elaborate on this tree construction algorithm. Given a built tree, the input symbol sequence can be compressed. However, to decompress a Huffman encoded data sequence, binary tree constructed during compression is required. Thus, the binary tree, that is basically comprised of mappings between symbols and encodings, has to be saved along with the compressed data sequence. In decompression stage, encoded sequence can be decoded into original data by tracing the tree until you reach a leaf node, which refers to a character.

While being a very influential and essential study, Huffman coding entails a couple of shortcomings. To be able to build a binary tree, probabilities/frequencies of the characters are required. Due to this, Huffman coding needs to go over the original sequence to learn the number of occurrences of symbols to be able to compress the data. Thus, it needs to scan the given data sequence twice during compression. When the input data sequence is large, using a two-pass algorithm may not be ideal, as it significantly slows down the execution. Later, to overcome this problem, following Knuth's study [15], many different adaptive (or also called dynamic) Huffman codings are proposed, which allow a one-pass implementation.

Another issue of Huffman coding is that it needs to store the constructed Huffman tree along with the compressed data, so that it can be decompressed later. However, most of the time the size of the constructed Huffman tree is around a few hundred bytes, which is a negligible sacrifice.

Recently, with the rise of deep learning and artificial intelligence, machine learning models are being used in many different applications. Deeper neural networks are shown to produce better models in terms of accuracy and this leads to larger models as big as few hundred megabytes. With their study, Han et al. [16] proved that achieving high compression ratios is possible by using Huffman coding on deep learning models. Thus, Huffman encoding continues to be a strong candidate among compression algorithms today.

2.3.4 Dictionary Based Algorithms

As famous as Huffman coding, dictionary based algorithms take a completely different approach. While Huffman coding provides us a symbol-oriented, information theoretical compression algorithm that is based on statistical models, dictionary based algorithms or dictionary coders do not encode single symbols. Instead, they try to encode variable length strings. As simple as it sounds, dictionary based algorithms achieve great performance in terms of compression ratio and speed. The basic idea is to identify frequent character patterns and encode them efficiently.

In dictionary coders, we basically have a data structure named *dictionary*, which contains a set of strings. During compression, it seeks matches between the input data sequence and the maintained dictionary data structure. In case of a match, it is replaced by a reference to the position of the string in the dictionary.

The maintained dictionary data structure may be static or dynamic or adaptive depending on the implementation. Obviously, while additions and deletions are not allowed in static dictionaries, dynamic dictionaries hold the strings previously found in the input stream and gets bigger/slower during compression.

Although static dictionaries are more trivial as a concept, their efficiency highly depends on the data. Also, the number of all possible static dictionaries is infinite, and choosing the optimal static dictionary is infeasible. Due to this, as expected, adaptive dictionary coders are much more common in data compression.

The most widely known dictionary based algorithms are the original methods proposed by Lempel and Ziv, **LZ77**[17] and **LZ78**[18], also known as LZ1 and LZ2. Later, Welch improved LZ78 algorithm and suggested another commonly used variate named **LZW** [19]. Then came LZMA, LZ4, LZHAM, Zstd and Brotli [20], distinct improvements to the original algorithms. This gave birth to the LZ-family of lossless data compression algorithms. For the sake of simplicity, only LZ77 and LZ78 are discussed in this section.

In LZ77, repeated occurrences of data is replaced by a reference to a single copy that is encountered earlier. Hence, the dictionary is actually a portion of the previously encoded data sequence. A sliding window that consists two buffers, search buffer and look-ahead buffer, is implemented to compress the data. The goal is to find maximum length matches in the search buffer and look-ahead buffer, by using pointers in both buffers of the window. Whenever a match occurs, we simply refer it to the previous occurrence with a triplet of *offset*, *length* and *codeword*. To spot these matches, the encoder keeps track of some amount of the most recent data. Naturally, as the size of sliding window increases, it is much more probable to find longer matches in the data. However, this ultimately means a higher memory usage during compression. Nevertheless, bigger buffers require faster searches to detect matches. Also, whenever a pattern repeats itself within a period that is larger than the window size, LZ77 simply misses this recurrence and such a case negatively affects the compression ratio of the algorithm. The rationale behind LZ77 is that if there are some patterns that repeat locally in data, a more efficient representation of the data is achievable.

Because of the aforementioned inconveniences of LZ77, its creators Lempel and Ziv suggested an improved version, LZ78. In LZ78, the sliding window approach is simply abandoned. Instead of search and look-ahead buffers, an explicit dictionary structure is used. In this case, dictionary entries are pairs of *indices* and *characters*, where *index* points to a previous entry in the dictionary, and *character* denotes the suffix character that should be appended to the pointed previous entry. For each symbol in the input data sequence, dictionary structure is searched for a match and then updated suitably. Depending on the conditions, the encoder outputs the last matching index and the character to be appended and compression continues. A problem with this approach is that the dictionary structure keeps growing as we encode the input data structure. To solve this issue, the dictionary may be pruned or reseted. Or, after reaching to a size, the dictionary can be switched into a static one.

Even though the algorithm is again improved with LZW implementation proposed by Welch, LZW has its own advantages as well as issues. It is used in UNIX's ***compress*** and *GIF* images.

Another highly acclaimed variate of LZ77 is **DEFLATE**, suggested by Deutsch [21] in 1996, which is employed by the compressor ***gzip*** as well as *PNG* images and the *ZIP* file format.

2.3.5 Arithmetic Coding

Arithmetic coding, introduced by Witten et al. [22], is a special entropy encoder variate that attains better compression ratios compared to Huffman coding and LZ family algorithms. It is predominantly used in lossless compression and provides nearly optimal encoding that is very close to the Shannon limit [22]. Being an entropy encoder, it aims to represent the given input sequence as compact as possible by considering the frequencies of characters.

Unlike other algorithms that replace symbols with codewords, arithmetic coding suggests a radically different approach by mapping whole data sequences into a single real number. That is, independent of the input data sequence length, arithmetic encoding outputs a real number between 0 and 1. To achieve this, it creates an interval for each symbol of the alphabet, based on their probabilities. This interval assignment process continues symbol by symbol until the very end of the input sequence. Initially, the interval for the input data sequence is [0, 1). As the algorithm progresses, interval of the data sequence shrinks. Consider the example given in the original study by Witten et al. [22]:

We have the following alphabet with prior probabilities:

Alphabet

Symbol	**Probability**	**Range**
a	0.2	[0, 0.2)
e	0.3	[0.2, 0.5)
i	0.1	[0.5, 0.6)
o	0.2	[0.6, 0.8)
u	0.1	[0.8, 0.9)
!	0.1	[0.9, 1)

Note that the initial interval [0, 1) is divided into subintervals according to the probabilities of the characters. Given a message, *eaii!*, we proceed with the compression as follows:

- Since the first character is *'e'*, compressor narrows the initial interval down to [0.2, 0.5).
- Next character, *'a'* has a probability of 0.2. Thus, it will shrink our interval into a fifth, which is $(0.5 - 0.2)/5 = 0.06$. Hence, the interval becomes [0.2, 0.26).
- This process goes on and our interval gets smaller in each iteration.

Initially	|	[0,	1)
After seeing	e	[0.2,	0.5)
	a	[0.2,	0.26)
	i	[0.23,	0.236)
	i	[0.233,	0.2336)
	!	[0.23354,	0.2336)

Decoding such a compression technique is unexpectedly straightforward. Suppose that, decoder only knows about the final interval and the alphabet along with the probabilities and ranges. Noticing that the given range [0.23354, 0.2336) is in [0.2, 0.5), decoder immediately deduces that the first character was *'e'*. Continuing in a similar fashion, decoder is able to decode the whole message without any other external information. However, decoder should be able to recognize the end of the input sequence and stop proceeding. To achieve this, an end-of-file (EOF) character must be agreed upon between encoder and decoder. In this case, *'!'* is the EOF character and decoder halts after seeing it.

Here is a figure from the original paper to clarify the compression phase:

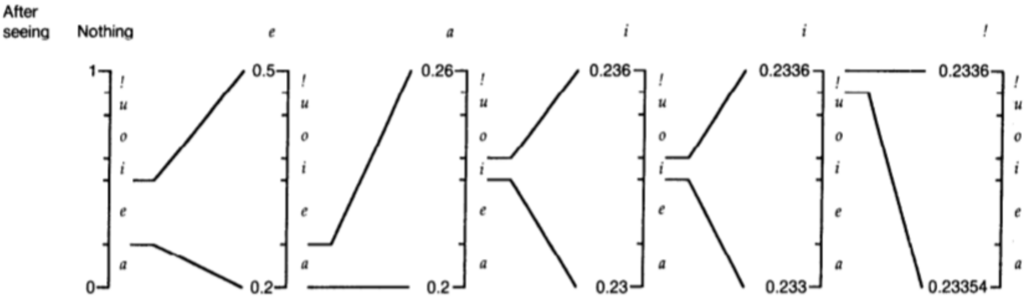

Figure 2.1: Representation of the arithmetic coding process.

Another issue of arithmetic coders is finding a suitable number in the resulting interval. That is, encoder needs to spot a real number that has a binary representation with the least number of bits. Since the whole interval uniquely identifies the given message, it is perfectly viable to use any real number in the resulting range.

Even though arithmetic coding is conceptually simple, implementing an arithmetic coder is indeed difficult compared to other algorithms. Shrinking ranges in each iteration requires the use of high precision arithmetic. Due to this, arithmetic coders are computationally expensive and slow, which is actually their biggest drawback.

Nowadays, research in arithmetic coding is mainly concerned with finding approximations to reach higher coding speeds without compromising compression ratio. To alleviate the algorithm's slow pace, many approaches such as range coding [23] and integer arithmetic coding are suggested. Recently, *Asymmetric Numeral Systems* (ANS), a novel approach introduced by Duda [24], has ended the compression ratio versus execution speed trade-off. In the study, it is shown that while ANS attains compression ratios as good as arithmetic coders, it can achieve this compression efficiency with speeds comparable to Huffman coders.

2.3.6 Burrows-Wheeler Transform

Burrows-Wheeler Transform, BWT, is introduced by Michael Burrows and David Wheeler in 1994. In their study [3], they present their implementation of transformation and the inverse transformation along with some variants.

BWT, also called the block sorting algorithm, simply reads a block of data and rearranges it such that the resulting symbol sequence is much more compressible compared to original sequence. Since the resulting string is just a permutation of the input sequence, BWT is not considered a compression algorithm. That is, the output block is comprised of exactly the same symbols that make up the input sequence.

Because Burrows-Wheeler transformation is reversible, the original ordering of a given block can be restored and it can be used in lossless data compression. Moreover, the transformation does not require the storage of any additional significant amount of data. Hence, BWT is said to be a *free* algorithm that improves the efficiency of compression algorithms in terms of compression ratio. Although BWT seems like the perfect algorithm especially for preprocessing stages of compression algorithms, it is a computationally expensive transform.

Theoretically, there are no constraints that prevent us from applying BWT to blocks of any size. The transformation is perfectly applicable. However, considering the complexity and the execution speed of the algorithm, current implementations force us to use rather smaller blocks of data. Naturally, as the size of the block gets larger, we have more opportunity to find recurrences in the data. Ideally, interpreting the whole file as a single block would give us the best possible permutation of data. Yet, it is simply infeasible considering the gigabytes-long files of our time.

A typical example of BWT is given below, using the string *'banana'*:

- We begin with appending the special BWT character, *'$'*, to the end of the input. Thus, our string becomes ***'banana$'***.

- Then we generate all possible permutations of the given input string:

banana**$**
anana**$**b
nana**$**ba
ana**$**ban
na**$**bana
a**$**banan
$banana

- Then, these generated permutations are sorted lexicographically. Note that, the special BWT character has a higher precedence.

$banan**a**
a$bana**n**
ana$ba**n**
anana$**b**
banana**$**
nana$b**a**
na$ban**a**

- Lastly, we extract the last column of this sorted matrix, and this is actually the outcome of BWT transformation.
- Hence, BWT(*'banana$'*) is ***'annb$aa'***. Note that, BWT was able to produce runs of characters even in such a short symbol sequence.

Even though, Burrows-Wheeler transformation is powerful, it does not always improve the compressibility of an input string. Moreover, it may even worsen the situation by destructing the runs that are already present in the data sequence. Consider the following non-example for the string ***'appellee$'***:

$$\begin{array}{lcl}
appellee\mathbf{\$} & & \$appelle\mathbf{e} \\
ppellee\mathbf{\$}a & & appellee\mathbf{\$} \\
pellee\mathbf{\$}ap & & e\$appell\mathbf{e} \\
ellee\mathbf{\$}app & & ee\$appel\mathbf{l} \\
llee\mathbf{\$}appe & \rightarrow & ellee\$ap\mathbf{p} \\
lee\mathbf{\$}appel & & lee\$appe\mathbf{l} \\
ee\mathbf{\$}appell & & llee\$app\mathbf{e} \\
e\mathbf{\$}appelle & & pellee\$a\mathbf{p} \\
\mathbf{\$}appellee & & ppellee\$\mathbf{a}
\end{array}$$

By sorting the generated permutations, we find that BWT(***'appellee$'***) is ***'e$elplepa'***. Note that the resulting string is less compressible compared to the original symbol sequence.

Unfortunately, inverse BWT transformation is not as straightforward as BWT. Although transformation and inverse transformation differ depending on the implementation, in this section, we discuss a simple version of the inverse transformation.

- In inverse transformation of BWT, we are oblivious about the initial symbol sequence and we only have the result of BWT, which is ***'e$elplepa'***.

- However, we can deduce the first column of the permutations by sorting the result of BWT, since they have the exact same symbols.

e		**$**
$		**a**
e		**e**
l		**e**
p	→	**e**
l		**l**
e		**l**
p		**p**
a		**p**

- Then, we prepend these columns and sort them again. After this sort operation, we again use the BWT result ***'e$elplepa'*** and sort the columns lexicographically.

- This process continues until we produce all of the possible permutations of the input string.

e$		**$a**		**e***$a*		**$ap**
$a		**ap**	*prepend*	**$***ap*		**app**
ee		**e$**	*BWT*	**e***e$*		**e$a**
le	*sort*	**ee**	*column*	**l***ee*	*sort*	**ee$**
pe	→	**el**	→	**p***el*	→	**ell**
ll		**le**		**l***le*		**lee**
el		**ll**		**e***ll*		**lle**
pp		**pe**		**p***pe*		**pel**
ap		**pp**		**a***pp*		**ppe**

- Note that, this process gave us the first three columns of the BWT permutations. Proceeding in this fashion, we are able to restore all possible permutations of the original string ***'appellee$'***. Thus, we can invert the BWT process and deduce the input symbol sequence.

To implement inverse BWT transformation, an NxN matrix can be used, where N denotes the length of input symbol sequence. However, such an approach would require immense amounts of memory and it is basically infeasible for large data sequences. As an example, a 1 MB block would require $2^{20} \times 2^{20} = 2^{40}$ bytes, which is a staggering 1 TB of space. Due to this, there are many improvements made to reduce the space requirements and time complexity.

Manzini [25] and Deorowicz [26] suggested improvements to the algorithm. Then Ferragina and Manzini [27] presented an implementation that employs suffix array data structure and this gave birth to another series of enhancements regarding the BWT algorithms that utilize suffix arrays. Kärkkäinen et al. devised a linear time suffix array construction algorithm [28]. Following this, Manzini and Ferragina [29] came up with a faster, lightweight suffix array construction algorithm that uses less memory when compared to its peers.

Many block sorting algorithms are emerged from the Burrows-Wheeler Transform. A competitive lossless data compressor, ***bzip2*** employs BWT and uses blocks of sizes in the range of 100 KB - 900 KB. BWT is generally used with Move to Front Transform and Run Length Encoding, as these techniques improve the overall algorithm's efficiency and allow higher compression ratios. As explained in Chapter 4, we have explored a similar technique in our study. However, it is found that MTF does not always improve the compression ratio efficiency. Nevertheless, it all boils down to the statistical properties of the data that is subject to the algorithm.

CHAPTER 3

RELATED WORK

3.1 Compressing Integers

Since our study is mainly concerned with compressing lists of integers, we discuss some integer encodings in this chapter. We also present Binary Adaptive Sequential Encoding (BASC) in Section 3.1.6, as it is used in the later stages of our algorithm.

Although many algorithms and techniques such as Prediction by Partial Matching (PPM), context mixing, etc. exists, only related material are examined in this chapter. After explaining a set of integer compression techniques, we elaborate on compressing inverted indices specifically, as our algorithm employs the very same data structure.

3.1.1 Variable Byte Encoding

First described by Thiel and Heaps in 1972 [30], Variable Byte (VB) Encoding is among the most trivial integer compression techniques. As the name suggest, it represents integers with variable number of bytes. The most common implementation of VB coding uses 7 bits of a byte to store the integer, and treats the remaining least significant bit as a marker that indicates whether this is the last byte that stores the integer, or the integer requires another byte that follows the current one. That is, in case of positive integers, VB can encode the integers in the range $[0, 128)$ with a single byte. However, it requires more bytes for larger integers. Some examples are given below (bold bits denote indicator bits):

- VB(6) = 0000110**1**

- VB(127) = 1111111**1**

- VB(128) = 0000001**0** 0000000**1**

Despite being a simple algorithm, given a sequence of integers, VB coding may drastically reduce the number of bits required to encode the whole sequence. When we have integers of distinct magnitudes (e.g., an integer sequence that is comprised of small numbers 1, 5, .., as well as larger ones such as 999999999), VB naturally becomes more effective.

Even though VB encoding is rather an old encoding that generally does not provide compression ratios as high as the other algorithms, it still is a fast alternative among integer compressors.

3.1.2 Simple9 & Simple16

Introduced by Anh and Moffat [31], Simple9 (S9) is another simple compression algorithm that offers a good combination of compression speed and ratio. It uses 32-bits long words to encode integer sequences. In S9 compression scheme, 4 bits are designated as status bits, and the remaining 28 bits are used for actual data storage. The status bits are basically utilized to determine the mode of this algorithm. That is, these bits tell us about the way the next 28 bits are used. Using these 4-bits long status headers, we can represent 16 different cases concerning the encoding of the next 28 bits. These cases include 28 1-bit integers, 14 2-bit integers, 9 3-bit integers(one bit unused) and so on. A table regarding these 9 cases are given as follows:

9 Cases of Simple9

Case	**Number of Integers**	**Length of Encodings**	**Unused Bits**
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

As shown in the table, in the worst case, which is case e, when we encode 5 integers that are 5-bits long, 3 of 32 bits are unused and wasted. Note that, the status header misuses the dedicated 4 bits as well, since they are used to encode 9 ways of storing integers instead of 16. Despite these inefficiencies, Simple9 proves itself as another fast alternative. Later, Simple16 is introduced as an improved version of S9 with 16 cases [32, 33].

3.1.3 PFOR and PFOR-Delta

PFOR and PFOR-Delta are relatively new algorithms suggested by Zukowski et al. [34] as a lightweight data compression technique to reduce the I/O bandwidth need of database and information retrieval systems. Researchers basically tried to leverage the super scalar capabilities of modern CPUs and achieved high instructions per cycle efficiency. Due to this strategy, they were able to attain compression/decompression speeds that are in the order of several gigabytes per second.

PFOR and PFOR-Delta are actually generalizations of an older technique called Frame of Reference (FOR), introduced by Goldstein et al. in 1998 [35]. However, since

PFOR (Patched Frame of Reference) and PFOR-Delta (PFOR applied on deltas) fix some inefficiencies of FOR, we believe elaborating on these newer algorithms are much more suitable.

FOR first divides the input integer sequence into blocks, e.g. of 128 integers. Then, it examines each block and calculates the range of values along with the minimum value in the block. Consider the following input sequence of 12 integers: {100, 105, 112, 113, 121, 127, 132, 132, 130, 137, 141, 155}. Note that, given integers are relatively small and can be represented in 96 bits, using 8 bits for each. Instead, FOR deduces that all the numbers in this partition are in the range [100, 155], and since the minimum value is 100, it subtracts 100 from every value in the partition. Hence, the original sequence becomes the following: {0, 5, 12, 13, 21, 27, 32, 32, 30, 37, 41, 55}. To encode this sequence, 6 bits are sufficient obviously. However, we need to store our reference value, 100, using 8 bits. Additionally, to be able to decompress the resulting integer sequence, 3 bits are required to record the fact that 6 bits per integer are used to encode this partition of integers. Hence, instead of 96 bits, the given input sequence can be stored in (6 * 12 + 8 + 3) = 83 bits. This reduction may seem inefficient at first. Yet, consider that the overhead of storing the reference value would be smaller, when the algorithm is applied on larger blocks.

Since it is a fairly cheap operation, Delta encoding may be used to decrease the values in the partition further. If we apply Delta encoding on {0, 5, 12, 13, 21, 27, 32, 32, 30, 37, 41, 55}, the sequence becomes {0, 5, 7, 1, 8, 6, 5, 0, -2, 7, 4, 14}. Except for the only negative value, -2, the rest of the integers can be encoded using only 4 bits per integer, which increases the compression ratio efficiency drastically. Note that, negative values can be handled in a special way. However, for the sake of succinctness, strategies concerning encoding negative values are not discussed here.

Needless to say, FOR exploits the locality in the data and provides high compression ratio efficiency whenever the ranges of integers are small. Yet, an integer with a higher magnitude compared to others would ruin this approach completely. To alleviate this issue, Zukowski et al. [34] came up with a patched version of FOR, named PFOR. In PFOR, a small bit width, *b*, is used to encode the majority of the values in a partition. On the other hand, the exceptions, which do not fit into *b* bits, are stored separately.

To find out the best possible bit width, b, PFOR tests various bit widths and chooses the one that results in the best compression ratio. On a different note, positions and the values of exceptions are encoded separately in a linked list. However, we deem such details unnecessary and believe that introducing the idea behind the algorithm is sufficient. In a similar fashion PFOR-Delta is basically PFOR applied on the deltas of the given partition.

Later, Lemire and Boytsov [36] improved PFOR with SIMD instructions and suggested SIMD-BP128 and SIMD-FastPFOR, by using a novel vectorized scheme that enables both faster decompression and higher compression ratios at the same time.

3.1.4 Binary Interpolative Coding

Introduced by Moffat and Stuiver [37], Binary Interpolative Coding(IPC) encodes a strictly increasing sequence of integers by using the knowledge of its neighbors. Employing such a strategy, it basically treats each integer separately and conveniently. Thus, this approach makes IPC sensitive to local alterations in the data.

Given a sequence .., x_{i-1}, x_i, x_{i+1}, .., since x_i is guaranteed to be in the range (x_{i-1}, x_{i+1}) we can deduce the maximum number of bits needed to store x_i through a simple subtraction, that is $log_2(x_{i+1} - x_{i-1} - 2)$. It is obvious that decoding such a sequence requires the prior knowledge of x_{i+1} and x_{i-1}, which constitutes the main problem of IPC.

Consider the example integer sequence given by Moffat and Stuiver [37], {3, 8, 9, 11, 12, 13, 17}. Then we apply Delta encoding and the sequence is converted into {3, 5, 1, 2, 1, 1, 4}. Suppose that the second integer in the list, 8, is known beforehand. Utilizing this information, given that the initial list is comprised of positive integers, we can deduce that the first integer is in the range $[1, 8)$. Hence, 3 bits is sufficient to encode it.

Then, suppose that the fourth integer, 11, is known as well. Such a knowledge would allow us to narrow down all possible values the third number can take immediately. Resulting viable range for the third number is $(8, 11)$, thus it is either 9 or 10, which can be encoded using just a single bit.

Moreover, as an extreme case, consider that the sixth integer, 13, is also known. Since the fourth integer is known and the input sequence is strictly increasing, it is certain that the fifth integer is 12. In such a case, encoding the integer in question is unnecessary and the algorithms requires zero bits to encode this integer.

To implement such an approach, IPC starts with the first and last values and encodes the median integer in the sequence. Then, similar to the process in binary search algorithm, it continues to compress the sequence recursively.

In their book, Witten et al. [38] compared IPC with other compression algorithms. According to their findings, even though IPC almost always provides the best compression ratios, it is unfortunately among the slowest compression algorithms [32] due to its complexity.

3.1.5 Recursive Bottom Up Coding (RBUC)

As explained in section 3.1.3, PFOR attempts to find a bit width, *b*, which is also called the selector, to encode the integers in a given sequence, even though some of them do not fit into *b* bits and encoded separately as exceptions. Recursive Bottom Up Coding (RBUC) follows a similar approach and treats integers in partitions. However, unlike PFOR, RBUC stores the required bit width values and proceeds with compressing these selectors recursively, until a single selector values is reached.

RBUC is introduced by Moffat and Anh [5] along with some other static codes in order to compress non-negative integers. As they stated in their study, their algorithms are suited for locally homogeneous values. Significance of this property should be clearer with an example. In their book, *Handbook of Data Compression* [39] Salomon et al. demonstrated how RBUC encodes a given non-negative integer sequence with the following example:

Given a set of integers $\{15, 6, 2, 3, 0, 0, 0, 0, 4, 5, 1, 7, 8\}$ of size $m = 13$, we begin with dividing this sequence into segments of size $s = 2$. Note that, since our set is an odd-length one, the last segment is comprised of just a single integer. Yet, this does not affect the encoding process at all and such a case is perfectly fine.

1. Initial sequence: {15, 6, 2, 3, 0, 0, 0, 0, 4, 5, 1, 7, 8}
2. Divided into segments: {(15, 6), (2, 3), (0, 0), (0, 0), (4, 5), (1, 7), (8)}

Then we deduce the largest integer in each segment, *i*, and use this value to calculate a selector value, b_i, for the corresponding segment. Selector values refer to the number of bits required to encode an integer in binary. Hence it is calculated as: $\lceil log_2(1+x) \rceil$, where *x* denotes a given integer.

3. Selectors for segments: {4, 3, 0, 0, 3, 3, 4}
4. Divide selectors into segments: {(4, 3), (0, 0), (3, 3), (4)}

Then, the integers are encoded with the standard binary code and the very same process continues recursively for the chosen selector values.

5. Initial sequence (Step 1) is encoded: 1111 0110 010 011 100 101 001 111 1000
6. Selector values (Step 3) are encoded: 100 011 011 011 100
7. Then the process continues as if the selector values are our initial set of integers.
8. In the end, produced binary sequences are concatenated in the reverse order, so that compressed sequence can be decoded easily. That is, binary encoded selector values appear before than the binary encoded initial sequence, in the resulting binary sequence. This is actually what makes RBUC a *"bottom up"* code.

Note that, four zeros that are present in the initial sequence are encoded with zero bits. Also, to be able to decode the whole input sequence correctly, size of the initial sequence should be stored as well. However, for the sake of clarity, such details concerning RBUC are not mentioned.

Obviously, as the values in segments get closer, they are encoded much more efficiently. Also note that the selector values are actually logarithms of original values, hence they get smaller and smaller as encoding continues recursively. On a different note, this recursion happens $\lceil log_2(m) \rceil$ times when the segment size, *s*, is 2.

The main drawback of this algorithm is the following: All the intermediate integer sequences until the last recursion step are required to be kept in memory during the encoding of an integer sequence. Due to this necessity, RBUC is considered as an offline algorithm. That is, encoded values basically cannot be used immediately.

3.1.6 Binary Adaptive Sequential Coding (BASC)

Binary Adaptive Sequential Coding (BASC) is another coding presented by Moffat and Anh [5]. Similar to PFOR and RBUC, given a non-negative integer sequence, BASC attempts to find selector values to encode the input sequence. However, instead of dividing sequences into partitions or segments, BASC uses a single selector value, then carries it to the next operation and updates it accordingly. Thus, it is able to exploit the local patterns in the data in a finer resolution compared to RBUC, since it treats integers individually.

In BASC, we determine a selector b' for each data symbol based on the selector b of the preceding symbol [5, 33]. Then codeword is emitted and the selector value is updated from b to b'. Given the current data x_i, its length or selector b' is $\lceil log_2(1 + x_i)\rceil$. If $b' \leq b$, then a zero is emitted and the integer is represented with b-bits. Otherwise, encoder emits $(b' - b)$ 1's followed by a zero, followed by the least significant $b' - 1$ bits of x_i. This way, we do not have to remember how many bits we used for each integer since selector value gets updated each time, and all we need to know is the first selector value b.

Consider the integer sequence given as an example in Section 3.1.5: {15, 6, 2, 3, 0, 0, 0, 0, 4, 5, 1, 7, 8} with size *m* = 13, and assume that initial selector, *b*, is 5. Note that, the pipe characters (|) in the encoded binary strings are given to separate the bits that are designated as flags and actual data.

- 15 yields b' = 4 $<$ *b* $\rightarrow$ encoder generates: 0|01111 and *b* becomes 4.
- 6 yields b' = 3 $<$ *b* $\rightarrow$ encoder generates: 0|0110 and *b* becomes 3.
- 2 yields b' = 2 $<$ *b* $\rightarrow$ encoder generates: 0|010 and *b* becomes 2.
- 3 yields b' = 2 = *b* $\rightarrow$ encoder generates: 0|11 and *b* becomes 2.

- 0 yields $b' = 0 < b \rightarrow$ encoder generates: $0|00$ and b becomes 0.
- Following zeros yield $b' = 0 = b \rightarrow$ encoder generates: 0 for each zero and b becomes 0.
- 4 yields $b' = 3 > b \rightarrow$ encoder generates: $111|0|00$ and b becomes 3.
- ...

In order to decode a BASC encoded integer sequence, we only need the values m = 13 and initial selector value b = 5. Thus, decoder knows how many codewords there are and how to read the first encoded integer.

For the sake of completeness, we briefly touch on how BASC decodes these integer sequences [39].

- If a codeword begins with a zero, read b bits, convert them into an integer n and calculate the number of bits required to encode n, k. Then update the selector value b by setting it to k and continue.
- Else, read consecutive 1's until a zero, skip that zero and store the count of 1's in a variable c. Then, decoder computes $b' = c + b$ and reads $b' - 1$ bits from the codeword. Lastly, it prepends a 1 and converts these bits into decimal. Also, b is setted to b' and the process continues.

In their study, Moffat and Anh [5] compared RBUC and BASC on different data and shown that these algorithms perform similarly in terms of compression ratio performance. Yet, BASC can be considered a much more useful algorithm in some use cases since it is an online algorithm.

3.2 Compressing Inverted Indices

Inverted indices are typical data structures used in information retrieval systems. As Yan et al. [2] described; an *inverted index I* is a collection of a set of *inverted lists* $I_{t_0}, I_{t_1}, .., I_{t_{m-1}}$ where I_t contains a *posting* for each document containing the term *t*. Each *posting* is composed of document id, term frequency and the position of occurrences of the term within the document. Thus, they can be denoted as $(d_i, f_i, p_{i,0}, p_{i,1}, .., p_{i,f_{i-1}})$ [2], where $p_{i,j}$ = *k* if *t* is the *k*-th word in document d_i.

Terms and documents may refer to different concepts depending on the use case. For example, in web search engines, HTML documents corresponding to websites comprise our documents and the terms are basically the words in these documents. In our study, we adopt the notion of inverted indices with a slight alteration. That is, in our case we just have a single document, which is the input file to be compressed and the hexadecimals that make up the file are our terms. In other words, instead of keeping the positions of terms in documents, we store the positions where the hexadecimals occur in the input file.

Information retrieval systems have a lot of problems to deal with such as tokenizing documents into terms, stemming, etc., whereas these problems are not encountered in EPIIC since we treat the input file as a sequence of hexadecimals. Because some languages like Chinese and Japanese do not use delimiters that divide words into tokens automatically, even the tokenizing process requires a lot of effort.

On a different note, phrases like ‘Middle East Technical University’ should not be tokenized into {middle, east, technical, university}, since this would change the semantics of the document. In such a case, the information retrieval system should be able to fetch documents related to the aforementioned university, instead of irrelevant documents that happen to include the tokens in a different context.

Additionally, as the set of documents gets larger, building an inverted index becomes a much more difficult task that is CPU and memory intensive. However, since we deal with just a single file in EPIIC and our algorithm compresses this file block by block, the process of building an inverted index becomes quite feasible.

On the other hand, information retrieval systems have their own tricks such as eliminating stop words, case folding, stemming and document reordering, to reduce the index size and improve the compression ratio performance. Delta coding is a common algorithm employed in compressing inverted indices. Because delta encoding works better on integer lists that have high locality, documents are reordered by using some clustering algorithms to increase locality in the data. As an example, Blandford and Blelloch use cosine similarity between documents and produce orderings for documents in their study [40]. Unfortunately, similar tricks and methods cannot be used in our case since EPIIC is a lossless compression algorithm.

Another challenge is introduced by the notion of multimodal information retrieval, which aims to fetch relevant documents from a multimedia document collection including text, image, audio and video files [41]. Yet, since we are concerned with compressing a built inverted index, we do not elaborate on this subject for the sake of succinctness.

Integer compression algorithms introduced in Section 3.1 are all used in compressing inverted indices. These methods are compared in terms of compression ratio and execution speed in many studies [2, 32, 31, 42]. According to the findings from Yan et al. [32], interpolative coding achieves the best compression ratio performance on inverted indices. However, as mentioned earlier, IPC is a comparatively slow algorithm due to its complexity. Note that, since execution time of a query in information retrieval systems is crucial, decompression speed is a significant measure while selecting a compression algorithm. Recently, Lemire and Boytsov proposed a fast algorithm, SIMD-BP128, that can decode billions of integers per second by leveraging the abilities of modern processors and SIMD instructions [36].

In our study, BASC is used to compress the positions of hexadecimals since it offers a nice trade-off between compression ratio and compression/decompression speed. Moreover, it is a trivial algorithm to implement. Nevertheless, compressing inverted indices is relatively an active research area and new techniques are constantly being developed.

CHAPTER 4

METHODOLOGY

4.1 Introduction and Motivation

Our study, EPIIC, aims to eliminate the redundancies in data by compressing the locations of hexadecimal values rather than the original data. To achieve this, it employs a couple of conventional compression algorithms such as run length encoding, move to front transformation, Burrows-Wheeler transformation, delta encoding etc. Since the concept of compressing positional data is intuitively more sensible when the data is structured orderly, some of these algorithms are used in preprocessing stage to make the data much more compressible.

The main motivation of this study is the impulse to find out whether we can discover regularities or patterns in positional information of hexadecimal characters. We are inspired by the way the data is kept in inverted index structures in information retrieval systems and search engines, hence the notion of compressing positional information. Also, the inverted index data structure enables us to apply the already available inverted index compression techniques. For example, a very trivial trick we adopted is the idea of keeping the information of gaps, instead of actual position values. The techniques we have employed and the overview of the algorithm are described thoroughly in the following section.

4.2 Method

In this section, we first describe the pipeline of the main algorithm, and then we explain the stages of different techniques and algorithms that EPIIC is comprised of.

The algorithm begins to perform its magic with a preprocessing stage. Then, we try to exploit the order in the data by building an inverted index out of preprocessed hexadecimal characters that comprises the data. After building the inverted index, we apply a couple of compression tricks and algorithms like delta encoding, which are already widely used in compressing inverted indexes [43, 44, 45]. Finally, we transform the built index into a compact structure and write it to a binary file according to the data format that is created for EPIIC.

In EPIIC, the whole pipeline of compression is done in a block by block fashion. In other words, after reading some fixed size of data block (e.g., 1 MB, 5 MB or 10 MB) from the file, we proceed by compressing it and then write the compressed block to the output file. This process is continued until there are no more blocks of data to compress.

Decompression is performed in a similar fashion as well. After a fixed size of block header is read from the compressed data, we decompress the corresponding block and write the contents into the output file. And the process carries on until the file is completely decompressed. In fact, decompression can be performed in parallel as the compressed data blocks do not depend on each other. Parallelization of the algorithm is argued in Chapter 6.

4.2.1 Preprocessing

Similar to many other algorithms, EPIIC starts with a preprocessing phase. During this initial phase, the file that we intend to compress is read as a sequence of hexadecimal characters. This may seem unintuitive when we consider that a single byte or a character is represented with 2 characters in hexadecimal format, ranging from 00 to FF. However, we treat these hexadecimals in packs of two in later stages. Hence, it can be said that our technique is a byte based encoding.

After transforming the file into a sequence of hexadecimals, the following transformations are applied to the data in the given order:

Burrows-Wheeler Transform (BWT) [3] is applied in the first step, since it converts the data into a more regular form. As mentioned in Chapter 2, BWT does not attempt to represent the data with less bits, instead it rearranges the positions of characters and creates runs of symbols. Note that during this process, we disrupt the initial form of hexadecimals that make up the data and we end up with a totally different permutation. For the sake of clarity, consider the hexadecimal sequence that corresponds to *"Hello World!"*, that is, *"48 65 6c 6c 6f 20 57 6f 72 6c 64 21 $"* (The spaces between hexadecimals are added for readability, and *'$'* denotes the special character used in BWT.). After the transformation, the aforementioned sequence becomes *"12 2f 47 6 $ 6 0c 82 5c c7 f5 46 66 66"*, which is actually the hexadecimal representation of the string *"/Gf?\??Fff"*, where question marks refer to non-printable characters, given that the special BWT character is removed after the transformation. Since the hexadecimals themselves were subject to BWT, this step alters the data radically, obviously. To our knowledge, no other study applies BWT directly to the hexadecimals instead of characters. Hence, this notion of revamping the original data completely is actually another novel contribution of this study.

In the first version of the algorithm, after applying BWT to data, we have used **Move To Front Transform (MTF)** [12] which again does not change the size of the input, yet it allegedly increases regularity by converting recurring characters into runs of zeros. However, this proved to be wrong in our case. Thus, we have abandoned using MTF in preprocessing stage. Effects of employing MTF during preprocessing is discussed in Chapter 5.

4.2.2 Run Length Encoding: A Simple Variation

Run Length Encoding (RLE) [4] is a very simple technique in which sequential recurrences of data are stored as a single data value and a count. As stated in Chapter 2, RLE has many different variations. However, in its most basic form, it transforms a sequence of symbols *"444833222225"* into the following: *"4381322551"*. Note that, since there is just a single *'8'* in the sequence, it will be converted into *'81'*. Hence, in such a case, RLE actually enlarges the data. Due to this, in our study, we use a slightly different version of RLE, that is, we convert the number of recurrences into counts only if a character repeats more than *twice*. And instead of keeping the total count, we store the number of remaining repetitions. To make it clear, consider the previous example of the sequence *"444833222225"*. Our version of RLE would alter this sequence of symbols into *"44183302235"*. Note that, non-recurring characters are left as they are.

Another distinction of our RLE variation is that it accumulates the symbols and counts separately. Thus, our version actually produces two separate sequences; namely the data string *"44833225"*, and the RLE counts vector *"{1, 0, 3}"*, when the aforementioned sequence is given as input. It may seem that such an encoding is undecodable at first. Yet, the trivial decoding algorithm is as follows:

"Read an integer from the corresponding RLE counts vector whenever a character recurs.".

We empirically found that compressing symbols and counts separately by using different algorithms is beneficial in terms of compression ratio. After separating counts from the symbols, we encode the RLE counts with Binary Adaptive Sequence Coding (BASC) algorithm. Since this gives us a binary string, we simply write it to a file and continue compressing the remaining symbol sequence in the following stages.

To our knowledge, the RLE scheme that we use is a novel variation of the algorithm, since RLE counts and runs are converted into different sequences of data and then compressed separately.

4.2.3 Building an Inverted Index

In this section, we present the very idea that makes this study novel. After encoding our data with RLE, the BASC-encoded RLE counts are written to the file and we are left with our RLE encoded symbols which is basically a sequence of hexadecimals.

In order to build an inverted index, we read two hexadecimal characters at a time from the given sequence, and we store their indexes in our data structure. Note that, since we consider hexadecimal characters in packs of two symbols, the number of all possible hexadecimal packs is 256, ranging from 00 to FF. Given a hexadecimal sequence $h_0, h_1, .., h_{m-1}$ of length m, our inverted index is a collection of a set of inverted lists of hexadecimal packs, which can be denoted as $(H_0, f_{H_0}, p_{i,0}, p_{i,1}, .., p_{i,f_{H_0-1}})$ where H_0 represents a hexadecimal pack in the range 00 to FF, f_{H_0} is the frequency of H_0

in the document and each following term, p_i, refers to the positions or indexes of H_0.

Consider a hex sequence such as '01 00 3D 04 40 00 02 20 02 03 00 07 10 03 03 03'. The inverted index for the given sequence is as follows:

$$\begin{aligned}
00, 3 &\rightarrow 1, 5, 10 \\
01, 1 &\rightarrow 0 \\
02, 1 &\rightarrow 6 \\
03, 4 &\rightarrow 9, 13, 14, 15 \\
04, 1 &\rightarrow 4 \\
05, 0 &\rightarrow (empty) \\
&\vdots \\
FF, 0 &\rightarrow (empty)
\end{aligned}$$

We can revert this operation and reconstruct the original hexadecimal sequence by simply allocating and filling an array according to the built index. Note that, since we store the positional information of hexadecimals, we actually know how much space we actually require, that is the maximum number in our inverted index. Hence, the process is completely reversible.

During compression, since we read two hexadecimal characters at a time from the given sequence, the algorithm requires an even-length sequence. Moreover, in our case, the length of a sequence is not guaranteed to be even due to RLE. Hence, to prevent odd-length sequences from disrupting our compression phase, a *zero* is inserted to the very end of the sequence, whenever we encounter with an odd-length sequence. As illustrated in Figure 4.1, parity information is added to the block flags in the block format, to be used in decompression.

4.2.4 Compressing the Built Index

In the preprocessing stage, our main goal was to transform data into a more compressible form. After building the inverted index, we apply a chain of compression techniques again to increase the compression ratio.

We start with a simple trick that we call the *fill value*. Since we keep the frequency of the hexadecimals while building the inverted index, we can obviously deduce the most frequent hexadecimal in a document. The main idea here is getting rid of the position information that belongs to the most frequent hexadecimal. Since we can recover the missing position information, this is doubtlessly a reversible transformation and we do not lose any information in this step. We do this by storing the most frequent hexadecimal as the *fill value*. To reconstruct the hexadecimal sequence during the decompression phase, whenever we cannot find the next index value in the inverted index, as the name suggests, we simply fill that missing index with the *fill value*. This is due to the fact that the inverted index must have all the positional index values from *0* to *compressed file block size - 1*. However, finding the minimum number on the index requires us to compare 256 integers (first values of the inverted lists, since they are sorted by the very nature of the algorithm), which is computationally expensive. To alleviate this, implementation of the decompression phase is altered significantly. In the latest version of the algorithm, we restore the hexadecimal sequence as follows:

- Allocate an array of size *'compressed file block size - 1'*
- Initialize the array with fill value
- Go over the inverted lists of hexadecimals, and just write the corresponding hexadecimal characters into the initialized array, according to the position values on the index

The time complexity of this approach is *O(n)*, since we iterate the whole inverted index structure once, which is actually the best we can do.

Note that this simple trick is especially useful when the frequency distribution of hexadecimals is non-uniform. By using the fill value trick, we actually replace the position information belonging to the most frequent hex with just a single byte, which is nothing but the hexadecimal itself.

If we apply the fill value trick on the inverted index example before, it becomes the following:

$$\begin{array}{lcl}
 & & Fill\_value \rightarrow 03 \\
00,3 \rightarrow 1,5,10 & & 00,3 \rightarrow 1,5,10 \\
01,1 \rightarrow 0 & & 01,1 \rightarrow 0 \\
02,1 \rightarrow 6 & & 02,1 \rightarrow 6 \\
03,4 \rightarrow 9,13,14,15 & \rightarrow & 03,4 \rightarrow (deleted) \\
04,1 \rightarrow 4 & & 04,1 \rightarrow 4 \\
05,0 \rightarrow (empty) & & 05,0 \rightarrow (empty) \\
\vdots & & \vdots \\
FF,0 \rightarrow (empty) & & FF,0 \rightarrow (empty)
\end{array}$$

Then, we apply **Delta encoding** to the remaining index values, which is basically replacing the actual values with gaps. Remember that we have defined inverted lists as $(H_0, f_{H_0}, p_{i,0}, p_{i,1}, .., p_{i,f_{H_0-1}})$, where H_0 represents a hexadecimal pack in the range 00 to FF, f_{H_0} is the frequency of H_0 in the document and each following term, p_i, refers to the positions or indexes of H_0. After applying delta encoding to the positional data in the inverted index, the structure of our inverted lists is altered as follows:

$$(H_0, f_{H_0}, p_{i,0}, p_{i,1}, .., p_{i,f_{H_0-1}})$$

$$\downarrow$$

$$(H_0, f_{H_0}, p_{i,0}, (p_{i,1} - p_{i,0}), .., (p_{i,f_{H_0-1}} - p_{i,f_{H_0-2}}))$$

$$\begin{array}{lcl}
Fill\_value \rightarrow 03 & & Fill\_value \rightarrow 03 \\
00,3 \rightarrow 1,5,10 & & 00,3 \rightarrow 1,4,5 \\
01,1 \rightarrow 0 & & 01,1 \rightarrow 0 \\
02,1 \rightarrow 6 & & 02,1 \rightarrow 6 \\
03,4 \rightarrow (deleted) & \rightarrow & 03,4 \rightarrow (deleted) \\
04,1 \rightarrow 4 & & 04,1 \rightarrow 4 \\
05,0 \rightarrow (empty) & & 05,0 \rightarrow (empty) \\
\vdots & & \vdots \\
FF,0 \rightarrow (empty) & & FF,0 \rightarrow (empty)
\end{array}$$

Finally, we encode the whole inverted index using **BASC**. As described by Moffat et al. 2006 [5] since it is an adaptive algorithm, it exploits the local regularities in the data and encodes the integers efficiently. To apply BASC encoding to our positional data, we first convert the index values of each inverted list, $(p_{i,0}, p_{i,1}, .., p_{i,f_{H_j}-1})$, into a vector of integers. Then we concatenate these vectors into a one large vector and use BASC to encode it. While concatenating these vectors, we also store their sizes in a separate vector in order to be able to reconstruct the corresponding vectors for each hexadecimal pack H_0 in decompression. In fact, by storing the sizes of inverted lists we actually save the frequencies of hexadecimals. This new vector of frequencies is also encoded with BASC.

Consequently, our inverted index is converted into two different vectors, the concatenated vector of positional data and the vector of hexadecimal frequencies $f_{H_0}, f_{H_1}, .., f_{H_{255}}$ i.e. the sizes of concatenated vectors, that are encoded with BASC algorithm, which basically converts integer sequences into binary strings (e.g. *"11001011001..."*). We conclude the compression phase of the algorithm by writing these binary strings into the file byte by byte. Note that, we use this pipeline of algorithms and techniques for each block of the file. A diagram describing the algorithm is illustrated in Figure 4.1.

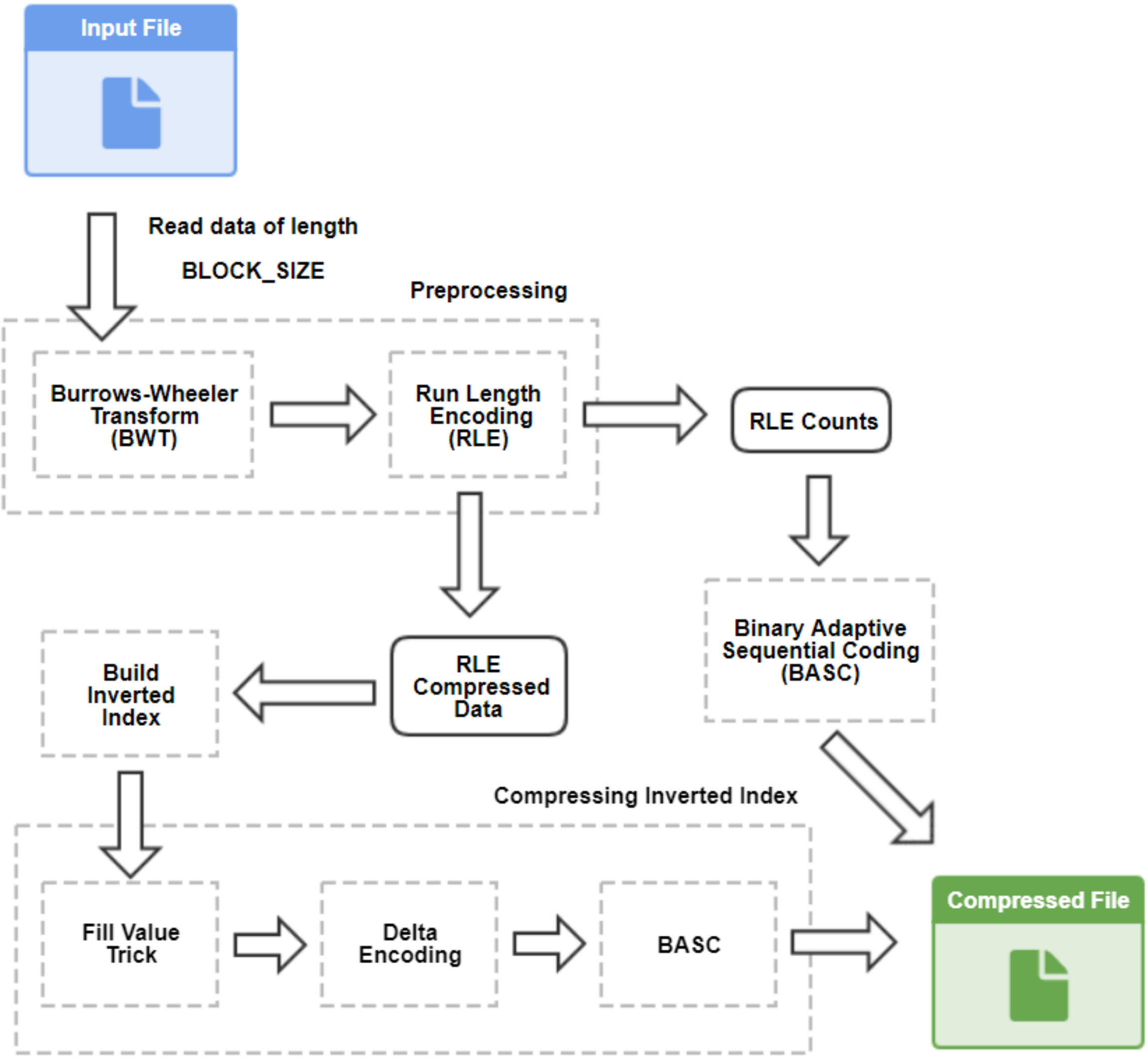

Figure 4.1: Compression diagram of EPIIC.

4.3 .eppi File Format

As a part of this study, we propose a special file format that fits our needs with the extension *".eppi"*. Since EPIIC is a block based compression algorithm, the file format is comprised of a file header and many blocks that describe the compressed data blocks. The file format is shown in Figure 4.2.

The file header is used to store supplemental metadata to process an .eppi file. The following metadata can be stored in the header of an .eppi file:

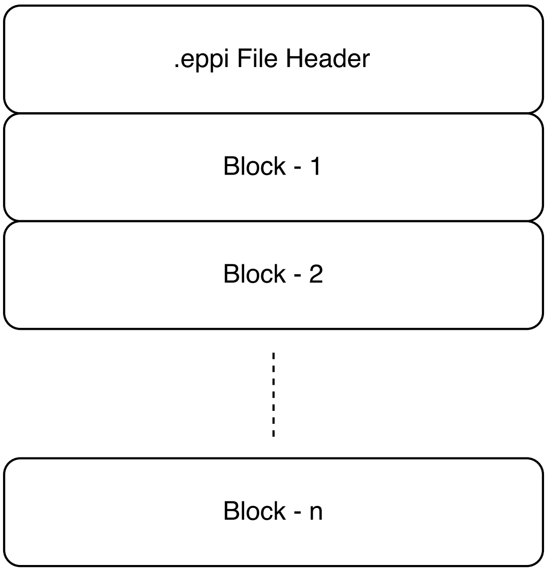

Figure 4.2: An overview of the *.eppi* file format.

- **Magic bytes:** The first couple of bytes of files are generally used to store magic bytes, i.e. the file signatures. To be able to recognize *.eppi* files and for the sake of following the customs, we have reserved the first 4 bytes for the following characters: **'E', 'P', 'P', 'I'**.
- **Non-printable character:** A non-printable character is required to prevent applications from misdetecting the file as a text file.
- **Versions:** Due to the possible algorithm alterations that may take place in the future, a major and a minor version of the algorithm should be stored to provide backwards compatibility. Two, single byte, unsigned integers should be enough to keep the version information.
- **Checksum:** A basic sanity check is required to ensure that the file is not changed after compression. A convenient hashing algorithm has to be used for this purpose.
- **Flags:** Reserving a couple of bytes for flags is probably wise, as they may come in handy in the future implementations of the algorithm. For the time being, we foresee that the following flags may be required:
  - **Endianness:** A bit to store the endianness of the machine that the file is compressed on.

  - **Total number of blocks:** Total number of blocks that the file consists may come in handy in a possible concurrent decompression implementation. Starting positions of the blocks may also be stored to ease the process.

Each block corresponds to *not compressed* data blocks of size of 1, 5 or 10 megabytes. As in many other block based compression algorithms, the users are faced with a trade-off when it comes to the block size. Using a higher block size allows BWT to exploit regularities in more data and consequently the algorithm compresses the files better. However, while increasing the compression ratio, such an approach results in longer compression/decompression times and higher memory usage. Thus, the appropriate block size to use depends on the use case. A more detailed explanation is given in Chapter 5.

The blocks that make up the compressed *'.eppi'* file are simply the concatenations of several outputs of the algorithm along with some metadata. In data communication protocols and file formats, type-length-value (TLV) scheme is a valid approach to encode information. It is even used in Internet protocol, version 6 [46]. TLV is basically concatenating type of the data, the length of it and the actual data. Type and length of the data are fixed-width and the data varies in length. In EPIIC, we have used a slight alteration called type-length-value-guard (TLVG) scheme, that introduces a guard byte, which is a value to check to ensure that the input *'.eppi'* file is self-consistent.

The following fields are included in the design of a block. Also, a visual representation is given in Figure 4.3.

- **Fill value:** A single byte fill value is put at the very beginning of each block. Remember that, the fill value indicates the most frequent hexadecimal value in the current block.

- **Index of special BWT character:** As stated earlier, to revert a BWT encoded sequence of data, index of the special BWT character in the last stage of the algorithm is required. In decompression, we simply insert the special BWT character into the appropriate index, just before BWT-decode phase.

- **Block Flags:** A single byte is reserved for possible flags that may be required for versions to come. Currently, the whole byte is used to determine whether the length of our RLE encoded data sequence is odd or even. The byte is set to 1 for odd sequences, and 0 for even ones. Since this information can be represented by using just a single bit, the remaining 7 bits may be used for different purposes in the coming versions of EPIIC.
- **Encoded RLE length (ERLE):** During compression, we have compressed RLE runs and RLE data separately. After applying BASC encoding to the RLE runs, we have written the resulting binary string to the file. This field stores the length of encoded RLE runs in bytes.
- **Byte remainder of ERLE:** BASC encoding does not necessarily produce complete bytes of data, that is, the length of the generated binary string is not always an integer product of 8. Thus, to read ERLE correctly, we need to store the byte remainder, i.e. the number of bits to be read from the last byte.
- **Encoded inverted list vector length (EILV):** While compressing the built inverted index, we concatenated each inverted list and created a single vector of integers. To be able to create the same inverted index during decompression, we need to know the number of values stored in each inverted list, that is, the frequency of the hexadecimals through *00* to *FF*. This field denotes the length of the BASC encoded vector of frequencies.
- **Byte remainder of EILV:** Similar to ERLE, we need to store byte remainder for EILV as well.
- **Encoded RLE Counts:** BASC encoded RLE counts are stored in this field.
- **RLE end marker:** This is a single byte guard indicating the end of encoded RLE counts.
- **Encoded Inverted List Sizes:** BASC encoded vector of hexadecimal frequencies is stored in this field.
- **Total compressed size of encoded inverted lists (TCS):** Even though the frequency of each hexadecimal is known, the total size of encoded inverted index is required to decompress the inverted index accurately.

- **Byte remainder of TCS:** Similar to ERLE, we need to store byte remainder for TCS as well.
- **Encoded inverted lists:** BASC encoded vector of concatenated inverted lists. This fields represents the actual data that makes up the inverted index.
- **Block end marker:** A single byte guard indicating the end of the current *'.eppi'* block.

4.4 Complexity and Constraints

In this section, we elaborate on the constraints of the algorithm and explore its complexity at different levels of granularity.

4.4.1 Overall Complexity

To determine the overall complexity of the algorithm let us first go over the algorithms that are used in EPIIC one by one.

Compression process starts with applying BWT encoding to the data block. In our algorithm we have used KS algorithm proposed by Kärkkäinen et al.[28]. It is a linear time suffix array construction algorithm, which is actually the computationally expensive part of BWT. Even though there are better algorithms and optimizations that offer an alternative to suffix array construction [47, 29, 48, 49, 50], we have employed the KS algorithm due to its simplicity.

After transforming the data with BWT, RLE is applied to the data. While encoding the data block with RLE, we simply iterate over the data and count the recurrences of characters. Thus, it is obviously $O(n)$.

Another encoding that is used frequently in our study is BASC. As stated in Chapter 3.1.6., BASC is an online encoding algorithm that requires a selector value and the next integer in the sequence. It converts an integer sequence to a binary string by performing simple operations such as subtractions and taking logarithms. As described by Moffat and Anh [5], BASC is a linear-time algorithm.

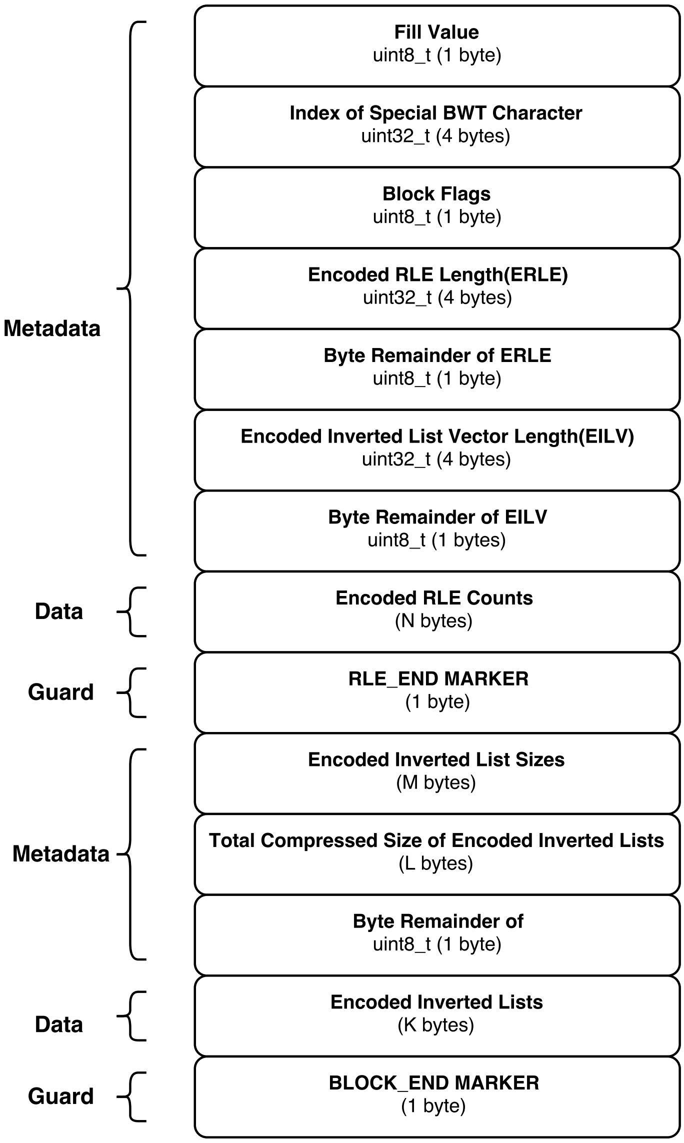

Figure 4.3: A block of the *.eppi* file format.

Building the inverted index in both compression and decompression is unexpectedly simple. During compression, we iterate over the compressed hexadecimal sequence two symbols at a time, and save the index values to the corresponding inverted list. Therefore, this operation requires exactly $O(n/2)$ push operations to inverted list vectors that make up our inverted index. Note that, we insert a *zero* to the end of the sequence if it is an odd-length sequence, and remove it in the decompression phase. On the other hand, in decompression we use the range constructor of C++ vectors to create 256 inverted lists, which has a time complexity of $O(256n)$. Although the number of inverted lists to initialize is constant, there is probably a better way to build the inverted index in this phase. Also, we encounter with a similar situation while reverting the built index to a sequence of hexadecimals in decompression. To obtain the hexadecimal sequence, we first initialize a vector with fill value and then edit it at necessary indexes while iterating each inverted list, which is again $O(256n)$.

Since all of these operations are applied to possibly many data blocks that make up the file, even though every single operation is $O(n)$, the overall complexity is actually $O(bn)$, where b denotes the number of blocks.

4.4.2 Constraints due to File Format

As it is stated in Chapter 4.3, *'.eppi'* file format defines the compressed data by using TLVG encoding scheme, which has fixed sized data fields as well as variable sized ones. The fixed sized fields place some constraints on our algorithm's ability to compress all kinds of files. Since markers (RLE_END and BLOCK_END) are defined as single byte fields that are independent of any other data, no constraint arises due to them. Likewise, due to the very nature of byte remainders, a single byte is sufficient as the maximum value to be stored in a byte remainder field is 7. In the same way, the fill value is defined as an 8-bit unsigned integer, which is adequate to store all possible hexadecimals in the range *00 - FF*. However, the following fields are both 32-bit unsigned integers (ranged from 0 to 4,294,967,295) and they define solid constraints:

- **Index of special BWT character:** This field places a constraint on the maximum size of a block that can be compressed, which is nearly 4 gigabytes due to the range of 32-bit unsigned integer. However, trying to encode a 4 gigabytes-long character sequence with BWT is simply infeasible.

- **Encoded RLE length:** Since BWT does not change the actual size of the input character sequence, RLE is applied to data of same length. Hence, this field limits the block size as the index of special BWT character does.

- **Encoded inverted list vector length:** Because the field is a 4 byte long unsigned integer, the length of the BASC encoded inverted list vector must be less than 4 gigabytes.

- **Total compressed size of encoded inverted lists:** Due to this field, the compressed size of encoded inverted lists of a single block must be less than 4 gigabytes.

Another constraint emerges from the very nature of fill value. Because the algorithm basically removes the positional information of the most frequent hexadecimal, EPIIC may fail to compress/decompress whenever one or more blocks are comprised of the same character. Even though the probability of the same character recurring for millions of times is extremely small, such a case is perfectly possible. The behavior of EPIIC on this condition is not tested thoroughly, but we leave this issue as a future work.

Compressing some files can backfire representing the same data with more bits. However, we believe that the discussed constraints due to file format would not cause any trouble easily.

4.5 Advantages

In this section, we discuss the advantages of EPIIC over other compression algorithms. Nevertheless, selection of the proper data compressor depends on the data and constraints which are based on use cases and applications.

4.5.1 Using Different Encodings

Setting aside the earlier steps of the algorithm including preprocessing, EPIIC actually provides the power to use any integer encoding algorithm on any kind of data. That is, nearly all of the steps of EPIIC; preprocessing, BWT, RLE, building the inverted index, fill value trick and delta encoding, should stay the same. However, the users are free to implement their own integer encoding algorithm and plug it to **EPIIC** - hence the name **E**ncoding **P**luggable **I**nverted **I**ndex **C**ompression. This is possible due to the very nature of inverted indexes and the use of positional information. By building an index, we basically convert hexadecimal representation of a file into a set of vectors of integers.

Even though EPIIC can be employed in encoding any kind of data, e.g. textual, audio, genomic, etc., it performs differently on each kind of data, which is anticipated, because there is no universal compressor that can represent every single data with less bits. That is, any compressor is expected to perform good in some subset of all possible permutations of bits and perform bad in others, in terms of compression ratio [51]. Thus, we encourage users to implement and try different integer encoding algorithms to see whether other algorithms perform better in their case.

4.5.2 Compress Once, Decompress Many Times

Recently, a new notion or paradigm “compress once, decompress many times” is on the rise in data compression community. It is argued that, some files are compressed once but then decompressed many times in real life scenarios. This concept proves to be true in case of game assets and high-performing distributed storage systems like Google’s BigTable, Facebook’s Cassandra and Apache’s Hadoop [52]. Brotli [20], a new compression algorithm developed by Google, used in Play Store applications follows this approach and saves gigabytes of bandwidth everyday. The fact that it requires a lot of resources is overlooked due to the benefits.

Through empirical observations, it appears that EPIIC follows this paradigm as well, as it requires less resources to decompress an EPIIC encoded file. Moreover, as discussed in Chapter 6, EPIIC’s decompression phase can be accelerated by implementing a multithreaded version of the algorithm. Since EPIIC is a block based algorithm and the blocks are decompressed independently of each other, such an implementation is absolutely viable.

4.5.3 Leveraging Related Research

Another advantage of EPIIC that we claim is that it can leverage the related research concerning inverted index compression. Whenever an advancement occurs in the ongoing related research, the techniques that are used can be mimicked by EPIIC. On the other hand, using complex algorithms in compressing the inverted index that we built may decrease the compression/decompression speed of our algorithm brutally.

CHAPTER 5

EXPERIMENTS AND RESULTS

In this chapter, we present our experiments and results concerning our study, EPIIC. The experiments of the proposed lossless data compression algorithm are done on a system with the following configurations:

- Intel® Core$^{\text{TM}}$i7-2670QM 2.20GHz CPU
- 8 GB RAM
- 64-bit Linux Mint 17 Qiana OS

The experiments are carried out on the files which are used in Squash Compression Benchmark [53]. As stated on the website of Squash, it provides a single API to access many compression libraries. Hence, it provides flexibility in choosing compression algorithms and facilitates conducting experiments. Note that, most of the files in Squash Compression Benchmark are taken from Canterbury Corpus [54] and Silesia Corpus [55]. Squash presents a plethora of information regarding the benchmark files such as compression ratio and compression/decompression speed. Moreover, it is possible to observe these measurements for different system configurations. In our experiments, we have compared our results with the results of a very similar system provided on Squash named *'hoplite'*, which has the following configurations:

- Intel® Core$^{\text{TM}}$i7-2630QM 2GHz CPU
- 6 GB RAM
- 64-bit Fedora 22 OS

In this chapter, we first explore the effect of block size on compression ratio and memory usage. Its effect on compression/decompression speed is not investigated thoroughly. However, it is observed that block size does not affect the execution speed of the algorithm as drastically as memory usage. Then we inspect the compression ratio performance of EPIIC and compared it to a series of state-of-the-art algorithms. Later, we analyze the compression ratios further and explain the reasons why our algorithm tends to work better on some files. We also probe EPIIC's compression/decompression speed and discuss the current state of the algorithm. Lastly, we argue about employing MTF as a preprocessing step and justify our decision with an experiment that has solid results.

5.1 Effect of Block Size

EPIIC is a block based compression algorithm. Since it allows different block sizes, the effect of block size is examined in this section. EPIIC uses 10-megabyte blocks by default. Whenever block size is not specified in experiments, it may be assumed that it is 10 megabytes long.

We first investigated the way block size affects compression ratio and memory usage of EPIIC. To describe the way compression ratio is affected, three files from Squash benchmark are chosen. These files are namely *enwik8*, the first 100 MB of English Wikipedia dump, *webster*, a text file that is comprised of The 1913 Webster Unabridged Dictionary, and *mozilla*, a .tar file of executables of Mozilla 1.0. These files are chosen because they realistically represent the files used out in the wild. Another reason is their greater sizes compared to the other benchmark files. Comparison of a large range of block sizes is much more preferable. However, such a goal is not attainable if the sizes of benchmark files are tiny. Also, *enwik8* is said to be a snapshot of human knowledge and it is used in benchmarks as well as contests such as the famous Hutter prize.

We have measured compression ratios for the aforementioned files with block sizes ranging from a single megabyte to a hundred megabytes. To be exact, the following are the block sizes we have used in this experiment: 1 MB, 5 MB, 10 MB, 20 MB, 30

MB, 40 MB, 50 MB and 100 MB. The block sizes are illustrated with a dot on the line graphs given in Figure 5.1. Note that, because its size is 41.5 MB, the compression ratio does not change for 50 MB and 100 MB blocks in *webster*'s case.

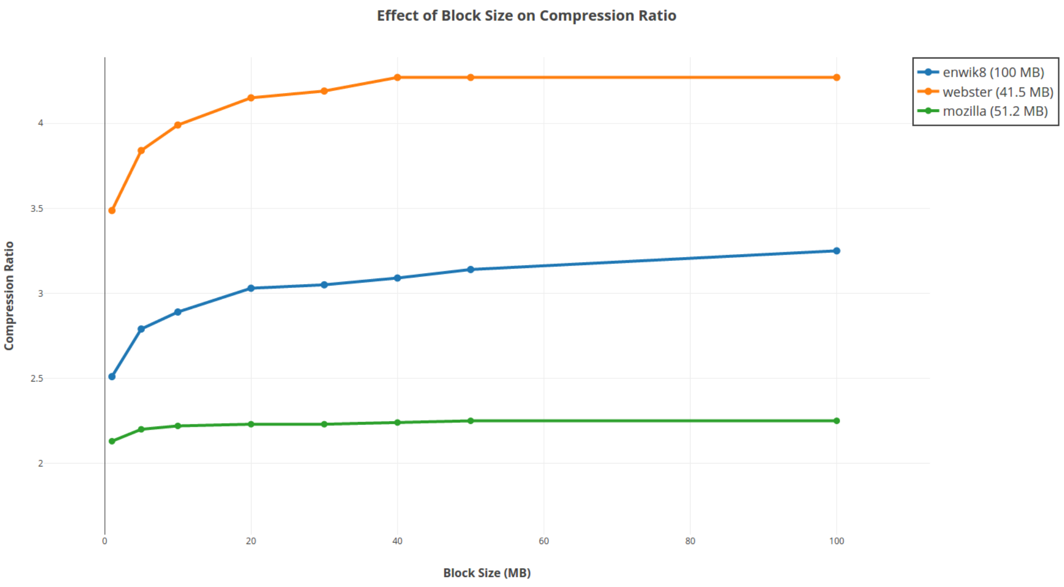

Figure 5.1: Effect of block size on compression ratio.

When block size increases, compression ratio tends to increase as well. This is due to employing BWT in the preprocessing stage. When the block size is large, since BWT is performed on a bigger block of data, it exploits the regularities in the file better. This suggests that we should use larger block sizes whenever we can. However, increasing block size also increases memory usage and execution time. Moreover, a better compression ratio is not always guaranteed. For some cases, we observed that increasing the block size only resulted in more memory usage, but the compression ratio remained unaffected, which is obviously not ideal. Such a case is illustrated in Figure 5.1. Although using a larger block size was beneficial for the benchmark files *enwik8* and *webster*, the algorithm could not increase the compression ratio for *mozilla*. Also, given the compression ratios of these 3 files, it can be argued that the optimal block size to use is 20 megabytes. Nevertheless, this is a trade-off and users should be able to pick the convenient block size for their use case.

5.2 Memory Usage

Another evaluation we have done is concerned with memory usage of the algorithm. Even though compression ratio and execution speed bears much importance, an algorithm that uses fewer system resources is also desirable. Since it is a block based algorithm, memory usage of EPIIC ultimately depends on block size and the size of the input file is completely irrelevant. In this experiment, we again used the benchmark file named *enwik8* and recorded EPIIC's memory usage every second during compression/decompression. The collected data concerning memory utilization is provided in Figure 5.2. For the sake of clarity, only the results for 1-MB and 10-MB blocks are given.

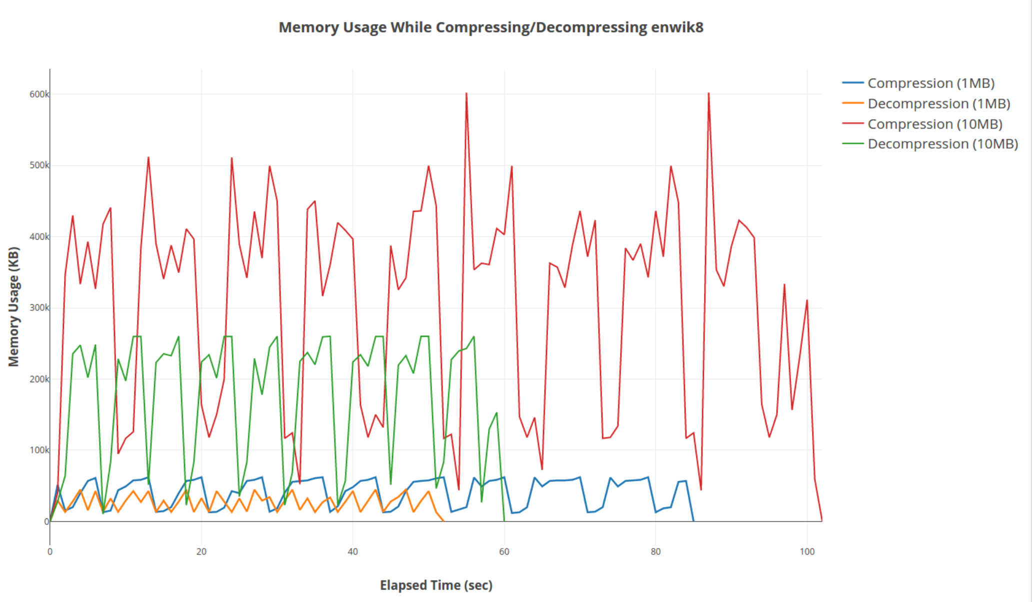

Figure 5.2: Effect of block size on memory usage during compression and decompression.

It is observed that altering block size affects memory usage of EPIIC drastically. The amount of memory required for EPIIC is empirically discovered to be approximately 50-60 times more than the block size. During compression, when 1-MB blocks are used, the memory usage is around 50-60 megabytes and it reaches up to 500-600 megabytes if we increase the block size to 10 megabytes. Memory usage for many other block sizes are tested and it is found that they follow the same pattern. Thus, we reckon that EPIIC has an average space complexity of *O(50b)*, where b denotes the block size in megabytes. Similarly, memory usage while decompression exhibits similar properties with an average space complexity of *O(25b)*.

Also, observe that the memory usage drops significantly in between compressing/decompressing different blocks. That is, when the algorithm is done with compressing/decompressing a block, it frees all the allocated memory and writes compressed/decompressed data block into the output file immediately. Then, EPIIC progresses to the next data block and memory usage increases again. In this fashion, memory utilization of EPIIC keeps oscillating during compression/decompression. Note that, memory usage during compression with 1 MB blocks seems much more stable due to our sampling frequency. As stated before, we record memory utilization at each second and the oscillation in case of 1 MB blocks is much more frequent in reality.

On a different note, block size seems to be affecting the algorithm's execution speed on a much smaller scale as opposed to memory utilization. While compressing *enwik8* takes approximately 85 seconds with 1 MB blocks, it takes just over 100 seconds with 10 MB blocks. Hence, when there is a tenfold increase in block size, EPIIC's memory usage increases tenfold as well due to linear complexity. However the compression time becomes only 1.17 times greater. Likewise, EPIIC's decompression speed is not affected dramatically either. However, we have not conducted further experiments on this issue and the relation between block size and compression/decompression speed shall be examined in more detail.

High memory usage puts a constraint on the applicability and availability of the algorithm on different systems. However, note that the current version of EPIIC is not optimized for memory usage and we claim that its memory footprint can be reduced. For starters, the current implementation performs operations that use a lot of memory unnecessarily, such as concatenating large integer vectors.

Since memory usage generally does not pose a serious bottleneck for data compression algorithms, it is difficult to find studies that focus on comparison of algorithms with respect to their memory usage. In their study, Sadler et al. [56] compared algorithms such as bzip2, ZLib, ncompress, LZO, PPMd for energy-constrained devices and demonstrated that memory usage of these algorithms can be as little as 10 kilobytes to 10 megabytes. On the other hand, Guetzli [57], a new JPEG encoder implemented by Google, is known to require 300 megabytes of memory per a megapixel of the input image [58]. Thus, selection of a compression algorithm absolutely depends on the use case and the current version of EPIIC may not be suitable for mobile devices due to its high memory usage.

5.3 Compression Ratio Performance

Compression ratio is a critical measure for compression algorithms and naturally high compression ratios are desired. In this section, we first provide experimental data concerning the compression ratios of benchmark files, and then we discuss EPIIC's performance.

To begin with, obtained compression ratios of EPIIC with 10-MB blocks for Squash benchmark files are provided in Figure 5.3. For the files used in Squash benchmark, it can be claimed that EPIIC yields a compression ratio of 2 to 3 on the average. That is, EPIIC is able to store and represent the same amount of information using only a half or a third of the bits required for the original file. Nevertheless, compression ratio performance absolutely depends on the file itself and it is difficult to suggest a range of compression ratio performance for compression algorithms.

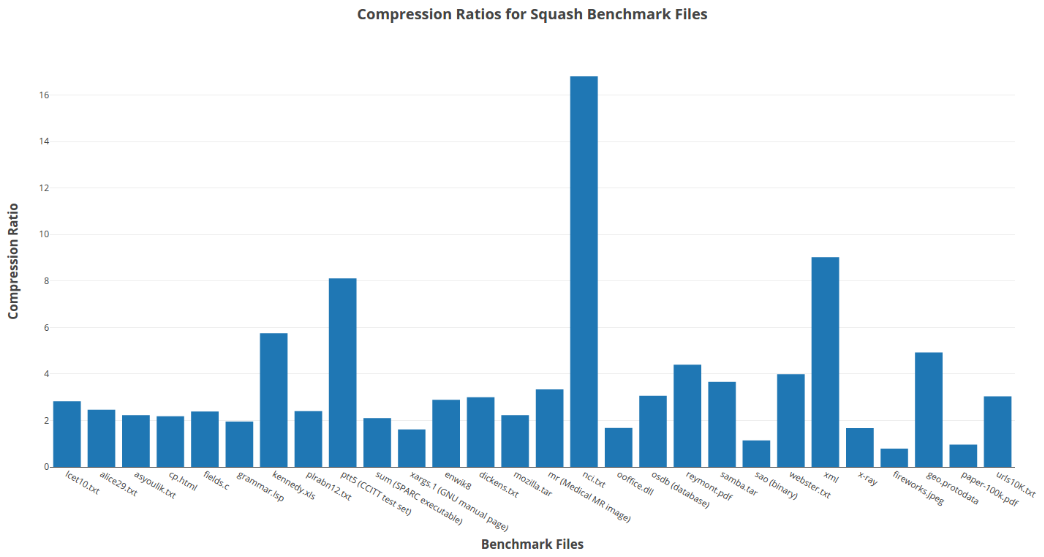

Figure 5.3: Compression ratios of EPIIC 10-MB for Squash benchmark files.

A benchmark file *fireworks.jpeg* is an already compressed file and it is expected to be compressed poorly, since compressed files have less redundancies and high entropy. In fact, the compression ratio of *fireworks.jpeg* is 0.785 and EPIIC actually increases the file size while attempting to compress the file. For such cases, a compression ratio check is required as a final step of the algorithm. In other words, if compression ratio for a given file is less than 1, the file should be left as it is. Such a feature ensures a lower bound of 1 for compression ratio performance.

Reporting compression ratios without any comparison is indeed vain. Thus, to be able to interpret our results properly, we have taken compression ratio performance data of various algorithms from Squash benchmark and simply appended our results to illustrate EPIIC's competitiveness clearly. Naturally, fast algorithms that provide high compression ratios are more desirable. For readers' convenience, we only provide graphs for just 4 benchmark files of Squash. For the remaining files, readers may compare the provided compression ratios of EPIIC on Squash benchmark's website manually. From Figure 5.4. to Figure 5.11, we present compression ratio versus compression/decompression speed graphs for benchmark files *mozilla*, *enwik8*, *dickens* and *xml* in the given order. While *dickens* is a collection of Charles Dickens' work in textual format, *xml* is a concatenation of 21 .xml files.

As shown in Figure 5.4, in case of the benchmark file *mozilla*, a .tar file of executables, EPIIC performs poor and provides us a compression ratio of 2.226. Nevertheless, it is observable that EPIIC is able to compete with algorithms such as lz4, lzo and lzg, which belong to the Lempel-Ziv data compressor family. Also, compression ratio versus decompression speed for *mozilla* is given in Figure 5.5. Note that, even though some algorithms like lz4 and density are not that successful in compressing the given file, they are able to compress/decompress the file with blazing speeds in the order of several hundred megabytes per second.

Note that the difference between compression speed and decompression speed of EPIIC is not observable on the figures. To shed some light on this subject, execution speed of our algorithm is discussed in Section 5.5. Nonetheless, although EPIIC's marker position on figures does not change much, we have provided both compression speed and decompression speed graphs so that the behavior of the other algorithms can be seen.

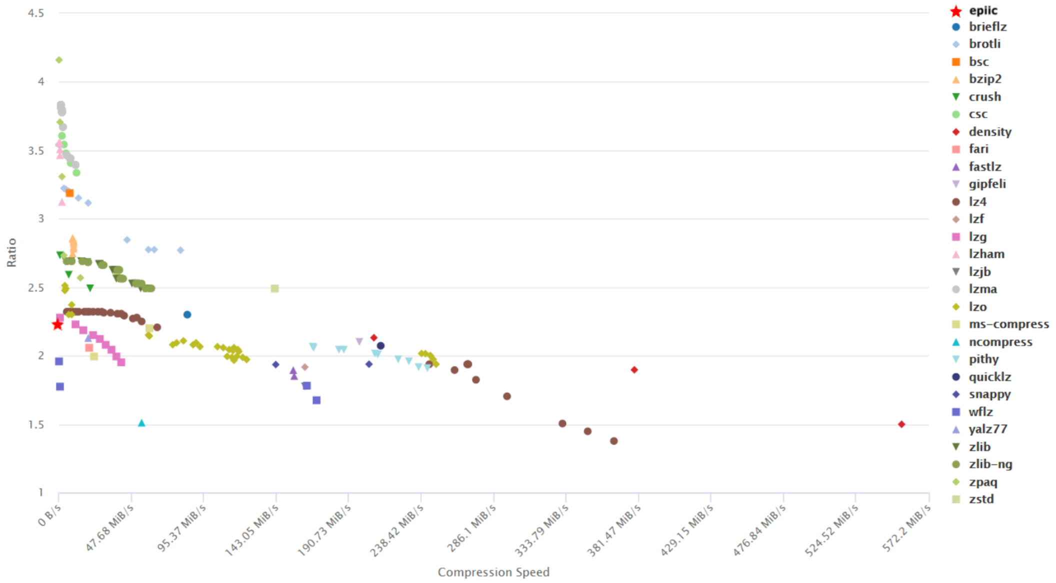

Figure 5.4: Compression ratio vs. compression speed comparison for *mozilla*.

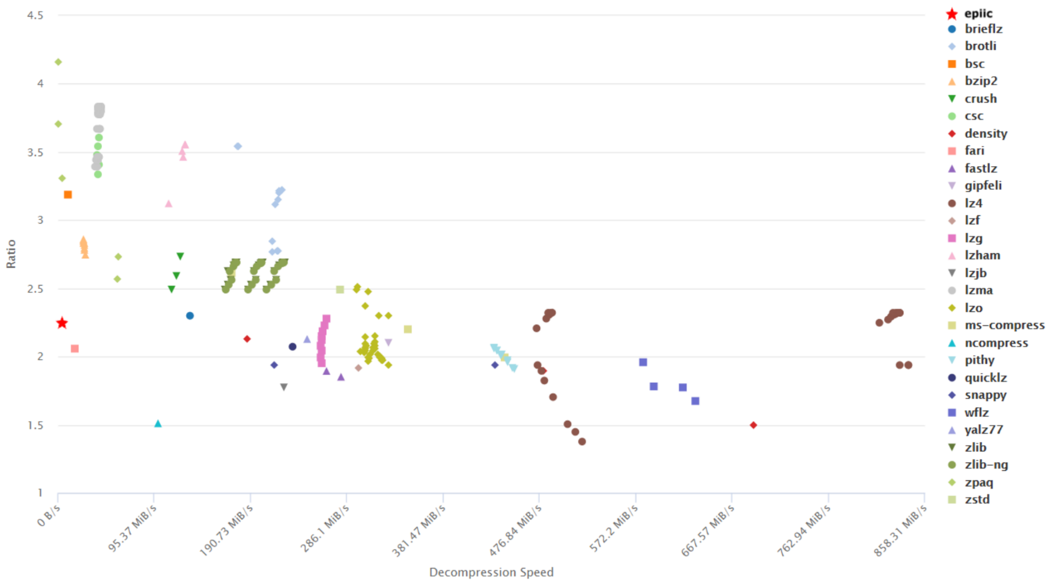

Figure 5.5: Compression ratio vs. decompression speed comparison for *mozilla*.

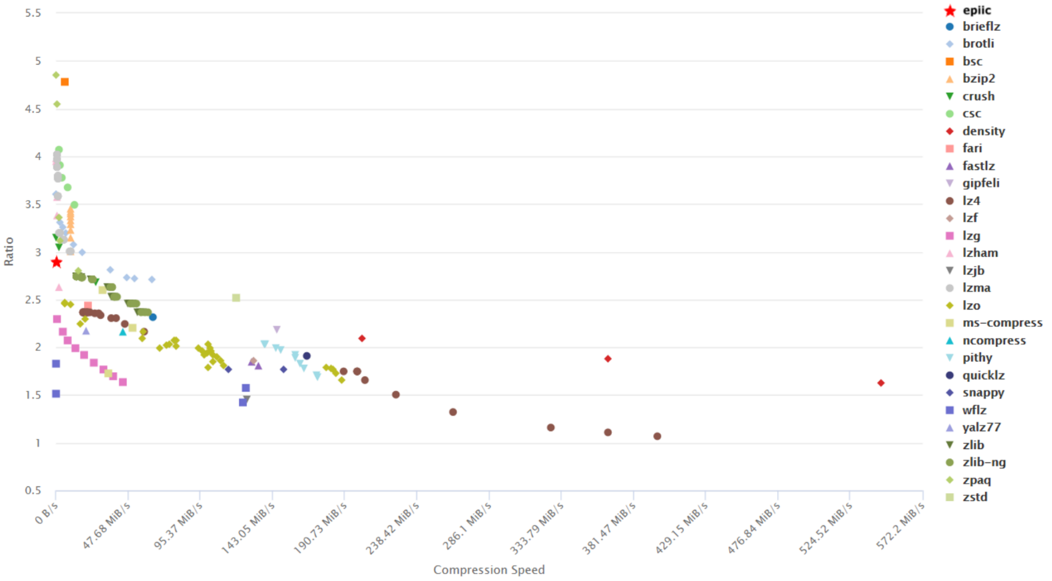

Figure 5.6: Compression ratio vs. compression speed comparison for *enwik8*.

Regarding an another Squash benchmark file, *enwik8*, we measure a compression ratio of 2.89 for EPIIC. As illustrated in Figure 5.6, considering that algorithms that result in better compression ratios, such as zpaq, lzma, lzham and crush, have execution speeds in the order of hundreds of kilobytes; our algorithm actually proves to be competitive and offers an alternative with a compression speed around 1 MB per second. Note that, measured compression/decompression speed for benchmark files are given in Figure 5.16, in Section 5.5.

As stated earlier, *enwik8* is considered to be a snapshot of human knowledge and we believe that being competitive in this case is indeed significant. Also, recognize that algorithms that provide high compression ratios like lzham and brotli can decompress *enwik8* with a speed in the order of hundreds of megabytes. In other words, the original size of the *enwik8* is 100 MB, and it is possible to compress/decompress it under a single second with higher compression ratios.

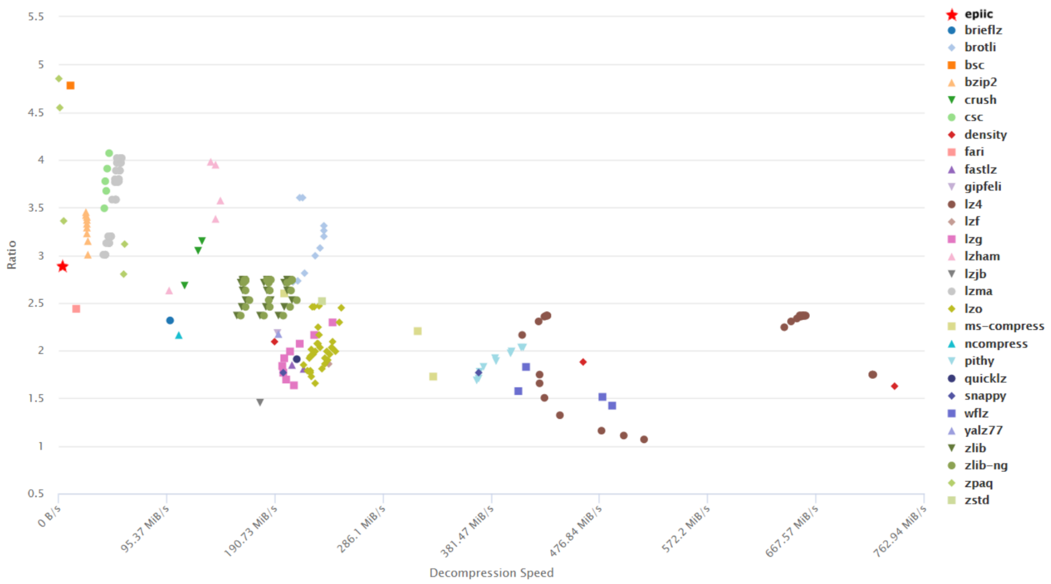

Figure 5.7: Compression ratio vs. decompression speed comparison for *enwik8*.

Another text file given in Squash benchmark is a file named *dickens*, a collection of Charles Dickens' work with a size of 10.2 MB. In this case, EPIIC is able to shrink the file to its third in size with a compression ratio of 3. A comparison of compression ratio and compression/decompression speeds are given in Figure 5.8 and 5.9. As shown in the figures, EPIIC performs on par with crush, lzma, brotli and bzip2 and surpasses algorithms like zlib, lz4 and zstd in terms of compression ratio. Yet, it would be fair to express that some of these algorithms that achieve poor compression ratios are actually optimized for high compression/decompression speed and reasonable compression ratio. In some real world use cases, the execution speed of the algorithm may be more critical than the compression ratio. Thus, lossless data compression algorithms on both extremes are required and used.

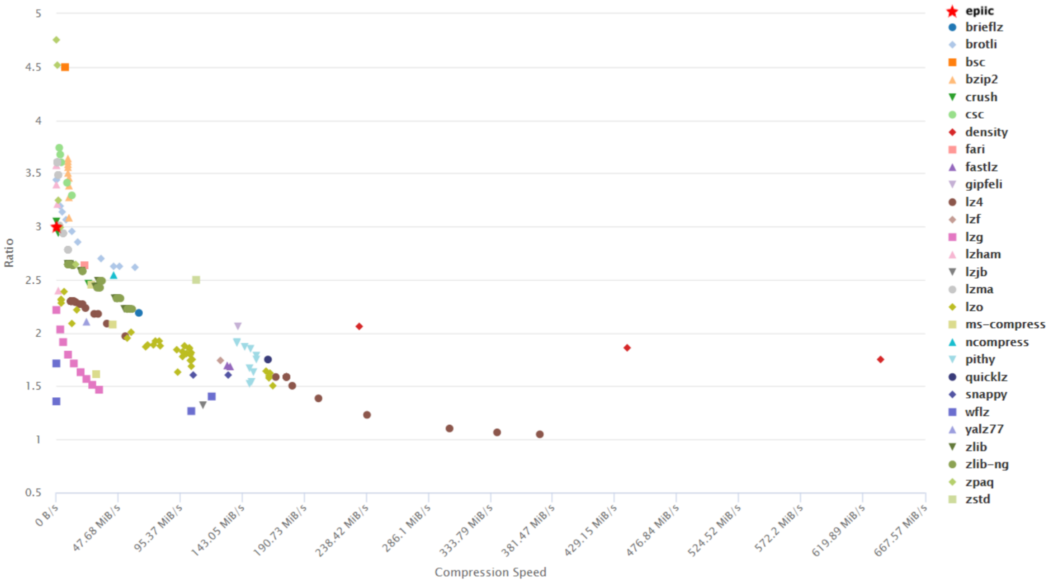

Figure 5.8: Compression ratio vs. compression speed comparison for *dickens*.

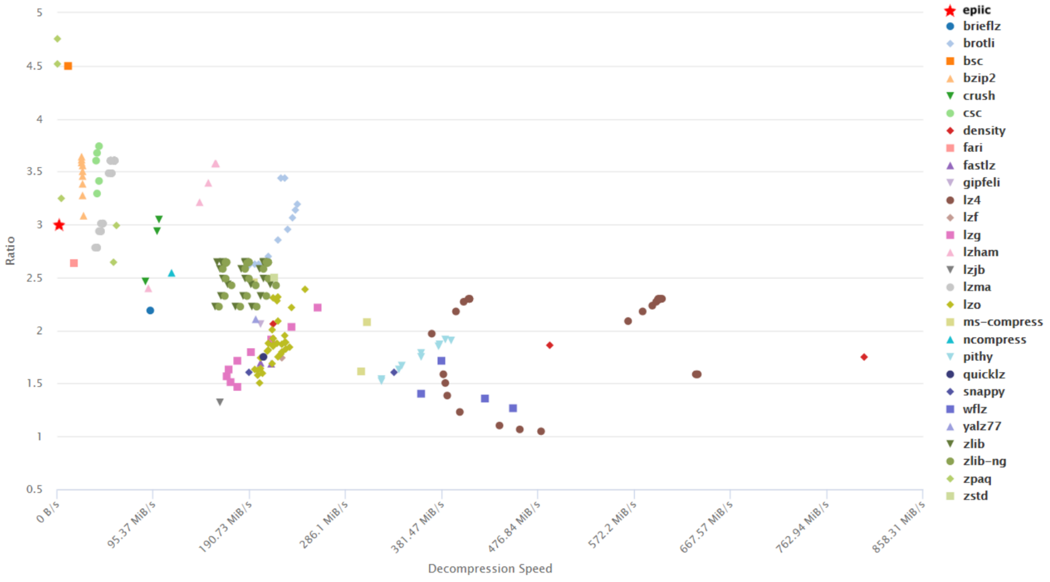

Figure 5.9: Compression ratio vs. decompression speed comparison for *dickens*.

Lastly, we have decided to include comparison graphs for the benchmark file *xml*. Since it is a concatenation of 21 distinct .xml files, we believe that this file, in a way, gives us a broader view concerning the compression ratio performance of EPIIC on .xml files. Experimental results regarding *xml* are given in Figure 5.10 and 5.11.

Aside from the fact that EPIIC has a slow compression/decompression speed, it proves its competence with a compression ratio of 9.02. While our study outperforms many algorithms including almost all of the LZ family algorithms except lzham and lzma, it behaves almost as good as csc, crush and lzma. Also, note that most algorithms are able to decompress *xml* with execution speeds over 200 MB per second.

.xml is known to be a common file format and it is used extensively in web development. Thus, presenting an alternative lossless data compression algorithm for .xml files is valuable, since employing such an algorithm would save inconceivable amounts of bandwidth each day.

When compared to other lossless data compression algorithms in terms of compression ratio, EPIIC often performs on par with commonly used algorithms such as lzma and brotli. It is found to be especially effective in text and .xml files. However, more experiments are required to truly measure the usability of EPIIC. Note that, the given compression ratio values represent EPIIC with 10 MB blocks and compression ratio tends to increase with larger block sizes. Although EPIIC is competent in terms of compression ratio, its compression/decompression speed is utterly poor. Nonetheless, this novel lossless data compression algorithm is still in its infancy and we believe that it has much more to offer.

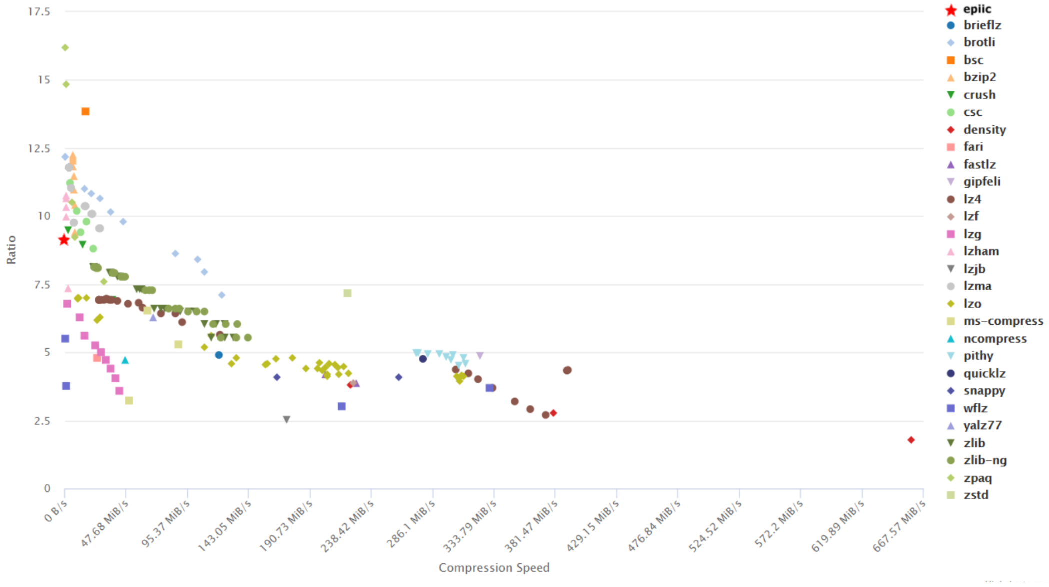

Figure 5.10: Compression ratio vs. compression speed comparison for *xml*.

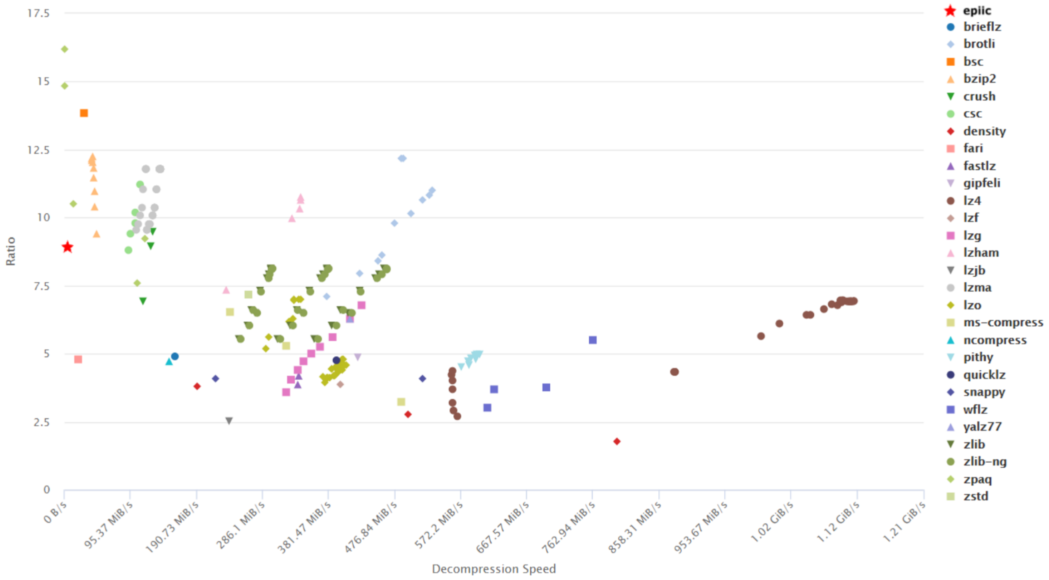

Figure 5.11: Compression ratio vs. decompression speed comparison for *xml*.

On a different note, observe that zpaq and bsc are almost always the best compressors for Squash benchmark files and they achieve state-of-the-art results in terms of compression ratio. Yet, they are not as fast as other algorithms and they are preferable when compression ratio is the highest priority. Hence, again it all boils down to the use case.

5.4 A Further Analysis of Compression Ratios

There are a couple of reasons why EPIIC performs better when used on some files and worse in others, in terms of compression ratio. The main reason is obviously the structure and the order of the hexadecimals that make up the data. However, in order to find more profound reasons we have delved into Squash benchmark files and analyzed their hexadecimal frequencies.

A noticeable explanation is non-uniformity of the frequencies of hexadecimals. When the file is comprised of hexadecimals with distinct frequencies, occurrence of a hexadecimal can be tremendously higher compared to another hexadecimal. In fact, some hexadecimals may not even be found in a file. As discussed in Chapter 4, we employ a simple trick named fill value and we discard the most frequent hexadecimal from our inverted index structure. Hence, this non-uniformity or irregularity in hexadecimal frequencies boosts the effectiveness of EPIIC regarding compression ratio. Actually, such a property literally means that the data is more orderly and thus more compressible. A histogram for hexadecimal frequencies of a benchmark file named *nci* is given in Figure 5.12. It is a great example that portrays the effect of our fill value trick with an astonishing compression ratio of 16.8.

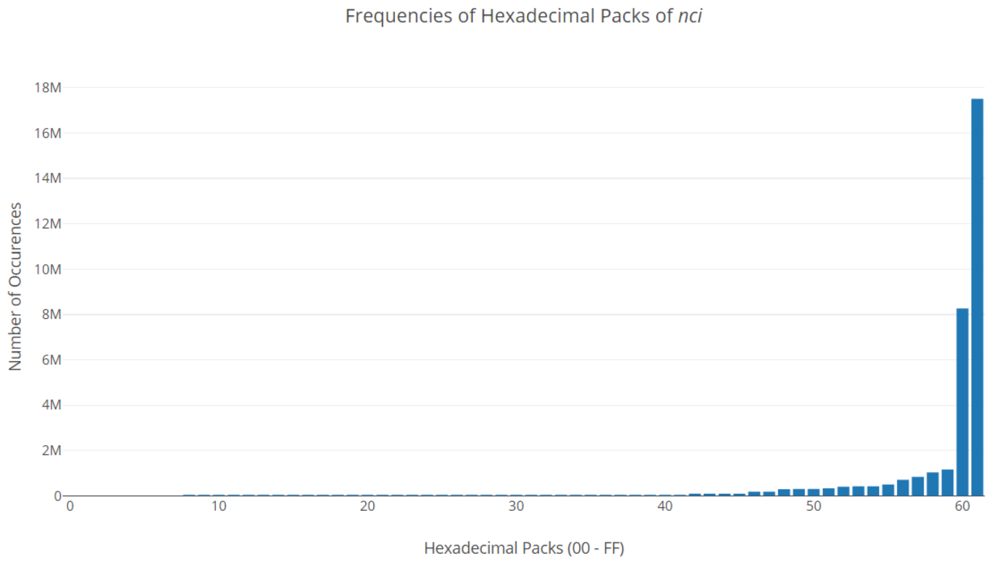

Figure 5.12: Histogram of hexadecimal frequencies of *nci*.

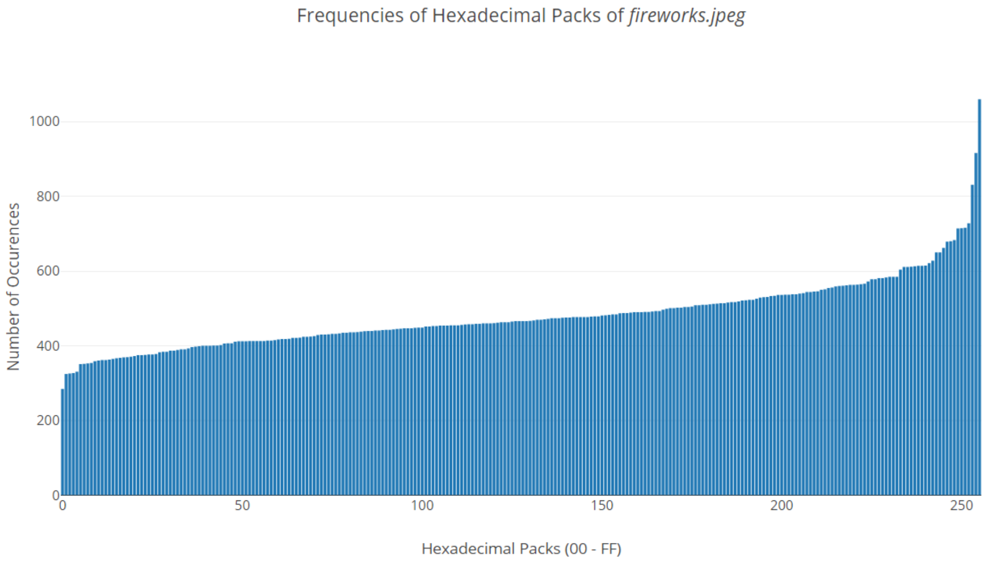

Figure 5.13: Histogram of hexadecimal frequencies of *fireworks.jpeg*.

As illustrated in Figure 5.13, another example is a file named *fireworks.jpeg* with a compression ratio of 0.785, which has an almost uniform histogram of hexadecimal frequencies. Frankly, the fact that *fireworks.jpeg* is an already compressed file is the reason for the poor compression ratio performance. However, the histogram for the benchmark file *sao*, given in Figure 5.14, would justify our point with a compression ratio of 1.14.

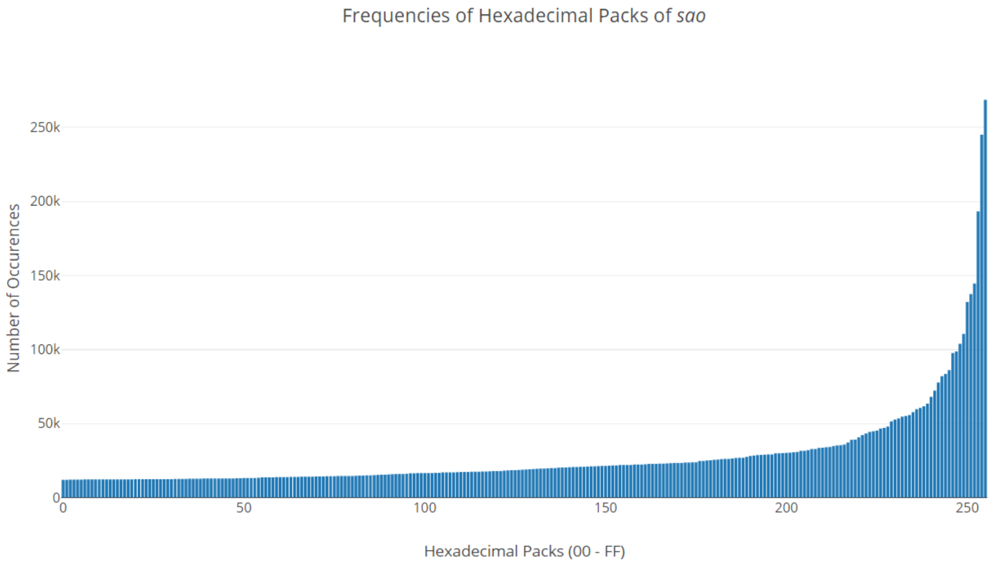

Figure 5.14: Histogram of hexadecimal frequencies of *sao*.

The main cause for the compression ratios for the given benchmark files to be distinct is absolutely the positions of hexadecimals. In the preprocessing stage, BWT transforms a given file block such that the same characters end up being closer after the operation. This process brings order to the inverted lists of hexadecimals. That is, the range of positions stored in inverted lists tends to shrink. This range even gets narrower after the Delta encoding stage, since it replaces original positions with gaps. As Moffat and Anh [5] argue in their study, BASC performs better in locally homogeneous sets of integers. In other words, for BASC to be effective, we should be able to use the same selector values for a given integer sequence. Thus, the worst case to employ BASC is when we have an oscillating integer sequence like *[1, 27864, 45, 688821, 127, ..]*. Therefore, EPIIC is expected to perform poorly on files that have the aforementioned property. A benchmark file called *mozilla*, a .tar file of executables, seems to have such a property since its compression ratio is just over 2 even though the fill value trick is immensely effective in this case. Corresponding histogram of hexadecimal frequencies is given in Figure 5.15. Note that, the most frequent hexadecimal occurs over 13 million times in *mozilla*, which means a staggering

13 megabytes of reduction in size due to fill value trick.

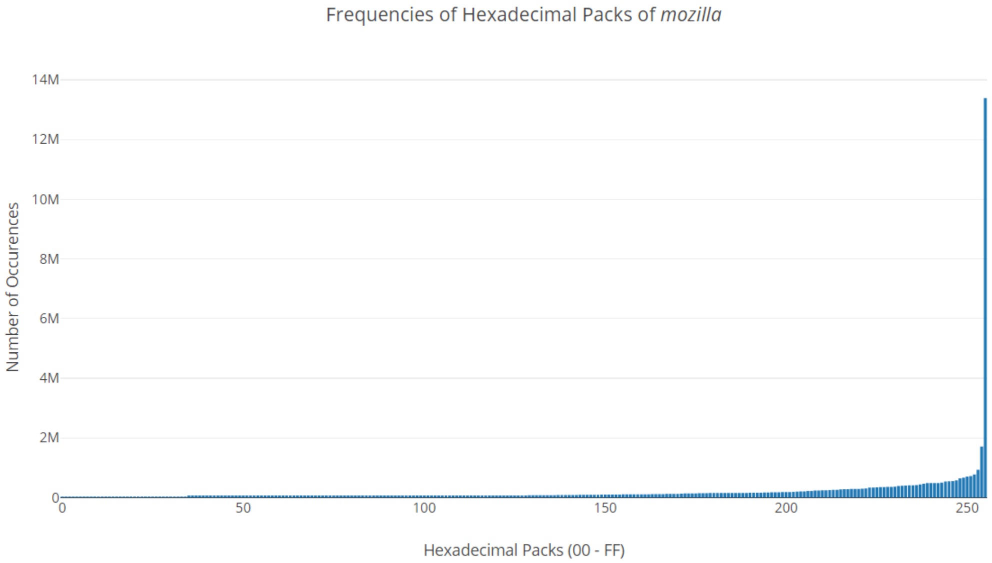

Figure 5.15: Histogram of hexadecimal frequencies of *mozilla*.

Another reason for EPIIC to compress some files better than others is the number of distinct hexadecimals found in a given file. There are 256 possible hexadecimal packs ranging from 00 to FF. However, a file does not necessarily have to include all of them. Intuitively, when a file has fewer distinct hexadecimal packs, this ultimately means that it has a lower entropy, hence it is more compressible. Also, this property affects and determines the position values stored in inverted lists. Fewer distinct hexadecimal packs suggest smaller gaps between recurrences. Figure 5.12 and 5.15 illustrate this issue. Note that while *nci* has just over 60 distinct hexadecimal packs, *mozilla* includes all possible 256 of them.

5.5 Compression/Decompression Speed

Another criterion for a good compression algorithm is its compression/decompression speed. While some algorithms prioritize compression ratio, others like Gipfeli [59] and Snappy [60] are optimized for fast execution speeds. A fast algorithm with decent compression ratios is desirable. However, finding that sweet spot in between is not trivial.

There are a lot of elements that determine the execution speed of an algorithm. Even though execution speed ultimately depends on the actual algorithm, it is possible to accelerate an implementation of an encoding by using special instruction sets such as single instruction, multiple data (SIMD) [61] and streaming SIMD extensions (SSE) [62]. Implementing a cache aware version of the algorithm is another way of speeding it up since it reduces cache miss rates and leads to faster compression/decompression. Performance of an algorithm also depends on implementation and the used programming language. C and C++ are among the languages which are preferred due to their intrinsic properties that make them fast. Most of the aforementioned compression algorithms are implemented in C and C++.

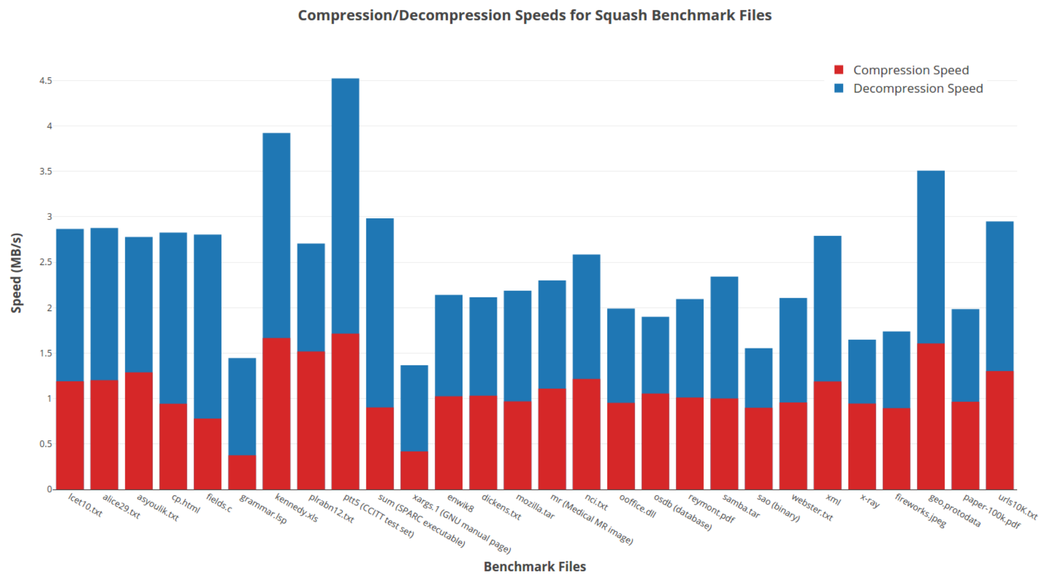

Figure 5.16: Compression/decompression speeds of EPIIC.

It is evident that the current version of EPIIC is no match for the other compression algorithms in terms of compression/decompression speed. Yet, considering that these algorithms are being developed for many years, EPIIC may be able to reach them in the coming versions of the algorithm. Nonetheless, we have inspected the relation between compression and decompression speeds of EPIIC. As seen in Figure 5.16, when the block size is 10 megabytes, decompression phase seems to be 2-3 times faster than compression. Also, note that EPIIC is implemented in C++11 and it is not yet optimized for execution speed. As stated before, this study is essentially a proof of concept and a further discussion of the subject is given in Chapter 6.

5.6 Effect of MTF on Compression Ratio

In the earlier versions of the algorithm, after transforming the input sequence with BWT, we have employed **Move To Front Transform (MTF)** [12] to make the data more compressible. In this section, we justify our reasons to discard MTF from our preprocessing pipeline.

MTF does not change the size of the input, yet it increases regularity by converting recurring characters into runs of zeros. The common MTF algorithm is applied to the data without any major alterations, except the fact that our alphabet contains the symbols *0-9* through *A-F*, as the process involves hexadecimal characters. Another slight difference is that we do not replace the characters with their integer indexes, instead we convert integers into hexadecimals too. That is, whenever the index of symbol of interest is greater than 9, we replace it by its hexadecimal representation. For example, if the symbol of interest is at index 11 in our symbol table, we replace the character with *'B'* instead of *'11'*.

During MTF, the symbol in question is replaced by its index in the symbol table. Since the symbol table is not static, we need to find the given symbol's position in the symbol table each time and insert it to the beginning of the table. To ease this update, we have utilized the *deque* data structure of C++, which is basically a double-ended queue. Hence, finding a symbol in a deque is $O(n)$ and inserting it to the deque is $O(1)$. Considering that this operation is performed for each symbol in the given character sequence, the time complexity is $O(n^2)$. Yet, since the alphabet we use is comprised of only 16 characters ($0-9$ and $A-F$), finding a symbol in such a tiny list is insignificant compared to the block size. Hence, time complexity of MTF is actually $O(16n)$, which is $O(n)$.

Although the time complexity of MTF is linear, this step considerably slows down the algorithm during both compression and decompression. Thus, we were curious about the effect of MTF on compression ratio and this led to an experiment. We have basically discarded MTF from our preprocessing stage and measured compression ratios for Squash benchmark files again. As illustrated in Figure 5.17, our version of MTF only slightly improves the compression ratio. Hence, since it simply does not worth the required time and resources, we have removed MTF transform from the preprocessing stage of EPIIC.

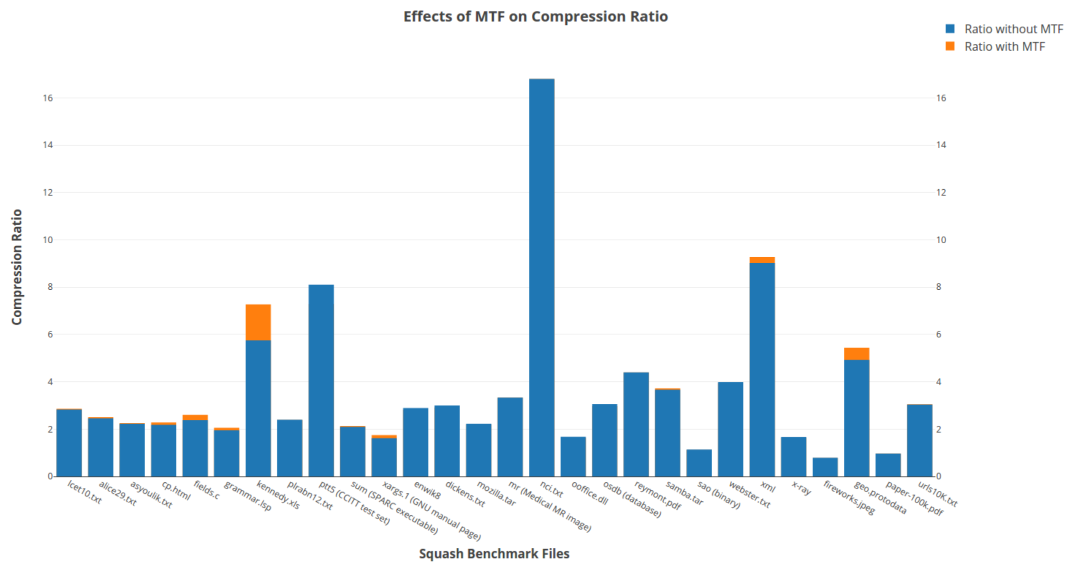

Figure 5.17: MTF's effect on compression ratio for Squash benchmark files.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The aim of this thesis was to create a lossless data compression algorithm with a brand new paradigm regarding compressing the positional information of hexadecimals that make up the file, and to study its performance and capabilities along with its constraints. Since our study constitutes evidence that such an approach is indeed effective, it may be followed by a group of next generation lossless data compression algorithms. Such algorithms may be referred as **positional information encoders**, **PIE**s, a term we have coined during this study.

In our study, we first briefly introduced the subject of data compression and mentioned about its historical background. In Chapter 2, we presented some encodings and algorithms that are commonly used in many compression algorithms. Later, we specifically elaborated on techniques implemented for inverted index compression and highlighted their strengths and shortcomings. In Chapter 4, we thoroughly explained our novel lossless data compression algorithm Encoding Pluggable Inverted Index Compression, EPIIC, that employs different encodings and transformations such as BWT, RLE, Delta and BASC. Lastly, we have demonstrated EPIIC's performance in terms of compression ratio, compression/decompression speed and memory usage and compared it to other compression algorithms.

Through our experiments, we found that in both text and .xml files, EPIIC performs on par with other compression algorithms such as brotli, crush, lzma and zlib in terms of compression ratio. As expected, it is not able to provide better compression ratios in files that are already compressed. Its performance on other types of files should be tested further.

At this stage, EPIIC runs with a compression speed around 1 MB per second, and a decompression speed around 2 MB per second. Hence, when compared to other lossless compression algorithms, it is safe to say that EPIIC is not competent in terms of execution speed and it is only faster than a few algorithms such as zpaq.

Moreover, EPIIC is measured to have an average space complexity of *O(50b)* for compression and *O(25b)* for decompression, where *b* denotes the block size in megabytes. Thus, in case of 10 MB blocks, EPIIC requires around 500 MB of memory during compression and 250 MB during decompression. Even though this does not pose a bottleneck for our algorithm, it eliminates the possibility of using EPIIC on devices that possess low memory.

To pave a way for the subsequent studies, we suggest some of the future work that may improve the algorithm thoroughly.

To begin with, we suggest an implementation of the algorithm using special instruction sets such as single instruction, multiple data (SIMD) [61] and streaming SIMD extensions (SSE) [62] as it is proven by many studies that these instruction sets speed up the algorithms by a considerable amount. In their study, Lemire and Boytsov demonstrated that it is possible to decode billions of integers per second by utilizing these special instruction sets. [36].

Another way to speed up the overall algorithm is to convert the current implementation into a cache aware one. Effects of cache awareness are presented in many studies [63, 64, 65]. Kowarschik and Weiß introduce and outline cache characteristics along with some cache optimization techniques in their study [66].

In addition to these suggestions, we propose implementing a multi-threaded version of both compression and decompression phases of EPIIC. Because it is a block based algorithm, it is trivial to achieve even faster compression/decompression with multithreading. Naturally, such a parallelized version will require more resources. Hence, reducing the memory footprint of the algorithm first is wiser. Nevertheless, considering the computational power and resources computers have nowadays, parallelizing EPIIC is completely feasible.

For further improvement of the algorithm we propose a couple more ideas concerning the encodings used in EPIIC. As discussed in Chapter 4, there are many suffix array sorting algorithms with different advantages. Instead of KS algorithm, a more suitable one may be utilized to upgrade the current BWT implementation.

In their study, Moffat and Anh [5] introduced some variations of BASC encoding. These variations may boost EPIIC in terms of compression ratio and compression/decompression speed. Although it is slow, binary interpolative coding [37] may be another alternative to BASC, due to its superior compression ratio. Recall that, experimenting with different encodings is straightforward and painless in EPIIC.

Lastly, although EPIIC is used in many different files during the experiments, the overall algorithm is not tested thoroughly. Hence, edge cases such as files with one or more blocks that are comprised of the same character, should be investigated and tested further.

In our study, we barely scratched the surface of a novel, inverted index based, lossless data compression algorithm and we believe that this innovative approach has much more to offer. Considering the storage needs and bandwidth constraints that grow day by day, novel approaches and techniques may prove their worth and strength in distinct use cases. In the coming versions, EPIIC can demonstrate its competence with better compression ratio, faster execution speed and lower memory usage. It may become a pioneer in this brand new technique of positional information encoding and it can lead to a completely new family of compression algorithms.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[2] H. Yan, S. Ding, and T. Suel, "Compressing term positions in web indexes," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 147–154, ACM, 2009.

[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[4] D. Pountain, "Run-length encoding.," *Byte*, vol. 12, no. 6, pp. 317–319, 1987.

[5] A. Moffat and V. N. Anh, "Binary codes for locally homogeneous sequences," *Information Processing Letters*, vol. 99, no. 5, pp. 175–180, 2006.

[6] S. M. Pincus, "Approximate entropy as a measure of system complexity.," *Proceedings of the National Academy of Sciences*, vol. 88, no. 6, pp. 2297–2301, 1991.

[7] C. Bandt and B. Pompe, "Permutation entropy: a natural complexity measure for time series," *Physical review letters*, vol. 88, no. 17, p. 174102, 2002.

[8] A. N. Kolmogorov, "Three approaches to the quantitative definition ofinformation'," *Problems of information transmission*, vol. 1, no. 1, pp. 1–7, 1965.

[9] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[10] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h. 264/avc video coding standard," *IEEE Transactions on circuits and systems for video technology*, vol. 13, no. 7, pp. 560–576, 2003.

[11] K. Brandenburg and G. Stoll, "Iso/mpeg-1 audio: A generic standard for coding

of high-quality digital audio," *Journal of the Audio Engineering Society*, vol. 42, no. 10, pp. 780–792, 1994.

[12] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[13] R. Mohanty and S. Tripathy, "An improved move-to-front (imtf) off-line algorithm for the list accessing problem," *arXiv preprint arXiv:1105.0187*, 2011.

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

[15] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, vol. 6, no. 2, pp. 163–180, 1985.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[18] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

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

[20] J. Alakuijala and Z. Szabadka, "Brotli compressed data format," tech. rep., 2016.

[21] L. P. Deutsch, "Gzip file format specification version 4.3," 1996.

[22] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[23] G. N. N. Martin, "Range encoding: an algolithm for removing redundancy from a digitised message," in *Video & Data Recording Conference, 1979*, 1979.

[24] J. Duda, “Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding,” *arXiv preprint arXiv:1311.2540*, 2013.

[25] G. Manzini, “The burrows-wheeler transform: theory and practice,” in *International Symposium on Mathematical Foundations of Computer Science*, pp. 34–47, Springer, 1999.

[26] S. Deorowicz, “Improvements to burrows–wheeler compression algorithm,” *Software: Practice and Experience*, vol. 30, no. 13, pp. 1465–1483, 2000.

[27] P. Ferragina and G. Manzini, “An experimental study of an opportunistic index,” in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pp. 269–278, Society for Industrial and Applied Mathematics, 2001.

[28] J. Kärkkäinen, P. Sanders, and S. Burkhardt, “Linear work suffix array construction,” *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 918–936, 2006.

[29] G. Manzini and P. Ferragina, “Engineering a lightweight suffix array construction algorithm,” *Algorithmica*, vol. 40, no. 1, pp. 33–50, 2004.

[30] L. H. Thiel and H. Heaps, “Program design for retrospective searches on large data bases,” *Information Storage and Retrieval*, vol. 8, no. 1, pp. 1–20, 1972.

[31] V. N. Anh and A. Moffat, “Inverted index compression using word-aligned binary codes,” *Information Retrieval*, vol. 8, no. 1, pp. 151–166, 2005.

[32] 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*, pp. 401–410, ACM, 2009.

[33] S. Büttcher, C. L. Clarke, and G. V. Cormack, *Information retrieval: Implementing and evaluating search engines.* Mit Press, 2016.

[34] M. Zukowski, S. Heman, N. Nes, and P. Boncz, “Super-scalar ram-cpu cache compression,” in *22nd International Conference on Data Engineering (ICDE’06)*, pp. 59–59, IEEE, 2006.

[35] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *Data Engineering, 1998. Proceedings., 14th International Conference on*, pp. 370–379, IEEE, 1998.

[36] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.

[37] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[38] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[39] D. Salomon and G. Motta, *Handbook of data compression*. Springer Science & Business Media, 2010.

[40] D. Blandford and G. Blelloch, "Index compression through document reordering," in *Data Compression Conference, 2002. Proceedings. DCC 2002*, pp. 342–351, IEEE, 2002.

[41] M. U. Bokhari and F. Hasan, "Multimodal information retrieval: Challenges and future trends," *International Journal of Computer Applications*, vol. 74, no. 14, 2013.

[42] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *Proceedings of the 17th international conference on World Wide Web*, pp. 387–396, ACM, 2008.

[43] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," in *Proc. Data Compression Conference*, 2000.

[44] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, "Compression of inverted indexes for fast query evaluation," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 222–229, ACM, 2002.

[45] A. Trotman, “Compressing inverted files,” *Information Retrieval*, vol. 6, no. 1, pp. 5–19, 2003.

[46] S. E. Deering, “Internet protocol, version 6 (ipv6) specification,” 1998.

[47] S. J. Puglisi, W. F. Smyth, and A. Turpin, “The performance of linear time suffix sorting algorithms,” in *Data Compression Conference, 2005. Proceedings. DCC 2005*, pp. 358–367, IEEE, 2005.

[48] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, “Better external memory suffix array construction,” *Journal of Experimental Algorithmics (JEA)*, vol. 12, pp. 3–4, 2008.

[49] G. Nong, S. Zhang, and W. H. Chan, “Linear suffix array construction by almost pure induced-sorting,” in *Data Compression Conference, 2009. DCC’09.*, pp. 193–202, IEEE, 2009.

[50] G. Nong, S. Zhang, and W. H. Chan, “Two efficient algorithms for linear time suffix array construction,” *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, 2011.

[51] G. E. Blelloch, “Introduction to data compression,” *Computer Science Department, Carnegie Mellon University*, 2001.

[52] A. Farruggia, P. Ferragina, and R. Venturini, “Bicriteria data compression: efficient and usable,” in *European Symposium on Algorithms*, pp. 406–417, Springer, 2014.

[53] E. Nemerson, “Squash compression benchmark.”

[54] M. Powell and T. Bell, “The canterbury corpus.”

[55] S. Deorowicz, “Silesia compression corpus.”

[56] C. M. Sadler and M. Martonosi, “Data compression algorithms for energy-constrained devices in delay tolerant networks,” in *Proceedings of the 4th international conference on Embedded networked sensor systems*, pp. 265–278, ACM, 2006.

[57] J. Alakuijala, R. Obryk, O. Stoliarchuk, Z. Szabadka, L. Vandevenne, and J. Wassenberg, “Guetzli: Perceptually guided jpeg encoder,” *arXiv preprint arXiv:1703.04421*, 2017.

[58] J. Alakuijala and R. Obryk, “Guetzli.”

[59] R. Lenhardt and J. Alakuijala, “Gipfeli-high speed compression algorithm,” in *Data Compression Conference (DCC), 2012*, pp. 109–118, IEEE, 2012.

[60] J. Dean, S. Ghemawat, and S. H. Gunderson, “Snappy.”

[61] R. J. Gove, K. Balmer, N. K. Ing-Simmons, and K. M. Guttag, “Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation,” May 18 1993. US Patent 5,212,777.

[62] S. K. Raman, V. Pentkovski, and J. Keshava, “Implementing streaming simd extensions on the pentium iii processor,” *IEEE micro*, vol. 20, no. 4, pp. 47–57, 2000.

[63] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, “Cache aware optimization of stream programs,” *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 115–126, 2005.

[64] N. Guan, M. Stigge, W. Yi, and G. Yu, “Cache-aware scheduling and analysis for multicores,” in *Proceedings of the seventh ACM international conference on Embedded software*, pp. 245–254, ACM, 2009.

[65] R. E. Ladner, R. Fortna, and B.-H. Nguyen, “A comparison of cache aware and cache oblivious static search trees using program instrumentation,” in *Experimental Algorithmics*, pp. 78–92, Springer, 2002.

[66] M. Kowarschik and C. Weiß, “An overview of cache optimization techniques and cache-aware numerical algorithms,” *Algorithms for Memory Hierarchies*, pp. 213–232, 2003.