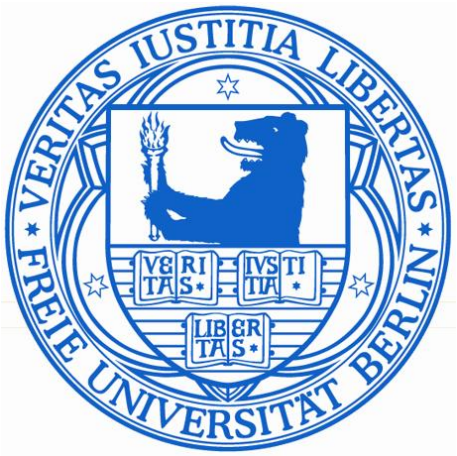
Master Thesis

A Wavelet Tree Based FM-Index for Biological Sequences in SeqAn

Jochen Singer

January 30, 2012



Freie Universität Berlin
Bioinformatik

Supervisor:
Prof. Dr. Knut Reinert

Second advisor:
Dr. Markus Bauer

Declaration of Originality

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the work of others has been acknowledged.

Jochen Singer

The technological development in the field of genome research has resulted in a massive generation of data that has to be stored and analyzed. The enormous amount of information demands special data structures and algorithms for an efficient analysis. Such an analysis often requires the identification of interesting sequences in genomes, which can be realized using full-text indices. Until recently, the major problem of this approach was its memory consumption, which now can be overcome using the well known FM-index. Therefore, in this thesis we extended the software library SeqAn that provides data structures and algorithms for analyzing biological sequences, with sophisticated FM-index versions designed for fast and memory efficient pattern search. We show that in comparison with existing FM-index implementations our variants are not only competitive to other approaches, but also outperform them.

Contents

List of Figures

List of Tables

1 Introduction

The discovery of the genetic blueprint of living organisms, the deoxyribonucleic acid (DNA), opened the door for deeper insights into the processes driving biological systems. Since many research fields like medicine and other life sciences profit of this knowledge, a lot of effort has been spent to decode the information hidden in the genome. This resulted in a fast development of new technologies generating vast amounts of sequence data to be analyzed.

For this reason, today the focus changed from data acquisition to efficient data storage and processing methods. An example addressing both issues is the so called *pattern recognition* in genomic sequences. This search is required in many genomic studies that make use of high-throughput next-generation sequencing (NGS) technologies. Here long sequences are split and then amplified into millions of shorter ones (so called reads) to identify their nucelotide sequence. To regain the original ordering of the reads, often they are mapped to a reference, where the enormous number of sequences that need to be processed requires sophisticated search strategies and data structures.

A lot of effort has been spent to develop methods that are both memory efficient and fast at the same time, since achieving this is a challenging task. One approach to derive suitable data structures is the Burrows-Wheeler Transform (BWT), which can be understood as a rearrangement of characters in a sequence. Therefore, it has been incorporated in several bioinformatic tools for read mapping, e.g. *SOAP2* [24], *BWA*, [22],*BWA-SW* [23] and *Bowtie* [20].

Another BWT based data structure is the FM-index, which is a both memory efficient and fast approach to pattern search that can be applied to a variety of applications.

For this reason, in this thesis we describe the design and implementation of the FM-index and several variants and extensions within the software library *SeqAn*[1]. This C++ based open source library is specialized on efficient data structures and algorithms to process biological sequences [5]. Therefore, the FM-index provides a desirable addition to increase the functionality of SeqAn.

1.1 Objectives

Since many applications for data analysis rely on efficient indexing strategies, we further extend the functionality of SeqAn by incorporating different variants of the FM-index.

In order to do so, our first aim is to identify and design the different components of the index. Therefore, we compare several existing approaches of FM-indices and modify them such that they fulfill the requirements of the library. In particular, the index needs to be:

- Optimized for biological sequences

[1] www.seqan.de

- Generically designed to guarantee generality as well as extensibility (the user should be able to easily exchange and modify the underlying data types of the index)
- Support fast pattern search
- Memory efficient
- Simple and robust, such that it can be understood and maintained by a variety of people

Further, we implement and combine the different components to create an FM-index fulfilling all the requirements described above.

Our next aim is to enable the index to process not only single strings, but also sets of strings. This is especially useful to efficiently represent and search NGS reads.

Finally, we aim at designing and implementing optimizations to reduce the search time requirements of our FM-index variants.

1.2 Structure of the Thesis

As described above, the FM-index combines memory efficiency with fast pattern search. Both terms will be explained in Section 2, together with the definition of frequently used terms and concepts. In addition, this section will introduce the Burrows-Wheeler transform as well as existing FM-index designs.

Section 3 will describe our approach to a FM-index implementation. In doing so, we will provide an extensive overview of the different data structures and their designs.

After describing our implementations, in Section 4 we will present the results of applying the FM-indices to sequences of different alphabets and text lengths. Further, we will compare our indices to other publicly available ones as well as the enhanced suffix array index that is already part of SeqAn.

Finally, Section 5 will summarize our achievements and provide an outlook on future work.

2 Research Context

For a better understanding of the algorithms and methods used and described in this thesis we will introduce some fundamental terms and concepts. This is followed by an introduction of the Burrows-Wheeler Transform and existing FM-index designs.

2.1 Definition and Notations

Since our goal is to design and implement an index over a sequence of characters, we start by formally introducing the concepts of *texts* and their underlying *alphabets*.

A text T is a concatenation of n characters, where the ith character is denoted by *T[i]*. We index the positions of T starting with *0*, such that the whole text is represented by *T[0, n-1]*. A *subtext* (also referred to as *substring*) is represented by *T[i, ..., j]* with $0 \leq i \leq j < n$. If $i = 0$, we call *T[0, ..., j]* a *prefix* of T and *T[i, ..., n-1]* refers to a *suffix* of T.

All characters of T belong to the same *alphabet* Σ, which is denoted as a set of distinct characters. Further, there exists a lexicographic ordering of the characters such that c_i is smaller than c_j, if c_i occurs before c_j in the lexicographic ordering of the alphabet. This relation is also expressed by $c_i < c_j$.

For a finite alphabet the number of characters defines its size $|\Sigma|$, which is usually small for biological sequences, such as genomes. A genome is often represented by four letters (*'A', 'C', 'G'* and *'T'*), which correspond to the four nucleobases adenine, cytosine, guanine and thymine of the DNA. A fifth letter (*'N'*) is sometimes introduced to cover positions with an unknown nucleobase.

In contrast to biological alphabets, we will also consider non-finite sequences with theoretical infinite large alphabets. In those cases $|\Sigma|$ is defined to be the number of different characters in T.

We refer to a character of the special alphabet $\Sigma = \{0, 1\}$ as a *bit*. A text based on such an alphabet is called a *bit string*[1]. A special kind of bit string is the *rank support bit string* that supports a *rank query* in time *O(1)*. A rank query determines the number of bits set up to a specified position.

Another fundamental data structure used in this thesis is the concept of *trees*, which are a special kind of *graph* [19].

A *graph* $G(V, E)$ consists of a set of nodes V and a set of edges E. The edge $e \in E$ connects two nodes of V and is defined as $e = \{v, w\}$ with $v, w \in V$. Furthermore, the edge e can be directed or undirected. In case of a directed edge, e is denoted by $e = (v, w)$, indicating that v is the *source* or *parent* node and w the *target* or *child* node. Therefore, e is an *outgoing* edge of v while it is an *ingoing* edge of w.

[1] Note that in this document we use *bit string* as a synonym for *bit vector*.

Even if two nodes v_0 and v_k are not directly connected via an edge $e = \{v_0, v_k\}$, they can be connected if there exists a set of edges $P = \{(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)\}$, called *path*. In the special case of $v_0 = v_k$ we call P a *cycle*.

A *tree* is a graph in which all nodes are connected via edges and that does not contain any cycles. A tree is called *rooted*, if it contains a node without any ingoing edges, which is then labelled as the *root* node. Nodes with no outgoing edges are called *leaves*. All other nodes are denoted as *inner nodes*. The special tree where the number of outgoing edges of all nodes is less or equal two is called a *binary tree*. Figure 2.1 provides a graphical representation of a tree and a binary tree.

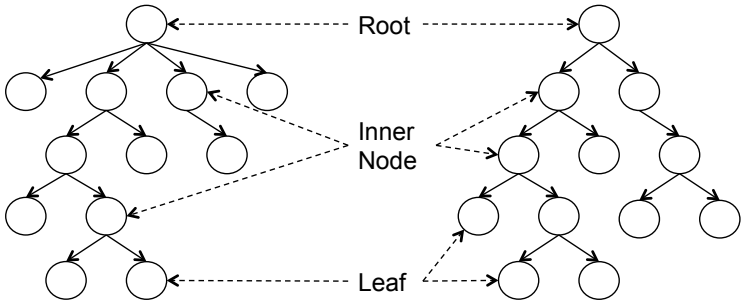


Figure 2.1: Graphical representation of a tree (on the left hand side) and a binary tree data structure (on the right hand side).

Further, a tree where the edges are labelled with characters and the nodes contain the number of occurrences of the sequence formed by concatenating the label of the edges to the specified node is called a *trie* [2].

Another special kind of tree is a *suffix tree*. A suffix tree of text T is a tree-like representation of all suffixes of T. Each edge of the tree is labeled with substrings of T, such that all suffixes are represented by a path from the root to a leaf [17].

A compact representation of a suffix tree is a *suffix array* (SA) introduced in [26] by Manber and Myers. In contrast to suffix trees, the SA does not store substrings of T, but positions of suffixes of T. In more detail, the array is constructed such that $T[SA[i]] < T[SA[j]]$, for $0 \leq i < j < n$. Therefore, the SA can be seen as an array containing the suffixes of T in lexicographic order, even though the SA does not explicitly store the suffixes of T[2]. Figure 2.2 illustrates a suffix tree and a the related SA.

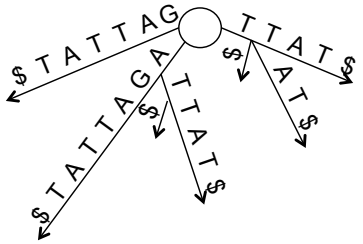
2.2 Full-Text Indices and Text Compression

Now that the fundamental concepts are introduced we change our focus to the basics of text compression.

A *full-text index* is a data structure that supports the fast search over large texts or text collections [29]. Therefore, this data structure is very interesting in the area of genome research, since a common task in this field is to identify patterns in genomes.

However, until recently, the memory consumption of those indices has been a problem [29], which especially holds in the analysis of the often very large genomic sequences. How-

[2]In this document SA will be referred to as an array of suffixes. However, in the actual implementations we only store positions of suffixes.



SA[x]	Suffix
7	$
0	A G A T T A T $
5	A T $
2	A T T A T $
1	G A T T A T $
6	T $
4	T A T $
5	T T A T $

Figure 2.2: An example of a suffix tree (on the left hand side) and the corresponding suffix array for the text "AGATTAT$" (on the right hand side).

ever, recent developments in the field of text compression have reduced the index size and introduced the concept of *self-indices*. A self-index is an index which has memory requirements proportional to its compressed text while at the same time enables the retrieval of any substring of the original text [29]. In doing so, a self-index effectively replaces the original text.

This concept is invaluable in the field of genome research, since it allows the search for patterns while compressing the data. Data compression aims at increasing data density by reducing redundancies [21]. In regard to text compression this is related to making use of regularities in a text in order to store its information and therefore using less memory than the original text file.

Text and data compression is an extensively studied research field and providing an appropriate overview would exceed the scope of this thesis by far. Therefore, the interested reader is referred to the available literature, e.g., [21], [31] or [32]. Nevertheless, in the next section we will introduce the notion of empirical entropy, since some FM-index implementations (see Section 2.5) depend on it.

2.2.1 Empirical Entropy

Data compression depends on the level of redundancy in a text, because using sophisticated methods it is not necessary to store all occurrences. The redundancy can be expressed as the level of disorder of a system, also called *entropy*. Hence, entropy provides information on the possible extend of data compression.

Note that the term entropy was first introduced to information theory by Shannon in [33]. As a consequence, in this field it is also known as *Shannon's entropy*, which is defined as:

$$H = -\sum p_i \log p_i, \tag{2.1}$$

for the set of probabilities p_1, ..., p_n.

As is illustrated in the example given in Figure 2.3 this definition can be used to determine the entropy of a system of two random variables. The example shows that Shannon's entropy is maximal if both random variables have the same probability, or in other words, when no prediction can be made which variable is more likely to occur.

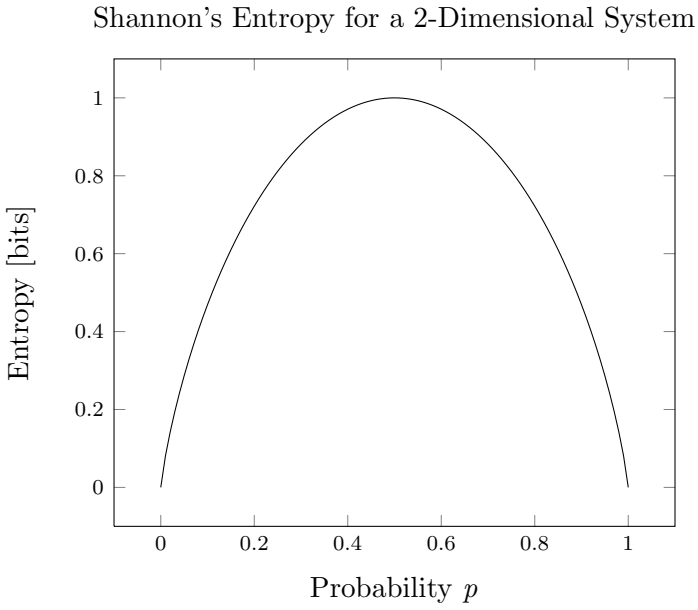


Figure 2.3: Graphical representation of the entropy of a 2-dimensional system. The two possible events have probabilities p and $q = 1$ - p, respectively.

In order to apply Shannon's entropy on a text T of length n, it is possible to replace the probabilities p_i in Equation 2.1 by the frequencies of the characters in T. In this case Equation 2.1 becomes

$$H(T) = -\sum \frac{n_i}{n} \log \frac{n_i}{n}, \tag{2.2}$$

where n_i represents the occurrences of character c_i in T.

In computer systems a character c is represented using a specified number of bits. The number of necessary bits depends on the alphabet, because every character needs a unique representation (also referred to as *codeword*). Since many texts only contain a small proportion of the available characters, it is possible to find a smaller bit representation of the characters and hence to compress the text.

If a fixed codeword is used to encode each character of a text, the best possible compression is achieved by using $\log \frac{n_i}{n}$ bits to encode each symbol [27]. Hence, H can be used to estimate a first bound for the best possible compression.

However, it is possible to achieve an even greater compression if also the characters preceding a codeword are taken into account, such as is done in the kth-order empirical entropy, defined in [27]:

$$H_k(T) = \frac{1}{|T|} \sum_{w \in \Sigma^k} |w_T| H(w_T), \tag{2.3}$$

where $w_T \in \Sigma^k$ is the concatenation of all characters of the alphabet Σ following string w (of length k) in T (for example, if T = Mississippi then $si_T =$ "sp'').

In [25] it is stated that N independent identically distributed variables can (for $N \to \infty$) be compressed with $N \cdot H(T)$ bits (where $H(T)$ denotes the entropy of each variable)

without loss of information, and with a high probability of loosing information if using fewer bits. For this reason, in contrast to $H(T), H_k(t)$ is suitable to describe the highest possible compression.

Compressing a text in the way that no information is lost will be the topic of the next section, where we will introduce the widely used *Huffman coding*.

2.2.2 Huffman Coding

Huffman Coding uses the entropy of a text in the way that symbols that are more likely to occur than others are encoded via shorter code words [18]. In addition, these codes are binary words that are chosen such that no code word is prefix of another one.

In [18], Huffman et al. present a strategy to create such a code. Note that even though the construction does not necessarily depend on a binary tree, we will make use of it for explanation purposes.

The code word construction works as follows:

1. All words of a text are sorted according to their frequency in the text.
2. A node for each word is created, storing the word itself and its frequency.
3. The two words with the smallest frequencies are combined to create a new parent node. The word associated with this node is the combination of its child node's words and the frequency is the sum of their frequencies.
4. Repeat step 3 until all nodes are connected and form a binary tree.
5. Starting from the root, label all edges such that:
   - Edges pointing to a left child node are marked with a *0*.
   - Edges pointing to a right child node are marked with a *1*.

The code symbol of a word is now the concatenation of the edge symbols on the path from the root to the leaf containing the word. The procedure is illustrated in Figure 2.4, where each character represents a word.

Huffman encoding uses the frequencies of words and in doing so their predictability to compress a text. Therefore, it serves as a good example to show the connection between entropy and compression, as explained in Section 2.2. In addition, there is a FM-index implementation employing Huffman coding, introduced in Section 2.5.4. Before explaining the details of the FM-index, we will introduce its fundamental backbone, the *Burrows-Wheeler Transform* (BWT). The BWT is of special interest in the field of data compression because it can be used to transform a text such that the transformation can be compressed to the kth order empirical entropy.

2.3 The Burrows-Wheeler Transform

The last section described the concepts of full-text-indices and text compression. In this section we will introduce a text transformation, the *Burrows-Wheeler Transform* (BWT), which can be applied to achieve both, efficient data storage and fast search query times.

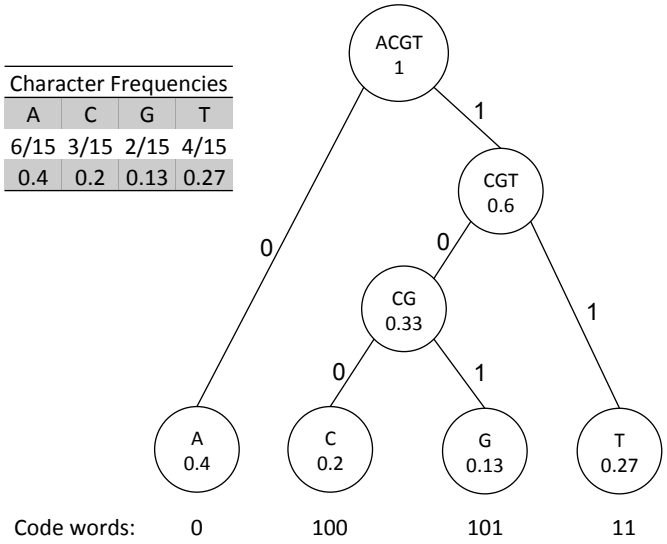


Figure 2.4: An example illustrating the Huffman encoding of the text "ACAGTATCTAGATCA".

We start by presenting the construction scheme and show the close relation between the BWT and suffix arrays. Thereby, special attention is paid to the properties of the BWT and its use for efficient pattern search.

2.3.1 Construction and Properties

The BWT is a sophisticated rearrangement of the characters of a given text T into T^{BWT} and was first described in [4]. The transformation consists of the following steps:

1. A special character *$*, which is lexicographically smaller than every other character in the alphabet, is attached to the end of T[3].
2. All cyclic rotations of the concatenation of T[i, n-1] and T[0, i-1] are formed, resulting in a table *CR*.
3. *CR* is lexicographically sorted to create a table *SCR*.
4. The last column of *SCR* is extracted to generate $BWT(T) = T^{BWT}$.

For example the BWT of "AGATTAT" yields "T$TGAATA", as shown in Figure 2.5.

The construction of *SCR* by sorting all cyclic rotations of T is time and memory consuming. However, a closer look at table *SCR* reveals a connection to the suffix array (SA) of T. To be precise, *SCR*[i] and $T[SA[i]]$ are equal up to *$*, as shown in Figure 2.6. This property is of great value, since the characters in last column of *SCR* are also the predecessors of the characters in the first column of table *SCR*. Hence, it is possible to construct T^{BWT}[i] by extracting the character $T[SA[i]-1]$. The only exception to this rule applies for the position in *SA* pointing to the first character in T, since there is no predecessor to $T[SA[0]]$. In this case $T^{BWT}[i] = \$$.

[3]This step is not part of the original construction in [4] and was introduced later for efficiency reasons.

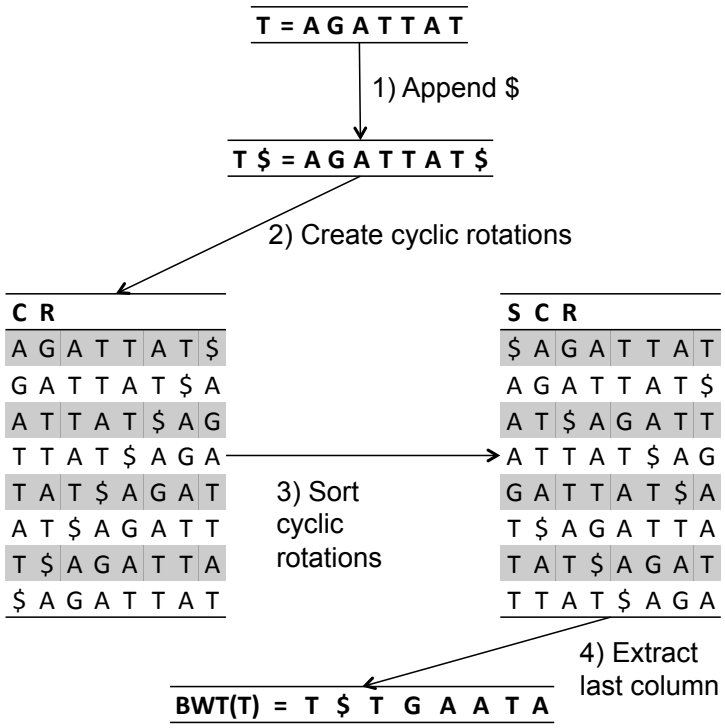


Figure 2.5: An example for the BWT construction scheme, using the text T = "AGATTAT".

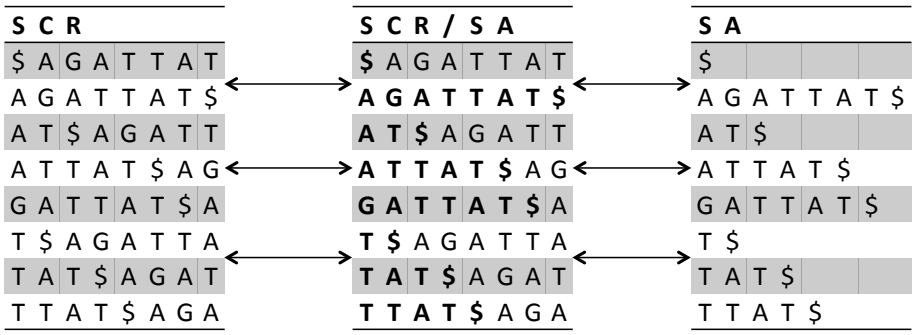
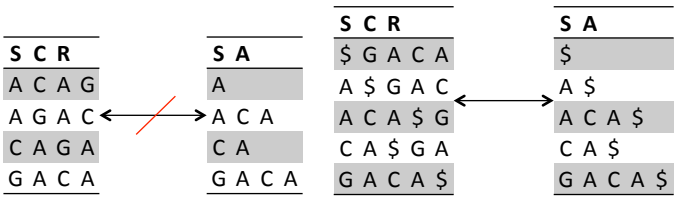


Figure 2.6: Graphical representation of the connection between table SCR and SA of a text T.

Note that Burrows and Wheeler did not include the character *$* in their proposed algorithm [4], hence it is not crucial for the general construction. However, this step becomes necessary if one is interested in using the well known and efficient suffix array construction algorithms. The usage of a suffix array construction algorithm depends on the relation between table SCR of T and the suffix array of T, which does not exist if *$* is not appended. The reason for this is a difference in the SCR tables of T with and without *$*, as can be seen in Figure 2.7.

The relation of SCR and SA, as described above, is one of the most important properties of the BWT. In the following we will focus on two additional features, which are the *last-*



(a) SCR and SA of "GACA". (b) SCR and SA of "GACA$".

Figure 2.7: Relationship between the tables SCR and SA of text "GACA" with and without appended $, where only in the latter case both tables correspond to each other.

to-first-mapping and the favourable compression behaviour.

Last-to-first-mapping The *last-to-first-mapping* (LF-mapping) describes the relation between the last column of *SCR*, from now on referred to as L, and its first column F. The LF-mapping corresponds to the following observation [7]:

Lemma: 1 *The ith occurrence of character c in L corresponds to the ith occurrence of c in F.*

For example, the second 'A' in Figure 2.5 (step 3) in L corresponds to the second 'A' in F.

Proof: 1 *Let $C = \{c_0, ..., c_k\}$, $c_i \in \Sigma$ and $c_i = c_j$ denote the set of occurrences of character c in T. Further, let $c_i \in C$ denote the ith c in column F. It is obvious that the ordering of characters c in F is determined by their suffixes in SCR (due to the construction of SCR). Because of the cyclic rotations also the ordering of the cs in column L is determined by their suffixes in SCR (which start in F). Therefore, the order of the characters c in F and L is determined in exactly the same way. Hence, the ith occurrence of character c in L corresponds to the ith occurrence of c in F.*

To realize the LF-mapping in *O(1)*, a *prefix-sum table* and an *occurrence table* of T^{BWT} are required. The prefix-sum table C of text T stores for every character c the number of characters smaller c in T. In contrast, the occurrence table Occ is a data structure that stores how many occurrences of c up to position i are observed in a given text T.

Using C and Occ, the LF-mapping of character c at position pos in T^{BWT} can be computed as follows:

1. Retrieve the number of occurrences of c up to and excluding position pos in T^{BWT} using Occ.
2. Retrieve the number of characters smaller c in T^{BWT} using C.
3. Compute the position of c in F by summing up the results of Steps *1* and *2*.

For example, for T^{BWT} = "T$TGAATA" in Figure 2.5 the LF-mapping for the first 'T' yields Occ('T', 0) + C('T') = 0 + 5 = 5. For the second 'A' in T^{BWT} the LF-mapping computes Occ('A', 5) + C('A') = 1 + 1 = 2.

As we will show in Sections 2.3.2 and 2.3.3, the LF-mapping makes it possible to retrieve T from T^{BWT} and search for patterns by simulating a suffix array search. Hence, the LF-mapping is a crucial property of the BWT.

Text compression using the Burrows-Wheeler Transform Another favourable feature of T^{BWT} is its compression behaviour, which we explain in the following. In general, the BWT results in a string containing substrings of the same characters (from now on referred to as runs). This property is induced by the sorting of the suffixes involved in the BWT construction (see previous section). This sorting causes occurrences of the same suffix to be adjacent in the SA. Therefore, it is likely to find the same character in the corresponding range of the BWT, because the number of different words in a text is limited and the same suffix is preceded by a certain one or a few characters.

A good example for this situation is an English text, if we look at the range of the suffix array containing all occurrences of "he". This tuple could be preceded by 'S', 's', 'T' or 't'. However, of all four possible combinations "the" appears to be the most likely one. Therefore, the BWT of the corresponding range will contain many 't's leading to runs of the same character.

Now one can make use of the structure of the rearranged text by applying compression techniques, such as replacing runs of the same character by a character combined with a number indicating the length of the run. This is a very simple technique, mentioned to illustrate the potential of compression schemes on the BWT of texts. In Section 2.4.1 we will show a different approach to text compression, but for more sophisticated techniques the reader is referred to the literature, for example [1].

2.3.2 Reversing

In addition to the compressibility introduced in the last section, it is possible to retrieve the original text by reversing the BWT. This is very useful, since for this reason it is sufficient to store T^{BWT} and to replace text T.

To understand how to retrieve the original text from T^{BWT}, we recall:

1. The predecessor of a character in F is the character in L in the same row.
2. The LF-mapping ensures that one can identify the same character in the first and last column.

Using the former described features and tables it is possible to reconstruct the original text from T^{BWT} with the *backward reconstruction* scheme:

1. Identify the position *pos* of the last character c of T[4].
2. Apply the LF-mapping on c to compute its position i in F.
3. Set *pos* to i.
4. Retrieve the predecessor p of c from $T^{BWT}[i]$.

[4]Note that the last character of the text is always the first character of the BWT, because of the appended '$'.

5. Set c to p.

6. Repeat steps 2 to 6 until T is recovered from T^{BWT}.

The procedure above is visualized in Figure 2.8 for T^{BWT} = "T$TGAATA".

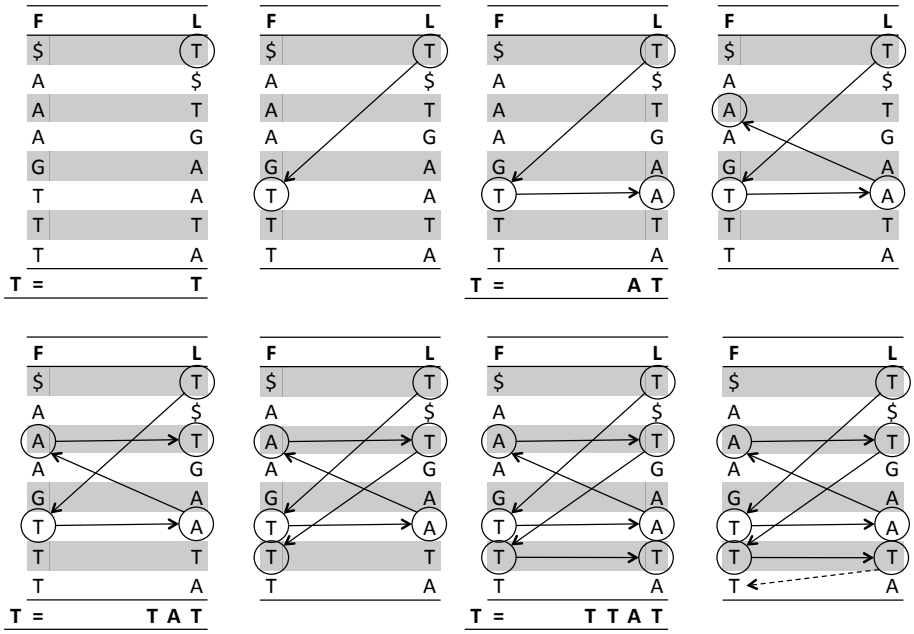


Figure 2.8: Illustration of the beginning of the reversing scheme from T^{BWT} = "AGATTAT".

The *forward reversing scheme* works in a similar fashion, apart from the major difference that one determines a characters successor by extracting the character in the same row, but first column. Afterwards, the inverse of the LF-mapping, called *FL-mapping*, has to be applied, which is marginally more complicated to implement than the LF-mapping [1].

As stated before, the LF-mapping can not only be used to compute the original string. Knowing the BWT it also allows to simulate a suffix array search, as is explained in the next section.

2.3.3 Searching

The search procedure for a pattern by using T^{BWT} is very similar to the reversing scheme introduced in the last section. Before going into more detail, we point out that suffixes with the same prefix are adjacent in the SA of T. Therefore, it is possible to split the search procedure into the *counting* and *localizing* phase. The counting phase describes the determination of a range of suffixes that share a specified prefix, whereas the localizing phase aims at identifying the patterns by following their links in the SA. As was introduced in [26], we refer to the whole procedure as *suffix array search*, which can be simulated using T^{BWT}.

Counting Phase Suffix array search is very time efficient, because it is possible to search simultaneously for all occurrences of a specified pattern P by first determining a lower

and an upper bound. The lower and upper bounds are pointers to positions in F, in the following referred to as *sp* and *ep*, respectively.

In addition to *sp* and *ep*, a pointer i is maintained that defines the current position in P. Further, let c denote the currently processed character of P.

For a pattern P of length m the search procedure initializes i to be m-1 and c with $P[i]$. The lower bound *sp* is initialized with $C[c]$ and *ep* with $C[c + 1]$. In doing so, *sp* and *ep* mark the first and the last occurrence of the last character of P in F, respectively.

After the initialization the search procedure determines *sp* and *ep* iteratively until the first character in P is reached, as follows:

1. $i = i$ - 1
2. $c = P$[i]
3. $sp = C[c] + Occ(c, sp - 1)$
4. $ep = C[c] + Occ(c, ep)$ - 1

$Occ(c, sp - 1)$ returns the number of cs in $T^{BWT}[0, sp - 1]$. Adding the number of characters smaller than c in T^{BWT} (Step 3) yields the position of the first c in the range *sp* to *ep* in F. Note that this only holds because the positions in a string are enumerated starting with 0. If the enumeration would start with 1, $C[c] + Occ(c, sp - 1)$ would point to the last c in the range from 0 to *sp* - 1. The calculation of Step 4 would have to be adjusted similarly.

Note that Steps 3 and 4 realize the LF-mapping for the first and last c in $T^{BWT}[sp, ep]$. The efficiency of those two steps relies on the availability of *Occ*, since otherwise it would be necessary to firstly locate the first and the last c in T^{BWT}[sp, ep] before applying the LF-mapping.

The whole counting phase is demonstrated in Figure 2.9 for the text "AGATTAT" and the pattern "TAT".

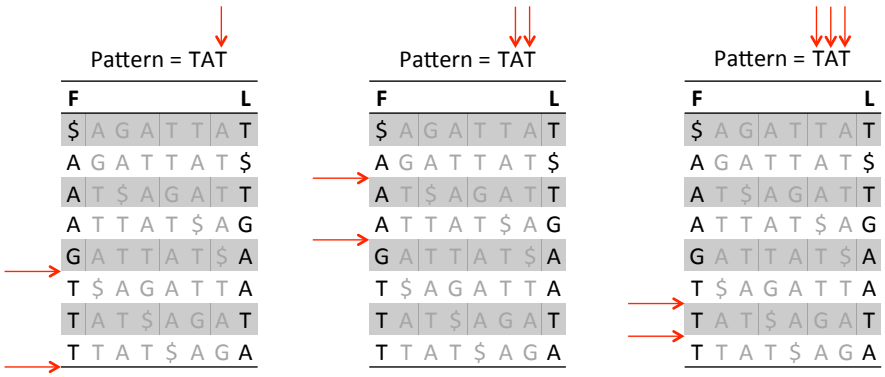


Figure 2.9: Searching for the pattern "TAT" in the text "AGATTAT". Note that the columns of table SCR between the first and the last column are only shown for demonstration purposes.

Localizing The counting procedure determines the lower and upper bound *sp* and *ep* that specify a range in SA of T pointing to occurrences of the pattern in the text. Therefore, the entries of the suffix array in this range reveal the position of matches between the pattern and the text.

However, identifying the exact positions can be a time consuming task, because in order to reduce the memory consumption many implementations do not store the entire suffix array, but only a fraction, and reconstruct missing entries on demand (see Section 3.2 for details). Therefore, the overall running time of the search is strongly influenced by the counting procedure and the suffix array access time. In the following, we will analyze the time requirements in more detail.

Running Time The running time of the search depends on the pattern length, the number of occurrences of the pattern in the text and the suffix array access time. Since searching can be separated into counting and localization, we differentiate the analysis of the search time in counting and localization.

The counting time is linear with respect to the length of the pattern, since in each iteration one character of the pattern is processed. Therefore, the dominant factor concerning the counting scheme is the pattern length. In contrast, the length of the text has no influence on the search time, which is one of the reasons why the BWT is interesting for large texts (such as genomes). In addition, the counting scheme is independent of the number of occurrences of a pattern in the text, since only the upper and lower bound of the range are determined, and the design of the suffix array has no influence since it is not accessed during the counting phase.

In contrast to the time needed to count the occurrences of a pattern, the localization does not depend on the pattern length. Instead, it is influenced by the number of pattern occurrences, since the suffix array has to be accessed for each occurrence individually. For this reason, the design of the SA is crucial for the localization time (for more details refer to Section 3.2).

This section explained how the BWT can be used to search for patterns in a text and demonstrated why this search procedure is interesting especially for large texts. In addition, we have shown how to efficiently construct the BWT, its properties and the reversing scheme. Furthermore, we explained how the BWT can be used to compress a text. In doing so, we illustrated that this method is exceptional suitable for pattern search in large sequences because it can be used to speed up search time and compress the underlying text at the same time. The next section will show how the BWT is incorporated into an index, the FM-index.

2.4 The Full-Text-Minute-Space Concept

Even though the *FM* in FM-index represents the first characters of the authors names (Ferragina and Manzini), the *FM* corresponds to full-text and minute space. A full-text index is a data structure built over a text T that supports the efficient search for substrings in T [11]. Minute space on the other hand denotes that the index is very memory efficient[5].

[5]In the following, *space* will be used as a synonym for memory.

The FM-index can be seen as a collection of tables with extra functionality (as illustrated in Figure 2.10). The composition of tables depends on the FM-index usage and can be divided into two categories. The first category consists of tables necessary to determine the number of pattern occurrences in a text. Hence, all tables supporting the LF-mapping belong to this category. To be precise, the LF-category consists of two tables, namely the prefix-sum table and an occurrence table. Recall, the prefix-sum table C of text T stores for every character c the number of characters smaller c in T. In contrast, the occurrence table is a data structure that stores how many occurrences of c up to position i could be observed in a given text T.

Using the occurrence table Occ on T^{BWT}, the ith character in T^{BWT} is the same as the one at position $Occ(i)+C(T^{BWT}[i])$ in F. Hence, the summation realizes the LF-mapping necessary for the pattern counting.

The second category comprises tables required for the localization of patterns in T. If the purpose of the index construction is to determine the number of occurrences of a given pattern *and* their locations in a text T, then the index needs to store a SA. Otherwise the SA can be omitted.

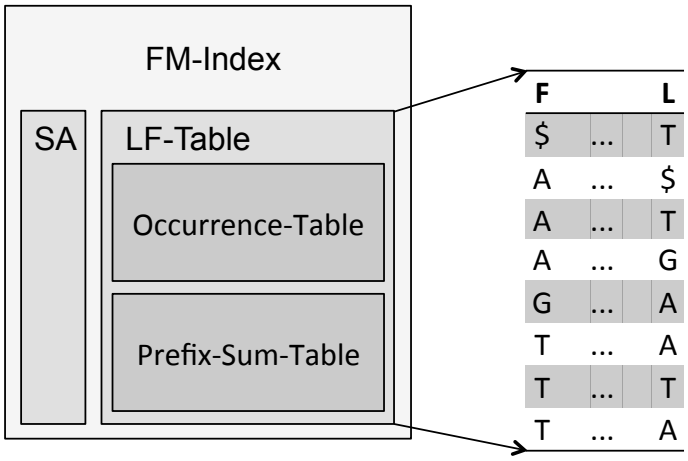


Figure 2.10: Graphical representation of the FM-index tables.

Compressed Suffix Array As mentioned in Section 2.3.3, a SA is required in order to determine the exact locations of a substring in text T. However, it is not necessary to keep the position of every suffix of T, since the LF-mapping allows to reconstruct a substring of T, starting at any position. Hence, it is sufficient to keep only a few suffix array entries and use the LF-mapping recursively to reconstruct T partially until a saved suffix array position is reached. In doing so, the memory consumption of this so called *compressed suffix array* (CSA) is only a fraction of the original space consumption. However, the fraction of saved positions has a great impact on the search time and should be chosen as a good trade-off between memory consumption and speed.

Prefix-Sum Table In contrast to the SA, for the prefix-sum table there exists no optimization mechanism in regard to the memory consumption. The reason for this is that the frequencies of different characters do not influence each other. Hence, one needs to

store the whole array. However, the prefix-sum table only accounts for a small fraction of the index, since the alphabet size of T is usually magnitudes smaller than the length of the text. This holds especially for biological sequences, for example genomes with their four to five letter alphabet, in contrast to millions of nucleotides.

In the rare case of dealing with a large alphabet one can omit the storage of the prefix-sum table. In this case the number of occurrences of character c is determined by summing up the results of occurrence enquiries of all characters smaller than c at the last position of T^{BWT}. However, this approach is very time consuming.

Occurrence Table While only minor changes have to be made in order to adapt the suffix array and the prefix-sum table, the design of the occurrence structure needs to be more sophisticated such that fast access and memory efficiency are guaranteed. In fact, the occurrence data structure is the crucial part of the FM-index and different designs have been implemented that we will explain in the following.

2.4.1 The Original FM-Index Concept

In their original work [7], Ferragina and Manzini first performed the Burrows-Wheeler Transform and then compressed T^{BWT}, denoted as CBWT, to guaranty a constant query time when determining the number of occurrences of a specified character c up to position i in T^{BWT}.

Their compression scheme is based on the assumption that T^{BWT} contains substrings of equal characters that can be represented in a memory efficient fashion. In more detail, the compression scheme consist of the following three steps:

1. **Move-to-front encoding** The move-to-front encoding (MTF) used in [7] is based on the technique introduced in [3]. The basic idea is to maintain a list of all characters (called move-to-front list - MTFL), which is initialized over the characters of the alphabet in lexicographic order. At each position the character is replaced with its rank in the MTFL and afterwards moved to the beginning. Hence, the modified text (called move-to-front text - MTFT) is turned into a sequence of integer values, most likely containing long runs of zeros due to the former applied BWT. For a graphical illustration of this first step see Figure 2.11.

2. **Run-length encoding** In order to reduce the memory usage of the runs of zeros a run-length encoding (RLE) is applied to MTFT. To be more precise, each substring 0^m is replaced by the product of:

   a) Calculating (m+1)

   b) Computing the binary representation of (m+1) and reverse the order of bits

   c) Discarding the most significant bit (which appears at the right hand side after the reversing)

   The binary representation is then stored using two new symbols **0** and **1**. For example 000 becomes:

   a) $(3 + 1) = 4$

   b) $4 = 100_2$, which is reversed to 001_2

c) $001_2 \rightarrow$ **00**

3. **Variable length prefix code** To complete the compression, the integer values in MTFT, as well as **0** and **1**, need to be represented memory efficiently. This is achieved by using two bits, 10 for **0** and 11 for **1**, respectively. The integer value m is transformed into $\lfloor log(i+1) \rfloor$ zero bits followed by the binary representation of *(m+1)*.

Step	T*	MTF
0	T $ T G A A T A	$ A C G T
1	4 $ T G A A T A	T $ A C G
2	4 1 T G A A T A	$ T A C G
3	4 1 1 G A A T A	T $ A C G
4	4 1 1 4 A A T A	G T $ A C
5	4 1 1 4 3 A T A	A G T $ C
6	4 1 1 4 3 0 T A	A G T $ C
7	4 1 1 4 3 0 2 A	T A G $ C
8	4 1 1 4 3 0 2 1	A T G $ C

Figure 2.11: Graphical representation of the move-to-front encoding.

The resulting binary string is a compressed representation of T^{BWT}. However, in order to be practical, a constant occurrence query time has to be guaranteed. In order to do so, Ferragina and Manzini partition T^{BWT} into blocks [7]. To be more precise, they introduce two layers of partitioning. The first layer consists of blocks which span a substring of T^{BWT} of length l, while the second layer consists of super blocks that cover a substring of length l^2. Afterwards, the blocks are compressed using the method described above. The partitioned and compressed T^{BWT}, enhanced with auxiliary information, can then be used to answer an occurrence inquiry in constant time. The main problem of this approach is that it is a theoretical approach that requires large amounts of memory, as we will show in the following.

For each bucket and super bucket the auxiliary information is composed of:

- Super buckets
  - NO_j - Stores the number of occurrences for all characters in $T^{BWT}[1, j \cdot l^2]$. In other words, NO_j stores the number of occurrences of the characters in T^{BWT} until super bucket j.
  - W - Stores the sum of required memory of the compressed buckets preceding the current one.
- Buckets
  - NO'_j - Stores the number of occurrences of a given character in T^{BWT} until the given bucket, starting from the preceding super bucket.
  - W - Stores the sum of required memory of the compressed buckets preceding the current one, starting from the preceding super bucket.

In addition, the following information are kept for the substrings of the buckets:

- MTF[j] keeps a picture of the state of the MTFL at the beginning of the encoding of T^{BWT} in the bucket j.
- S - Table S is the most space consuming and complicated table of the original proposed FM-index. It can be interpreted as a 4-dimensional array that stores the number of occurrences of a given character in a given substring. In more detail, S[c, h, BZ_j, MTF[j]] stores for every character c for every length h ($h \leq l$) for every possible compressed string BZ_j and every possible picture of the MTF-list the number of occurrences of c.

Using the data structure introduced above, one can determine the number of occurrences of a given character at a specific position in constant time. This is done by adding the information of the preceding super bucket and bucket to the value in table S. However, the obvious weak point of this approach is the space consumption of table S, as is explained in the following.

Consumption of Table S As explained above, table S can be referred to as a 4-dimensional array. Each dimension is described in the following:

- The key for the first dimension is a character c.
- The key for the second dimension is the length of a substring in a bucket.
- BZ_j is used as a key for the third dimension and represents a binary string of size l'. This size depends on the used encoding scheme. In this case $l' = (1 + 2\lfloor log\Sigma \rfloor) * l$ [10]. Note that [10] states that each possible compressed bucket BZ_j is represented.
- The last dimension represents all possible MTF list states, which are $2^{|\Sigma| |log|\Sigma|}$.

The overall memory consumption of table S is dominated by the factor $2^{l'}$. Even for a very small alphabet of 5 different characters and a small bucket size of 8 it holds that $l' = 40$, and therefore S contains more than 10^{11} entries. Hence, S would be larger than 1.3TB, even if one would assume that only one bit per entry is required (which is not possible in practice). For this reason, the proposed occurrence table structure is not practical. In the following section we present different approaches designed to overcome this problem. Note that reducing the buckets size is no solution because then the other introduced data structures would become too memory demanding.

2.5 Overview of Different FM-Index Implementations

As the last section showed, the originally proposed FM-index requires large amounts of memory for table S if l is not chosen to be very small. However, smaller l increase the number of required buckets and super buckets. Therefore, it is necessary to reduce the memory consumption of the occurrence data structures in order to design a practical BWT based framework for indexing. There exist several different approaches dealing with this problem, which we will introduce in this section. Since the major differences are related to the occurrence data structure we will focus the attention on this part. We start by presenting a simple solution that is strongly related to the original proposed index (see [7] for details).

2.5.1 A Practical Implementation of the Original FM-Index

In [8], Ferragina and Manzini present a practical solution of the FM-index described in Section 2.4. In contrast to the original FM-index, this solution simply refrains from storing table S. In doing so, they are able to reduce the memory consumption of the index at the cost of speed at query time. In order to compute the occurrence of a given character c up to a given location i in T^{BWT}, the substring of the bucket containing i is decompressed and the number of occurrences of c from the beginning of the bucket to i are counted.

The major advantage of this approach is the small memory consumption. In fact, using move-to-front encoding followed by run-length encoding and a statistical compressor on T^{BWT}, it is not only competitive to the best known compression schemes, but possibly superior [9].

However, the process of decompression necessary in each occurrence query involved in the LF-mapping is time consuming. For this reason, several FM-index implementations with different approaches to overcome this shortcoming have been published, such as the run-length FM-index that will be introduced next.

2.5.2 Run-Length FM-index

Even though the approach described above is very memory efficient [8], it is not optimal concerning the time consumption for pattern query times. In contrast, the run-length FM-index introduced in the following does not need to decompress sections of T^{BWT}. Instead an occurrence query is performed in constant time.

The run-length FM-index described in [28] uses runs of equal letters in T^{BWT} for the compression. An array R, which is a concatenation of representatives of each run of characters, and a bit vector B marking the start positions of the character runs, replace T^{BWT} (see Figure 2.12). An additional bit vector B' marks the starting positions of the sorted character runs. The sorting scheme sorts the runs according to their character value. However, the order of runs of equal characters remains conserved. Further, a prefix-sum table C_R of R is constructed.

Applying these data structures, Makinen and Navarro [28] use the following two equations for the LF-mapping:

$$C[c] + Occ(T^{BWT}, c, i) = select(B', C_R[c] + 1 + Occ(R, c, rank(B, i))) - 1 \tag{2.4}$$

for any $c \in \Sigma$ and $1 \leq i \leq n$, such that $BWT[i] \neq c$. Note that the function *rank* returns the number of bits set up to position i and *select* returns the position of the ith bit set to 1.

$$C[c]+Occ(T^{BWT}, c, i) = select(B', C_R[c]+Occ(R, c, rank(B, i)))+i-select(B, rank(B, i)) \tag{2.5}$$

for any $c \in \Sigma$ and $1 \leq i \leq n$, such that $BWT[i] = c$.

Nevertheless, this approach needs an additional data structure to ensure a constant occurrence query time using R. In order to do so, array R is replaced with a two dimensional array of bit values that encode R. In detail, position i in row R_c is set to 1, if and only if R[i] = c (see Figure 2.13). It is worth mentioning that the rank determination of a specified position in a bit string needs additional information, as described in Section 3.1. However, this additional information only needs sub-linear space [28].

BWT	A	A	A	A	C	C	A	G	G	T	T	T	T	A	A	A	T	C	C	C	T	T	T	A	A	A	G	A	T	C	C	C	A	A
R	A	C	A	G	T	A	T	C	T	A	G	A	T	C	A																				
B	1	0	0	0	1	0	1	1	0	1	0	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1	0
B‘	1	0	0	0	1	1	0	0	1	0	0	1	1	0	1	0	1	0	0	1	0	0	1	0	1	1	0	0	0	1	1	0	0	1

Figure 2.12: An example illustrating the data structures involved in the run-length FM-index.

R	A	C	A	G	T	A	T	C	T	A	G	A	T	C	A
R_A	1	0	1	0	0	1	0	0	0	1	0	1	0	0	1
R_C	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
R_G	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
R_T	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0

Figure 2.13: Binary representation of R.

2.5.3 A Wavelet Tree Based FM-Index

Even though the run-length FM-index has a better query time than the original index, it is not as memory efficient. Now, alone for the two bit vectors B and B' $2n$ bits have to be stored. In addition, the size of R strongly depends on the length of zero runs that determine the length of a single row in R. Those shortcomings are overcome in the wavelet tree based version of the FM-index [12] at the cost of pattern query speed, as we illustrate next.

A *wavelet tree* is a binary tree of bit strings that represents a given text T [16]. For an alphabet Σ and a text of length n the tree needs $O(\log n)$ bits of storage and supports the determination of the character of a specified position in $O(\log |\Sigma|)$ time. In addition, it allows to obtain the number of occurrences of a given character up to a specified position in $O(\log |\Sigma|)$. The only requirement is the availability of a data structure that supports rank operations on bit strings in constant time (see Section 3.1 for details). Nevertheless, the wavelet tree can be seen as a 0th-order compressor [12]. Details on wavelet trees can be found in Section 3.4. For now it is sufficient to treat wavelet trees as 0th-order compressors, which at the same time represent the occurrence table of a text.

In order to achieve a better compression, in [12] the authors make use of the recently introduced *compression boosting* [9, 10, 13]. The idea behind compression boosting is to use a 0th-order compressor (that does not incorporate any context information) as a kth order compressor (that uses the "best possible" context), via the booster [12]. Such a compression booster is introduced in [6]. It determines the best compressible contexts t_0, t_1, ..., t_{n-1} of a text T (in our case T^BWT).

Combining the different techniques Ferragina and Manzini first use the booster to get the contexts t_i of T^BWT of text T. Afterwards, the beginning of each t_i is marked in a bit string. In the last step the prefix-sum table $C_i[0, ..., n-1]$ and a wavelet tree are computed for every context.

Note that even though the theoretical compression is close to the kth-order entropy, the overall result strongly depends on the length of the contexts, since the structure of the wavelet tree of each context needs additional memory as well as the prefix-sum table.

In contrast, the following approach for an efficient FM-index implementation does not need to store a tree or several of the prefix-sum tables. Instead, it is based on a Huffman encoding (see Section 2.2.2) before computing the occurrence table.

2.5.4 A Huffman Based FM-Index

A different approach to the original, run-length or wavelet tree based FM-indices is the Huffman based FM-index presented in [14] and [15].

The idea of this approach is simple: first use Huffman encoding (Section 2.2.2) on the text to obtain *T'*. Afterwards, the BWT is applied on *T'* to determine the bit string *B*. This bit string already realizes the occurrence table in the FM-index. This method has the advantage that the determination of an occurrence in *B* is reduced to a rank query. For the same reason there is no need for an additional prefix-sum table. However, in order to search for a pattern, the pattern itself has to be Huffman encoded as well. Further, an extra bit string *B'* is needed to mark the beginnings of the Huffman code words, since Huffman-transformed patterns can match in the middle of a codeword in *T'*.

The major advantage of this approach is its simplicity. Further, the authors claim that their method is the fastest and smallest index compared to the previously described ones [15]. This comes as a surprise because the utilized compression scheme is a *0*th-order compressor as opposed to the kth-order compressor scheme used in [12]. Furthermore, the search procedure has to Huffman encode each pattern, which needs extra preparation.

We postpone the discussion of possible reasons for the superior performance of the Huffman based FM-index to the next section, where we will compare the different approaches and reason why we chose to implement a wavelet tree based FM-index version in this project.

2.6 Reasons for a Wavelet Tree Based FM-Index in SeqAn

With the introduction of the original FM-index concept (see Section 2.4.1) it was shown that, due to the size of the occurrence table, this method is not feasible for a practical implementation. As presented in Section 2.5, various approaches exist that try to efficiently realize the occurrence table, however for our implementation we have chosen to use a method based on wavelet trees. In the following we will give reasons for our choice.

The practical implementation of the original FM-index [8] described in Section 2.4 has a major drawback, which is the dependence on a partially decompression of T^{BWT}. This step is time consuming and has a negative effect of the bucket sizes of the occurrence table. In addition, the compression scheme is a 0th-order compressor which does not justify the time trade-off. Therefore, this approach seems to be rather unsuitable for the SeqAn software library.

The next approach by Mäkinen and Navarro [28] does not need to decompress T^{BWT}. Instead, two bit strings of length n and a 2-dimensional auxiliary table (which strongly depends on the underlying text) are constructed, such that the search procedure is independent of the alphabet size. In doing so, Mäkinen and Navarro created a fast index. However, the dependency of their approach on long runs of zeros in the run-length encoded BWT is a bottleneck. We will see later that our approach only needs $2n$ bits for the occurrence table for *DNA* (for general alphabets $log|\Sigma| \cdot n$), without the need for an extra auxiliary table.

In [12], Ferragina and Manzini introduced the first FM-index with a kth-order compressor. Their approach yields a very good theoretical compression. However, it is questionable whether this compression is much better than an *0*th-order compressor, since they have a memory overhead due to the requirement of storing the tree structure for every context t_i. In addition, this approach will waste memory by storing the bit strings of the wavelet trees. The reason for this is that it is not possible to address a single bit on a computer. Instead, one has to allocate a block of bits. If the length of the context t_i is not a multiple of this block size memory will be wasted, or one has to shift bits which will have an negative impact on the pattern query time. Furthermore, the boosting scheme introduces more complexity, which is not desired for a simple and robust data structure.

In contrast to the wavelet tree based approach by Ferragina and Manzini [12], the Huffman based FM-index [15] uses a *0*th-order compressor. However, the authors claim to be more memory efficient than the other FM-index variants, if applied to DNA. A reason for this phenomenon could be that the runs of zeros produced by the move-to-front encoding are short, which has a negative effect on the memory consumption of the run-length FM-index (see Section 2.5.2), and that the kth-order entropy is not much smaller than the *0*th-order entropy, which negates the theoretical advantage of the wavelet tree based approach. In this case the previously discussed memory issues of the wavelet tree outweigh the theoretical advantage of the wavelet tree based FM-index. Further, the fastest index is the Huffman based approach [15], though it requires each pattern to be Huffman encoded before the pattern search.
Transforming the pattern into a Huffman code creates complexity and becomes a disadvantage if the pattern has to be changed constantly during the search, which is the case in many bioinformatic applications, such as read mapping, where sequencing errors have introduced changes in the patterns.

Considering all advantages and disadvantages, we chose to implement a wavelet tree based FM-index, because it is a fair trade-off between dependency on the alphabet, compression efficiency and pattern query time. However, in contrast to the approach described in [12], we chose to omit the compression booster and build a single wavelet tree over T^{BWT}. In doing so, we make use of a *0*th-order compressor and create a simple and robust index, which is especially fast on biological sequences. To summarize, the reasons for choosing a wavelet tree based FM-index are:

- In contrast to [8], no decompression of any subsequences of T^{BWT} is necessary.
- No modification of the pattern is necessary such as required in [15]. Therefore, our approach is also suited for applications in which the pattern is modified, e.g. read mapping algorithms.
- The wavelet tree is a *0*th-order compressor. Therefore our theoretical compression is at best as good as the one obtained in [12]. However, we only need to store one tree structure and a few bit strings. Since the bit strings are rather long, the ratio of contained information to memory allocated is very high, which is not necessarily the case in [12]. In fact, for the DNA alphabet the memory consumption of the occurrence table of the new wavelet tree based FM-index is about half of the one required in [15]. In this case, our implementation needs about *2n* bits (plus auxiliary

memory for the constant time rank query support) while the one in [15] needs *2n* bits for B and *2n* bits for B' (for the alphabet $\Sigma = \{A, C, G, T\}$)[6].

- The design and implementation of a data type *wavelet tree* will include a desirable feature in SeqAn, because it can replace more costly data structures in other applications.
- Wavelet trees are a simple and robust data structure. Hence, their implementation is easily maintained and understood also by non-experts, which is an important feature of a fast developing software library.

[6] This statement is based in the assumption that on average two bits are required to encode one nucleotide. This is a reasonable assumption, since in most genomic sequences the frequencies of the four nucleotides are fairly similar.

3 Approach and Methodology

In the last section we have shown that the composition of the FM-index has a strong influence on the index compressibility and pattern search time, where in particular the occurrence table needs to be designed carefully. We will present two approaches for an occurrence table design, namely a wavelet tree based FM-index and, for comparison, a FM-index version based on a 2-dimensional occurrence table.

Before providing a detailed explanation of the construction of the occurrence table, we will present the design of data structures that are part of both FM-index versions, such as the rank support bit string and the compressed suffix array.

An overview of the different data structures is given in Figure 3.1.

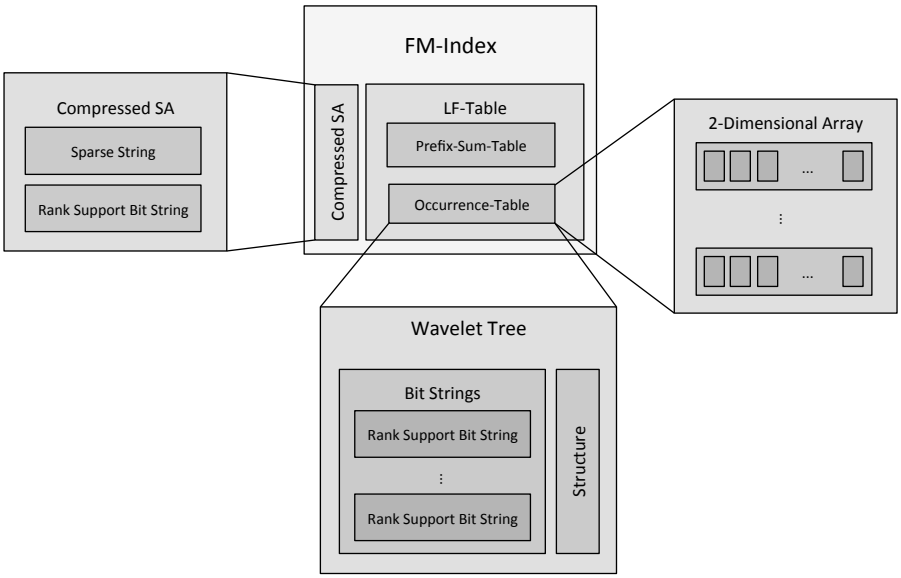


Figure 3.1: Structure of the FM-index with its main tables.

3.1 A Constant Time Rank Query Data Structure

A rank query of a specified position i on a bit string B returns the number of bits set in $B[0, i]$. Without additional information, each position of B needs to be accessed to check whether or not the corresponding bit is set. The running time of this approach is $O(n)$, which is problematic for large bit strings.

The main idea how to turn a bit string into a rank support bit string is to associate rank information with intervals of the bit string. The interval information is then used such that only a constant number of operations has to be performed.

In order to do so, we divide the bit string into buckets of size l and super buckets of size l^2, as was done in [7] and [10]. Afterwards, we create a bucket string BS of size $\frac{n}{l}$ and a super bucket string SBS of size $\frac{n}{l^2}$. The entry i in SBS stores the number of bits set in $B[0, i \cdot l^2 - 1]$, while $BS[i]$ returns the number of bits set in $B[\lfloor \frac{i}{l} \rfloor, i \cdot l]$. A graphical representation of the bucket structure is shown in Figure 3.2.

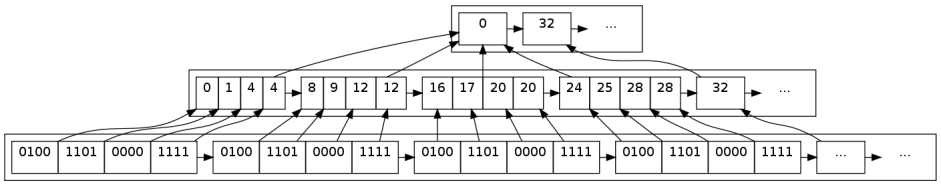


Figure 3.2: Structure of a rank support bit string.

Using this bucket scheme, a rank query of position i on B is reduced to the summation of a super bucket, a bucket and the rank of bits of the bucket containing i. More formally, the rank query is reduced to:

$$rank(i) = SBS[\lfloor \frac{i-2}{l} \rfloor - 1] + BS[\lfloor \frac{i}{l} \rfloor - 1] + rank(B[\lfloor \frac{i}{l} \rfloor, \lfloor \frac{i+1}{l} \rfloor - 1]). \tag{3.1}$$

Note that $rank(B[\lfloor \frac{i}{l} \rfloor, \lfloor \frac{i+1}{l} \rfloor - 1])$ has to process at most l many bits. Since l is a fixed constant the whole rank query needs *O(1)* time.

Even though, the search time directly depends on the chosen l. Therefore it is desirable to decrease l as far as possible. On the other hand, Figure 3.3 shows that the memory consumption increases with smaller l.

Another restriction on l is the availability of data types usable to implement the bit string. Since the smallest addressable unit is the *byte*, it is impossible to directly address each bit of a bit string. Instead, it is necessary to implement the bit string as a string with an underlying data type, e.g. *bool, char, int*, etc., and use bit shift operations as well as logical operators to access a specified bit.

For example, in order to check whether the 4th bit in a bit string bS, implemented as a string of chars, is set, the following operations have to be performed:

1. $result = bS[\lfloor \frac{12}{8} \rfloor] >> (12 \ modulo \ 8 - 1) = bitString[1] >> 3$
2. $result = result \ \& \ 1$

Therefore, serial counting the number of bits set in a bucket requires at most l shift operations and l logical operations.

However, there exist methods to count the number of bits in a non-serial fashion, which requires even less operations, such as implemented in *__builtin_popcount*, provided by the GCC compiler[1]. In order to apply *__builtin_popcount*, the data type used to implement the bit string has to be *unsigned int*, *unsigned long* or *unsigned long long*. Therefore, l can not exceed 64.

For three reasons, we chose *unsigned long long* to be the underlying default data type of our bit string. Firstly, we can make use of *__builtin_popcount* and sacrifice memory

[1] http://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/Other-Builtins.html, received January 28, 2012

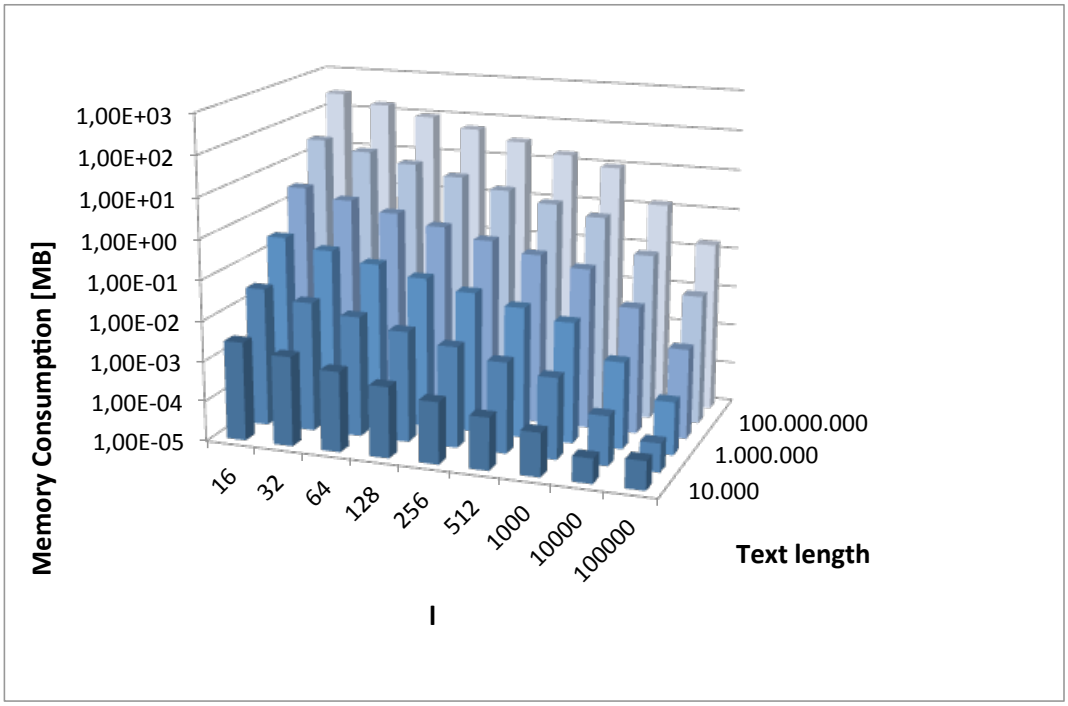


Figure 3.3: Memory consumption of the bucket and super bucket string for different l's and text length.

efficiency for speed. Secondly, $l = 64$ is more memory efficient than using *unsigned int* or *unsigned long* (the size of the latter depends on the operating system) and thirdly an l that differs from 64 would require additional bit shift operations, since the bucket borders could be in different entries of the underlying string.

As a consequence, we made *unsigned short* the default data type of the bucket string. An *unsigned short* requires 16 bits, which is larger than the 12 bits necessary to encode $l^2 = 4096$, and at the same time it is the smallest directly accessible data type. Again we sacrifice memory for speed, to be precise 4 bits per entry.

In contrast to the bucket string, the super bucket string is independent of the data type of the bit string. Instead, it depends on the text length, which is different from instance to instance. Therefore, we chose *unsigned int* to be the default underlying data type of the super bucket string. In doing so, we limit the text length of the default implementation to 4,294,967,296 characters, which is sufficient for most applications.

In addition to the default implementation, the user can use metafunctions already implemented in SeqAn (such as *Fibre*, introduced in Section 3.6) to overload the default settings of the rank support bit string.

By carefully choosing the underlying data types of the rank support bit string, it is possible to reduce the memory consumption of the index, since the rank support bit string is used in other data structures, such as the compressed SA, which we explain in the following.

3.2 The Compressed Suffix Array

The suffix array of the original text is essential to recover the positions of a pattern in the text. Nevertheless, the suffix array needs at least $\Omega(n \cdot logn)$ bits of memory. This is problematic for large sequences and can be reduced by using a compressed suffix array (CSA).

Instead of storing the positions of all suffixes of a text T, a selection of entries is kept. In the case of the FM-index this is a valid solution, since it is possible to retrieve the position of a suffix in between two stored positions by consecutively using the LF-mapping. In other words, if there is no entry for a specified position x in CSA, the original text is recovered until a text position with a corresponding entry CSA$[y]$ is reached. Therefore CSA$[y] + i$, with i being the number of necessary LF-mappings, represents SA$[x]$.

The SA access time depends on the fraction of stored entries in the CSA, as well as its values. For example, if every xth position in the uncompressed SA is chosen to be stored, in the worst case the CSA could consist of neighbouring text position. In this case the SA access time would be *O(n)*.

In order to achieve an access time of $O(\frac{1}{x})$ we use a rank support bit string to mark every xth position in the text and store those suffix array positions in the CSA. Hence, the CSA consists of a sparse string storing SA values and an indicator bit string.

Obviously there is a trade-off between memory consumption and running time. Considering this, the user should choose low compression rates if enough memory is available.

3.3 The Prefix-Sum Table

Like the SA described in the section above, the prefix-sum table is a data structure required by all FM-index variants.

For small alphabets it is sufficient to create an array of the size of the alphabet. The different entries are then addressed using a mapping function, conserving the ordering of the characters during the mapping process.

Note that the described method is not applicable to large alphabets. In these cases the prefix-sum table is implemented as an array of tuples. Each entry stores the character it represents and the number of characters smaller than this character. Note that it is not any longer possible to directly address an entry. Instead, it is necessary to use a mapping function or search for the location in the array. We have chosen the latter, because a mapping would either require additional memory or perform a search as well.

3.4 Wavelet Trees

In contrast to the previous data structures, in the following we will describe an index specific one, namely the wavelet tree.

A wavelet tree encodes a given text T as a binary tree. The tree is constructed by defining subtexts for each node which are then encoded by bit strings. Those are generated by comparing elements of the subtexts to a pivot element p. Each character c smaller than p is represented by a '*0*', while characters greater or equal than p are encoded by a '*1*'. Now the bit string defines the strings of the child nodes, where all characters represented by '*0*' form the new substring of the left child and all characters encoded

by '*1*' define the substring of the right child node. A wavelet tree for the sequence: "AGCTAGCTCATACAGGGTATGACCAGTACGACAG" is shown in Figure 3.4.

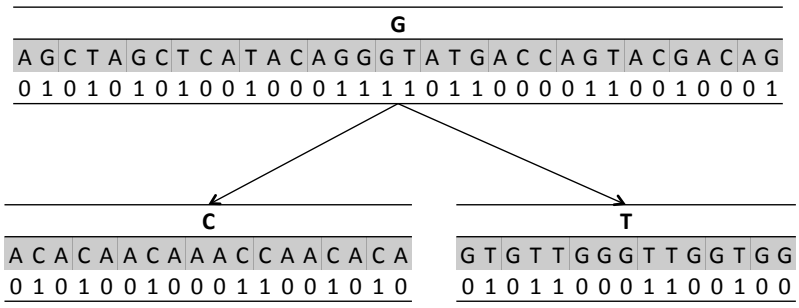


Figure 3.4: Graphical representation of a wavelet tree. Note that the character strings are shown for clarity but are not stored in the actual tree structure.

In order to compute the rank of a given character c in T up to position i, one determines the rank of *0s* if c is smaller to the pivot element p of the current node or the rank of *1s* if c is greater than p. The result is a new position i that is used in the next computation. Afterwards, this procedure is repeated in the left child, if c is smaller than the current p, or in the right child, if c is greater or equal than p, until the last node containing c[2] is reached. In total this procedure requires $O(log|\Sigma|)$ many rank queries for an balanced wavelet tree, leading to the proposed running time. However, it is still possible to increase the performance, as we will show in the next section.

3.4.1 Optimal Rank Query Time in Wavelet Trees

The former explained running time of $O(log|\Sigma|)$ can be optimized by a more sophisticated choice of pivot elements p.

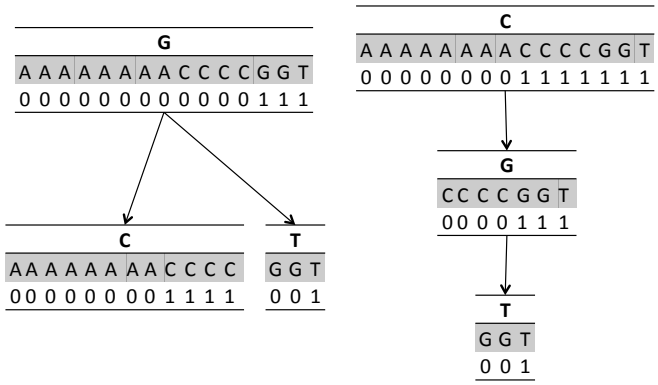
We reduce the number of involved nodes on the average rank query by choosing the pivot elements according to the frequency of the characters in text T. To be more precise, p is chosen such that the numbers of *0*s and *1*s in a bit string is as equal as possible. The influence of the pivot elements on the tree structure is illustrated in Figure 3.5.

The structures of the trees in Figure 3.5 have a strong impact on the rank query time. For example, if one wants to compute the rank of 'A' at any position using the tree in Figure 3.5(a), only one rank query is needed, whereas 'G' or 'T' require three rank queries. However, on average this is better than using the tree of Figure 3.5(b), assuming that the frequencies of the characters in text T reflect the expected number of rank queries of the characters.

3.4.2 Wavelet Tree Implementation Design

How to achieve an optimal rank query time was described in the last section, where we have shown that the tree structure has a major influence. However, the impact of the implementation design on query time and memory consumption must not be neglected and will be described in the following.

[2]The last node containing c does not have to be a leaf, as can be seen in Figure 3.5(b)



(a) Unbalanced wavelet tree (b) Balanced wavelet tree

Figure 3.5: Influence of the pivot elements on the wavelet tree structure.

A fast solution for determining the position in a wavelet tree would be to label all nodes with their corresponding position in an array, as can be seen in Figure 3.6.

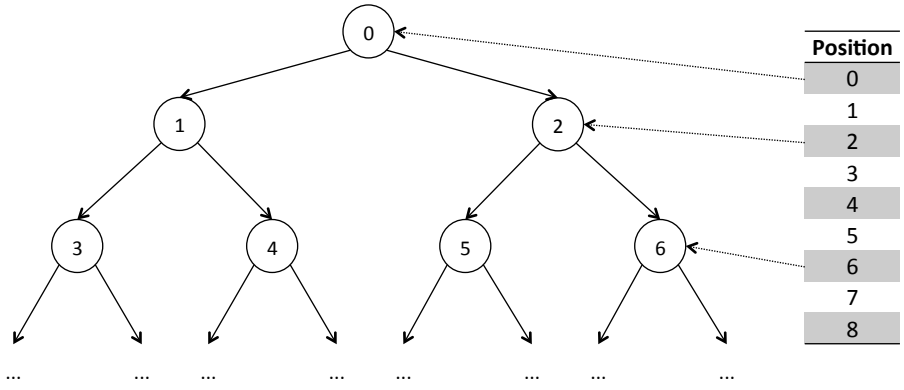


Figure 3.6: Possible implementation structure of a wavelet tree.

However, this approach has a major drawback, namely its memory consumption. For example, in order to store the nodes of Figure 3.5, seven entries would have to be generated, even though the graph consists of only three nodes. This effect becomes even more prominent with larger alphabet sizes. In fact, using this scheme one would have to consider $2^{|\Sigma|} - 1$ many entries, even though the number of nodes, including leaves, in a wavelet tree is at most $|\Sigma| - 1$.

For this reason, our wavelet tree structure is encoded using an array in which every entry stores the pivot element of its corresponding node and a link to the right subtree. We do not need to store the location of the left subtree, because, per default, its location is one to the right of the current node in the array. Nevertheless, we need to encode the possibilities of having no child node, having only the left or right subtree as a child node or having two subtrees. Therefore, a link to the root encodes the situation of having no child nodes, a *1* or *2* showing that only the right, respectively the left, child node exists and values greater than *2* indicate two child nodes. Figure 3.7 provides a graphical

representation of our approach.

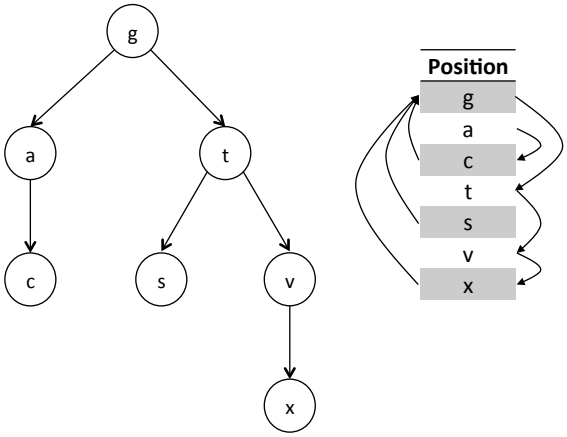


Figure 3.7: Memory efficient design of a wavelet tree structure.

This section provided detailed information on the usage of wavelet trees as a substitute for the compression scheme in [7] that is memory efficient and fast at the same time. The next section will introduce our design of an occurrence table based on a 2-dimensional array.

3.5 The 2-Dimensional Occurrence Table

In contrast to wavelet trees described above, a simple 2-dimensional occurrence table is memory inefficient. To be precise, the memory consumption is $O(|\Sigma| \cdot n)$ for a text T over the alphabet Σ of size n, since for every character $c \in \Sigma$ the array keeps for every text position $i < n$ the number of cs up to i in T.

Because the frequencies of different characters in T are usually independent of each other it is impossible to retrieve information on the number of occurrences of one character if its frequency is not known. Therefore, it is required to store information of all characters to realize the occurrence table. However, it is unnecessary to store the number of cs up to i for each position $i < n$. Instead, it is sufficient to keep the number of occurrences of c up to i in specified intervals. The number of occurrences of c between two stored positions x and y with $x < y$ can be computed by counting the number of occurrences of c in T[x + 1, i] and adding them to $Occ[x][c]$.

3.6 Appropriate Data Types of the FM-index

During the last sections we have introduced all data structures required by the different FM-index versions. In this section we will describe how appropriate data types for these structures are chosen.

First of all we want to recall one fundamental concept of the SeqAn library, the usage of templates to specify data types at compile time. This design ensures a speed advantage since data types do not have to be determined at running time. On the other hand, all data types have to be known at compile time, which introduces a problem for the FM-index.

The underlying data types of the compressed SA, the 2-dimensional occurrence table as well as the data types of the rank support bit string depend on the length n of the text T. Since the largest possible value necessary to be stored by those data structures is (n - 1), it is sufficient that their strings have underlying data types of $\lceil \log(n-1) \rceil$ bits.

In order to ensure the usage of the most efficient underlying data types for the different tables of the FM-index, in our implementation we made use of the metafunction *Fibre*, which returns the type of a specified data structure. Note that there exists a *Fibre* for each text dependent data type of each data structure.

For example, the *Fibre* of the compressed SA defines the sparse string to be a *string* of *unsigned int*s. We chose unsigned int because it can store values between 0 and 4,294,967,295 which is sufficient for most applications.

In addition to the default implementation, the user can specialize the *Fibres* for a specific purpose. For example, if the text length is smaller than 65,536 it is sufficient to use a data type of 16 bits, such as *unsigned short*, as the underlying data type for the sparse string of the compressed SA.

In addition to the data types, the strategy to represent the *$* has to be chosen carefully, which we will explain in the next section.

3.7 Handling of Alphabet Extrinsic Characters

The last sections have shown our design of the FM-index tables. However, in order to implement a robust FM-index applicable on all alphabets special attention has to be paid to the alphabet extrinsic character *$*. As explained in Section 2.3.1, the *$* sign poses an essential modification to the original proposed BWT construction that is necessary to use suffix arrays during the FM-index computation. In the following we propose several approaches to represent the *$*.

A simple solution would be to use the smallest character c of the alphabet as a substitute for the *$* sign. Using the smallest character is essential, since the *$* has to be the lexicographic smallest character. However, this solution is only practical in cases where c does not appear in the underlying text of the index. Because this restriction can not be met in many cases, such as in the case of genomic sequences (encoded in a DNA alphabet), the proposed idea is not practical for our purpose.

Another solution would be to create a new alphabet. However, this approach is only practical for certain alphabets. Therefore it is not possible in many cases, especially if the user defines a special alphabet.

A practical solution is to substitute the *$* with the least frequent character c in the text and store its position. In doing so, it is possible to circumvent the problem of creating a new alphabet for T^{BWT} and the text is not limited to certain characters of the alphabet. The obvious drawback is an additional check in each occurrence query to determine if it is necessary to subtract the result by 1. This is the case when determining the number of occurrences of the least frequent character behind the *$* position. Note that by choosing the least frequent character as the *$* substitute we reduce the number of necessary checks to a minimum.

In addition to the previous approach that we implemented for the wavelet tree based as well as the 2-dimensional array based FM-index, we designed an alternative approach for the wavelet tree version of the FM-index. Again, we substitute the *$* with a character

c, but in order to circumvent the check necessary in the previous version we insert an additional node in the wavelet tree.

This extra node v is inserted as the child of the leaf l containing c. Therefore l becomes an inner node. In addition, we assign c to be the pivot element of node v and resize the length of its bit string to be the number of cs (note that the dollar is included in this number). Afterwards, the bits of the new bit string are set accordingly.

This approach is superior concerning rank query time in situations where the frequency of one of the characters is very low. In those cases the leaf containing the *$* will be reached seldomly. Therefore, the additional node influences the rank query on a very low level within the tree, but it makes the dollar position comparison dispensable. Hence, for certain text compositions the alternative *$* handling will be more efficient.

3.8 Search Speed Optimization

The compressed suffix array introduced in the last section has a strong influence on the localization of a pattern in the original text. However, since the pattern search also depends on the counting method, in this section we will show two alternatives to enhance the search time by modifying the counting procedure.

3.8.1 Range Control

Our first approach is based on the idea that unique patterns do not require the determination of the upper and lower bounds *sp* and *ep* in every step of the counting routine. Instead, it is sufficient to determine only one of the two (since they must be equal), which reduces the time requirements.

An advantage of the range control optimization is its simplicity, though a major drawback is the dependence on the occurrence determination of a character at a specified position, since this step has a logarithmic relation to the alphabet size. In the following we present an alternative that overcomes this shortcoming at the cost of additional memory allocation.

3.8.2 Text Verification

In order to minimize the number of necessary LF-mappings, now we introduce a text verification based search procedure.

The idea is similar to the one presented above: it is not necessary to determine both *sp* and *ep* for unique patterns. However, instead of computing one of the two bounds, one verifies the remaining prefix of the pattern against the text once *sp* and *ep* are equal. The advantage of such a procedure is the direct access and verification of characters. On the other hand, the memory requirement increases because the whole text hast to be stored, since the reconstruction using T^{BWT} would mean to indeed mimic the range control optimization scheme presented above.

3.9 Suffix Trie Simulation

The search speed optimizations are strategies to make the FM-index more suitable for pattern searches. However, the described optimizations do not take into consideration that

in many applications the pattern contain errors. For example, the reads in a sequencing project are usually error prone, such that a strict pattern search does not yield the desired results.

Therefore, we implemented an iterator which can be used to traverse T^{BWT} as if traversing a suffix trie. Note that we do not store the trie structure, but instead generate parts of the trie on demand. In general, the user can choose to go down, go right or go up in the trie. In the first case the child node of the current node following the edge with a user defined label is generated. If no label is specified, the node following the edge with the lexicographic smallest label is constructed. In the second case, the node in the same trie following the edge with the next larger edge label compared to the current incoming edge label is constructed. Finally, in the last case, the parent node becomes the current node and all information on the former current node is discarded. In detail, the iterator works as follows:

The iterator processes T^{BWT} starting with the last occurrence of a specified character, similar to the counting procedure described in Section 2.3.3. The major difference is an additional stack that stores for every processed node x of the trie a range $[sp_x, ep_x]$, instead of overwriting *sp* and *ep* in each iteration. In doing so, a history of steps is created that can be used to identify patterns that differ from the text.

The bounds sp_x and ep_x specify a range in F (first row of *SCR*, see Section 2.3) containing all words who's last characters match the edge labels from the root to the current node x of the trie. Note that since the procedure starts with the last occurrence of a specified character, the words are "spelled backwards" compared with the trie (see Figure 3.8(a)). Therefore, the last character of the word has to match the label of the first edge of the trie starting from the root. The second last character matches the second edge and so on. Hence, the procedure described above generates a suffix trie of the reversed text T (see Figure 3.8(b)). In order to use the iterator of the suffix trie of T the index has to be built using the reversed text.

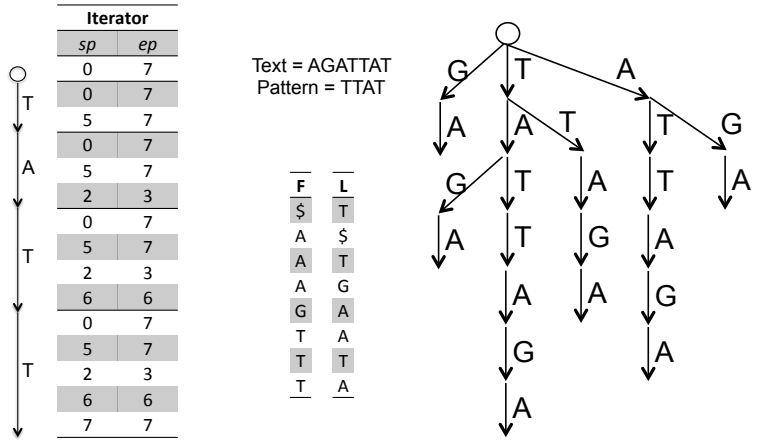


Figure 3.8: On the left hand side a schematic overview of the trie construction for the pattern "TTAT" is presented, whereas (b) show the constructed trie of text "AGATTAT".

3.10 Extension to Sets of Strings

Since often biological questions involve several sequences in contrast to a single sequence, we extended our FM-index variants to be also applicable to string sets.

A very simple adaptation to cope with string sets would be to concatenate all strings before applying the FM-index. However, this can lead to matches that span over the borders of two adjacent strings and is therefore no valid solution in reality. In order to circumvent such wrong matches, one could either verify the matches returned by the search procedure or add a special character to the ends of the strings. After using the search routine, in both cases one has to identify the position of the string the pattern matched to within the string set as well as the matching position in the string.

To avoid this extra effort we follow a different approach, in which the search procedure directly returns the correct string as well as the pattern position within the string. As described above, we add a special symbol (which is the $ already introduced in previous sections) to the end of each string. Afterwards, we directly build the suffix array of the string set in contrast to first concatenate the strings. For this reason, the suffix array is an array of pairs, where each entry represents a position within the string set and a position within a string.

To implement this concept into our FM-index variants, all indices (except the wavelet tree based one that incorporates an extra node for the $ sign) have to be adapted to store more than one $ position. Simply storing these positions as integer values would require to determine the number of characters substituted by a $ up to a specified position, which is time consuming. Therefore, we chose to encode the $ positions with a bit string, which enables us to retrieve the number of substituted characters up to a specified position in constant time.

4 Results and Discussion

As stated in the introduction, our goal is to design and implement a practical FM-index version that is both memory efficient and guarantees fast pattern search times. We have shown how we achieve those two goals in the previous section and will analyze our implemented indices in the following.

4.1 Experimental Set-Up of the Benchmarking

Four main factors influence the memory consumption and pattern search time, which are text and pattern length, as well as alphabet size and suffix array compression rate. The impact of these factors is analyzed in a benchmarking study performed under the following set-up.

We generated artificial texts by concatenating randomly chosen characters of a former specified alphabet. To ensure equal character frequencies this sampling follows an equal distribution. The alphabets chosen for the text generation are *Dna, Dna5, AminoAcid* and of type *unsigned char*, which contain 4, 5, 24 and 224 characters, respectively[1]. Further, we generated patterns of different lengths by randomly extracting subsequences of the former created texts, which ensures that each pattern occurs at least one time (otherwise the pattern search would be aborted after too few iterations, which would induce a bias in the running time analysis).

While the memory consumption was measured by storing the different index tables on hard drive, we divided our running time analysis according to counting and localization phase. The counting phase time requirement was determined by searching a fixed number of 100,000 patterns. Note that due to the speed of the search procedure an accurate measurement was achieved by repeating the process until the overall time exceeded 30 seconds. The actual running time needed to search for 100,000 patterns was then derived by dividing the overall time by the number of necessary repetitions. In contrast, we analyzed the time requirements of the localization phase by measuring the suffix array access time. This was done by iterating over the suffix array and accessing each entry. Similar to the counting phase time measurement, we repeated the process until the overall time exceeded 30 seconds and then computed the time necessary to access 100,000 positions.

Note that throughout this section the 2-dimensional occurrence table based FM-index is referred to as *2dOcc*, the wavelet tree based FM-index that substitutes the \$ sign is referred to as *WTS* and the wavelet tree based FM-index incorporating an extra node for the \$ sign as *WT*.

Further, our implementations are also applied to real data sets to give an impression of their performance not only in simulations, but also in real world scenarios. Note that now we do not divide the running time analysis into counting and localizing, but measure

[1]Note that we excluded 32 characters from the 256 characters of the *unsigned char* data type from our benchmarking, since they represent control characters such as the *end-of-file* symbol.

the overall pattern search time. In doing so we also consider the impact of the number of matches between pattern and text.

As an example for a comparably small genomic sequence (4.6MB in size) we have chosen the *Escherichia coli* genome (accession: NC_012947). The second genome belongs to *Drosophila melanogaster* (accession number: PRJNA164, 132.2MB in size) and the largest genome to be analyzed is the genomic sequence of *Homo sapiens* (accession number: PRJNA42201, 3.1GB). All genomes are taken from the NCBI reference sequence database (*http://www.ncbi.nlm.nih.gov/*).

In addition, we used the real world data sets to compare our new implementations not only with each other, but also with the enhanced suffix array index of SeqAn[2], as well as FM-index implementations taken from the Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/index.html).

All benchmarks were performed on a machine with 2 x Intel(R) Xeon(R) CPU X5550 @ 2.67GHz and 50GB of RAM.

4.2 Memory Consumption

We begin our verification by analyzing the memory consumption of the FM-index versions, first depending on different compression rates of the suffix arrays (SA). Afterwards, we will analyze the memory consumed by the 2-dimensional occurrence table data structure (2dOcc) (see Section 3.2) and the wavelet trees (see Section 3.4). Finally, the different FM-index implementations are examined with respect to their overall memory dependence on alphabet size and text length.

4.2.1 Memory Consumption of the Compressed Suffix Arrays

A major contributor to the memory consumption of the FM-index is the suffix array. For this reason, the size is reduced by storing only a fraction of all values present in the uncompressed SA. This is possible because values in between two stored SA positions can be computed using the BWT of T (T^{BWT}, as described in Section 3.2).

A graphical representation of the memory consumption for differently compressed suffix arrays (CSA) is given in Figure 4.1. The CSAs have been constructed using a text of 10 million characters.

As can be seen in Figure 4.1, the memory consumption increases with decreasing SA compression. Even though this behaviour is expected, the following points are worth mentioning:

1. There is a non-linear relation between the suffix array compression and its memory consumption.

2. The memory consumption of the CSA with a compression factor of 1 is larger than the memory consumption of a normal SA.

Both observations can be explained with the memory consumption of the rank support bit string (RSBS) used to indicate positions in the CSA that store a value. The RSBS

[2]The following flags were used for compilation: *-DSEQAN_HAS_ZLIB=1 -DSEQAN_HAS_BZIP2=1 -DSEQAN_HAS_EXECINFO=1 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -W -Wall -Wno-long-long -fstrict-aliasing -Wstrict-aliasing -pedantic -Wno-variadic-macros -DSEQAN_ENABLE_TESTING=0 -O3 -DNDEBUG -DSEQAN_ENABLE_DEBUG=0*

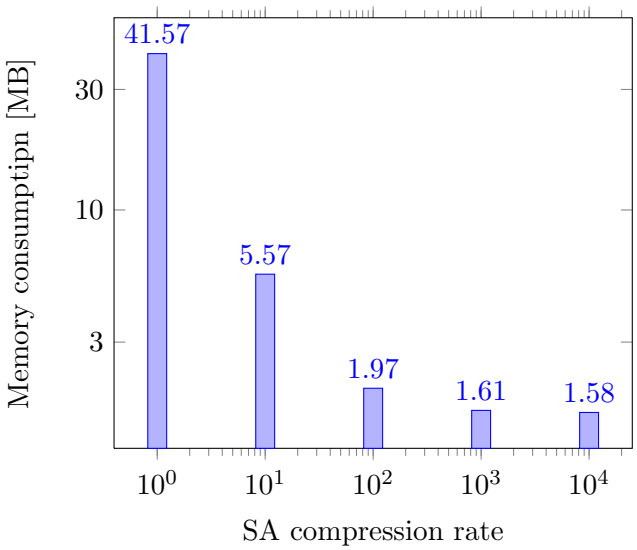


Figure 4.1: Graphical representation of the memory consumption for differently compressed suffix arrays. Note that both axis are scaled logarithmically.

requires 1.57MB, which explains the difference between the 41.57MB, required by the uncompressed SA, and the memory consumption of a normal SA (which is 40MB). In addition, since the memory required by the RSBS is independent of the SA compression it contributes as a constant factor to the overall CSA memory consumption. Hence, it explains the non-linearity between SA compression and CSA memory requirement. To be more precise, the memory consumption of the RSBS is a lower bound of the overall compressibility of a CSA.

We want to point out that for high SA compression rates the memory consumption is dominated by the requirements of the RSBS and hence the differences between the memory consumption of highly compressed SAs are minor. In contrast, the SA access time has a linear dependence on the SA compression, as we will show later. Hence, considering both properties, a very high SA compression is impractical.

On the other hand, Figure 4.2 shows that there is a linear dependence on the text length.

4.2.2 Memory Consumption of the Occurrence Tables

In contrast to the memory consumption of the CSA, the sizes of the occurrence table data structures (Occ) depend not only on the text, but also on the underlying alphabet. The effects of the alphabet on the Occs are shown in Figure 4.3. The Occs were build using a text of 10 million characters over different alphabets with equal frequencies for each character.

Even though the 2-dimensional occurrence table of Figure 4.3 stores only every 100's row, it is the most memory intensive data structure for all alphabets. In contrast, the most memory efficient data structure is the wavelet tree in which the \$ is substituted by the least frequent character.

The high memory consumption of the 2-dimensional occurrence table data structure is the result of a linear dependence of the alphabet size combined with a linear dependence

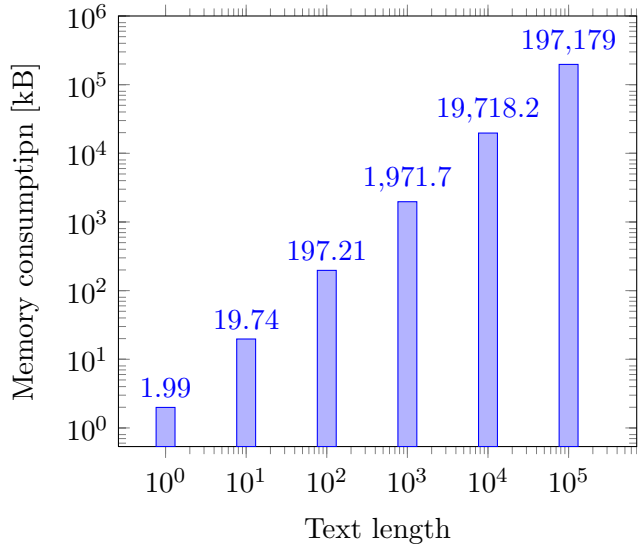


Figure 4.2: Graphical representation of the memory consumption of a CSA with a compression factor of 100 for different text lengths.

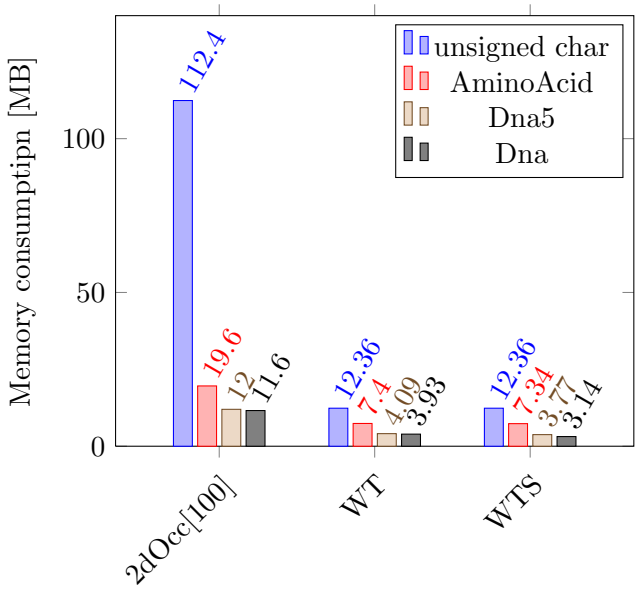


Figure 4.3: Graphical representation of the memory consumption for different occurrence table data structures. The underlying text consisted of 10 million characters drawn from the alphabets *unsigned char, AminoAcid, Dna5* and *Dna*. Note that both axes are scaled logarithmically.

on the text length. In addition, since entries between two stored rows need to be recomputed, it is necessary to store T^{BWT} explicitly. Therefore, similar to the behaviour of the

CSAs, the memory consumption of the 2-dimensional occurrence table converges towards a lower bound, independent of the alphabet or the compression rate. Figure 4.4 provides a graphical representation of the described dependence.

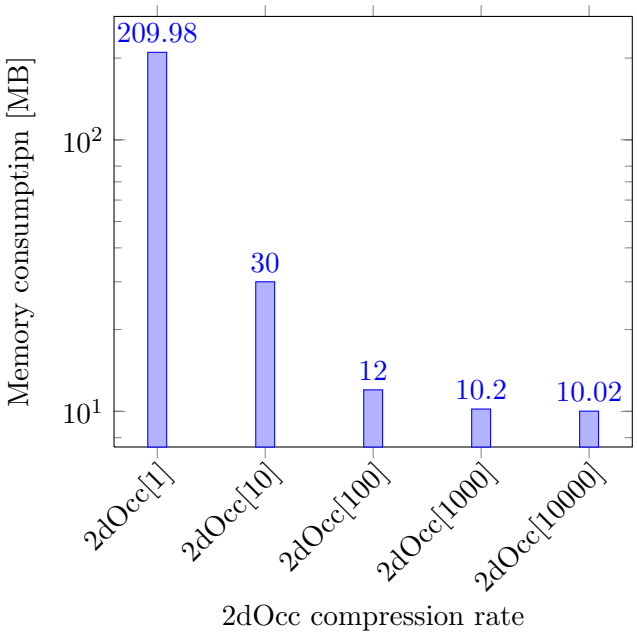


Figure 4.4: Graphical representation of the memory consumption for differently compressed *2dOcc*s. Note that the memory consumption axis is scaled logarithmically.

In contrast to the 2-dimensional occurrence table data structure, T^{BWT} represents the theoretical upper bound (plus some auxiliary data) on the memory consumption for the wavelet trees in Figure 4.3. Since the characters have approximately the same frequencies, the tree is balanced and has a depth of $log_2|\Sigma|$. For the character string this is approximately 8. Hence, each character is contained in 8 bit strings. Since 8 bits per character is also the memory consumption of T^{BWT} string, as claimed T^{BWT} represents an upper bound.

However, in Figure 4.3 one can observe that the wavelet trees actually need more memory than necessary to store T^{BWT} as a string (12.36MB versus 10MB). The reason for this observation is the additional memory requirement of the rank support tables of the rank support bit string.

Since for the alphabets *Amino Acid, Dna5* and *Dna* it holds $log_2|\Sigma| < 8$, the memory requirement for the corresponding wavelet trees is smaller than the original text size. The smaller the alphabet, the smaller the memory requirements. Therefore, the wavelet tree is most memory efficient for *Dna* strings.

Note that the wavelet tree version that incorporates an extra node for the $ needs more memory due to the extra node.

Concerning the text length: similar to the CSAs, the memory consumption of the occurrence table data structures show a linear dependence on the text length, as shown in Figure 4.5.

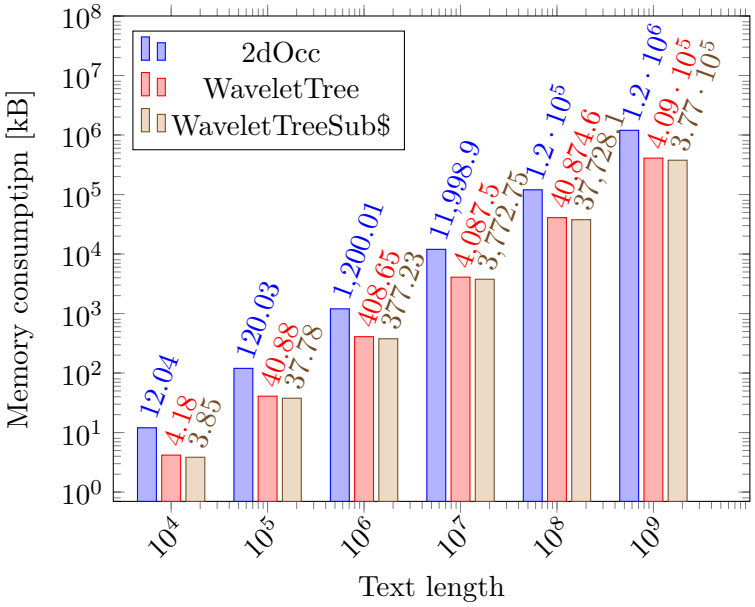


Figure 4.5: Graphical representation of the memory consumption of occurrence table data structures of different text lengths. The alphabet of the underlying text is *Dna5*. Note that both axes are scaled logarithmically.

4.2.3 Memory Consumption of the FM-index

The last two sections have shown the memory consumption of the CSA and the occurrence table data structures with respect to different alphabets and text lengths. In this section we will analyze the memory consumption of the whole FM-index. We start by focusing on the influence of the alphabet on the index.

In order to do so, we build the different indices over a text of 10 million characters drawn from different alphabets with equal character frequencies. Since there is no alphabet dependence of the CSA regarding to the memory consumption, the reason for the differences between the indices in Figure 4.6 are the occurrence table data structures. Hence, Figure 4.3 and Figure 4.6 show a similar distribution of the bars, which only differ in the assigned values.

Note that with a SA compression factor of 10, for *Dna* and *Dna5* alphabets both wavelet tree based FM-index versions require less memory than the original text file. For a SA compression factor of 100 the compression is so effective that the wavelet tree based FM-indices for *AminoAcid* texts require less memory than the original file and for *Dna* and *Dna5* alphabets the index size is even reduced to 50-60% of the original file.

The indices are not only sensitive to the alphabet size, but are also influenced by the text length. However, since we have illustrated that the CSA as well as the occurrence table data structures show a linear dependence on the text length (see previous sections), the results of this section can be transferred to shorter or longer texts.

After analyzing the memory requirements of the different FM-index implementations, in the next section we will have a detailed look on the time requirements of the pattern search.

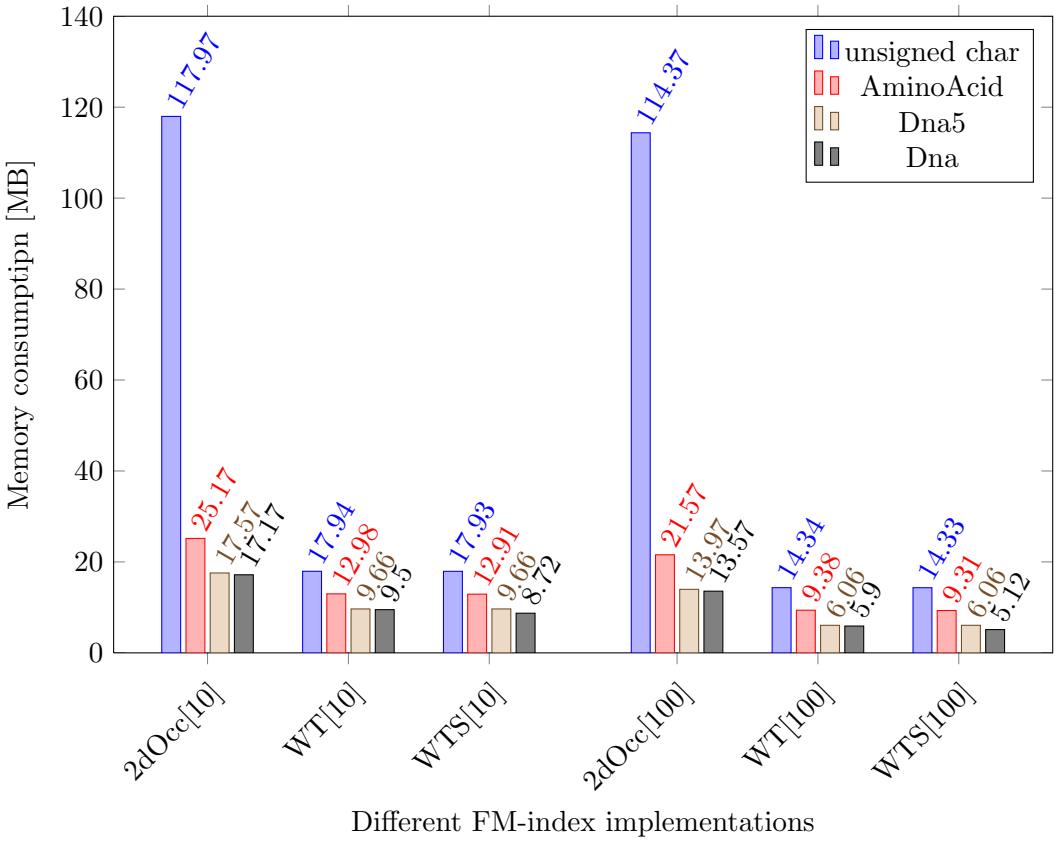


Figure 4.6: Graphical representation of the memory consumption for different indices. The underlying text consisted of 10 million characters drawn from the alphabets *unsigned char, AminoAcid, Dna5* and *Dna*.

4.3 Search Time Analysis

In the last section we have shown that the wavelet tree based FM-index implementations can be parametrized such that the memory requirements are less than the size of the original file. In addition to its memory efficiency, the FM-index also facilitates a fast pattern search, as we will show in this section. We begin by examining the time requirements of the counting procedure in dependence on the text and pattern length as well as the alphabet. Afterwards, we will focus on the SA access time and time requirements of the whole search procedure.

4.3.1 Counting Time Dependence on the Text Length

As described in Section 2.3, in theory the counting procedure is independent of the text length. However, in practice we do observe that counting requires more time for longer texts. This dependence is visualized in Figure 4.7, where the time consumption of the counting procedure for the different FM-index implementations and text lengths is illustrated. In the presented scenario the pattern length was fixed to 100 and the text type was *Dna5*.

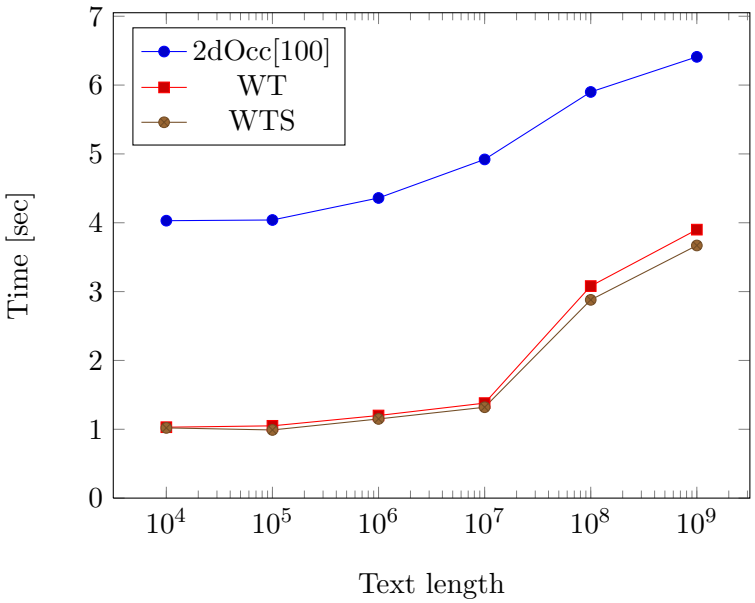


Figure 4.7: Graphical representation of the time required by the counting procedure to determine the number of matches between the original text and 100,000 patterns. While the alphabet was fixed to Dna5 and the pattern length to 100, the text length has been increased exponentially. The occurrence table compression of the 2dOcc based FM-index was fixed to 100. Note that the text length axis is scaled logarithmically.

Figure 4.7 shows that for the wavelet tree based FM-indices the counting time is approximately constant (apart from a slight increase) for texts up to 10^7 characters. For larger texts, the time requirements are more than doubled. For the 2-dimensional array based FM-index the counting time stays constant for text lengths up to 10^5 characters before increasing, though the increase is not as prominent as in the case of the wavelet tree based indices. In general, it is worth mentioning that considering the whole range of text lengths, the overall increase in the time requirement in almost equal for all indices. However, it is clear to see that the wavelet tree based variants are much faster than the 2dOcc based version (this can be adjusted when decreasing the occurrence table compression).

Since the procedure of counting is theoretically only influenced by the pattern length, it comes as a surprise to observe a dependence on the text length. The reason for the differing behaviour in practice is hardware related, for instance there occur cache effects in case the program needs to access information that is not already present in cache but has to be loaded beforehand, which increases the time consumption (for more details, refer to [30]).

Since the cache is very limited in size, large data sets tend to increase the number of necessary information loads. Such a situation can be observed during the counting procedure. The value of the lower bound sp, as well as the value of the upper bound ep (refer to Section 2.3.3), can change to any position of T^{BWT} from iteration to iteration. Therefore, different parts of the rank support bit strings in the wavelet tree need to be present in cache in order to minimize the running time. Since this is not possible for large

texts, due to the limited cache size the running time changes for different text lengths.

A visualization of cache effects is shown in Figure 4.8, which presents the time necessary to access 4 billion positions in bit strings of increasing length. The positions are accessed sequentially with a step size of 4096. In doing so, we guaranty that information needs to be reloaded once the whole bit string does not fit into cache.

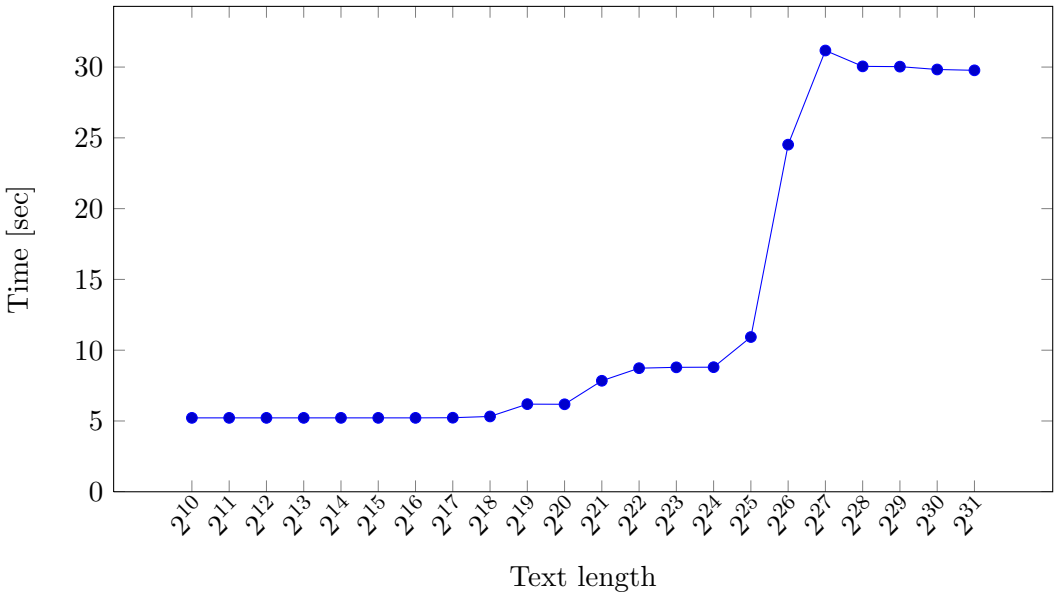


Figure 4.8: Time necessary to access a rank-support-bit-string 4 billion times with different underlying text lengths. Note that the text length axis is scaled logarithmically.

Note that the graphs in Figure 4.7 differ from the graph in Figure 4.8, because not every re-computation of *sp* and *ep* lead to cache misses. Further, for the 2dOcc based FM-index cache effects already occur for shorter text lengths, since in comparison to the wavelet tree based FM-indices this FM-index requires more memory.

4.3.2 Counting Time Dependence on the Alphabet

In contrast to the text length, there is a logarithmic relation between the alphabet size and the counting time of the wavelet tree based FM-indices. Since the number of nodes in a wavelet tree increases with increasing alphabet size, the number of rank queries necessary to determine the number of occurrences of a specified character until a specified position in T^{BWT} grows as well. The described situation is shown in Figure 4.9.

Note that in regard to the different alphabets the time required by the counting procedure is approximately constant for the 2-dimensional array based FM-index. In contrast, the counting times of both wavelet tree based FM-index versions differ for the various alphabets. The minimal time consumption is required for the smallest alphabet, *Dna*, closely followed by *Dna5*. Between the wavelet tree based FM-indices, the times for *Dna* and *Dna5* differ only slightly. In contrast, the running time more than doubles from *Dna5* to *AminoAcid*, as well as from *AminoAcid* to *unsigned char*. Since the alphabet sizes between *Dna5*, *AminoAcid* and *unsigned char* grow exponentially, there cannot be a linear relationship between the counting time and alphabet size. However, if we take the loga-

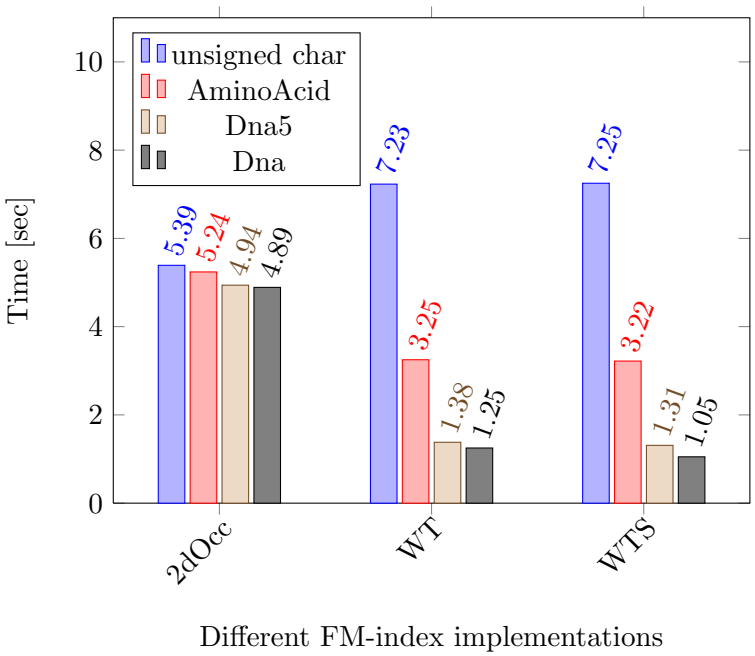


Figure 4.9: Graphical representation of the time required by the counting procedure to determine the number of matches between the original text and 100,000 patterns. While the text and the pattern length are fixed, the alphabet size was 4, 5, 24 and 224 for *Dna, Dna5, AminoAcid* and *unsigned char*, respectively.

rithm to the base of 2 of the alphabet sizes and set these results in relation to the counting time, a linear correlation is revealed. Therefore the wavelet tree based FM-indices show a linear relation between the logarithm of the alphabet size and counting time, as claimed in Section 2.5.3.

4.3.3 Counting Time Dependence on the Pattern Length

Since the FM-index is built over the text and not over the searched patterns, the patterns themselves are not preprocessed. Therefore, there is a linear relationship between the pattern length and the time required by the counting routine of the FM-index search. A graphical representation of the dependence is provided in Figure 4.10.

The fastest index is the wavelet tree based implementation in which the $ is substituted by the least frequent character. The second fastest is the wavelet tree based version with an extra node for the $. This shows that the processing of the extra node is more time consuming than the processing of the substituted $.

4.3.4 Suffix Array Access Time Dependence on the Text Length

In the last section we have discussed the counting time dependence of the FM-index with respect to text and pattern length, as well as alphabet size. To complete the search time analysis, in the following we will focus on the second part of the search routine, the localization procedure. In order to do so, we will first concentrate on the SA access time with respect to the text length and then analyze the influence on SA compression.

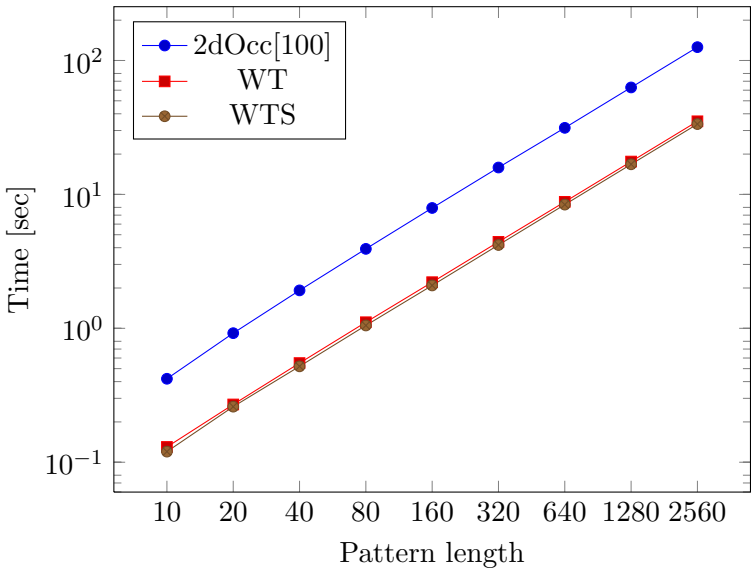


Figure 4.10: Graphical representation of the time required by the counting procedure to determine the number of matches between the original text and 100,000 patterns. While text length and alphabet have been fixed, the pattern length has been increased exponentially. Note that both axes are scaled logarithmically.

From a theoretical point of view there is no dependence between the text length and the SA access time. However, Figure 4.11 shows that the time consumption increases with increasing text length, which moreover is not linear. To be more precise, there is a small increase of the time consumption for texts up to 10^7 characters. Afterwards, the time consumption is more than twice as high.

Since no part in the computation of an entry in the CSA is influenced by the text length, the in practice observed dependency is hardware related. A similar situation occurred when we analyzed the counting time dependence on the text length in Section 4.3.1. Note that graphs presenting the counting time of the wavelet tree based FM-indices are very similar to the one in Figure 4.11. This comes as no surprise, since in order to compute the CSA entries the LF-mapping has to be applied, which is the same procedure as the computation of the bounds *sp* and *ep*. Therefore we omit a detailed analysis and instead refer the reader to Section 4.3.1.

4.3.5 Suffix Array Access Time Dependence on the Suffix Array Compression

In contrast to the text length, in regard to the SA compression there is a linear influence on the SA access time. With increasing compression the number of CSA entries that need to be computed increases, as well as the number of LF-mappings necessary to compute one SA entry (for details see Section 3.2), as is illustrated in Figure 4.12.

At this point we have analyzed the dependencies of the two phases of the search procedure. However, the overall search time also depends on the number of matches between the pattern and the original text and is therefore highly problem specific. Nevertheless, a rough search time estimate can be computed by summing up the counting and the SA access time (for one entry) and multiplying the sum with the number of expected matches.

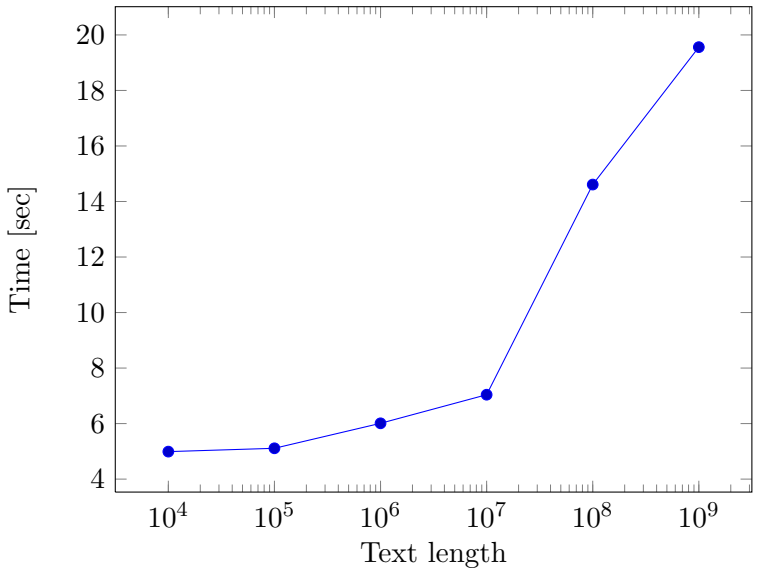


Figure 4.11: Graphical representation of the SA access time of a CSA with a compression factor of 100 for different text lengths. The presented results reflect the time necessary to access 100,000 CSA positions. Note that the text length axis is scaled logarithmically.

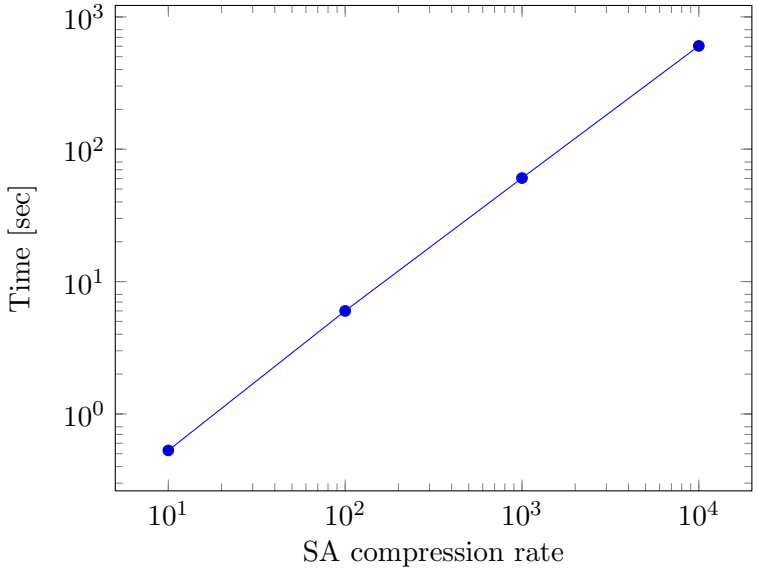


Figure 4.12: SA access time dependence on the SA compression. The presented results reflect the time necessary to access 100,000 CSA positions. Note that both axes are scaled logarithmically.

Since our aim was to design and implement an FM-index that is suitable for biological sequences we will analyze the overall search time using genomes of different organisms, namely *Escherichia coli*, *Drosophila melanogaster* and *Homo sapiens*, in the next section.

4.4 Comparison of the Indices on Real Data

In this section we will analyze the search time of the different FM-indices with various SA compression rates using the *Drosophila melanogaster* genome. In addition, we provide results for a much smaller and a much larger genome, of *Escherichia coli* and *Homo sapiens*, respectively.

We start the analysis by comparing indices with a SA compression factor of 1. Since no LF-Mapping is involved when retrieving a position in the CSA we expect this setting to yield the fastest results.

Figure 4.13(a) shows that the 2dOcc FM-index with an occurrence table compression rate of 10 is the fastest index, followed by the two wavelet tree based FM-index implementations. The increased compression rate of 100 in the second 2dOcc FM-index makes this index the slowest. Note that the axes of Figure 4.13(a) are scaled logarithmically. Therefore, the difference between the two wavelet tree FM-indices and the less compressed 2dOcc FM-index are much smaller than the difference between the two wavelet tree FM-indices and the higher compressed 2dOcc FM-index. For details refer to Table 4.2 at the end of this section.

The Figures 4.13(b) and 4.13(c) show the search times for the indices of Figure 4.13(a) with a SA compression rate of 10 and 100, respectively. While the order of the indices remains, their progressions are different. To be more precise, especially Figure 4.13(c) shows that the search time is not doubled for the 2dOcc based FM-indices, as is observed in Figure 4.13(a). In fact, the search time is almost constant for the two indices for pattern lengths of 40 and 80. The same trend can be observed for the wavelet tree based FM-indices, though not as prominent.

The reason for this behaviour is a domination of the localization phase during the search for small patterns. In contrast to Figure 4.13(a), in Figure 4.13(c) a major proportion of the time is spent to determine the corresponding entry in the CSA. With increasing pattern length this effect decreases, since most of the search time is spent during the counting phase.

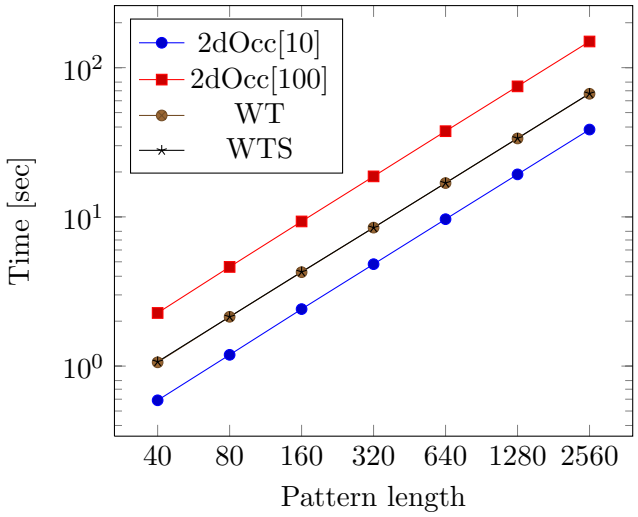
In addition, the localization phase is strongly influenced by the number of matches between the patterns and the genome. As can be seen in Table 4.1, there are more matches between the genome and the patterns for smaller pattern lengths, which increases the time spent in the localization phase.

Pattern length	Number of matches
40	317,241
80	198,210
160	159,338
320	140,981
640	125,719
1280	113,418
2560	103,503

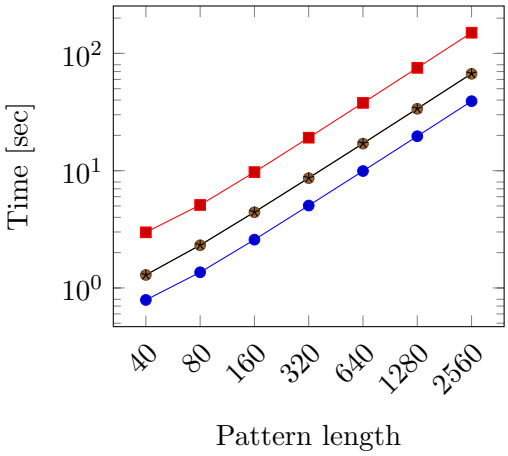
Table 4.1: Number of matches between patterns of different lengths and the *Drosophila melanogaster* genome.

As stated above, the described effect is not as strong for the wavelet tree based FM-

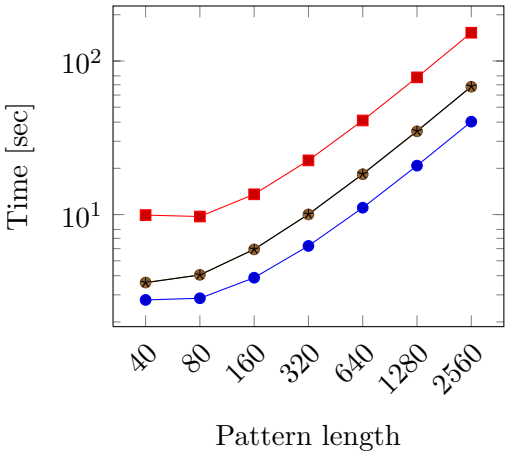
indices. A reason for this is that they are much more memory efficient and therefore less prone to cache effects than the 2dOcc based FM-indices.



(a) Search times for different FM-indices with a SA compression of 1.



(b) Search times for different FM-indices with a SA compression of 10.



(c) Search times for different FM-indices with a SA compression of 100.

Figure 4.13: Graphical representation of the time required by the different FM-indices to find 100,000 patterns in the *Drosophila melanogaster* genome. The compression rates of the SA are (a) 1, (b) 10 and (c) 100. Note that both axes are scaled logarithmically in each figure.

In addition to the search time differences of the indices in regard to a specified SA compression rate, we are also interested in the differences of the same index regarding various SA compression rates. Therefore, we calculated the pairwise difference in the search time between the indices of a SA compression rate of 1 and 10, as well as 1 and 100. The results are visualized in Figure 4.14(a) and Figure 4.14(b), respectively.

From the first look, the graphs in Figure 4.14(a) do not seem to follow a common

tendency. However, it has to be mentioned that the overall search times are so close that in the figure even small differences appear to have a strong impact such that tendencies might be covered. Hence, for a better understanding we further normalized the obtained differences with the running times of the indices with compression rate 1. The results are presented in Figure 4.15(a), where it is now clear to see that for the pair of compression rates 1 and 10 the differences in search time indeed decrease with increasing pattern length.

For the pair of compression rates 1 and 100 this tendency is already visible for the not normalized differences presented in Figure 4.14(b), because compared to the first pair the search time differences are far larger. Figure 4.15(b) further supports this observation.

The reason for the relation between search time and pattern length can be explained with the high percentage of time spent in the localization phase, as discussed above. With increasing pattern length this effect is reduced, such that the difference between the two SA compression rates do not as strongly influence the overall search times.

All in all, as expected the search time requirements increase with increasing SA compression. Though, it is worth mentioning that the loss in speed is relatively small for the step from compression rate 1 to 10.

4.5 Memory versus Time Consumption

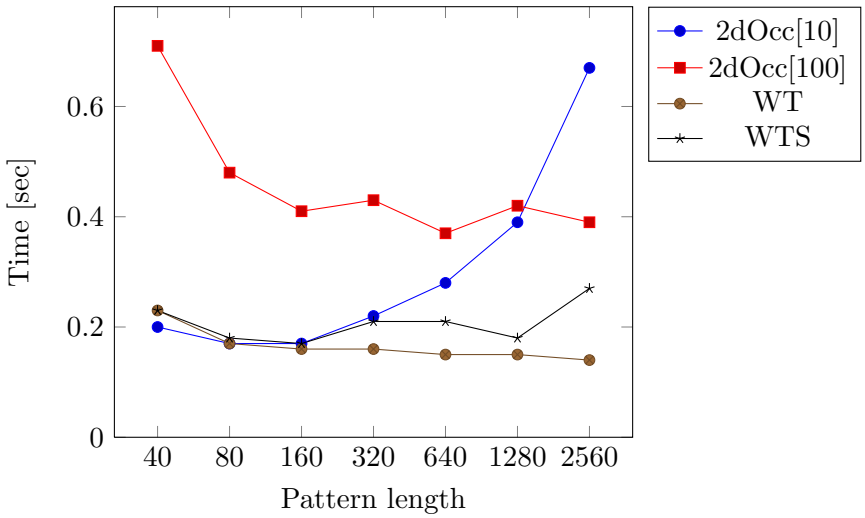
In the last section we have first analyzed the memory consumption of the different FM-index implementations and discussed their counting and search time requirements. In the following we will put those two properties into relation.

Figure 4.16 shows the time to search for 100,000 patterns of length 320 in the *Drosophila melanogaster* genome, depending on the size of the corresponding index. We only included one of the two wavelet tree based FM-index implementations, since their values are almost identical and they would not be distinguishable in the figure.

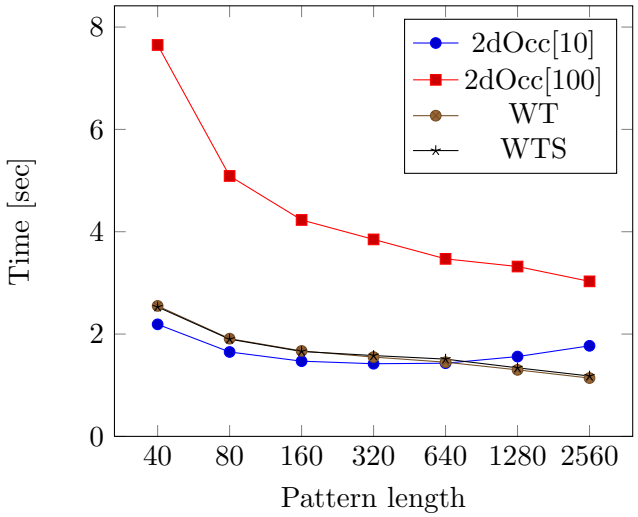
As shown in the last section, there is a trade-off between memory consumption and search time. However, Figure 4.16 shows that increasing the SA compression from 1 (blue) to 10 (red) reduces the memory consumption by at least 50% while no significant speed reduction can be observed. A further increase in the SA compression does reduce the memory consumption, however, the reduction is relatively small. On the other hand, there is a significant increase in the search time. Hence, it seem to be appropriate to at least use a compression rate of 10, while there is a trade-off for further compression rates.

Comparing the indices with respect to their occur table, it is obvious that the 2dOcc based FM-index with the smaller Occ compression is the fastest, followed by the two wavelet tree based indices. However, while the memory consumption almost quadruples from the wavelet tree based versions to the 2dOcc version, the search time of the 2dOcc versions is not even twice as high compared to the wavelet tree based variants.

If we repeat the experiment, but search for patterns of length 40, the situation changes, as is shown in Figure 4.17. Because the localization phase has a strong impact on the overall search time, the 2dOcc FM-index implementation with an Occ compression of 10 and the two wavelet tree FM-index versions are almost equal in terms of time consumption. At the same time the memory consumption does not change in comparison to Figure 4.16, such that the wavelet tree based FM-indices show no disadvantages with respect to the 2dOcc FM-index implementations.



(a) Search times difference between indices with a SA compression of 1 and 10.

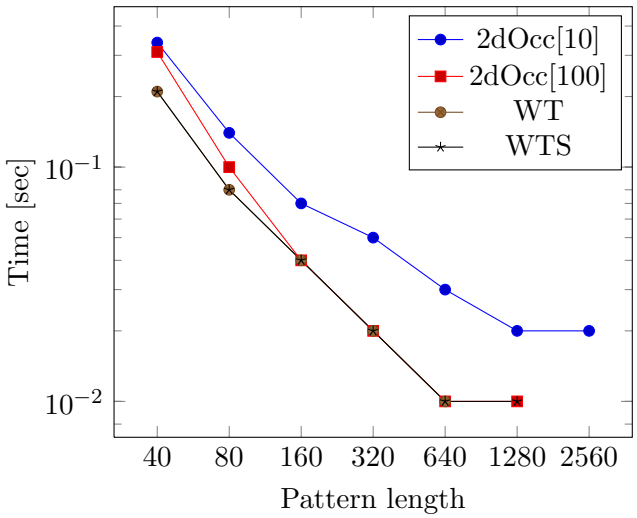


(b) Search times difference between indices with a SA compression of 1 and 100.

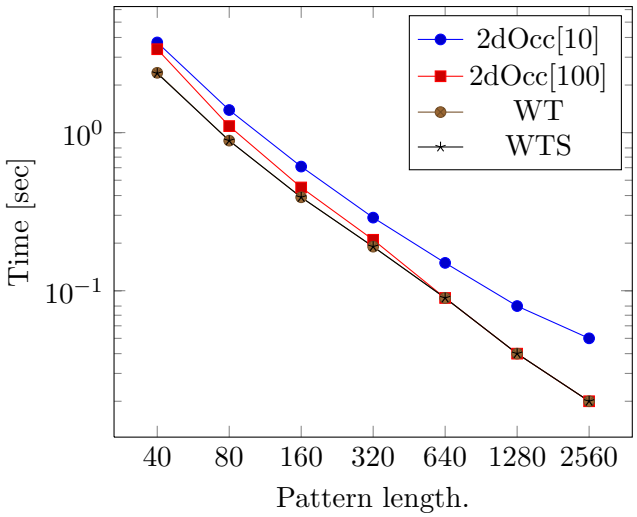
Figure 4.14: Graphical representation of the difference in search time between the indices whit a SA compression of 1 and the indices with a compression rate of 10 and 100 respectively. Note that the pattern length axis is scaled logarithmically in both figures.

4.6 Comparison with Other Indices

So far we have compared our new FM-indices amongst themselves. In this section we compare the new indices with the *enhanced suffix array index* (ESA-index, in the following referred to as ESA) implemented in SeqAn, an FM-index (FM-index version 2) implemented by Paolo Ferragina and Rossano Venturini (from now on PC_V2) as well as a run-length encoded FM-index implemented by Veli Mäkinen and Rodrigo González (from now on



(a) Normalized Search times difference between indices with a SA compression of 1 and 10. An division by 0 occurred for the read length of 2560, therefore the coordinates are not shown.



(b) Normalized search time differences between indices with a SA compression of 1 and 100.

Figure 4.15: Graphical representation of the normalized difference in search time between the indices with a SA compression of 1 and the indices with a compression rate of 10 and 100 respectively. Note that both axes are scaled logarithmically in both figures.

PC_RL). The latter two are available at the Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/index.html[3]).

In contrast to the results presented in the earlier sections, we compiled the code for the

[3]There is also an alphabet friendly FM-index version available. However, we were not able to include this version in the benchmarking because the program would terminated throwing an error. In addition the version 1 of the normal FM-index is also available, however, it follows a different syntax.

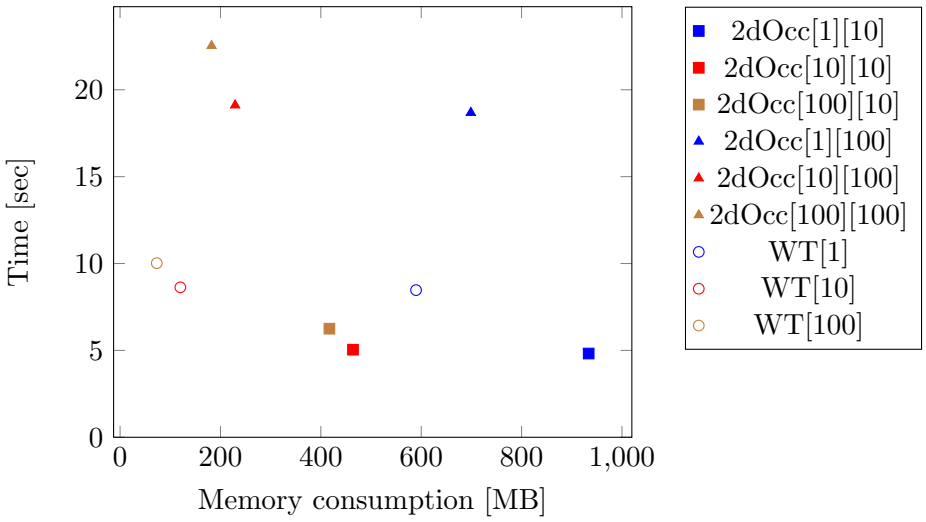


Figure 4.16: Graphical representation of the time required by the different FM-indices to find 100,000 patterns of length 320 in the *Drosophila melanogaster* genome versus their memory consumptions. Note that 2dOcc[x][y] denotes a 2-dimensional occurrence table based FM-index with a SA compression rate of x and a occurrence table compression rate of y.

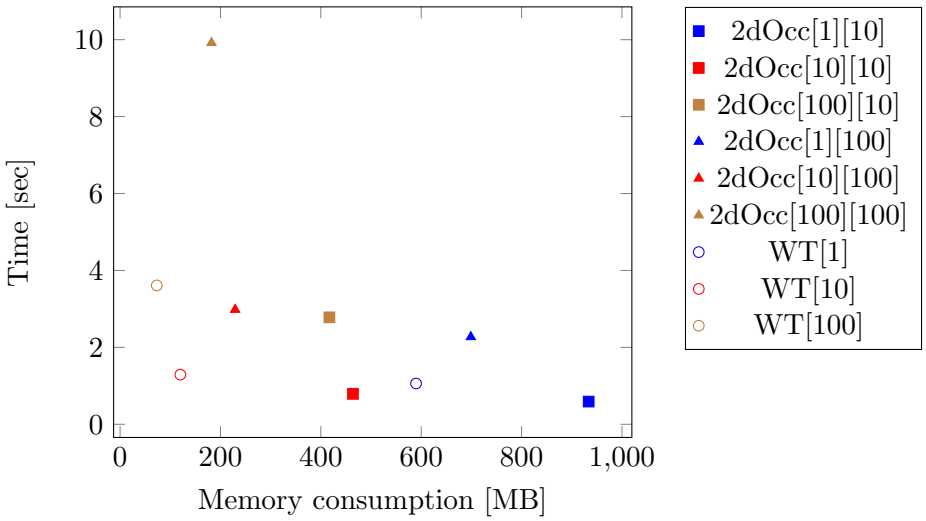


Figure 4.17: Graphical representation of the time required by the different FM-indices to find 100,000 patterns of length 40 in the *Drosophila melanogaster* genome versus their memory consumptions. Note that 2dOcc[x][y] denotes a 2-dimensional occurrence table based FM-index with a SA compression rate of x and a occurrence table compression rate of y.

following benchmarks on a 32-bit machine. This was necessary because the indices from the Pizza&Chili Corpus depend on a 32-bit architecture[4].

[4]The code was compiled using the command make with *CFLAGS=-g -O9 -lm -fomit-frame-pointer -W*

We start our comparison with a relatively small genome, the *Escherichia coli* genome, which is 4.6 MB in size. The results for the different indices are shown in Figure 4.18(a),where it is obvious that the ESA is the fastest of all indices. For smaller pattern lengths it is about 8 times as fast as the second fastest index, the new wavelet tree based FM-index in which the $ is substituted (from now on WTS). However, the distance in time increases with increasing pattern length, because in our experiment ESA does not show a linear dependence on the pattern length. WTS is followed by the other new wavelet tree implementation (WT), with a small increase in time. The difference of this index to the following 2dOcc FM-index with an occurrence table compression rate of 10 (referred to as 2dOcc[10]) is small as well. In contrast, the gap to the next fastest index, the PC_RL is larger. In fact, PC_RL is about as fast as the 2dOcc FM-index with an Occ compression rate of 100 (from now on referred to as 2dOcc[100]) and is 3 to 4.5 times slower than the wavelet tree based FM-indices. The gap to the last index, the PC_V2 is even larger. This index is 30 to 50 times slower than the new wavelet tree based FM-indices.

On the other hand, Figure 4.18(b) shows that the slowest index is also the most memory efficient. However, the difference between this index and the wavelet tree based one is very small, which means that one has to sacrifice a lot of speed to gain a small memory advantage. Further, the only index faster than WT is ESA, which is in return also the most memory intensive. Even though it is about 25 - 60 times faster, it also requires 10 times as much memory.

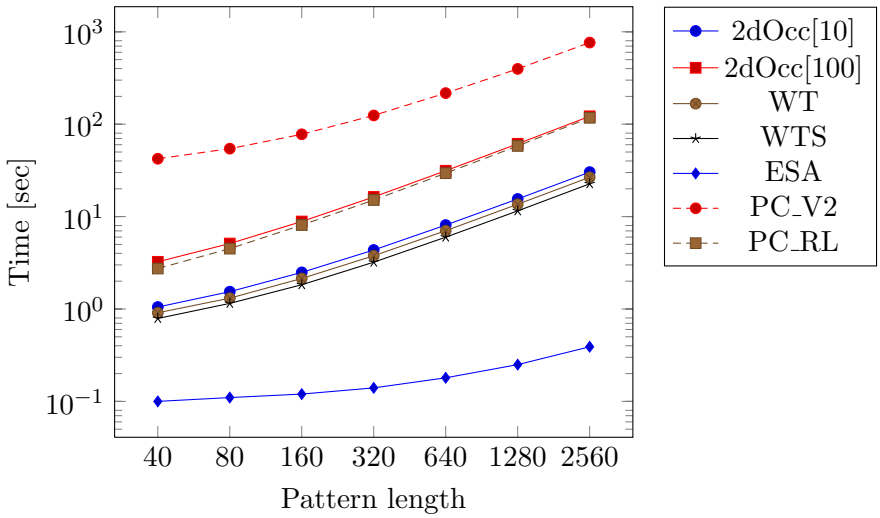
The reason for the speed superiority of ESA is the lower number of computations to adjust the lower and upper bound within the SA, as well as its cache efficient implementation (as is explained later).

Repeating the experiment with the *Drosophila melanogaster* genome, which is 132.2MB in size yields the results presented in Figure 4.19(a). The results show that the fastest and the slowest indices are ESA and PC_V2, respectively, as was observed for the *Escherichia coli* genome. In contrast, the order of the indices in between those two has changed. The second fastest index is 2dOcc[10], followed by the wavelet tree based FM-indices. The order of 2dOcc[100] and PC_RL has changed as well. Therefore, the indices from the Pizza&Chili Corpus are the slowest in this experiment.

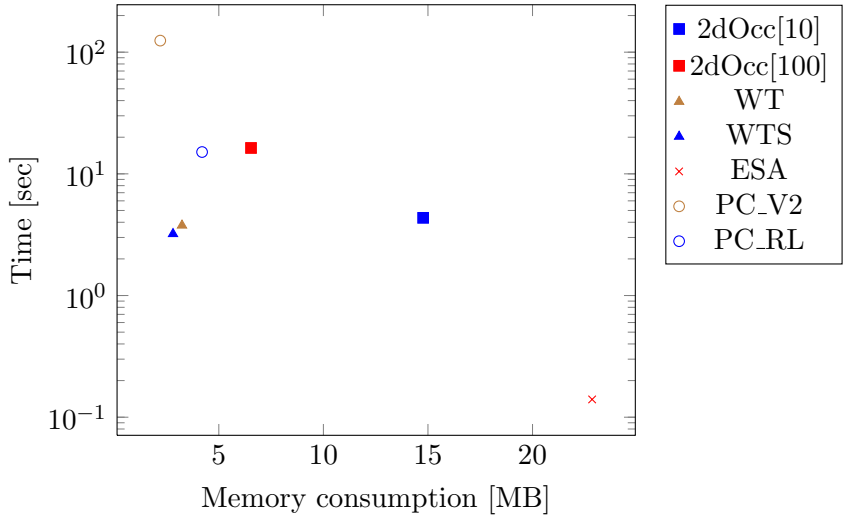
The reason for the changed order of 2dOcc[10] and the wavelet tree based FM-indices are cache effects. While the rank queries during the LF-Mapping (involved in the counting phase and the localization phase) do not cause cache problems for the small *Escherichia coli* genome, they do for the larger *Drosophila melanogaster* genome. Because the rank queries are necessary in the counting and localization phase for the wavelet tree based FM-indices, those two are stronger affected than 2dOcc[10], since the rank queries only influence the localization phase of 2dOcc[10].

Since the compression rate has an impact on both memory consumption and running times of the indices, Figure 4.19(b) puts both properties in relation to each other. The most memory efficient index is still PC_V2. However, similar to the *Escherichia coli* genome, the index is only slightly more memory efficient while it is magnitudes slower. In addition, the ESA-index is also still the fastest, but the most memory consuming as well. In contrast, the 2dOcc based FM-index with an Occ compression of 10 is now the second fastest. Further, its speed advantage is about 50%, while the index consumes about 5 times as much memory as the wavelet tree based ones (the two wavelet tree based FM-indices

-Wall -Winline -DDEBUG=0 -DNDEBUG=1



(a) Search times for the *Escherichia coli* genome. Note that both axes are scaled logarithmically.



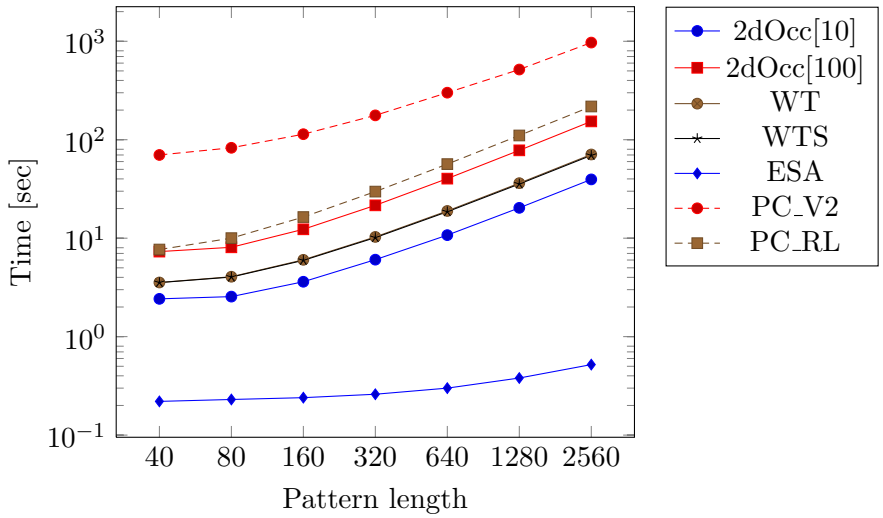
(b) Search time in dependence on the memory consumption for the *Escherichia coli* genome for a pattern length of 320. Note that the time axis is scaled logarithmically.

Figure 4.18: Search times for the *Escherichia coli* genome for different pattern lengths and in dependence of the memory consumption.

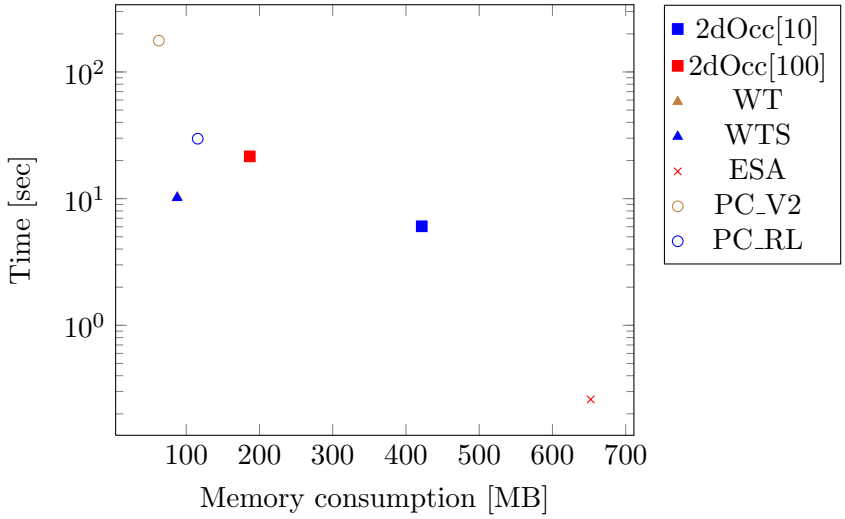
are represented by one symbol in the figure).

Finally, Figure 4.20(a) shows the running times for the *Homo sapiens* genome. Note that the two Pizza&Chili Corpus indices are missing. Applying them to the *Homo sapiens* genome resulted in an error message. Therefore, we only compare the FM-indices with ESA.

The order of the indices is the same as in Figure 4.19(a). However, the figure shows that the search times do not follow the same pattern as for the *Escherichia coli* or *Drosophila melanogaster* genomes. Instead, the time spent on the patterns of length 40 is magnitudes



(a) Search times for the *Drosophila melanogaster* genome. Note that both axes are scaled logarithmically.



(b) Search time in dependence of the memory consumption for the *Drosophila melanogaster* genome for a pattern length of 320. WT and WTS can not be distinguished in this figure. Note that the time axis is scaled logarithmically.

Figure 4.19: Search times for the *Drosophila melanogaster* genome for different pattern lengths and in dependence of the memory consumption.

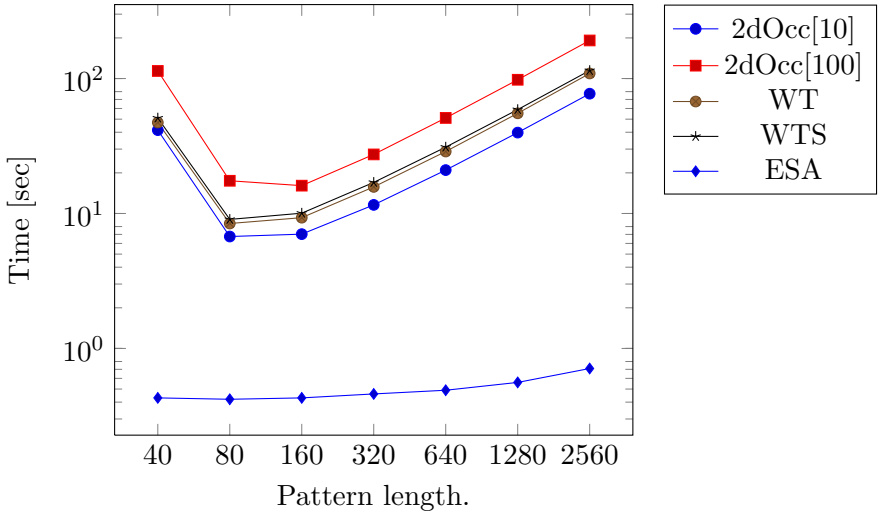
larger than the time spent on the other pattern length. Only the patterns of length 2560 require more time.

The explanation can be found in Table 4.2, where it is shown that the number of matches for patterns of length 40 is about 10 to 40 times larger than for the other patterns. Therefore, the localization phase requires a large amount of time, especially for the used SA compression of 100.

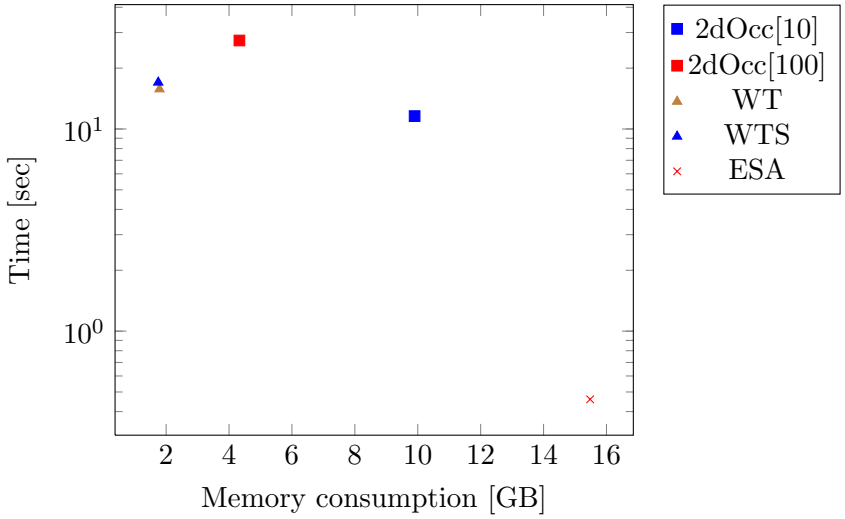
Figure 4.20(b) illustrates that the trade-off between time and memory consumption became larger in comparison with the smaller genomes. Even though the relation of the

memory consumption is similar, ESA is now up to 150 times faster than the wavelet tree based versions.

One of the reasons for this behaviour are cache effects. In contrast to the search using T^{BWT}, the more cache efficient ESA-index not only decreases the distance between the lower and upper bounds *sp* and *ep*, but also decreases the range with respect to the whole suffix array. Therefore, once the distance between *sp* and *ep* is small enough to fit the whole range into cache, it does not have to be reloaded. Hence, the extra time caused by cache misses is very small.



(a) Search times for the *Homo sapiens* genome. Note that both axes are scaled logarithmically.



(b) Search time in dependence of the memory consumption for the *Homo sapiens* genome for a pattern length of 320. Note that the time axis is scaled logarithmically.

Figure 4.20: Search times for the *Homo sapiens* genome for different pattern lengths and in dependence of the memory consumption.

		2dOcc[10]	2dOcc[100]	WT	WTS	ESA	PC_V2	PC_RL	# matches
E. coli	Memory [MB]	14.77	6.54	3.24	2.81	22.85	2.20	4.20	
	Time [sec]	1.05	3.24	0.91	0.79	0.10	42.26	2.74	109352
		1.54	5.11	1.308	1.15	0.11	54.38	4.5	108161
		2.49	8.85	2.14	1.83	0.12	77.76	8.10	106288
		4.35	16.32	3.77	3.21	0.14	124.43	15.11	103570
		8.10	31.36	7.04	5.99	0.18	217.05	29.49	101599
		15.52	61.38	13.61	11.53	0.25	396.82	58.16	100340
		30.31	121.73	26.65	22.63	0.39	764.72	117.34	100094
D. melanogaster	Memory [MB]	421.54	186.73	87.91	87.57	521.80	62.60	115.70	
	Time [sec]	2.42	7.31	3.55	3.54	0.22	70.03	7.67	317241
		2.55	8.08	4.06	4.03	0.23	82.81	10.01	198210
		3.61	12.31	6.03	5.95	0.24	113.81	16.39	159338
		6.05	21.56	10.30	10.14	0.26	176.78	29.73	140981
		10.74	40.26	18.91	18.56	0.30	300.45	56.54	125719
		20.33	78.04	36.22	35.50	0.38	516.39	110.46	113418
		39.54	153.60	70.82	69.36	0.52	969.91	217.62	103503
H. sapiens	Memory [GB]	9.9	4.33	1.79	1,75	15.5			
	Time [sec]	41.47	113.81	47.23	50.96	0.43			4244238
		6.75	17.48	8.43	9.04	0.42			437934
		7.03	16.08	9.31	10.05	0.43			122597
		11.58	27.40	15.74	17.00	0.46			102819
		20.97	51.06	28.79	31.02	0.49			100731
		39.76	97.94	55.26	58.88	0.56			100389
		77.22	191.52	109.06	114.65	0.71			100299

Table 4.2: Search time and memory consumption for different indices and pattern lengths.

All in all, we have shown that the wavelet tree based FM-indices are much more memory efficient than the ESA-index, and for the larger genomes even outperform the Pizza&Chili Corpus indices. In addition, all of our new FM-indices are faster than the indices of the Pizza&Chili Corpus. Further, in contrast to the Pizza&Chili Corpus indices all of the SeqAn indices were able to cope with the *Homo sapiens* genome. We have also illustrated that while the ESA-index is the most memory intensive index, it is by far the fastest, which is especially obvious for large sequences. In order to overcome this shortcoming of our new FM-index variants we implemented two speed optimization methods that are presented in the next section.

4.7 Time Optimization Methods

In this section we will present the results of the new speed up optimizations for the FM-index that have been described in Section 3.8.1 and Section 3.8.2.

We applied the two optimization methods to the *Drosophila melanogaster* genome and obtained the results shown in Figure 4.21.

While there are only marginal differences between the range optimized and the original FM-index search, shown in Figure 4.21(a), a significant improvement in the search time can be observed for the text optimized method, shown in Figure 4.21(b).

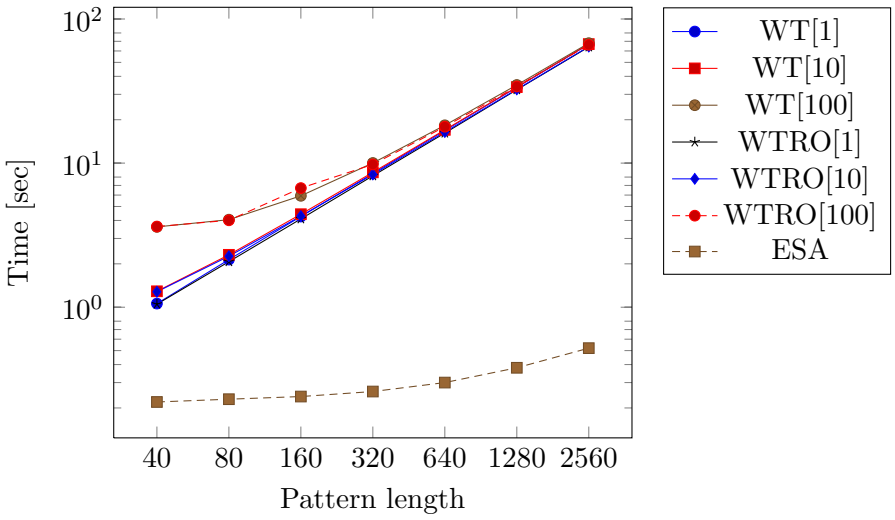
One possible explanation that range control does not enhance the speed are again cache effects, since (as already discussed in former sections) the counting as well as the localization phase are prone to cause cache misses. However, even though the two bounds *sp* and *ep* have to be computed, the necessary information might only be loaded once into cache when *sp* and *ep* are close enough to each other. Therefore, the range control method did not achieve the desired results, because it still was not able to circumvent cache effects.

In contrast, the text verification optimization reduces the effect of cache effects dramatically. As soon as *sp* and *ep* are equal, the method identifies the location of the remaining prefix in the original text. Due to the compression of the SA, first the location of the pattern has to be computed (which causes cache effects). However, afterwards cache effects only occur if the identified text or the pattern exceed the cache size limit. Further, the remaining prefix of the pattern is already verified during the localization phase, such that this step does not have to be repeated.

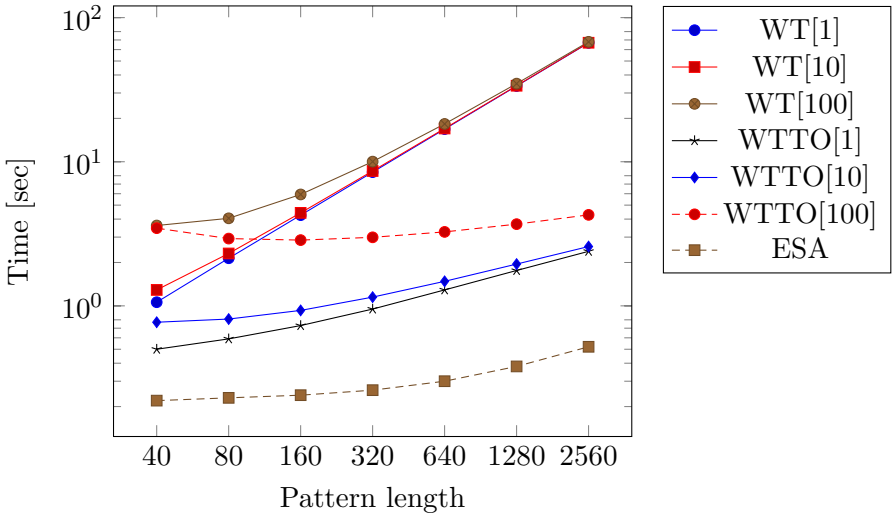
It is interesting to see that the most significant decrease in time consumption can be observed for a SA compression rate of 100. The reason for this is that the localization is already integrated into the search and does not consume additional time. This especially holds for short patterns, since in those cases the localization phase demands a high percentage of the overall time. However, it does not explain why the overall time decreased from pattern length 40 to 80 and from pattern length 80 to 160. The reason for this behaviour is the higher number of matches, as can be seen in Table 4.2.

4.8 Application to String Sets

After we have extensively described and discussed the results gained for single strings, we will change the focus to sets of strings. Since the search procedure for strings and string sets is identical and the additional memory consumption for the SA (which is now a string of pairs, as described in Section3.10) is the same for all indices, we will not repeat the



(a) Search times for the *Drosophila melanogaster* genome for the original FM-index search and the range control optimization methods as well as for the ESA-index.



(b) Search times for the *Drosophila melanogaster* genome for the original FM-index search and the text verification optimization methods as well as for the ESA-index.

Figure 4.21: Search times for the *Drosophila melanogaster* genome for the original FM-index search, the range control and the text verification optimization methods as well as for the ESA-index. WTRO denotes the range control optimization applied to WT, while WTTO denotes the text optimization method applied to WT. Note that both axes are scaled logarithmically in each figure.

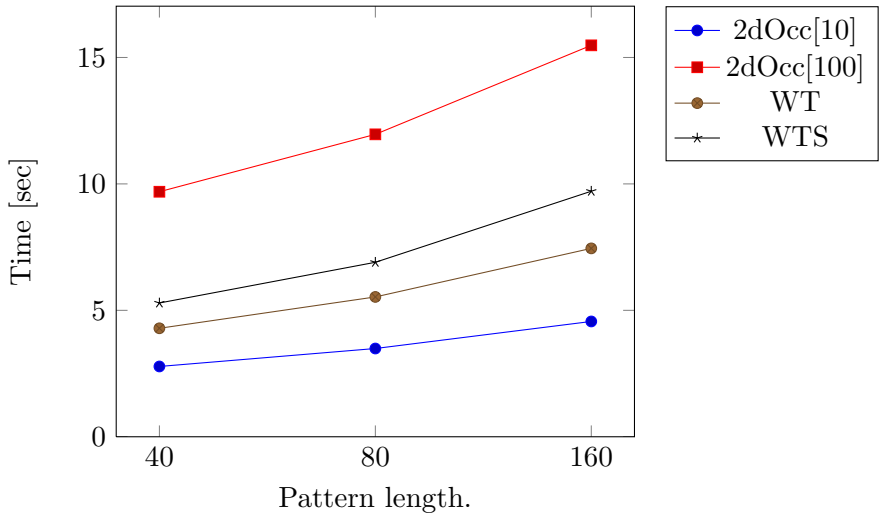
whole analysis of the previous sections. Instead, we will only point out the main differences of the two input formats that can be seen in Figure 4.22.

Note that in difference to the benchmarking of single strings, now we use a text that is not contiguous, but consists of 100,000 strings of length 320. Those strings, as well as the patterns, were generated in the same fashion as before.

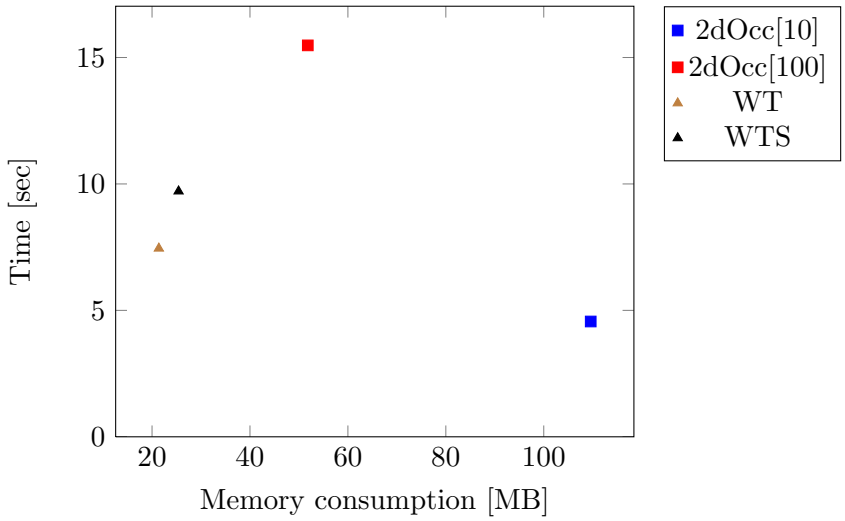
In contrast to the search time behaviour of the FM-index on strings, now the wavelet tree based index that incorporates an extra node representing the $ is both faster and more memory efficient than the other wavelet tree based version.

The reason for this is the number of $ signs. While only one position has to be stored in the case of a single string, there are several in the case of a string set, depending on the number of included strings. We solved this problem using a rank support bit string (RSBS) indicating the $ positions. The extra node requires less memory than the RSBS, which explains the smaller memory consumption for the FM-index implementation incorporating the additional node.

The speed advantage is also a result of a more compressed representation of the $ positions for the FM-index with the extra node for the $. Here cache effects occur not as often, because the $ signs are closer together, reducing the probability of the need to load new information into the cache. In contrast, the $ signs of the FM-index substituting the $ can be distributed over the whole length of T^{BWT}.



(a) Search times for the different FM-index implementation using a set of strings as input. Note that the pattern length axis is scaled logarithmically.



(b) Search time in dependence on the memory consumption.

Figure 4.22: Search times of the different FM-index implementation taking a set of strings as input.

5 Conclusion and Outlook

The technological development in the field of genome research has resulted in a massive generation of data that has to be stored and analyzed. The enormous amount of information demands special data structures and algorithms for an efficient analysis. This need is met by the open source library SeqAn that aims at providing sophisticated and up-to-date software for biological sequences.

In this thesis we designed, implemented and compared different FM-index variants in SeqAn. We compared several existing designs with a focus on their realization of the occurrence table data structure, the core module of the FM-index. Based on the results, we chose the wavelet tree data structure to be the most suitable for this task.

In order to realize a memory efficient wavelet tree that quickly returns an occurrence query, we designed a sophisticated tree structure and implemented a rank support bit string, which is able to return the rank of a specified position in constant time. To complete the FM-index we also designed a compressed suffix array as well as a prefix-sum table that is applicable to non-finite alphabets.

We implemented two wavelet tree based variants that process the $ sign necessary for an efficient Burrows-Wheeler Transform. The first variant is based on a substitution of this sign, while the second one incorporates an extra node for the $. Furthermore, for a comparison we created a third FM-index version that is based on a 2-dimensional array.

In a benchmarking study we analyzed the different FM-index versions in regard to pattern search time and memory efficiency. We also applied our methods to real world data sets and compared them to the enhanced suffix array index of SeqAn (ESA-index) and two FM-index implementations from the Pizza&Chili Corpus.

The benchmarking showed that the wavelet tree based variants are far more memory efficient than the array based index. Further, depending on the occurrence table compression rate, the wavelet tree based implementations are also faster.

In comparison to other indices, our wavelet tree based variants are much more memory efficient than the ESA-index, and for the larger genome even outperform the Pizza&Chili Corpus indices. In addition, all of our new FM-indices are faster than the indices of the Pizza&Chili Corpus. We have also shown that while the ESA-index is the most memory intensive index, it is the fastest one, which is especially obvious for large sequences. Therefore, two speed optimization extensions have been incorporated in the new FM-index variants.

Further, we provided a suffix trie iterator that can be used for inexact pattern search.

Another characteristic of our FM-index implementation is its ability to also accept sets of strings as the input.

Further, future versions shall be extended to except an even larger range of input instances. For instance, though our focus was to provide a data structure efficient for biological sequences it shall also be applicable to other kinds of texts, for example those based on very large alphabets. We already adapted the prefix-sum table to handle large alphabets, though (due to time constrains) exhaustive testing and verification remains a

future task.

Furthermore, it would be interesting to exchange our wavelet tree structure implementation with another one including pointers from a parent node to its child nodes. In doing so, at the cost of additional memory one could spare a few of the currently necessary comparisons and hence further decrease the running time.

Another speed optimization approach would be to exchange the whole wavelet tree with another occurrence table data structure in which the number of occurrences is encoded with a rank support bit string for each character. In doing so, one could discard the tree traversal and directly access the correct bit string, again at the cost of additional memory.

All in all, we created a generically designed and powerful extension to the SeqAn library that is both memory efficient and supports fast pattern searching.

Acknowledgements

I like to thank my two supervisors, Prof. Dr. Knut Reinert and Dr. Markus Bauer, for giving me the opportunity to work on this exciting project.

I want to thank David Weese for your continuous support, ideas and comments and that you were always available when I had questions or difficulties. I enjoyed working on the project, even though I had some challenging times.

A big thank you to Franziska Zickmann for your continuous support and for keeping be grounded during stressful times. Thank you for spending hours reviewing ideas and helping me to overcome any difficulties.

In addition I want to thank Kathrin Trappe, René Rahn, Leon Kuchenbecker and Konrad Ludwig Moritz Rudolph for your useful advises and support.

Last but not least I want to thank my family and friends for your continuous support.

Bibliography

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 1 edition, 2008.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

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

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *Systems Research*, Research R(124):1–24, 1994.

[5] A. Döring, D. Weese, T. Rausch, and K. Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 2008.

[6] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52:688–713, 2005.

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390 – 398, Washington, DC, USA, 2000. IEEE Computer Society.

[8] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 269–278, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[9] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 655–663, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[10] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52:552–581, 2005.

[11] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3, 2007.

[12] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly fm-index. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer Berlin / Heidelberg, 2004.

[13] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of burrows-wheeler compression algorithms. In Ricardo Baeza-Yates, Edgar Chávez, and Maxime

Crochemore, editors, *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 129–143. Springer Berlin / Heidelberg, 2003.

[14] S. Grabowski, V. Mäkinen, and G. Navarro. First huffman, then burrows-wheeler: A simple alphabet-independent fm-index. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 435–447. Springer Berlin / Heidelberg, 2004.

[15] S. Grabowski, G. Navarro, and R. Salinger. A simple alphabet-independent fm-index. In *International Journal of Foundations of Computer Science (IJFCS)*, pages 230–244, 2006.

[16] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[17] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[18] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[19] D. E. Knuth. *The art of computer programming. Vol. 1, Fundamentals algorithms.* Addison-Wesley Pub. Co., 3 edition, 1997.

[20] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[21] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19:261–296, 1987.

[22] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[23] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[24] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[25] D. J. C. Mackay. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 1st edition, 2003.

[26] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[27] G. Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48:407–430, 2001.

[28] Veli Mäkinen and Gonzalo Navarro. Run-length fm-index. In *In Proc. DIMACS Workshop: The Burrows-Wheeler Transform: Ten Years Later*, pages 17–19, 2004.

[29] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39:1–66, 2007.

[30] S. A. Przybylski. *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[31] D. Salomon. *Data Compression: The Complete Reference*. Springer, 2004.

[32] K. Sayood. *Introduction to Data Compression, Third Edition (Morgan Kaufmann Series in Multimedia Information and Systems)*. Morgan Kaufmann, 3 edition, 2005.

[33] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27:379–423, 1948.