Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science
and Engineering

Diploma Thesis

FM-Index Implementation

Tomáš Lakatos

Supervisor Ing. Jan Holub, Ph.D.

Master Study Program: Electrical Engineering and Information Technology

Specialization: Computer Science and Engineering

January 2008

Prohlášení

Anotace

FM-index (*Full-text* index in *Minute* space) je indexová struktura nad libovolným textem T. Využíva vlastnosti Burrows-Wheelerova transformace ke snížení místa potřebného na uložení indexu a k podpoře operací *count(P)* a *locate(P)* pro zjištení počtu a pozici výskytů vzoru P v textu T a *extract*, která umožňuje získat zpět celý původní text nebo jeho libovolnou část.

V této práci prezentujeme přehled datových struktur a algoritmů použitých FM-indexem se zaměřením na současný výzkum konstrukce Suffix Array, jehož poznatky lze využít pro časově a paměťově efektivní konstrukci FM-indexu. Také navrhujeme implementaci této struktury a porovnáme naše výsledky s programy které mají podobný účel.

Abstract

The FM-index (*Full-text* index in *Minute* space) is a self-index structure over an arbitrary text T. It utilizes properties of the Burrows-Wheeler Transformation to decrease the space needed to store the index, and to support the operations *count(P)* and *locate(P)* for counting and locating the occurrences of the pattern P in the text T, and the operation *extract* for extracting the whole original text or any part of it.

In this report we give an overview of the data structures and algorithms used by the FM-index, with special focus on recent research of Suffix Array construction, which can be used in space and time efficient construction of the FM-index. We also propose an implementation of the FM-Index structure, and compare our results to other programs for similar purposes.

Acknowledgements

I would like to thank my supervisor Ing. Jan Holub, Ph.D. for the idea of the thesis, and for his valuable help not only with the thesis but also with the technical details of the formatting. I also owe him thanks for his patience and understanding during the whole course of development.

I would like to thank my friend Martin Štrbáň for consulting the thesis with me and helping me put my thoughts in order.

I owe great thanks to my family for their support, especially my mother who was willing to listen to my thoughts, theories and explanations, helping me with her encouragement and her remarks and corrections.

I would like to express my gratitude to Dávid Ádám and Daan van Yperen for finding the time to read this thesis and correct many of my errors and mistakes. Their work was indispensable for making the text understandable and presentable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Indexing text to speed up searching for patterns has been a hot topic for almost two decades. The exponential growth of the amount of text data is putting higher and higher pressure on the speed of both building the indices and searching for patterns. Also, when indexing very large files, the memory occupancy of the process of building the index becomes important.

The most efficient approach considering the speed of pattern matching is the suffix tree, which stores the text suffixes in a (sorted) tree structure, thus supporting search and locate operation in time linear with pattern length. On the other hand, its space occupancy of both memory and storage space makes it unacceptable in most applications.

The suffix arrays considerably improve the space occupancy of an index compared to suffix trees, at a price of a small slowdown in the search time. However, in their primitive implementation, they still require the storage of the original text, as well as a data structure that occupies asymptotically the same space as the text itself.

The FM-index, described by Ferragina and Manzini in [6] and [7] is a full-text index structure based on the Burrows-Wheeler transformation [5]. It exploits the compressibility of the transformed text and its ability to represent the suffix array of the text to create a kind of *compressed suffix array*. With this compressed representation of the original text and some auxiliary data structures the FM-index is able to support fast searching for patterns in the original text.

1.2 State of the Art

An implementation of the FM-index is presented in [7] by Ferragina and Manzini. This implementation is optimized mainly for English texts and documents of similar properties, using several algorithms that have very good performance for typical cases, but perform poorly in the worst case situation. While the executable files are freely downloadable from the Internet, the source codes of this implementation are not readily available.

1.3 Contribution of the Thesis

In this thesis we present an implementation of the FM-index that can be used for research purposes. Where it was applicable, we tried to include multiple approaches to implement some features, and to compare their performance among each other on both theoretical and experimental level. We also utilize results from recent research in the area of suffix array construction, which introduced a variety of methods that seem to have the potential of outperforming the algorithms used in the implementation of Ferragina and Manzini [7]. We also present an experimental comparsion of our implementation with other utilities for compression and compressed-domain pattern matching.

We hope that this document can serve not only as a description of our implementation, but also as a starting point and a reference work for those who decide to improve our program or write their own.

1.4 Organization of the Thesis

In Chapter 2 we define the basic terms used in this text. Chapter 3 explains the working of the Burrows-Wheeler Transformation and the FM-index in detail and gives an overview of the algorithms that can be used in the efficient construction and functioning of the FM-index. Chapter 4 describes our implementation of the FM-index. We also introduce some ideas to improve the performance of our implementation. The experiments conducted with our implementation are reported in Chapter 5. They include comparsion of various approaches to implement key parts of the FM-index and comparsion of our implementation as a whole with other programs of similar purpose. The last chapter summarizes our results, and contains suggestions for future research.

Chapter 2

Preliminaries

Definition 2.1 (Alphabet)**.** An *alphabet* Σ is a finite non-empty set of *symbols* (or *characters*). The size of the alphabet $|\Sigma|$ is the number of symbols in contains.

Definition 2.2 (Integer alphabet)**.** The alphabet $\Sigma = \{0, 1, \ldots, n-1\}$ is called an *integer alphabet* of size n.

Definition 2.3 (String)**.** A *string* over the alphabet Σ is an arbitrary sequence of symbols from Σ. The n-th symbol of the string w will be denoted $w[n]$, starting with zero. The term *text* used in this document is considered mathematically equivalent to *string.*

Definition 2.4 (Set of all strings)**.** The *set of all strings* over Σ is denoted Σ^*.

Definition 2.5 (Length of string)**.** The *length of string* w is the number of symbols in string $w \in \Sigma^*$ and is denoted $|w|$.

Definition 2.6 (Substring)**.** The string x is a *substring* of the string y, if $y = uxv$, where $x, y, u, v \in \Sigma^*$. We will denote substrings starting at position a and ending at position b in the string y as $y[a..b]$.

Definition 2.7 (Prefix)**.** The string x is a *prefix* of the string y, if $y = xv$, where $x, y, v \in \Sigma^*$.

Definition 2.8 (Suffix)**.** The string x is a *suffix* of string y, if $y = ux$, where $x, y, u \in \Sigma^*$.

Definition 2.9 (Cyclic shift)**.** The string x is a *cyclic shift* (*rotation*) of the string y if there exist two strings $u, v \in \Sigma^*$ such as $y = uv$ and $x = vu$.

Definition 2.10 (Lexicographic order)**.** Let $\leq$ be a total ordering over the alphabet Σ. The *lexicographic order* produced by the ordering $\leq$ is a total ordering over Σ^* such as:

1. $\varepsilon \leq w \qquad \forall w \in \Sigma^*$,
2. if $u, v \in \Sigma^*$, $|u| = 1, |v| = 1$, then $u \leq v \iff u[0] \leq v[0]$,
3. if $u = u_1u_2, v = v_1v_2 \in \Sigma^*$, $|u_1| = |v_1|$, then $u \leq v \iff u_1 \leq v_1 \vee (u_1 = v_1 \wedge u_2 \leq v_2)$.

Note 2.1 (Lexicographic ordering of integer alphabets)**.** For integer alphabets we use the *lexicographic order* produced by the natural order of symbols.

Definition 2.11 (Suffix array)**.** The *suffix array SA* is a sequence (array) of all suffixes v_n of a string w such as $w = uv_n, |u| = n \quad n \in \{0, 1, \dots, |w| - 1\}$, where $SA[x] \leq SA[y] \iff x \leq y$.

Definition 2.12 (Text index)**.** A *text index* is a data structure over a text T supporting fast queries for pattern searching and information retrieval. The strategies for constructing text indices are numerous and vary greatly. An example is a word-based index which can answer queries of type "Does the word w occur in the text T?" very fast. This kind of index is used for example in web search engines for quickly finding documents that are relevant to the user.

Definition 2.13 (Full-text index)**.** A text indices built over the text T supporting pattern searching operations, where the pattern can be an arbitrary substring of the original text, is called *Full-text index*.

Definition 2.14 (Self-index)**.** A *self-index* is a text index that does not need access to the indexed text for its function and supports the operation *Extract* for extracting the indexed text T (and optionally also its parts). Such an index can be used to represent the original text, eliminating the need to store T explicitly.

Definition 2.15 (Code, encoding, decoding)**.** A *code* is an ordered triple (S, C, f), where S is the input alphabet, C is the output alphabet, and f is a one-to-one mapping of S to C^* called the encoding function. The application of the encoding function to a symbol from S to get a string from C^* is called *encoding*. Strings can be encoded using a code by encoding all its symbols and writing their codes sequentially (for $w \in S^*$ $f(w) = f(w[0])f(w[1]) \dots f(w[|w| - 1])$).

The code is called *unambiguous* if for all $y \in C^*$ there exists at most one string $x \in S^*$ such as $f(x) = y$. The process of finding for a given $y \in C^*$ the string $x \in S^*$ such as $f(x) = y$ is called *decoding*.

Definition 2.16 (Data compression)**.** A *(lossless) data compression* method is a *code* for which the encoded representation $y \in C^*$ of the text $x \in S^*, f(x) = y$ requires less storage space than the original text x.

Note 2.2 (Data compression)**.** In the sense of this formal definition, any code that reduces the space requirements for at least one string can be considered a compression method. However, when talking about compression algorithms we are considering codes that reduce the space requirement for the *majority* of the strings it will be applied to.

Note 2.3 (Storage space)**.** The storage space required by the strings is a function of the encoding used to store the original string and its encoded representation on a medium. For our purposes we will consider that the strings are stored on media using an encoding where each symbol of the alphabet Σ occupies $\log_2 \Sigma$ bits.

Definition 2.17 (Compression ratio)**.** The *compression ratio cr* of a code encoding the text $x \in S^*$ is the ratio of space requirements for the original text x and its encoded representation $f(x) \in C^*$ ($cr = \frac{|x| \log_2 |S|}{|y| \log_2 |C|}$).

Note 2.4 (Text corpora for evaluating compression methods)**.** The reader will note that the compression ratio, which is used as an evaluation measure for compression methods depends not only on the code used, but also on the encoded string. Considering that compression methods generally vary greatly in efficiency (compression ratio) depending on the input text, with the growing amount of methods for text compression it became necessary to construct a collection of text that can be used for fair evaluation of compression methods.

The *Calgary Corpus* is a collection of typical texts from real life. It was created by Ian Witten and Tim Bell in the late 1980s for the purpose of evaluating lossless compression methods, and was used for this purpose during the 1990s and even up to this day. The *Canterbury Corpus* was designed in 1997 by Ross Arnold and Tim Bell to replace the Calgary Corpus. It contains files that were carefully chosen to represent the typical files used in the modern days.

Definition 2.18 (Information entropy)**.** The (zero order) *information entropy* (also called *Shannon entropy*) is a measure of information content in messages. Because any finite alphabet can be mapped to an integer alphabet, we can consider without loss of generality an information source that generates symbols of an integer alphabet Σ, $|\Sigma| = n$, each with probability $p(n)$ and independently on the symbols previously generated (source without memory).

The entropy of each generated symbol i defined as $H(i) = -p(i)\log_2 p(i)$ measures the minimal number of bits (symbols of a binary alphabet) necessary to encode this symbol in a message so the receiver can know with certainty which symbol was generated.

The information content in a text can be expressed as the sum of entropies of symbols in it, where the probability of a symbol i occurring in the text T can be expressed as $p(i) = \frac{count(i)}{|T|}$, where $count(i)$ is the number of occurrences of symbol i in the text T. The entropy of the text T thus becomes:

$$H(T) = -|T| \sum_{i=0}^{n-1} p(i)\log_2 p(i).$$

Because lossless data compression methods preserve information, the entropy of a text expresses the theoretical limit of its compressibility.

Definition 2.19 (k-th order information entropy)**.** Most information sources in reality do not meet the requirement that the generated symbols should be independent on previously generated ones. The definition of information entropy can be extended to such sources the following way.

Consider an information source generating symbols from an integer alphabet, with probability depending on its inner state. Suppose that the inner state of the source depends on the last k symbols it generated. The probability of a symbol $i \in \Sigma$ being generated in the state $s \in \Sigma^k$ is $p_s(i)$.

Looking at a string generated by this source we can judge these probabilities the following way. Let us construct the strings w_s as a concatenation of all symbols occurring in state s (immediately following the string s). The probability of each character in these strings can be expressed as $p_s(i) = \frac{count(i,w_s)}{|w_s|}$, where $count(i, w_s)$ refers to the number of occurrences of the symbol i in the text w_s. The sum of entropy over the whole string w_s

is exactly the zero-order entropy of this string $H_0(w_s) = -|w_s| \sum_{i=0}^{n-1} p_s(i) \log_2 p_s(i)$. By summing the entropies of these strings over all states, we get the definition of the k-th order entropy of a text:

$$H_k(T) = - \sum_{s \in \Sigma^k} |w_s| H_0(w_s)$$

.

Note 2.5 (k-th order entropy)**.** The k-th order entropy of a text is lower or equal to any lower order entropy of the same text, thus employing compression algorithms that take into account the memory of the information source can theoretically achieve better compression ratios. It should be noted however that the first k characters of the text are not considered in the definition of the k-th order entropy because the state in which these symbols occurred is not known.

Chapter 3

The FM-index

3.1 Basic Properties

The FM-index is a full-text self-index over an arbitrary text utilising the properties of the Burrows-Wheeler transformation to achieve sublinear space occupancy while supporting efficient pattern searches on the original text. The index structure itself contains a compressed representation of the Burrows-Wheeler transformed version of the original text, and some auxiliary data structures that enable the support of pattern matching without decompressing the whole text.

The index occupies $O(H_k(T)) + O\left(\frac{|T|}{bsize}\right) + O\left(\frac{|T|}{mdist}\right)$ bits of storage space, where $bsize$ and $mdist$ are constants chosen at the time of building the index. The operations supported by the index are $Count(P)$ for counting the occurrences of the pattern P in the text T in $O(|P|bsize)$ time, $Locate(P)$ for locating the starting position of all occurrences of pattern P in the text T in $O(occ \cdot bsize \cdot mdist)$ time (occ is the number of occurrences of the pattern in the text), and the operation $Extract(a, b)$ for extracting part of the original text (with starting position a and ending position b) in $O((b - a)bsize)$ time.

The choice of the constants $mdist$ and $bsize$ represents a trade-off between storage space and search time, giving a flexibility to the user to set them according to his specific needs to create big indices supporting fast pattern searching or tiny indices with limited ability to match patterns, or anything in between.

3.2 The Burrows-Wheeler Transformation

Let us create the matrix M from the text T using the following steps:

1. append to the end of T a special character $\#$ smaller than any other text character;
2. form all rotations (cyclic shifts) of the text $T\#$;
3. sort these rotations in lexicographic order.

In Figure 3.1 the construction of the matrix M over the example text “swiss_miss” is shown. This matrix can be represented by the suffix array over the text, because each row can be represented by the starting position of that row in the original text. We now show

Rotations of T	Sorted rotations
	F L
swiss_miss#	#swiss_miss
wiss_miss#s	_miss#swiss
iss_miss#sw	iss#swiss_m
ss_miss#swi	iss_miss#sw
s_miss#swis	miss#swiss_
_miss#swiss	s#swiss_mis
miss#swiss_	s_miss#swis
iss#swiss_m	ss#swiss_mi
ss#swiss_mi	ss_miss#swi
s#swiss_mis	swiss_miss#
#swiss_miss	wiss_miss#s

Figure 3.1: Example of the Burrows-Wheeler transform

that the last column of this matrix (denoted L or T_{BWT}) is also sufficient to represent the original text, as well as the matrix M.

Before the lexicographic sorting of the rows, the rows of matrix M contains all rotations of the original text. Since this matrix is symmetric towards transposition, the columns of this matrix also contain rotations of the original text. After the sorting operation shuffles the rows, all columns of the matrix M contain permutations of the original text, and the first column in particular contains the characters of the original text in lexicographic order.

Thus, if we use the last column to represent the text, we can produce the first column by sorting all of its characters. Because all rows of the matrix are rotations of the original text, the character on the last column must originally precede the character on the first column in each row, thus we are able to produce all two character long factors of the original text. By sorting these character pairs in lexicographic order we can produce the first two columns of the matrix (see Figure 3.2).

By repeating this step it is possible to fill up the matrix M column by column by sorting the n-tuples that are produced in the n-th step, and putting before them the character on the last column of the according row to produce $n+1$-tuples. This shows that the last column of the text is sufficient to represent the original text.

The process of sorting the text to get the last column of matrix M is called the Burrows-Wheeler transformation, and we denote this last column as T_{BWT}, or Burrows-Wheeler transformed text. The BWT-ed text can be constructed from the suffix array by writing $T_{BWT}[n] = T[SA[n]-1]$ for each $n \in \langle 1, |T|-1 \rangle$ and $T_{BWT}[SA[0]] = \#$.

Of course this approach to reconstruct the original text would be very inefficient, especially considering the fact that we would like to avoid the unnecessary representing of the matrix M explicitly. Luckily there is a more efficient method of getting back the original text from the BWT-ed representation.

Note that the characters in the BWT-ed representation are ordered by the lexicographic order of the suffixes following them in the original text. That means that the number of the row starting with the same instance of a particular character *ch* at the

Sorting text characters to produce the first column	Sorting pairs to produce the second column
`F         L`	`F         L`
`#.........s`	`#s........s`
`_.........s`	`_m........s`
`i.........m`	`is........m`
`i.........w`	`is........w`
`m........._`	`mi........_`
`s.........s`	`s#........s`
`s.........s`	`s_........s`
`s.........i`	`ss........i`
`s.........i`	`ss........i`
`s.........#`	`sw........#`
`w.........s`	`wi........s`

Figure 3.2: First two steps of inverse transformation

position i in the BWT-ed text can be computed as $C[ch] + occ(ch, i-1)$, where $C[ch]$ is the amount of characters lexicographically smaller than ch in the text, and $occ(ch, n)$ is the number of occurrences of character ch in the transformed text up to the n-th position (in $T[0, n]$). This formula gives the exact number of suffixes that are lexicographically smaller then the suffix starting with the character $ch = T_{BWT}[i]$, because all such suffixes start with a character smaller than ch or with character ch that is before the i-th position in T_{BWT}.

We call the application of this formula the *LF-mapping* or last-to-first mapping (denoted $LF(i)$), because it maps the row that contains a particular character as the last one to the row that contains it as the first one. We also know that the last character in row $LF(i)$ precedes the first character in the same row in the original text. That means, that by applying the LF-mapping to the character $T_{BWT}[i]$ we get the character $T_{BWT}[LF(i)]$ that precedes it in the original text.

The first row in the matrix M necessarily has to start with the end of text character #, because it is lexicographically the smallest. This means that the last character on the first row (the first character in T_{BWT}) is the last character of the original text T. By applying the LF-mapping repeatedly to this character we are able to get back the original text, starting with the last character, and moving one character for each LF-mapping towards the start.

Note 3.1. The original implementation of the Burrows-Wheeler transformation [5] performs one pass over the BWT-ed text to compute two arrays that support the LF-mapping operation in constant time. The array $C[ch]$, $ch \in \Sigma$ contains for each character the total number of characters in the text that are lexicographically smaller or equal than ch, and the array $P_1[i]$, $i \in \{0, 1, \ldots, |T|-1\}$ that contains the number of occurrences of the character $T_{BWT}[i]$ in $T_{BWT}[0..|T|-1]$.

3.3 Supporting the Pattern Matching Operations

The goal of the FM-index is fast support of pattern matching operations *Count*(P) and *Locate*(P) for counting the occurrences of pattern P and finding their starting positions in the original text, and the operation *Extract*(a, b) for extracting substrings of the original text. In this section we describe the basic ideas behind the FM-index introduced by Ferragina and Manzini in [6] and [7] to augment the ability of the Burrows-Wheeler transformation to support these operations.

3.3.1 The Operation Count()

To provide support for the LF-mapping operation, a representation of the first column of the matrix M and the function $occ(ch, n)$ to count occurrences of the character ch in the string $T_{BWT}[0..n]$ are necessary. The array C used in the inverse BWT to represent the first column is small enough to be constructed at the time of building the index and stored in the index in $O(|\Sigma|)$ space.

The array P_1 representing the function occ in the inverse BWT however is not fit for our purposes, because the time complexity of building it is linear with text size, and we want to achieve sublinear time bounds for pattern matching. Shifting its computation to the time of constructing the index would solve this problem, but it would introduce the necessity of storing this array with the index in $O(|T|)$ space.

In [6] Ferragina and Manzini introduce the idea of constructing a different array to support the LF-mapping that requires less storage space while retaining the ability of fast LF-mapping computation. We logically partition the BTW-ed text into $\frac{T}{bsize}$ blocks of size $bsize$ at construction time, and store the number of occurrences of the character ch before the i-th block of the text in the array $P_2[i][ch]$. This array requires $O\left(|\Sigma|\frac{T}{bsize}\right)$ space to store. If the block size is significantly larger than the alphabet size, this array can be stored in less space than the array $P_1[i]$ used in the implementation of the BWT.

With the help of the new array P_2, the number of occurrences $occ(ch, n)$ of the character ch in the text $T_{BWT}[0..n]$ can be retrieved in time proportional to the block size in two steps. We can divide $T_{BWT}[0..n]$ into two parts in such way that the first part $T_{BWT}[0..k]$ contains $\lfloor\frac{n}{bsize}\rfloor$ whole blocks and the remaining part $T_{BWT}[k+1..n]$ contains a prefix of the block $\lfloor\frac{n}{bsize}\rfloor + 1$. The number of occurrences of ch in the first part can be retrieved in constant time from the entry $P_2[\lfloor\frac{n}{bsize}\rfloor][T_{BWT}[n+1]]$, and the occurrences in the second part can be counted by parsing the according substring in $Obsize$ time.

Note 3.2. The LF-mapping computation always calls the function occ in the form $occ(T_{BWT}[n], n-1)$. This means that if the character ch does not occur in a block i, no calls to occ that would require retrieving the entry $P2[i][ch]$ can happen, thus making this entry obsolete. The possibility to utilise this fact for reducing the index size will be addressed later in the text.

This array, together with the array C supports fast computation of the LF-mapping (in constant time if $bsize$ is a constant) without the need of storing excessive amounts of data in the index. Now we show how the ability of computing the LF-mapping is sufficient for supporting the operation *Count*(P).

```
Algorithm $Count(P)$
1) $i = |P| - 1$, $c = P[i]$, $sp = C[c]$, $ep = C[c+1] - 1$
2) while $((sp \leq ep) and (i \geq 3))$ repeat 3) – 6)
3)     $c = P[i-1]$
4)     $sp = C[c] + occ(c, sp - 1) + 1$
5)     $ep = C[c] + occ(c, ep)$
6)     $i = i - 1$
7) if $(ep < sp)$ then return 0 else return $(ep - sp + 1)$
```

Figure 3.3: Pseudo-code of the *Count*(P) operation.

Lemma 3.1. The arrays C and P_2 together with the BWT-ed text are sufficient for counting the occurrences of an arbitrary pattern P in the original text T.

Proof. Note that since the matrix M contains all text suffixes, there must be exactly one row starting with the pattern P for each occurrence of this pattern in the text. Also, since the rows are sorted in lexicographic order, all rows starting with P must occupy a contiguous region in the matrix, so it is sufficient to find the first and the last such row. In case that P contains a single character $P[0]$, the first and the last row starting with this character (denoted sp and ep) can be found using the array C, which can be considered a representation of the first column of the matrix. ($sp = C[P[0]] + 1$, $ep = C[P[0] + 1]$)

For patterns longer than a single character, we can apply a recursive approach. If we know which rows start with the pattern $P[1..|P| - 1]$ (let sp and ep denote the number of the first and the last such row, respectively), we can find the rows that start with P by applying the LF-mapping to all such rows that end with the character $P[0]$. Because after the LF-mapping these rows occupy a contiguous region of the matrix, it is sufficient to find only the first and the last row starting with $P[1..|P| - 1]$ and ending with $P[0]$.

Let us denote these rows with x and y, then it holds that the first row starting with the pattern P is $LF(x) = C[P[0]] + occ(P[0], x - 1)$ and the last row starting with P is $LF(y) = C[P[0]] + occ(P[0], y - 1)$. Because sp is the first row starting with P[1..|P|-1] and x is the first such row that ends with $P[0]$, it must be true that $P[0]$ does not occur between the positions sp and $x - 1$ in T_{BWT}. This means however, that the call $occ(P[0], x - 1)$ and $occ(P[0], sp)$ yield the same result. This eliminates the need to find the exact value of x, because it is only used when the call to occ is made, and in this call sp can be used instead of it with the same result.

A similar reasoning can be applied to show that ep can be used instead of y in the call to find the last row starting with pattern P, the only difference being that $P[0]$ occurs at the position y that precedes (or is equal to) the position ep, so we must subtract this singe occurrence when replacing $y - 1$ with ep in the function call ($occ(P[0], y - 1) = occ(P[0], ep) - 1$). □

The pseudo-code of the operation *Count*(P) (as per [7]) is shown in Figure 3.3.

```
Algorithm $Locate(i)$
1)  let $v = 0$ and $j = i$
2)  if row $j$ is marked return $pos(i) = v + pos(j)$
3)  let $j = LF(j)$ and $v = v + 1$ and go to step 2)
```

Figure 3.4: Pseudo-code of the *Locate*(i) operation.

3.3.2 The Operation Locate()

The operation *Locate*(P) first executes *Count*(P) to find all rows that start with pattern P. Given an occurrence of the pattern P in the text, represented by the number of the row in M starting with this particular occurrence, we would like to know its starting position in the original text. This operation is basically the inverse of the Suffix Array. (For row i we are looking for the number $pos(i)$ such that it holds that $SA[pos(i)] = i$.)

This could be accomplished without the need for further auxiliary data by an algorithm similar to the inverse BWT. As noted before, we know that the first row in matrix M starts with the terminating character #, so we know that $pos(0) = |T|$. By applying the LF-mapping we find the row k that starts with the character preceding # in the original text ($pos(k) = |T| - 1$). By repeating this step until $k = i$ and counting the amount of LF-mapping steps v needed to accomplish this, we can say that $pos(i) = |T| - v$.

However this process requires in the worst case $v = |T|$ steps to locate the first character in the original text. This is because at the start of the algorithm, the only known value of the *pos* function is the last character of the original text. If we could provide several entry points into the LF-mapping process for which the position in the original text would be known, we would find locations of the pattern in the original text faster.

To this end, the database of the so called *marks* is constructed (see [6]). We logically mark a subset of rows from matrix M, for which we store their starting position in the original text. Supposing that the row that corresponds to the start of the original text is marked, Algorithm 3.4 can be used to find the starting position $pos(i)$ of the row i in the original text.

The row corresponding to the start of the text has to be marked to ensure that Algorithm 3.4 always finishes. We use the LF-mapping step to traverse the original text backwards until we find a position corresponding to a marked row, and return the sum of the position of this row and the amount of LF-mapping steps conducted.

Let k and l be two marked rows $pos(k) < pos(l)$ such as it holds that there is no row m ($m \neq k$ and $m \neq l$) for which $pos(k) < pos(m) < pos(l)$. We will call these marked rows neighboring, and we will call $dist(k, l) = pos(l) - pos(k)$ the distance between them.

The maximal number of the LF-mapping steps conducted in our algorithm is bounded by the maximal distance between two neighboring marked rows. We denote this distance *mdist*. The value of *mdist* and the speed of answering the two queries "Is row j marked?" and "What's the position of j in the original text?" from the algorithm depends on the strategy that is used to select the marked rows and the implementation of the Marks Database. We will further examine this matter in the part of this text devoted to implementation details.

3.3.3 The Operation Extract()

The operation *Extract*(a, b) is used to retrieve the substring $T[a..b]$ from the original text. If we assume that we can find the position i in the BWT-ed text that corresponds to the position $b = pos(i)$ in the original text, we can extract the substring $T[a..b]$ by starting at the row i and applying the LF-mapping $b - a$ times, retrieving one character for each step.

The Marks Database can be used for finding i. It is sufficient to find a marked row j that corresponds to the text position $pos(j)$ such as $pos(j) > b$, and apply the LF-mapping $pos(j) - b$ times to the j-th row. For maximal efficiency during this operation it is necessary to find the row whose distance from b is the smallest. Depending on the implementation of the Marks Database this operation ranges from constant time to time linear with the Marks Database size to find j, and additional $(b-a)+mdist$ LF-mappings to find i and extract the desired substring.

Note 3.3. This algorithm is primarily meant for extracting small portions of the original text, for example the immediate context of the searched patterns. For large text portions its use becomes inefficient, because the LF-mapping step using the array P_2 takes time proportional to the block size. For text portions considerably larger than the number of blocks in the text it becomes more efficient to construct the array P_1 instead of P_2.

3.4 Compressing the BWT Output

In the previous text we focused mainly to the speed of the pattern matching operations. The data structures introduced to augment the speed of pattern searches using the BWT-ed text occupy additional space. To reduce space requirements of the index, the output of the BWT should be compressed.

Apart from representing the matrix M, the important property of the Burrows-Wheeler transformation is that it groups together characters that appear in similar context (more specifically characters followed by suffixes that are lexicographically closest). In the most common real life texts (texts in natural languages, programming language sources, and even executable files just to mention a few) the probability of character occurrences is affected greatly by their context.

For example in English text the string “he ” (note the empty space at the end of the string) is most commonly preceded by the characters t, T, s, S and the space character. This means that in a matrix M build over an English text most rows starting with “he ” will end with these characters. This tendency of the BWT to group together several different characters can be considered valid for any text where the context partially defines the characters occurring, and can be exploited to improve compressibility of the text.

The tendency of the BWT to produce runs of the same character interleaved with several occurrences of a small selection of other characters can be augmented by employing a move-to-front encoder [3] to encode the BWT output. This encoder codes every character in a text with the number of different characters seen since its last occurrence. (The characters that were not seen yet are usually encoded according to an initial table or with exceptions.)

Applying this coding to the BWT output yields for most real life texts a large amount of zeroes interleaved mostly with small numbers. The occurrence of large numbers tends to be much less common. This output is well suited for compressing with statistical encoders.

In fact, applying the BWT and the move-to-front encoding enables the zero-order statistical encoder to encode the original text with efficiency close to k-th order statistical encoders, because the BWT groups characters with similar context (of size k up to the size of the whole text) and the move-to-front coder transforms long portions of text containing only several distinct characters into small numbers. The probability of occurrence decreases fast with increasing numbers, which can be exploited by all zero-order statistical compression schemes. It even makes it possible to predict the probability of the symbols occurring without first calculating an exact statistical model for the text being compressed.

The choice of the compression method has great impact on the resulting performance of the FM-index. To support the counting of occurrences in the Burrows-Wheeler transformed text avoiding the need to decompress the whole text, the ability to split the text into separately decompressible blocks must be possible. Also, every LF-mapping step requires a call to the function counting *occ*, which decompresses one block of the transformed text for each call to count occurrences in that block. Since all supported operations depend on the LF-mapping, their speed depends heavily on the speed of the decompression of blocks.

Block-wise decompression of the compressed text requires another external structure, called the *block index*, which stores the starting positions on the blocks in the compressed text. Also, the inner state of the compression algorithm must be known at the start of each block. This can be satisfied by saving the current inner state in the header of each block, or by reinitializing the inner state at the start of each block.

The compression rate of the algorithm together with the selection of constants *bsize* and *mdist* determines the resulting size of the index. The fact that both the decompression time and the index size grow with growing block size introduces a metric for comparing the suitability of different compression schemes for the purpose of being used in the FM-index. Slower algorithms with better compression ratios can be speeded up at the cost of storage space by choosing a smaller block size and vice versa. If we select the block size in such a way that two algorithms with different compression ratios result in roughly the same index size, we can compare their suitability for our needs by comparing the speed of the supported operations.

3.5 Sorting the Text Suffixes

The construction Burrows-Wheeler transformed text requires the lexicographic sorting of all suffixes of the text. This sorting operation is the bottleneck of the FM-index construction in both time complexity and memory usage. Sorting the text suffixes using the compare-and-exchange approach has a time complexity of $O(n \log n)$ comparsions, with various optimizations (like the one suggested by Burrows and Wheeler in [5]) only reducing the constant factor and/or the average time complexity of the comparsion of two strings.

While this sorting approach is optimal for data sets where the elements of the set are not related, the suffixes of a text are highly correlated, thus better approaches can be found for sorting them. In the last decades a large amount of research was devoted to fast and memory efficient construction of suffix arrays, which also require sorting of the text suffixes and can be transformed into the BWT in a singe pass over the text as shown earlier in this text.

Historically, the first algorithm to construct the suffix array was throught suffix tree construction. The suffix tree is constructed in linear time, then traversed (also in linear time) to construct the suffix array of the text. This approach is however seldom used in practice because of its complexity, the high memory requirements of the suffix tree, and because it is outperformed by most more recent algorithms.

Most of the algorithms to construct suffix arrays fit in one of two groups, those which are designed with minimal space occupancy in the mind (also often called lightweight algorithms), and those optimized for minimal worst case run time. Of course it is not possible to draw a distinct line between the two groups, as the more recent algorithms tend to perform better in both run time and memory occupancy than the older, more outdated ones.

Many of the algorithms with low memory occupancy have poor worst-case performance but are optimised for real life data, outperforming algorithms with lower worst case time bounds. The modified quicksort of Burrows and Wheeler in [5], the *doubling scheme* of Manber and Myers ([15]) the *qsufsort* of Larsson and Sadakane [14], the *deep-shallow sorter* of Ferragina and Manzini [8], the *difference cover* algorithm of Burkhardt and Kärkkäinen [4] and the *bucket-pointer refinement* algorithm of Schürmann and Stoye [16] are all based on this approach.

Recent breakthroughs in research produced algorithms that can build the suffix array of a text in linear time, and which are practically faster and occupy less space than the construction using suffix trees. In 2003, three independent solutions for constructing the suffix arrays in linear time were found. Ko and Aluru [13] use a recursive sorting algorithm (*S-L sort*), Kim et al. [12] use a binary divide-and-conquer approach and Kärkkäinen and Sanders [11] use a ternary divide-and-conquer algorithm. More recently, in [10] Hon et al. introduced an algorithm for linear time suffix array construction in $O(n)$ bits of working space.

In the following, an overview of these algorithms is presented. For comparsion purposes in terms of space occupancy, we assume an architecture where integers and pointers occupy 4 bytes and a constant alphabet with $|\Sigma| \leq 256$, so characters of the string can be stored in 1 byte.

In [5], Burrows and Wheeler suggest an optimised version of quicksort to accomplish sorting of text suffixes. The basic idea is to first radix-sort the suffixes by their first two characters, then quicksort the buckets produced by the radix pass. This basic idea performs poorly if the text has many repeated strings, so they also suggest improvements to eliminate this weakness. One is to sort suffixes starting with long runs of the same character separately, the other is to limit the depth of comparsion in quicksort, then double it should the need arise.

The worst-case time complexity of the algorithm is $O(n^2 \log n)$, since quicksort uses $O(n \log n)$ comparsions of suffixes, and each comparsion takes $O(n)$ time in the worst case. The expected run time with the improvements is much better, since comparsion time comes close to constant for most real data. The minimal space occupancy of the algorithm is $5n$ bytes, which is the theoretical minimum for any algorithm that stores an array of indices into the text (which are sorted), and the text itself.

Manber and Myers' sorting algorithm uses a bucket sorting scheme by first k characters of the suffixes, where k is doubled after each sorting pass. The first step is a radix-sort of the suffixes by their first character. In every other step, it sorts the buckets that compared equal in the previous step, using twice as long prefixes of the suffixes as in the step before, until every bucket contains only one suffix. The comparsion of the suffixes can be done in constant time by using the results from previous sorting steps and some auxiliary data structures. Thus, the resulting time complexity of the algorithm is $O(n \log n)$ in the worst case, and $O(n)$ expected run time can be reached with some improvements. The data structures that are used in the algorithm can be fit into $8n$ bytes including the original text and the output.

Larsson and Sadakane's *qsufsort* ([14]) improves Manber and Myers' sorting algorithm by using a ternary radix-sort as the sorting step, and by carefully avoiding comparsion of already sorted suffixes. It has the same worst-case performance as Manber and Myers' algorithm, and also the same space occupancy. However, its implementation is more clean and straightforward, and according to experimental results in [14], it is 5-10 times faster on real data than Manber and Myers' algorithm.

The basic idea of the *deep-shallow* sorter introduced in [8] by Ferragina and Manzini is to apply a different sorting algorithm for suffixes that have short common prefixes (shallow sorting), and for suffixes with long common prefixes (deep sorting). As a first step, radix-sort is used to split suffixes into $|\Sigma|^2$ buckets by their first two characters. Then these buckets are sorted separately with the deep-shallow sorter, using the data from already sorted buckets to reduce the amount of character comparsions. The Bentley-Sedgewick multikey quicksort with a capped depth of comparsion is used as the shallow sorter. The remaining unsorted suffixes are then sorted by means of the deep sorter, which is a collection of sorting algorithms coupled with an advanced heuristic that selects the most efficient of them (depending mostly on the amount of strings to be sorted) for sorting the current selection of suffixes.

The worst-case time bound of the algorithm is $O(n^2 \log n)$, but on real life data it outperforms *qsufsort* on almost all instances (see [16] and [8]). Its biggest advantage is that it is extremely lightweight, requiring only $5n + cn$ bytes of space, where $5n$ bytes are used to store the input and the output, and c is a constant chosen by the user at run time. In accordance with experiments documented in [8], the authors suggest that the choice of $c = 0.03$ offers a good trade-off between speed and space occupancy. Thus the final space occupancy of the deep-shallow sorter is $5.03n$.

The *difference cover* algorithm of Burkhardt and Kärkkäinen ([4]) operates by selecting a sample of the suffixes D, where $|D| = \sqrt{n}$. This selection is performed using the mathematical construct difference cover modulo $v = \log n$, (or any other $v \in 2, 3, \ldots, n$ for a space/time trade-off). The sorting of this selection ensures that all the remaining suffixes can be sorted by using the previous results, in a similar way that qsufsort and deep-shallow sorting uses data already available to speed up the sorting process.

The resulting worst-case run time of the algorithm is $O(vn + n \log n)$ and it occupies $O(n/\sqrt{v})$ space in addition to the $5n$ space required to store the input and the output. The choice of $v = \log n$ results in $O(n \log n)$ time complexity, and $O(n/\sqrt{\log n})$ space occupancy. Experiments on real data show that the difference-cover algorithm performs roughly as well as qsufsort, but occupies less space. It performs quite well on artificially constructed strings that render the deep-shallow sorter useless. The authors suggest, that by more efficient implementation the performance of this algorithm can be improved to match that of the deep-shallow sorter, thus eliminating its weakness of extremely bad performance on some data.

The *bucket-pointer refinement* (or *bpr*) method of Schürmann and Stoye ([16]) also relies on reusing the already sorted suffixes to speed up the sorting. The main difference between it, and the algorithms described so far is that the *bpr* makes use of not only the totally, but also partially sorted suffixes. It also applies advanced heuristics in the choice of the sorting algorithm, in a similar way as Ferragina and Manzini's deep sorter.

The theoretical upper bound in run time of the *bpr* is $O(n^2)$, but the authors state that this is a provable pessimist estimate of the upper bound, and that the proof of any better bound of the run time remains an open question. The experimental run times are comparable to those of the deep-shallow sorter on real life data. On artificial data with extremely high LCP (on which the deep-shallow sorter fails) it performs comparably to algorithms with a linear asymptotic run time. The space occupancy of the structures used is $8n$ including the space for the input and the output.

From the family of linear time sorting algorithms, the *S-L sort* of Ko and Aluru [13] uses a recursive approach. First the suffixes are classified in groups such as the i-th suffix is classified as S if $T[i] < T[i+1]$ and as L otherwise. This classification can be done in linear time. Given the sorted order of one of these groups, it is possible to sort all suffixes in linear time. The recurrent step of the algorithm consists of numbering the so called S-type substrings of T (assuming S type suffixes are being sorted) that have the form of $T[i, j]$, where $i < j$ and i, j are the starting position of S type suffixes, and $\forall x \in (i, j)$ x is the starting position of a L type suffix. The S-type substrings are replaced by their numbers, and the algorithm is performed recursively until the trivial case.

The authors state, that this algorithm runs in linear time, because the run time obeys the recurrent formula $T(n) = T(\lceil \frac{n}{2} \rceil)) + O(n)$, where $T(1) = O(1)$. However, they supply no experimental data, which makes it hard to judge the efficiency of the algorithm without first implementing it. The space occupancy of the algorithm is $O(n \log n)$.

Kärkkäinen and Sanders' *skew* algorithm ([11]) is based on Farach's scheme for linear time suffix tree construction. It makes use of recursion similarly to S-L sort, first counting the suffix array of the suffixes with starting positions i, where $i \bmod 3 \neq 0$ recursively, then using this result to construct the suffix array of the remaining suffixes. Finally, it merges the two suffix arrays.

The algorithm builds the suffix array in linear time in the worst case, but according to experiments in [16], it is outperformed by both the *bpr* and the *deep-shallow sort* on real data. Also it is about two times slower than the divide and conquer algorithm of Kim et al. The space occupancy of the algorithm is $O(n \log n)$.

In [12], Kim et al. propose a divide and conquer based algorithm that has a similar basic concept as the skew algorithm. They divide the suffixes into odd and even ones. The suffix array of the odd suffixes is constructed recursively using the same approach. The suffix array of the even suffixes is constructed in linear time from the odd suffix array. Finally, the two arrays are merged to form the suffix array of the original text. The merge step is non-trivial in this case, and it requires the support of auxiliary information (constructed in linear time) to fit in the linear time-bound.

According to experiments of Schürmann and Stoye [16], the divide and conquer algorithm outperforms the *skew* algorithm in almost every case. The price for this is paid in slightly larger space occupancy, and by more complex implementation. The asymptotic run time and space occupancy are the same as of the skew algorithm. Note that the *bpr* and the *deep-shallow sort* still perform better on real data.

The most compact one of all linear-time construction algorithms is proposed in [10] by Hon et al. It exploits the divide and conquer approach to incrementally construct the compressed suffix array of a text. The actual construction process is quite complex. Amongst algorithms with linear time complexity, this one is the first to achieve linear space occupancy. Sadly, the lack of experimental data at this time makes it hard to compare the performance of this method with others.

From these algorithms, we the *deep-shallow sorter* of Ferragina and Manzini [8] and the *skew* algorithm of Kärkkäinen and Sanders in our implementation. These two algorithms complement each other well. We can use the *deep-shallow sorter* in the general case, and the *skew* algorithm in the rare cases where the deep-shallow sorter fails to finish in reasonable time.

Including the algorithms *bpr* (Schürmann and Stoye [16]), *divide and conquer* (Kim et al. citeSAR8) and the linear time and linear space algorithm of Hon et al. [10] would improve the performance of our implementation at index construction time, and are left as a possible future improvement.

Chapter 4

Implementation

4.1 Development Platform

The main goal of this thesis was to produce an implementation of the FM-index that can be used for research purposes. The properties that were required include modularity, ease of modification, space and time efficiency and portability. The program was to be equipped with a command line interface and its output designed with ease of further processing in our experiments in mind.

The GNU/Linux platform was chosen for development purposes because it provides a wide range of easy to use and efficient tools for software development, batch processing, text processing, etc. We chose the programming language ANSI C for developing our implementation. It is the native language of the Linux operating system providing easy access to kernel functions for measuring time complexity and memory occupancy of our application. The availability of the wide range of ANSI C compilers for different platforms also ensures good portability.

For development and testing, the compiler *gcc* and the debugger *gdb* were used. The program was tested under a 32-bit little endian architecture. For exploiting improvements supported by 64-bit architectures further modifications of the used algorithms would be needed. Also, the 32-bit addressing of file pointers along with some other implementation details limits the file size our program is able to handle to about 4 gigabytes. This however will probably not inhibit the user of the program in most practical cases, and it should be more than sufficient for research purposes.

We tried to develop the application so it does not rely on machine endianness, but unfortunately we did not have access to big endian machines for testing purposes, so the functionality of the developed software can not be guaranteed under such platforms.

An implementation of the FM-index by Ferragina and Manzini is already freely available on the Internet, but it does not include the source codes. Because our main motivation was to to fill in the gap and produce a new implementation with available source codes for research purposes, we chose the algorithms to be implemented mainly for their value for research and experimenting over their efficiency where it was applicable. In many cases we tried to provide multiple algorithms and compare them instead of adapting an already available optimized implementation. This limits the usefulness of our implementation as a production-quality program for compression and indexing. The modularity of the implementation however should make it possible to replace parts of the program with highly optimized algorithms from other sources to improve its performance.

The appendix A of this document serves as an end user's manual for the developed application. The functioning of the application as a whole and of its parts is described in this chapter, hopefully in sufficient detail for providing an in-depth reference manual for those deciding to modify it for their own use.

4.2 General Overview

For the purposes of constructing the FM-index and pattern matching, the input file is considered to be a string over the integer alphabet $\{0, 1, 2, \ldots 255\}$, each byte of the input file representing one character of the string. This is in accordance with how text files are stored on most systems, allowing the FM-index to perform pattern matching in the way it conforms to the common understanding of pattern searching in texts. It also allows us to work with practically any input file as a stream of bytes without the need to understand what the file represents.

Our implementation of the FM-index consists of four main modules, the *Control Module*, the *Compression Module* that also is responsible for construction of the FM-index database, the *Decompression Module* that is used for extracting the whole original text from the compressed representation and the *Pattern Matching Module* that provides support for the pattern matching operations.

The *Control Module* provides a command line user interface for controlling the operation of the other modules. It processes the input from the command line, prepares the running of other modules by setting up the global parameters of the index, loads the input files in memory when appropriate and executes the module requested by the user.

The *Compression Module* calls the BWT sub-module to perform the Burrows-Wheeler transformation over the input text, building during this process the array C and the *Mark Database*. Then it calls the compression sub-module that compresses the BWT output and builds the array P_2 and the *Block Index*, and stores the FM-index structure in the output file.

The *Decompression Module* utilises the ideas presented by Burrows and Wheeler in [5] for efficient decompression of the original file. This method does not require the additional structures built during the compression apart from the compressed representation of the original text, and it is much faster for retrieving the whole original text than the *Extract*() operation of the FM-index. The module also provides an interface for block-wise decompression used by the Pattern Matching Module.

The *Pattern Matching Module* implements the operations $Count(P)$, $Locate(i)$ and $Extract(a, b)$ supported by the FM-index database. It relies on the function $occ(ch, n)$ that computes the occurrences of the character ch in the first n characters of the Burrows-Wheeler transformed text. This function accesses the Decompression Module to decompress one block and count character occurrences in it for each of its calls.

A sketch of the different modules and the FM-index database, including the data flow between the modules are depicted on Figure 4.1.

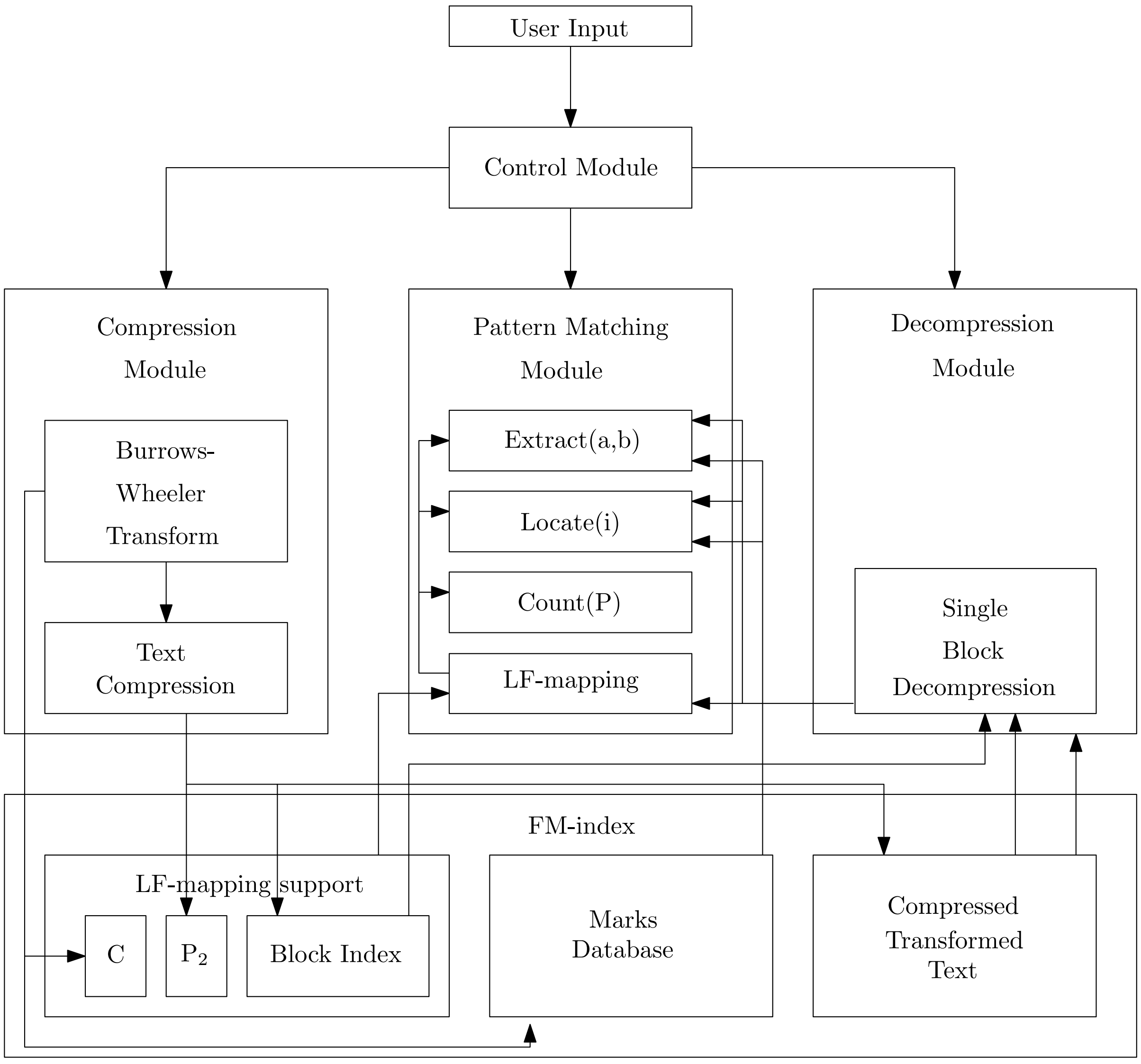

Figure 4.1: Block diagram of implementation modules and the FM-index structure

4.3 The Compression Module

The Compression Module is responsible for applying the Burrows-Wheeler transformation to the input text and for compressing the output of the BWT into separately decodable blocks. It also builds the auxiliary data structures required for efficient operation of the Pattern Matching Module during these steps.

After computing all required information, the FM-index database is stored in the output file specified by the user for use of the other modules. The output file format will be described in detail later in this chapter.

4.3.1 Constructing the BWT

The sub-module responsible for constructing the Burrows-Wheeler transformation performs any preprocessing of the input text as required, then calls an algorithm for suffix array construction over the original text. From the large amounts of algorithms available for suffix array construction (3.5 in Chapter 3) we have chosen to include two, namely the *dssort2* algorithm of Ferragina and Manzini [8] and the *skew* algorithm Kärkkäinen and Sanders [11].

The choice of these two algorithms is left to the user and can be made using the command line interface. The *dssort2* algorithm has low memory occupancy and is optimized for natural language texts, and is the default behavior of the application.

The *skew* algorithm was included because of its linear time bound in the worst case, and for comparsion purposes. The original implementation has been modified to comply to the ANSI C standard. The $O(n \log n)$ memory occupancy of this algorithm makes it unfit for use on excessively large files, and for most input texts the *dssort2* algorithm is faster. (see Chapter 5 for experimental comparsion).

After computing the suffix array, one pass is performed over the input text, in which the suffix array is transformed into the BWT-ed text according to the formula $T_{BWT}[n] = T[SA[n] - 1]$ for each $n \in \{1, 2, \ldots, |T| - 1\}$ and $T_{BWT}[SA[0]] = \#$, the number of occurrences of each character is stored in an array and the Marks Database is computed. The array C is then computed from the occurrence counts in a single pass over the input alphabet.

The BWT introduces the end-of-file symbol # into the text. This character occurs only once in the BWT-ed text, but it increases the alphabet size to 257, making it impossible to represent characters in a single byte. To avoid storing the output of the BWT as a 4 byte integer, we introduce a structure called *ext_ str* that stores the symbols of the original alphabet in the BWT-ed text in one byte using the same representation as in the original text, and stores the position of the end of file character explicitly as an integer. Functions to access this structure as if it was an integer representation of the text are implemented.

The main access functions represent the end of file character as the signed integer -1 (note that it has to be smaller than all other symbols of the alphabet). At each writing of the string a test is performed whether the written character was the end-of-file character, and its position in the text is updated if needed. Reading the string performs a similar check and returns -1 if the end of file character was accessed. To avoid producing rotations of the text explicitly, accesses that are out of range of the text size are computed modulo n, where n is the size of the text. The byte that conforms to the position of the end-of-file character in the string representing the BWT-ed text in the structure is never accessed, and its content is undefined.

Because the characters of this extended alphabet are used for indexing arrays, and in the ANSI C language arrays are indexed with integers starting with 0, it was necessary to include an extra access function to the structure that returns numbers in the range 0 to 257. The return value of this access function (called *skew_get*) is identical to the return value of the standard access function *ext_str_get* incremented by one, apart from one significant difference. It returns zero for all accesses that are out of the range instead of simulating cyclic rotations. This behavior is required by some algorithms using it.

4.3.2 Compressing the BWT Output

The sub-module for compressing the BWT-ed text generates during its operation the compressed representation of the original text along with the array P_2 and associated structures, and the Block Index that is used for finding the start of blocks in the compressed text so they can be retrieved separately.

The algorithm takes the output of the BWT character by character. The structures associated with the current block are updated, and the input character is then passed to the compression chain. The current length of the compressed output is tracked for use in the block index. When the end of a block is detected, the structures are saved in memory and reset, and the *end-of-block* character is passed to the compression chain before resuming the processing of the text. At the end of input, another *end-of-block* character is written. For the compression and decompression algorithms, the *end-of-block* character is considered to be the end of input, and after encountering it they perform the necessary operations for finalizing the block and reset their inner state to the initial value.

For compressing the BWT we decided to use a move-to-front encoder [3] followed by an *adaptive range encoder*. The move-to-front encoder is implemented using an array of 257 integers (one for each symbol of the alphabet of T_{BWT}), in which the symbols are ordered according to the number of distinct symbols seen since its last occurrence. After reading the symbol *ch* this array is searched for the entry containing this symbol and its index is passed as the output. Then this symbol is moved to the index 0 in the array, pushing all symbols between its previous position and the start of the array one position further. The array representing the MTF is initialized so that every entry contains its index as a value.

The adaptive range encoder was selected mainly for experimental purposes. In [7] Ferragina and Manzini mention experimenting with range encoding and they state they found it inferior to the Multiple Table Huffman encoding used in their implementation, but they supply no experimental data to back up this statement. Since range encoding theoretically achieves better compression ratios than Huffman encoding, we considered this worthy of further examination.

The basic idea of range encoding is to represent the whole input text as a single number from the range $\langle min, max\rangle$. Every character is associated with a part of this range according to the probability of its occurrence. After reading a character, the working range is reduced to the range associated with this character replacing the original range. This step is repeated for all input characters, and after reading the input, the text is represented by an arbitrary number from the resulting range. The original value of *max* is assumed large enough to be able to code any text with a single integer.

Because the implementation details of the range encoder are quite complex, and these details are not important for being able to understand the functioning of the application, we decided not to describe them at this point but to devote a separate section in this chapter to it.

The range encoder is currently the only compression method implemented in our application. Including other compression methods that might be better suited for use in the FM-index is left as future work. The comparsion of the speed and compression ratio of the MTH and the range encoder can be achieved utilising Ferragina's and Manzini's implementation from [7].

4.4 The Range Encoder/Decoder

The range encoder used to compress the output of the MTF transformation was implemented according to the instructions presented in the course 36KOD (Data Compression) led by Ing. Jan Holub, Ph.D. on the Faculty of Electrical Engineering of the Czech Technical University in Prague [9].

The basic idea of range encoding is to represent the input text as a single integer from the range $\langle 0, max\rangle$, where the number max is assumed to be $2^n - 1$, n being "large enough", theoretically infinite and in practice exactly the size of the compressed output in bits.

Each symbol from the input alphabet is associated with a portion of the available range. The range is represented by the (very high precision) numbers *low* and *high*, which are initialized (in their binary representation) to theoretically infinite number of zeros (for *min*) and ones (for *max*) respectively.

Symbols of the input text are associated with a part of the available range to encode them in such way, that all symbols occupy a part proportional to its number of occurrences. In a static compression scheme the number of occurrences is counted before the compression for all symbols, and each symbol *ch* is associated with a portion of the range that has the size $(high - low) * \frac{occ(ch)}{|T|}$, where $occ(ch)$ denotes the number of occurrences of *ch* in text T and $|T|$ is the size of the input text. The actual ranges associated with the symbols can be (without loss of generality) ordered in a way that $high(a) < low(b) \iff a < b$ and $high(a) = low(a) + \left((high - low) * \frac{occ(ch)}{|T|}\right)$.

In a dynamic compression scheme, the number of occurrences for each symbol is initialized to non-zero numbers (to be able to represent them at their first occurrence), and the variable *count* is initialized to the sum of the initial occurrences. At compression time, this statistical model is updated after each input character, incrementing its occurrence count, and the total number of characters stored in *count* by one. The updated model is used for encoding the subsequent character. The size of the range associated with characters becomes $(high - low) * \frac{occ(ch)}{count}$.

As symbols are read, the current working range is shrunk to the range associated with the current symbol. The working range is becoming smaller with each symbol read, the numbers *low* and *high* getting closer to each other in every step. Because of this, their highest order bits start to become identical during the course of compression, and remain that way until the algorithm terminates. Since we know that the result of the compression has to be a number between *low* and *high*, we can send these identical high order bits directly to the output, thus eliminating the need to store them in the memory.

The ability to write out the high-order bits can be exploited to represent the numbers *high* and *low* in practice without the need of working with extremely large (or infinite) number representation. Instead we can use two limited size numbers (in our case 32-bit integers) to represent a "window" into the theoretical working range. We do not have to consider the bits to the "left" of the window, because they were already written to the output. The bits to the "right" of the window can be assumed zeroes for the number *high* and ones for the number *low*. This assumption ensures that our current working range is slightly smaller than the theoretical maximum possible, because we replace the unstored bits "right" of the window with a pessimistic estimate, thus losing a small part of the range (making the compression ratio insignificantly smaller than the theoretical best). At this cost we avoid representing infinitely large numbers, and the algorithm retains its ability of lossless data compression.

One problem that arises from this representation of ranges is the possibility of *underflow*. It occurs when our 32-bit representations of *high* and *low* get too close to each other to represent a big enough range for continuing with the encoding. For example should the situation arise where *high* is represented (in binary) by "10^{31}" and low by "01^{31}", then the following steps either do not change them, resulting in no further characters written to the output, or become identical, resulting in the whole of them being shifted to the output, and replaced by symbols shifted in from the right. This would mean *high* becoming zero and *low* becoming *max*, the meaning of these values having no sense as the representation of a range.

The occurrence of an underflow and the errors resulting from it can be prevented with help of the following consideration: if it occurs that the $n + 1$ highest order bits of *high* are "10^n" and the $n+1$ highest order bits of *low* are "01^n" we can be sure that all of these bits will become identical and subsequently shifted to the output at the same time. The only question remains if they become "01^n" or "10^n".

When the first one of these bits is shifted to the output, we know that the n bit following it can be written aswell, and we also know their value. Because of this we can delete these n bits as soon as we detected them and only keep track of their amount n, filling up their spaces by shifting more bits in from the right.

The only question to remain open is which number to choose from the available range when the input string ends. We have decided that the singe number *high* will be excluded from the range associated with input symbols, and will represent the end of file character. Thus when the input ends (or the end of input character is encountered), the current value of *high* is shifted to the output.

Decoding works in exactly the same way as encoding. In the initial phase it initialises the values of *high* and *low* and the occurrence counts exactly in the same way as the encoder, and fills a 32-bit buffer representing the window into the result of the encoding. The symbol that is associated with the range that this number falls into is written to the output and the values *high* and *low* and the statistical model are updated exactly as in the encoding phase. Every time *high* and *low* are shifted because their high order bits being identical, the window in the input is also shifted, erasing its high order bits and filling its low-order bits from the input file. The decoding process finishes when the value of *high* and the number in the input window match (this value means the end-of-file symbol).

Since the range encoder and decoder operate with files as they were streams of bits, we include the structure *bitfile* in our implementation. This structure and the functions associated with it are responsible for emulating bit-wise access to files, and the initialization and finalization of such files. The structure contains a buffer for storing a part of the file it is working with, and a counter for tracking how many bits in the buffer are currently used.

4.5 The Structures of the FM-index

4.5.1 The Block Index

The Block Index tracks the starting positions of the compressed blocks in the output of the compression step. Since the compressed output is considered a stream of bits by the compression/decompression chain, we currently use two numbers to store these positions. A 4 byte integer is used for storing the byte offset of the block's starting position, and a 1 byte integer is used to store the bit offset in the starting byte.

This structure for storing the starting position of blocks is far from ideal. The bit offset in a byte can only have 8 different values, and encoding it in a whole byte wastes 5 bits for each block. It would be possible to save the byte used for storing the bit offset by aligning the start of each block to whole bytes. The alignment wastes on average 4 bits per block in the compressed representation, but reduces the index size by 8 bits per block, thus saving several bytes.

Another possibility would be to use 3 bits in a 32-bit number to store the bit offset of the block, and use the remaining 29 bits to store the byte offset, but this would limit the compressed file size we are able to work with to 2^{29} bytes.

However, we decided not to implement these optimizations. The reduction in the index size they achieve is 4 to 8 bits per block. Since the block sizes usually used in the index is in hundreds to thousands of bytes, the difference in the index size they would make ranges around tenths of percents. We judged that this slight difference is not worth the increased complexity of the encoder (in the first case) or the limitations introduced (in the second case).

4.5.2 Supporting fast LF-mapping

The LF-mapping relies on the array C and the function $occ(ch, n)$. The array C stores for each $ch \in \Sigma$ the number of characters in the original text that are lexicographically smaller than or equal to ch in the entry $C[ch]$. This array is created during compression time in a single pass over the BWT-ed text (that contains the same characters as the original) to count character occurrences, and a single pass over the alphabet to add to the occurrence counts the sum of occurrences of characters smaller than ch. It is then stored in the FM-index in 1024 bytes (a 4 byte integer for every 256 character of the alphabet).

The function counting the occurrences relies on the strategy described in Section 3.3.1. When the function to count the occurrences is called with parameters $occ(ch, n)$, it finds the block in which the n-th character is located, decompresses it if needed and counts the number of occurrences of ch in that block before the n-th character.

The array P_2 is constructed during the pass over the BWT-ed text at compression time, storing for each block b the occurrence count of all distinct characters ch before that block in the entry $P_2[b][ch]$. The array contains a four byte integer for each block and each character of the alphabet, occupying 1024 bytes per block in the basic case.

As suggested by Ferragina and Manzini in [7], this array is further augmented by a bit mask for each block. The bit mask includes one bit for each character in the alphabet (thus occupying 8 bytes per block) for keeping track if a given character occurs in a given block. It is implemented as an array $bm[b][ch]$ accessed with functions for bit-wise reading and writing.

This bit mask contains zero for in the entry $bm[b][ch]$ if the block b does not contain the character ch. Ferragina and Manzini state that this property can be used to speed up the function for counting occurrences because it can save the decompression of a block if the from the bit mask we know that the given character does not occur in that block. This is however rarely exploited in our case, because the LF-mapping only calls the function to count occurrences with parameters $occ(ch, n)$ if ch is the $n+1$-th character of T_{BWT}. The probability of n and $n+1$ being in different blocks, and ch not occurring in the block of n is very small.

On the other hand, the bit mask can be utilised to reduce the size of the index. If the character ch does not occur in block b, the entries $P_2[b][ch]$ and $P_2[b+1][ch]$ are equal. When saving the index it is enough to save one of them, and when loading the index both can be reconstructed using the bit mask and the saved entry.

In practice this means, that for each zero in the bit mask we eliminate the need to store one entry of the array P_2, reducing the index size by 4 bytes. The cost of saving the bit mask is 8 bytes per block, meaning that introducing it saves space whenever it contains more than two zeroes per block. Because the BWT tends to group together a small number of distinct characters and because most text do not use all 256 symbols of the alphabet, the index size reduction enabled by the bit mask usually out-weights its cost by large amounts.

Ferragina and Manzini also suggest using a multi-level block scheme, grouping several blocks into super-blocks. Each super-block is equipped with a header containing the occurrence counts before the super-block and a bit mask similarly to the blocks. The entries of the array P_2 then contain the character counts for each block since the beginning of the super-block. This practically does not introduce any opportunity to speed up the program, but it could also be used to reduce index size.

The extra structures introduced by the super-blocks occupy 8 bytes for the block index and another 4 bytes for every distinct character in the super-block. This information enables saving the storage space of a single bit per block for each character not contained in the super-block, but this saving would complicate the storage of block bit masks unnecessarily for saving only a few bytes.

If the super-block size is limited so it can be addressed by integers occupying less than 4 bytes, the space required for the array P_2 can be reduced by reducing the size of its entries. Limiting the super-block size to 64 kilobytes would reduce the size of the array P_2 to half, thus introducing considerable savings in index size.

Even though the multi-level block scheme can reduce the space needed by the index, we decided *not to implement* it. Assuming a constant super-block size its reduction of space requirements increases as the size of blocks decreases, and the index size reduction is close to zero as block size approaches half of super-block size, in extreme cases even limiting block size to super-block size and wasting storage space.

We feel that this behavior is slightly counter-productive. The selection of the block size introduces a trade-off between storage requirements and pattern matching speed. Users preferring small space consumption over fast pattern matching would gain no benefits from super-blocks, and it would limit the maximum block size they could choose. Users preferring speed over space would benefit from the ability of super-blocks reducing the index size at the cost of slight slowdown because they tend to use small blocks. But their choice of small blocks means they prefer speed over size, in which case they might not want the space reduction of super-blocks even at the cost of only a small slowdown.

The implementation of super-blocks and the solution of this controversy is left as a possible future improvement.

4.5.3 The Mark Database

The operations *Locate*() and *Extract*() rely on the Marks Database. Its function is to store for a selection of rows from the matrix M the starting position of their corresponding suffixes in the original text. It has to be able to answer queries "Is row i marked?" (denoted $is\_marked(i)$ in the following text), "What's the starting position $pos(i)$ of row i?" (denoted $pos(i)$) and "What marked row i is nearest to a given position in the original text?" (denoted $pos^{-1}(i)$ which is equal to $SA[i]$ for marked rows).

The first of these queries is executed in every LF-mapping step during the execution of the *Locate*() procedure. The other two are executed once for every call to *Locate*() and *Extract*() respectively. The marking strategy and the implementation of the Marks Database determines not only the performance of these queries, but also the value of $mdist$, which limits the maximum number of LF-mapping steps during the *Locate*() operation.

Ferragina and Manzini [6] propose a marking strategy, where every $mdist$-th position of the original is marked. It means that the maximum number of LF-mapping steps during a *Locate*() operation is limited by $mdist$, its value being chosen by the user at the time of building the index. Storing the pairs of i and $pos(i)$ in an array $marks$, where $marks[\lfloor \frac{i}{mdist} \rfloor] = pos(i)$ takes $4\lceil \frac{|T|}{mdist} \rceil$ bytes of space. In this primitive implementation the queries for $is\_marked(i)$ and $pos(i)$ can be answered in time linear with the database size, and the query $pos^{-1}(i)$ can be answered in constant time.

Another marking strategy, proposed by Ferragina and Manzini in [7] is to select one character from the text ch such as the count of this character is as close as possible to the proportion of the text to be marked (as chosen by the user), and mark the rows ending with this character. With this strategy the Marks Database can be stored in an array $marks$ where $marks[occ(ch, i)] = pos(i)$. This enables answering queries $is\_marked(i)$ in constant time by looking if $T_{BWT}[i] = ch$, and answering queries $pos(i)$ in time proportional to the cost of counting occurrences. The time required to answer the query $pos^{-1}(i)$ becomes linear.

The downside of the second strategy is that the value of $mdist$ becomes the biggest distance between two neighboring characters ch in the original text, giving us no guarantee of limiting the amount of LF-mapping steps executed in the *Locate*() operation. Their experiment show that most natural language texts characters have a tendency to occur more or less regularly, making this marking strategy a good option for practical cases.

We decided to implement a marking strategy that guarantees the limiting of LF-mapping steps in the *Locate*() operation and is able to answer *is_marked*(i) queries in constant time. The performance of the other two queries is less important, because they are only executed once during the *Locate*() and *Extract*() operations. The path we follow is similar to the first strategy of Ferragina and Manzini presented, marking every *mdist*-th position of the original text and storing them in an array where $marks[\lfloor \frac{i}{mdist} \rfloor] = pos(i)$.

To speed up the query *is_marked*(i) we construct at the time of loading the index a bit mask of the size $|T|$ bits, storing in each bit 1 if the corresponding row is marked and 0 if it is not. This bit mask is not stored in the index, and takes $O(\frac{|T|}{mdist}$ time to build at the time of loading the index, occupying $\frac{|T|}{8}$ bytes in memory. This enables the answering of *is_marked*(i) queries in constant time at the cost of having to execute an extra pass over the array representing the Marks Database.

The memory occupancy of this bit mask can be rather large for large texts, but in practical cases where the block size is not considerably larger than about 10 kilobytes it will still occupy less space than the rest of the index. In cases where the this memory consumption could mean significant slowdown, the user can choose to disable the building of the bit mask, and the program falls back to answering *is_marked*(i) queries by traversing the Marks Database in linear time.

4.5.4 Output File Format

The output file constructed by the compression module contains all of the necessary structures for the operation of the pattern matching and decompression modules. Where it is not stated otherwise, integers are saved in byte order depending on machine endianness. This limits the use of the index to machines having the same endianness as the machine it was created on and is planned to be fixed in a future version.

The first several bytes of the file function as a header. In the first four bytes, the characters “F” and “M” followed by the hexadecimal “magic number” 0x7C19 in little endian notation serve the purpose of identifying the file as and FM-index structure. If the first four bytes of the file do not match when the decompression or pattern matching operations are executed, the application terminates assuming that the file does not match the requested format. In the next four bytes, a pointer to the position where the compressed representation of the BWT-ed text ends and the parameters and structures of the FM-index are stored.

The following four bytes contain the size of the original text over which the index was built. After this, the compressed representation of the BWT-ed text follows. The data following this is aligned to whole bytes, with excess bits not occupied by the compressed BWT-ed text filled up with zeroes.

The compressed BWT-ed text is followed by the constants *bsize* and *mdist* and the position of the *end of file* character in the BWT-ed text. After this, the array C and the *block index* are written. Then for the 8 byte bit mask tracking what characters occur in the given block are outputted for each block.

The next part of the file contains a representation of the array P_2. Only the entries for which the corresponding entry in the bit mask contains 1 are saved, because the rest of the entries can be reconstructed from this data at the time of loading the index.

```
Algorithm $occ(ch, n)$
1)  $occurences \leftarrow 0$
2)  if ($block\_bit\_mask[k/bsize][ch]$)
3)      if block_cached?($k/bsize$)
4)          $block \leftarrow block\_cache[k/bsize]$;
5)      else
6)          $block \leftarrow$ decode_block($k/bsize$);
7)          $block\_cache[k/bsize] = block$;
8)      endif
9)      for $i \leftarrow 0$ to $n$ mod $bsize$ do
10)         if $block[i] = ch$
11)             $occurences \leftarrow occurences + 1$;
12)         endif
13)     done
14) endif
15) $occurences \leftarrow occurences + P_2[n/bsize][ch]$;
16) return $occurences$;
```

Figure 4.2: Pseudo-code of the $occ(ch, n)$ operation.

The final part of the index contains the array *marks* that represents the Marks Database. This part of the index might be missing if the user opted for not creating it at the time of building the index. Files that do not include the Marks Database are indicated by the value 0 saved in the position of the variable *mdist*, and these files do not support the operations *Locate()* or *Extract*(). However, they can still be used by the operation *Count*() and for extraction of the whole original text.

4.6 Pattern Matching Operations

In this section we present the details of how the pattern matching operations function. We devote special attention to the optimizations included in our application.

4.6.1 Occ()

The function $occ(ch, n)$ counts the occurrences of the symbol *ch* in the text $T_{BWT}[0..n]$. For this means it utilises the information saved in the array P_2 and the bit masks of the character occurrences. If the character *ch* occurs in the block that contains the n-th character in T_{BWT}, it needs to decompress this block and traverse it, counting the occurrences of *ch* in this block up to the position n.

Because every LF-mapping performed during the pattern matching operations contains a call to this function, the performance of all supported operations depends greatly on its performance. If the operation does not need to decompress blocks it finishes in constant time, but this case is very uncommon. Thus the speed of decompression is very important for the performance of the FM-index.

Both the speed of decompression and the time it takes to traverse the decompressed block for counting the occurrences depends linearly on the size of the blocks. Thus reducing the size of the blocks greatly improves the overall performance of the index.

The amount of the LF-mappings performed during one execution of the application can vary greatly. Executing the operation *Count*() for a short pattern only involves calling the *occ*() function several times, while calling the *Locate*() function for a pattern occurring many times in the text can involve thousands of calls to *occ*(). In these cases a lot of effort is usually wasted into decompressing and traversing the same blocks over and over. While the traversing of the blocks can not be avoided without storing excessive amounts of data in memory, the decompression can be limited to occurring only once by saving the decompressed parts of T_{BWT}.

These decompressed parts can occupy at most as much memory as the original text. This ensures that the memory occupancy of the pattern matching does not grow far beyond the memory occupancy of the construction of the index. The memory needs of this optimization make it reasonable to allow the user to turn it off, but we do not recommend this. We expect that even in the case that storing the cache would cause the application to run out of memory, the swapping operation would probably still be faster than the decompressing the blocks that are used repeatedly.

The pseudo-code of the function $occ(ch, n)$ is depicted on Figure 4.2.

4.6.2 Count()

The procedure $Count(P)$ counts the occurrences of the pattern P in the original text, using the array C and the the LF-mapping supported by the function $occ(ch, n)$. To achieve this it finds the rows of the matrix M prefixed by the pattern P. The number of the required LF-mapping steps depends on the length of the pattern.

The number of occurrences of the pattern P found in the original text is written to the standard output, then calls the operations $Locate(i)$ and $Extract(a, b)$ with appropriate parameters are made as needed, according to the user's request. $Locate(i)$ is called for each row i of the matrix M prefixed by P if the user has requested the positions of occurrences of the patterns found. The operation $Extract(a, b)$ is called if the user has requested to extract the contexts of the occurrences. The values of parameters become $a = Locate(i) - x$ and $b = Locate(i) + |P| + x$, where x is the size of the context.

The operation of the procedure $Count(P)$ was already described in detail in Section 3.3.1 of Chapter 3. The implementation of this procedure follows the pseudo-code presented on Figure 3.3.

4.6.3 Locate()

The function $Locate(i)$ uses the LF-mapping operation and the queries $is\_marked(i)$ and $pos(i)$ of the Marks Database to locate the starting position of the suffix associated with the i-th row of the matrix M in the original text.

The pseudo-code for this function was presented on Figure 3.4. Note that the call to LF-mapping includes in this case two requests for compressed blocks. One is made to retrieve the symbol $ch = T_{BWT}(i)$, and the other is included in the call to the function $occ(ch, i-1)$. The caching of decompressed blocks enables us to perform the decompression

```
Algorithm $Extract(a, b)$
1)  if $(a < 0)$ $a \leftarrow 0$ endif
2)  if $(b > |T|)$ $b \leftarrow |T|$ endif
3)  if $(a > b)$ Error("Bad call to Locate()")
4)  $i \leftarrow pos^{-1}(b)$, $j \leftarrow pos(i)$
5)  while $j > b$ do
6)      $i \leftarrow LF(i)$
7)      $j \leftarrow j - 1$
8)  done
9)  $T[j] \leftarrow T_{BWT}[i]$
10) while $j > a$ do
12)     $i \leftarrow LF(i)$
13)     $j \leftarrow j - 1]$
14)     $T[j] \leftarrow T_{BWT}[i]$
15) done
16) return $T[a..b]$;
```

Figure 4.3: Pseudo-code of the *Extract*(a, b) operation.

only once in the majority of cases, because unless $i \bmod bsize = 0$, these two calls request the same block to be decompressed.

The performance of this function relies heavily on the performance of the queries to the Marks Database and the value of *mdist* (see Section 4.5.3 for details).

4.6.4 Extract()

The operation *Extract*(a, b) returns the substring $T[a..b]$ of the original text. It first checks if the bounds of the requested substring are withing the scope of the original text, and adjusts them if necessary. In the case that $a > b$ the operation indicates an error.

The actual retrieval of the substring $T[a..b]$ is achieved by first executing the query $pos^{-1}(b)$ of the Marks Database to find the row i of the matrix M associated with the text position $n \geq b$, then executing $n - b$ LF-mappings to reach position b and $b - a + 1$ LF-mappings to extract the requested text.

Similarly to the function *Locate*(i), each call to the LF-mapping includes two decompression requests, in most of the cases referring to the same block. Note that retrieving large parts of the original text using this operation is very inefficient compared to retrieving the original text, because every call to the function $occ(ch, i)$ involves decompressing and parsing a block. If the requested substring of the text is larger than several symbols (approximately the number of blocks in the text), it is recommended the user decompresses the whole text instead using a call to the decompression module.

The pseudo-code of this operation is presented on Figure 4.3.

Chapter 5

Experiments

5.1 Machine and Platform Specification

All measurements described in this chapter were executed on an Intel(R) Pentium(R) 4 3.00 GHz with 1 GB of RAM, running Ubuntu Linux 7.10 (gutsy) with kernel version 2.6.22-14-generic. The source code was compiled using *gcc* version 4.1.3 with the optimization switch *-O3*. The run times were measured using the POSIX function *times* as the sum of user and system time.

5.2 Description of the Test Files

For evaluating the performance of our implementation, we used files contained in the *Canterbury Corpus*, the *Large Corpus* [1] and the *Calgary Corpus* [2]. The *Large Corpus* was used for evaluating the space and time efficiency of the compression part, because these files are large enough for precisly measuring the run time of the algorithms on them. For measuring the compression rate and the search time of our program and for comparing its pattern matching performance to the original implementation of Ferragine and Manzini [7] we used all three above mentioned corpora (the large Canterbury Corpus, the standard Canterbury Corpus and the Calgary Corpus).

A list of files used contained in these corpora, along with their size and short description can be found in Table 5.1.

5.3 Selecting the Constants

The file *paper2* from the *Calgary Corpus* was chosen for the preliminary tests to gauge the effect of the constants *bsize* and *mdist* on the performance of the construction, search and decompression algorithms and the compression ratio. All measurements were repeated five times, the tables presented contain the minimal times achieved. In the graphs we decided to show all five results. This should make it easier to distinguish the trends in the data from random noise affecting the measurements.

For measuring the run time of the queries *Count* and *Locate* we use the pseudo-random number generator of the ANSI C language *rand()* with a fixed seed for generating 100 random (4 to 8 symbols long) words. The pattern matching queries are then executed for

Corpus	File size	File name	Description
calgary	111261	bib	Bibliography (refer format)
calgary	768771	book1	Fiction book
calgary	610856	book2	Non-fiction book (troff format)
calgary	102400	geo	Geophysical data
calgary	377109	news	USENET batch file
calgary	21504	obj1	Object code for VAX
calgary	246814	obj2	Object code for Apple Mac
calgary	53161	paper1	Technical paper
calgary	82199	paper2	Technical paper
calgary	513216	pic	Black and white fax picture
calgary	39611	progc	Source code in "C"
calgary	71646	progl	Source code in LISP
calgary	49379	progp	Source code in PASCAL
calgary	93695	trans	Transcript of terminal session
canterbury	152089	alice29.txt	Novel in english
canterbury	125179	asyoulik.txt	Shakespeare play
canterbury	24603	cp.html	HTML source
canterbury	11150	fields.c	C source
canterbury	3721	grammar.lsp	LISP source
canterbury	1029744	kennedy.xls	Excel Spreadsheet
canterbury	426754	lcet10.txt	Scientific text
canterbury	481861	plrabn12.txt	Poetry
canterbury	513216	ptt5	CCITT test set
canterbury	38240	sum	SPARC executable
canterbury	4227	xargs.1	GNU manual page
large	4047392	bible.txt	The King James version of the bible
large	4638690	E.coli	Complete genome of the E. Coli bacterium
large	2473400	world192.txt	The CIA world fact book

Table 5.1: Files used in our experiments.

Block size	Construction time	Decompression time	Average *Count()* time	Average *Locate()* time	Compression ratio
100	$0,52$	$0,09$	$0,0010$	$0,00069$	$455,54$
200	$0,27$	$0,07$	$0,0016$	$0,00083$	$256,30$
300	$0,19$	$0,06$	$0,0018$	$0,00090$	$189,01$
400	$0,15$	$0,06$	$0,0023$	$0,00101$	$154,71$
500	$0,16$	$0,05$	$0,0027$	$0,00109$	$134,19$
600	$0,11$	$0,06$	$0,0030$	$0,00115$	$120,42$
700	$0,11$	$0,06$	$0,0034$	$0,00119$	$110,30$
800	$0,09$	$0,05$	$0,0036$	$0,00127$	$102,60$
900	$0,08$	$0,05$	$0,0041$	$0,00132$	$96,85$
1000	$0,10$	$0,05$	$0,0044$	$0,00141$	$92,12$
1250	$0,07$	$0,05$	$0,0052$	$0,00150$	$83,19$
1500	$0,07$	$0,05$	$0,0063$	$0,00168$	$77,45$
2000	$0,07$	$0,05$	$0,0075$	$0,00190$	$70,43$
3000	$0,04$	$0,05$	$0,0104$	$0,00237$	$62,86$
4000	$0,05$	$0,04$	$0,0131$	$0,00280$	$58,83$
8000	$0,05$	$0,05$	$0,0220$	$0,00443$	$53,09$

Table 5.2: The effects of block size.

each of these words, and the run time is averaged throught the number of calls. To give realistic estimates of run times the block cache is cleared between calls and the time it takes to load the index from the disk is not included.

5.3.1 The constant *bsize*

For measuring the effect of *bsize* on the run time of the program and the compression ratio achieved, we fixed the value of the constant *mdist* to the ad-hoc value 32, and run the compression, decompression, count and locate algorithms with *bsize* set to values from the range of 100 to 8000 bytes. The total number of occurences of these 100 patterns in the text was 3333. A summary of our results is in Table 5.2, and the graphs of the compression and decompression time, compresstion ratio and the pattern matching operations as a function of block size are depicted on Figures 5.1, 5.2 and 5.3 respectively.

We can see from 5.1 that while the decompression time (of the whole text) is largely independent on the block size, the compression time grows noticably as the block size gets smaller. This is mostly caused by the work needed to save the array P_2, which involves bit-by-bit parsing of all block bit masks and writing a 4 byte value for each set bit in them. The large amount of disk accesses in the causes the system time used by the program to rise linearly with the number of blocks. Buffering these disk writes to reduce the overhead caused by this computation is left as a future improvement.

The size of the data that support the LF-mapping depends linearly on the number of

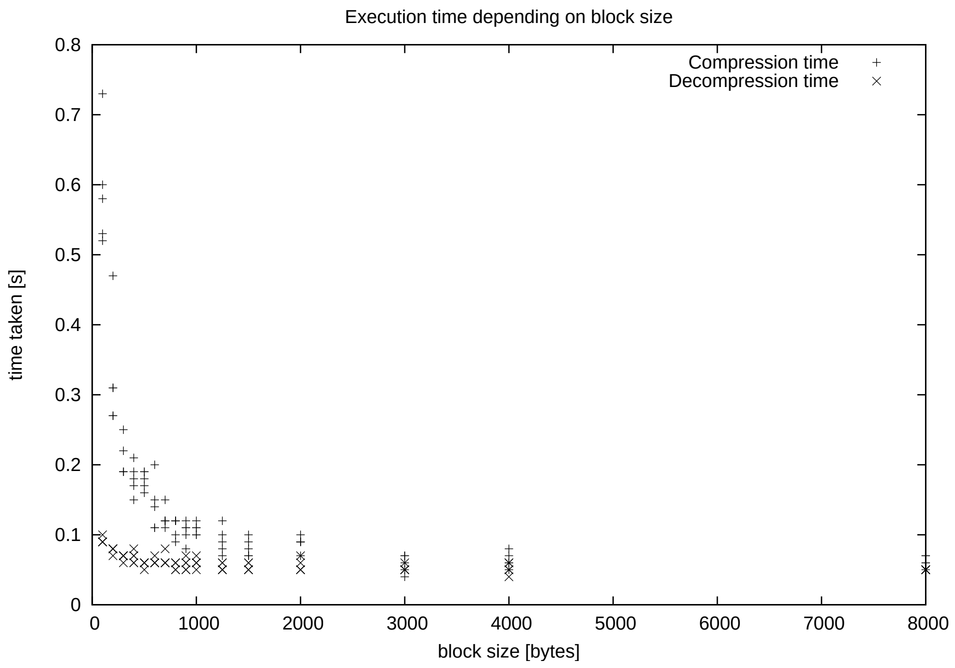

Figure 5.1: Construction/decompression time as a function of block size

blocks, and this dependance can be seen on Figure 5.2. For the block size of 8000 bytes this data occupies neglectable space compared to the Marks Database and the compressed representation of T_{BWT}, and grows steadily as the number of blocks grows.

The average performance of the pattern matching queries depends on the block size almost perfectly linearly (see Figure 5.3. The slight deviation for larger block sizes is justified by the block caching mechanism. For larger blocks, the chance of repeated requests to decompress the same block rises. With block caching, repeated requests to access the same block do not result in their repeated decompression. The effect of the block cache gets more visible for *Locate()* queries, because they involve a lot more calls to the LF-mapping for each pattern than the *Count()* query.

From this experiment we judged that the block size of 2 kilobytes is a reasonable trade-off between index size and pattern matching query time, so it was selected as the default value for the program and is used (unless specified otherwise) in the other experiments presented.

5.3.2 The constant *mdist*

For measuring the effect of *mdist* on the run time of the program and the compression ratio achieved, we used the value $bsize = 2048$ and executed compression, decompression, count and locate algorithms with *mdist* ranging from 4 to 128. The patterns that were searched were generated with the same algorithm as in the previous experiment. The total number of occurences of these 100 patterns in the text was 3333. A summary of our results is presented in Table 5.3, and the graphs of the compression and decompression

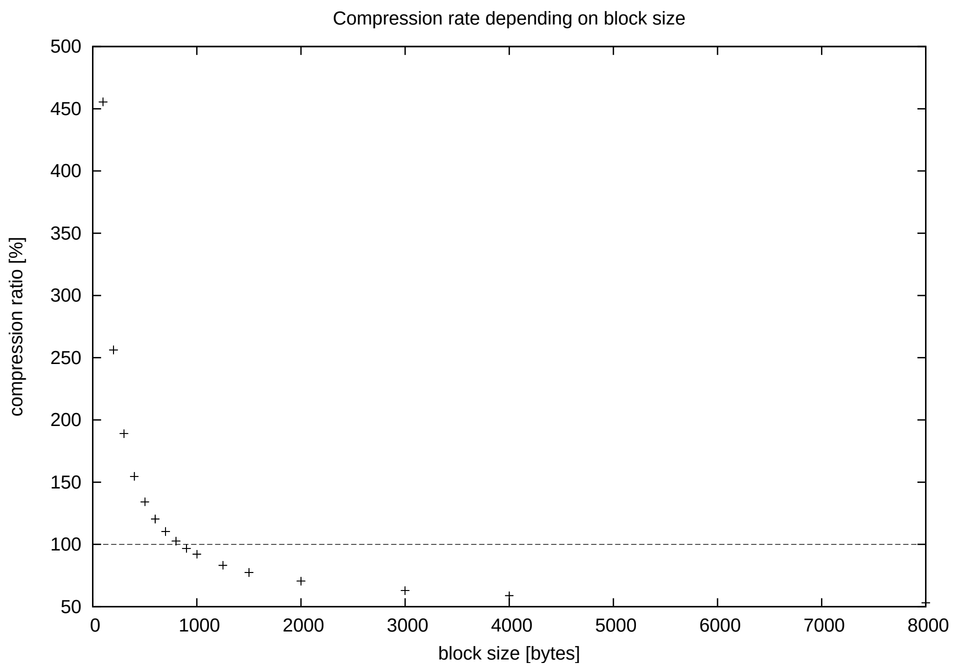

Figure 5.2: Compression ratio as a function of block size

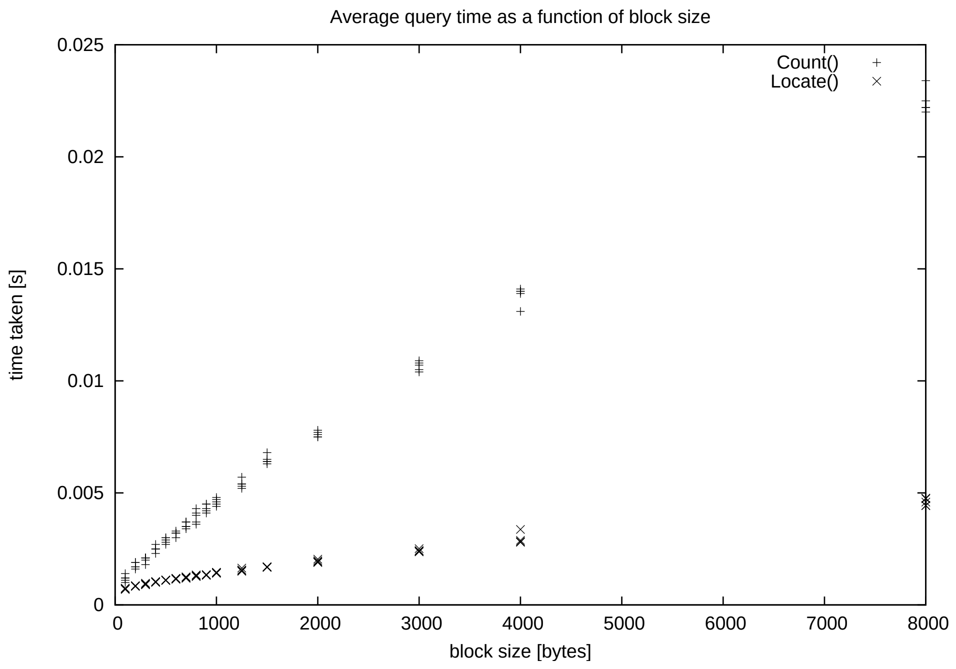

Figure 5.3: Query time as a function of block size

Mark distance	Construction time	Decompression time	Average *Locate()* time	Compression ratio
4	$0,07$	$0,05$	$0,00057$	$157,36$
6	$0,06$	$0,05$	$0,00075$	$124,03$
8	$0,07$	$0,05$	$0,00092$	$107,36$
12	$0,06$	$0,05$	$0,00118$	$90,69$
16	$0,06$	$0,05$	$0,00138$	$82,36$
24	$0,06$	$0,05$	$0,00169$	$74,03$
32	$0,07$	$0,05$	$0,00191$	$69,86$
64	$0,06$	$0,05$	$0,00296$	$63,61$
128	$0,07$	$0,05$	$0,00471$	$60,49$

Table 5.3: The effects of mark distance.

time, compresstion ratio and the pattern matching operations as a function of the mark distance are depicted on Figures 5.4, 5.5 and 5.6 respectively.

From Figure 5.4 we judge that the construction and decompression times are largely independent on this constant. It has only a neglectable effect on construction time implied by the need to save the Mark Database, which is slower with its growing size. The decompression does not use this part of the index at all, and its run time is entirely unaffected by the selection of *mdist*.

The space occupied by the Marks Database has a linear dependance on the amount of marked rows. This trend is observed on Figure 5.5. Because, unlike the data supporting LF-mapping where we are able to avoid storing parts that can be reconstructed from the remaining data, the Marks Database has to be stored in the whole, this dependancy is more regular.

The operation *Count*() is independent on the Marks Database, so its performance was not included in the results of this experiment. In the operation *Locate*(), the mark distance bounds the maximum amount of LF-mapping steps executed during this operation. Assuming random queries, the average amount is half of *mdist*. This fact can be observed on Figure 5.6 depicting the linear dependance of the run time of *Locate*() operations on the amount of marked rows (which is the inverse value of *mdist*.

From this experiment we conclude that the mark distance only has visible influence on the Marks Database size and the performance of the *Locate*() operation. Because of their linear dependance on either the mark distance or the amont of marked rows, the choice of the constant *mdist* represents a straightforward trade-off between file size and pattern matching performance. The user is encouraged to set the value of this constant according to his needs without the fear of affecting other parts of the implementation negatively.

We have chosen the value $mdist = 32$ as default and we will use this value in further experiments unless denoted otherwise.

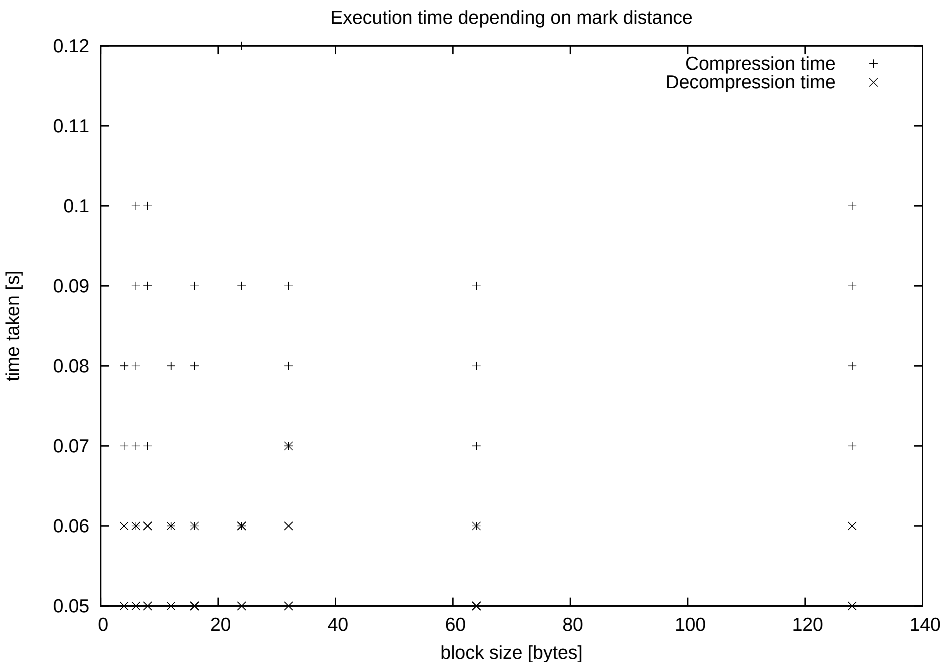

Figure 5.4: Construction/decompression time as a function of mark distance

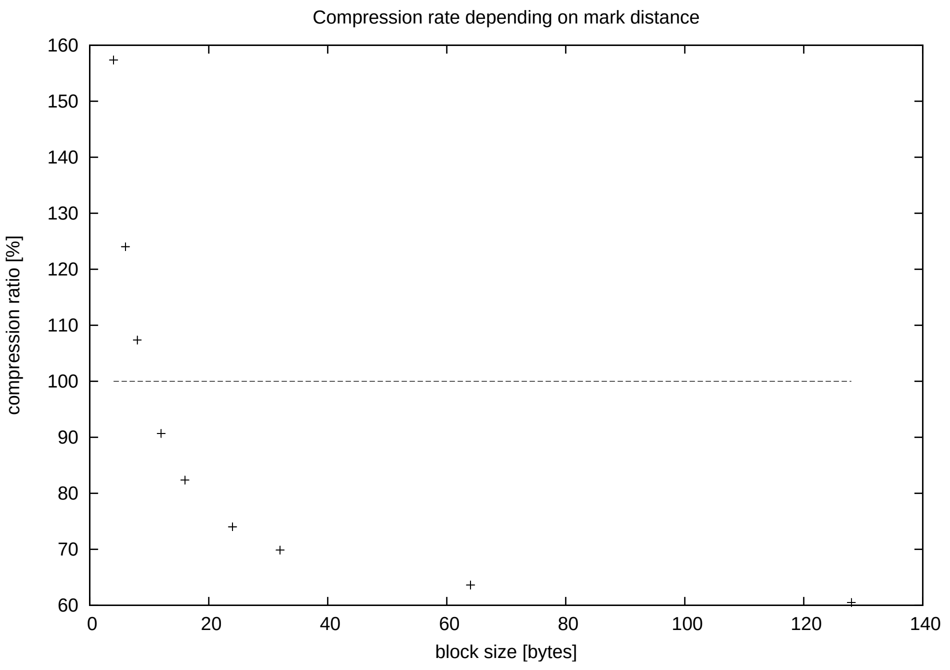

Figure 5.5: Compression ratio as a function of mark distance

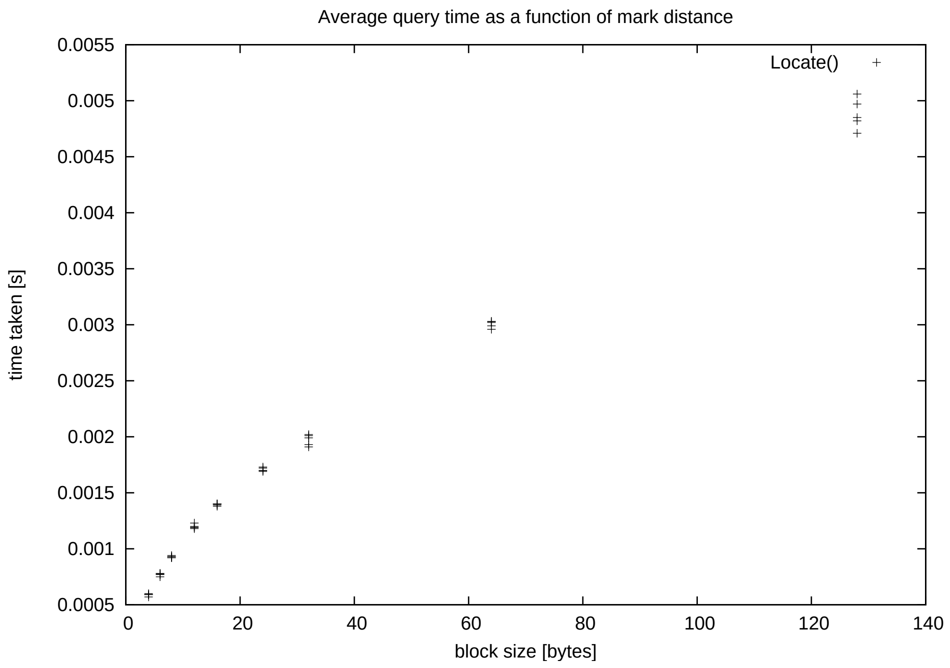

Figure 5.6: Query time as a function of mark distance

5.4 Evaluating the Range Encoding

To evaluate the performance of our range encoder in both compression rate and in compression and decompression speed, we utilised the implementation of the FM-index by Ferragina and Manizin [7]. This program includes a Multiple Table Huffman encoder/decoder which was found the most efficient by Ferragina and Manzini, and the run times of the compression algorithm are reported by their program.

We used the files of the *Large Corpus* to measure the compression and decompression time of the MTH encoding and our range encoding, along with the compression ratio. All measurements were repeated five times, and the minimal achieved values are shown in Table 5.4. The compressed file size includes the LF-mapping support structures that were created with the block size set to 2 kilobytes. The Mark Database was disabled in both algorithms.

The results from this experiments concur with Ferragina and Manzini's conclusion that the range/arithmetic encoding is outperformed by the MTH coder. Our (largely unoptimized) range encoder is about two times slower in compressing data and about three times slower in retrieving it, and it also has considerably worse compression rate. Replacing this encoder with a more efficient one will be our primary concern in future improvements.

File name	MTH			Range coder		
	Comp-ression time	Decomp-ression time	Comp-ression ratio	Comp-ression time	Decomp-ression time	Comp-ression ratio
bible.txt	0.58	0.58	24.20	1.16	1.83	39.20
E.coli	0.78	0.80	27.91	1.47	2.18	34.54
world192.txt	0.34	0.36	24.18	0.68	1.14	43.58

Table 5.4: Comparsion of our range encoder with MTH encoding.

5.5 Comparsion of the Implementation with Others

In this section we present a comparsion of our implementation with the implementation presented by Ferragina and Manzini [7]. All experiments presented here were conducted with the block cache turned off in both implementations, which should allow for better comparsion of the algorithms used. Enabling the cache would result in reducing the disadvantage of our program caused by its slow decompression, making it harder to judge the improvements that could be achieved with a more efficient compression scheme.

To measure average *count* and *locate* times we used 100 randomly generated words of length 4 to 8 symbols as patterns, and averaged the run time throught all operations. Because currently our implementation is unable to handle patterns containing the zero character which denotes the end of the pattern, and the implementation of Ferragina and Manzini requires the words in the word list to be separated by the end-of-line character, we replaced all occurences of these characters in the random patterns with the *space* characters.

In files that had an extremely large number of these forbidden symbols, their replacement resulted in the generation of a word list where none of the patterns occured in the original text. We judged that omitting these files from our experiments will not affect the value of the results greatly, so they were excluded from the experiment. The affected files are *geo* and *pic* from the Calgary Corpus and *kennedy.xls* and *ptt5* from the Canterbury Corpus.

The Table 5.5 summarizes the results on the files of the Calgary, Canterbury and Large corpora. The implementation of Ferragina and Manzini is denoted as *BWI* in this table and in the following text of this section.

The compressed files generated by our algorithm are in most cases about 50% larger than those produced by the *BWI* program. This is caused by the inefficiency of our compression method used, and should be fixed in a future version of our program.

Because both implementations use the same algorithm for the operation *Count*, and preliminary experiments with the range encoder have shown that it is about three times slower in decompressing than the MTH coding used in the BWI program, we expected the *Count* operation of our program to be slower by a similar amount. However, the measurements have shown it to be 5 to 10 times slower than the BWI *Count* operation.

	BWI			Our program		
File name	Comp-ression ratio	Average count time	Average locate time	Comp-ression ratio	Average count time	Average locate time
bib	38.33	0.0006	0.0030	61.21	0.0072	0.00236
book1	44.10	0.0007	0.0028	66.72	0.0083	0.00273
book2	41.36	0.0007	0.0037	65.68	0.0081	0.00301
news	48.75	0.0009	0.0084	71.64	0.0089	0.00184
obj1	90.41	0.0014	0.1246	112.33	0.0060	0.00115
obj2	62.82	0.0011	0.0798	99.14	0.0071	0.00148
paper1	47.47	0.0009	0.0032	70.30	0.0069	0.00305
paper2	45.98	0.0005	0.0026	69.86	0.0078	0.00202
progc	49.33	0.0008	0.0063	70.90	0.0064	0.00279
progl	36.76	0.0008	0.0041	57.75	0.0060	0.00104
progp	37.63	0.0006	0.0019	57.80	0.0061	0.00088
trans	35.07	0.0011	0.0085	56.43	0.0055	0.00136
alice29.txt	41.79	0.0008	0.0057	63.30	0.0070	0.00166
asyoulik.txt	45.19	0.0010	0.0026	65.99	0.0013	0.00250
cp.html	48.36	0.0007	0.0046	72.52	0.0068	0.00126
fields.c	44.52	0.0005	0.0028	69.52	0.0036	0.00160
grammar.lsp	52.38	0.0006	0.0021	86.22	0.0025	0.00111
lcet10.txt	39.18	0.0009	0.0044	61.82	0.0081	0.00159
plrabn12.txt	43.22	0.0005	0.0032	65.21	0.0081	0.00276
sum	66.43	0.0015	0.0033	100.55	0.0048	0.00159
xargs.1	61.49	0.0007	0.0035	98.06	0.0030	0.00179
bible.txt	32.17	0.0007	0.0028	51.70	0.0086	0.00363
E.coli	32.47	0.0006	0.0034	47.04	0.0095	0.00414
world192.txt	32.36	0.0006	0.0028	56.08	0.0083	0.00322

Table 5.5: Comparsion of search times.

We found that this difference was probably caused by the fact that our time measurements included some initialization code (namely the resetting of the block cache) in addition to the *Count* operation. Because we were unable to verify what exactly is included in the time the BWI reports as average count time, and because we judged that our count times without the cache resetting are close to the expected values, we decided not to repeat this lenghty experiment.

The *Locate* operation of our implementation has shown to be very efficient, outperforming the BWI's *Locate* operation on all measured files of the Calgary and Canterbury corpora, and being very close in performance of the BWI on the Large corpus. The difference is caused by the marking strategy we used. Ferragina and Manzini's marking strategy selects one symbol with frequency close to the amount of marked rows (as per user request), and relies heavily on this symbol being evenly distributed in the input text.

On files that meet this requirement, the performance of both implementations comes within 50% of each other. On texts that do not meet this requirement, our implementation achieves superior results, reducing the time requred by the *Locate* operation to 50-80% in most cases, and achieving a speedup of over 50 times on the binary files *obj1* and *obj2* that seem to be extremely ill suited for the BWI marking strategy.

Considering that the block decompression in our implementation is about three times slower, we expect that by replacing the range encoder with an encoding scheme of speed similar to MTH we can achieve further improvement of the *Locate* operation performance. The speedup achieved by reducing the decompression time three times can be expected to be in the range of $1,5$ to 2 times for uncached operation. With this improvement we expect that our marking strategy will outperform the one used in the BWI implementation in all cases.

Chapter 6

Conclusion and Future Work

6.1 Summary of Results

In this thesis the possibilities of data compression and indexing using the FM-index structure introduced by Pablo Ferragina and Giovanni Manzini [6, 7] were analysed. An implementation of this structure and its supported operations *Count(P)*, *Locate(P)* and *extract(a, b)* was developed, and its performance on typical input files was evaluated. This implementation fills the gap caused by the absence of an open-source implementation of this structure and its associated algorithms for research purposes.

Recent research results in suffix array computation were summarized, and the algorithms *dssort2* (Ferragina and Manzini [8]) and *skew* (Kärkkäinen and Sanders [11]) were adapted for computing the Burrows-Wheeler transformation [5] in the construction of the FM-index.

The range encoding mechanism was implemented according to the instructions presented in the course 36KOD (Data Compression) led by Ing. Jan Holub, Ph.D. on the Faculty of Electrical Engineering of the Czech Technical University in Prague [9], and its performance was evaluated. This encoding algorithm has shown to be inferior in performance to compression methods used in the original FM-index implementation by Ferragina and Manzini.

Several ideas to improve the performance of the operations supported by the FM-index were introduced, the most important of them being a new marking strategy that improves the performance of the *Locate(P)* operation considerably. Utilising this new marking strategy entirely removes the dependance of the pattern matching performance of the FM-index on the structure of the input text. The theoretical speedup of this marking strategy has been confirmed by our experiments.

6.2 Suggestions for Future Research

The main drawback of this implementation has shown to be the utilization of the range encoder. We expect that by replacing the encoding scheme with a more efficent one, we could improve the performance of our implementation to match the performance of Ferragina's and Manzini's implementation of the FM-index.

Further modifications of the representation of the FM-index structure analysed in this thesis could be implemented, and their properties could be exploited to reduce the index size and/or to improve the query performance.

The construction time of the FM-index can be further reduced by utilising the most recent achievements of suffix array construction. The algorithms developed by Schürmann and Stoye [16], Kim et al. [12] and Hon et al. [10] all show great promise, and adapting them could enchance the speed of FM-index construction as well as eliminate the weaknesses of the algorithms currently used.

The portability of the implementation could be further improved by adapting it to work on big endian hardware and by reviewing the FM-index file format to be compatible with both big endian and little endian hardware without any need of conversion.

Bibliography

[1] R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Designs, Codes and Cryptography*, pages 201–210, 1997. http://corpus.canterbury.ac.nz/.

[2] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989.

[3] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.

[4] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, Lecture Notes in Computer Science, pages 55–69. Springer, June 2003.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, 1994, 1994.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society.

[7] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Symposium on Discrete Algorithms*, pages 269–278, 2001.

[8] P. F. Giovanni Manzini. Engineering a lightweight suffix array construction algorithm. In *Algorithms - ESA 2002: 10th Annual European Symposium*, Lecture Notes in Computer Science, pages 135–142. Springer, September 2002.

[9] J. Holub. Statistical compression methods (slides for course 36KOD). Technical report, Department of Computer Science and Engineering, Czech Technical University in Prague, August 2007. http://service.felk.cvut.cz/courses/X36KOD/X36KOD-04-aritmet-4.pdf.

[10] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, volume 00, pages 251–260, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[11] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[12] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *CPM*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003.

[13] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 200–210. SpringerVerlag, 2003.

[14] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.

[15] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[16] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO 2005)*, pages 77–85. SIAM, January 2005.

Appendix A

User Manual

Synopsis: `fmi <command> <options>`

Supported commands and their options:

```
fmi c <input file> [-v] [-mX] [-bY] [-ds] [-skew]
```

The command *c* is used for creating the FM-index over the file *input file*. The output is written to the file `<input file>.fmi`. The switches *-m* and *-b* can be used for setting the constants *mdist* and *bsize* to X and Y respectively. Note that there is no space between the switch and the number. The switches *-ds* and *-skew* can be used for selecting the suffix array compression method used in the index construction. The switch *-v* enables the outputting of information about the time used to construct the index. The default behavior is `fmi c <input file> -m32 -b1024 -ds`.

```
fmi d <index file> [-v]
```

The command *d* is used for retrieving the whole original text from the index. The output file name is created by replacing the extension `.fmi` with the extension `.ori` in the input file name. The switch *-v* enables the outputting of timing information.

```
fmi n <pattern> <index file> [-v] [-nocache]
```

The command *n* is used for counting the occurences of `<pattern>` in the original text represented by the `<index file>`. The switch *-v* can be used for enabling the reporting of the time taken. The switch *-nocache* disables the caching of decompressed blocks, forcing the program to decompress them every time they are accessed.

```
fmi l <pattern> <index file> [-xN] [-v] [-nocache] [-nobitmask]
```

The command *l* is used for counting the occurences of `<pattern>` in the original text represented by the `<index file>` and reporting the positions in which they occur. The switch *-x* enables printing of the context of size N characters (on both sides) surrounding every occurence. The switch *-v* can be used for enabling the reporting of the time taken.

The switch *-nocache* disables the caching of decompressed blocks, forcing the program to decompress them every time they are accessed. The switch *-nobitmask* disables the bit mask supporting constant time checking if a given row is marked. In this case the program falls back to parsing the whole Marks Database for asserting of a row is marked. The use of this switch is heavily discouraged.

```
fmi x <index file> [-sN] [-eM] [-v] [-nocache]
```

The command *x* is used for extracting an arbitrary part of the original text represented by the `<index file>`. The starting and ending positions of the extracted part can be set to N and M using the switches *-s* and *-e* respectively. The switch *-v* can be used for enabling the reporting of the time taken. The switch *-nocache* disables the caching of decompressed blocks, forcing the program to decompress them every time they are accessed.
The default values of the starting and ending position of the extracted part are the start and the end of the original text. Be aware that using this command for extracting large portions of the text is very slow. Consider using the command *d* instead if you need to decompress more than a small part of the original text.

In the case that the program encounters an unknown command or any of the mandatory arguments are missing, it outputs it's proper usage and exits. The mandatory arguments must precede any optional switches. The optional switches can be entered in any order. In the case of incompatible switches, the last one on the command line takes precedence.