

Dynamic FM-Index for a Collection of Texts with
Application to Space-efficient Construction of the
Compressed Suffix Array

Diplomarbeit
im Fach Naturwissenschaftliche Informatik
von
Wolfgang Gerlach

28. Februar 2007

Betreuer: Dipl.-Inform. Klaus-Bernd Schürmann
Dr. Veli Mäkinen
Prof. Dr. Jens Stoye



Contents

1	Introduction	1
1.1	Related Work	2
2	Prerequisites and Basic Concepts	7
2.1	Notation	7
2.2	Entropy	8
2.3	Queries	8
2.4	Red-Black Tree	9
2.5	Suffix Tree	9
2.5.1	Using the Suffix Tree	11
2.5.2	Asymptotics	11
2.6	Suffix Array	12
2.6.1	Concept	12
2.6.2	Using the Suffix Array	12
2.6.3	Enhanced Suffix Array	13
2.6.4	Construction	13
2.7	Rank and Select on a Static Bit Vector in Constant Time	13
2.7.1	Zaks' Sequence	14
2.7.2	Static Bit Vector with Rank	15
2.8	Rank and Select on General Sequences	17
2.8.1	Linear Representation	18
2.8.2	Balanced Wavelet Tree	18
2.8.3	Huffman-shaped Wavelet Tree	20
2.8.4	k-ary Wavelet Tree	21
2.9	Burrows-Wheeler Transformation	21
2.9.1	Reverse Transformation	22
2.9.2	Compression	23
2.10	Backwards Search	23
2.11	Sampled Suffix Array	24
2.12	FM-Index	25
3	Direct Construction of the Burrows-Wheeler Transform	27
3.1	Single Text	27
3.2	Collection of Texts and the Pseudo BWT	29

4	Implementation of a Dynamic FM-Index	33
4.1	Dynamic Bit Vector with Indels	33
4.1.1	Data Structure	33
4.1.2	Algorithms	34
4.1.3	Asymptotics	35
4.2	Dynamic Sequence with Indels	35
4.3	Dynamic FM-Index for a Collection of Texts	36
4.3.1	Table C	38
4.3.2	HANDLE and POS	39
4.3.3	DSSA – Dynamic Sampled Suffix Array	41
4.3.4	Implementation Remarks	42
4.3.5	Asymptotics of the Dynamic FM-Index	43
5	Experiments	45
5.1	Experiment 1: Different Block sizes	45
5.2	Experiment 2: Comparison of the BWT construction	48
5.3	Experiment 3: Dynamic versus Static FM-Index	50
6	Summary and Prospects	53
	Bibliography	57

List of Figures

2.1	Suffix tree for “mississippi\$” with edge labels.	10
2.2	Suffix tree for “mississippi\$” with indices.	10
2.3	Suffix tree for “mississippi\$” with suffix links (grey).	11
2.4	Suffixes of <i>mississippi</i> in sorted order.	12
2.5	Assignment of bits to the nodes of the tree and its new leaves.	14
2.6	Representation of traditional tree (left) with B and array of values (right).	15
2.7	Computation of $\text{rank}_s(10)$ in the Wavelet Tree for “mississippi”.	19
2.8	Conceptual BWT matrix $M(t)$, where $t = \text{“mississippi$”}$	22
3.1	Transformation of $\text{bwt}(\text{“ississippi$”})$ into $\text{bwt}(\text{“mississippi$”})$	27
3.2	Suffixes of “mississippi\$” (left) and the corresponding BWT(right).	28
3.3	Difference between $\text{bwt}(\text{“c$b$”})$ (left) and $\text{bwt}(\text{“ac$b$”})$ (right).	30
3.4	Differences between $M(\text{“dddxz$xz$...”})$ and $M(\text{“dddxz$bxz$...”})$	30
3.5	Reconstruction of “dddxz\$” using correct BWT (left) and pseudo BWT (right).	31
4.1	HANDLE and POS.	40
5.1	Effects of different LOGN values on the time usage.	46
5.2	Effects of different LOGN values on the space usage.	46
5.3	Several time-space tradeoffs with varying LOGN.	47
5.4	Zoomed in: Several time-space tradeoffs with varying LOGN.	47
5.5	Time comparison of DFMI, bwt, and dnabwt.	49
5.6	Space comparison of DFMI, bwt, and dnabwt.	49
5.7	DFMI versus FMI: DNA	51
5.8	DFMI versus FMI: Proteins	51
5.9	DFMI versus FMI: English texts	52

List of Algorithms

1	Reverse Burrows-Wheeler transformation	22
2	BackwardsSearchCount($p[1..m], L[1..n]$)	24
3	Find new handle in HANDLE	41

1 Introduction

The system of nature, of which man is a part, tends to be self-balancing, self-adjusting, self-cleansing. Not so with technology.

(E. F. Schumacher – *Small is Beautiful*, 1973)

The storage of huge amounts of electronic data and the fast availability of these data have become indispensable in our today's knowledge-based society. But the development of electronic storage capacity and speed doesn't keep up with the exponential growth of data. In biology, for example, new high-throughput technologies for DNA sequencing, microarray gene expression analysis, and mass spectrometry produce huge amounts of data that need to be stored.

To address the problem of storage of an increasing amount of data, many data compression techniques are available. They are established in many fields of our everyday life that deal in some way with electronic data transmission and storage. There are two kinds of data compression, lossless and lossy compression. Lossless compression allows the exact retrieval of the original data. Lossy compression, in contrast, which is used for example for sound, images or videos, discards information. The main idea is to discard that kind of information that is less important for sound, images or videos to be recognized by a human, thus reaching a higher compression rate¹.

With the fast increasing amount of data another problem arises: Without any additional data structures, search within the stored data takes time linear to the size of the data, which is unfeasible if there is much data. Furthermore, if the data is compressed, most common compression techniques require that the whole data gets decompressed to allow searches within the data or retrieve parts of the data. Suboptimal algorithms, hardware costs, and -limits not only affect the time a user is waiting for a response to his inquiry to some database, but also restricts the potential for scientific and commercial applications.

The problem of increased look-up times can be solved by using data structures which enable sublinear-time look-up, called *indices*. There are two kinds of indices available, word-based and sequence-based indices. Word-based indices partition the text into

¹The ratio between the length of the original and the length of the compressed data.

words and store for each word the information where it occurs in the text. Those indices enable fast queries and are space-efficient. A disadvantage is that they are not applicable to biological sequences like DNA, proteins, or audio and video signals. Sequence-based indices allow to search efficiently for any substring within the text. Classical examples of such sequence-based indices are the suffix tree and the suffix array. The problem with these indices is that they require space several times the text size in addition to the text itself. Thus, with the increasing amount of available data, space-efficient or *succinct indices* have become important.

The idea behind the development of compressed data structures is to achieve the smallest possible representation of a data structure while maintaining its functionality. Furthermore, the operations on the data structure should be supported in asymptotic optimal time and without decompressing the whole structure. A succinct index is an index built on top of the text to provide fast search functionality while using space proportional to that of the text itself. The suffix tree and the suffix array are not succinct as they require $\mathcal{O}(n \log n)$ bits of space in contrast to the $n \log \sigma$ bits of space of the text, where n is the length of the text and σ is the size of the alphabet. A *compressed index* is an index that uses space proportional to that of the compressed text and a *self-index* is a compressed index that, in addition to providing search functionality, can efficiently reproduce any substring. Thus, the self-index replaces the text.

Our contribution is the first implementation of a *dynamic FM-index for a collection of texts*, a self-index whose space usage is bound by the 0-order entropy of the text to be encoded. The index allows to insert or delete texts of the collection, which avoids reconstruction of the whole index as it is necessary for static indices even for small modifications. We further show that an immediate result of the dynamic FM-index is the space-efficient construction of the *Burrows-Wheeler Transformation*.

1.1 Related Work

Some of the most basic succinct data structures that have been studied extensively in the last years are those for the *Partial Sums* problem and its variants.

The *Partial Sums problem* [RRR01] has two positive integer parameters, the item size $k = \mathcal{O}(\log n)$ and a maximal increment of $\delta_{\max} = \log^{\mathcal{O}(1)}(n)$. For a sequence of n integer numbers, $A[1], \dots, A[n]$, where $0 \leq A[i] \leq 2^k - 1$, two operations are provided:

- $\text{sum}(i)$: returns $\sum_{j=1..i} A[j]$
- $\text{update}(i, \delta)$: sets $A[i] \leftarrow A[i] + \delta$, for some integer δ with $0 \leq A[i] + \delta \leq 2^k - 1$ and $|\delta| \leq \delta_{\max}$

The *Searchable Partial Sums problem* is an extension of the Partial Sums problem. The additional operation *search*, also called *select*, is defined as:

- $\text{search}(i)$: returns $\min\{j \mid \text{sum}(j) \geq i\}$

The *Searchable Partial Sums with Indels problem* is an extension of the Searchable Partial Sums problem that allows to insert and to delete numbers of the sequence:

- $\text{insert}(i, x)$: inserts number x between $A[i - 1]$ and $A[i]$
- $\text{delete}(i)$: deletes $A[i]$ from the sequence

The word *indel* is an abbreviation that is used to indicate that something supports both, (in)sertions and (del)etions.

The *Dynamic Bit Vector problem* is a restricted version of the Searchable Partial Sums problem, for which $k = 1$. Furthermore, the operations sum , update , and search are replaced by the corresponding operations *rank*, *flip*, and *select*. For rank and select the above definitions of sum and search can be used. flip is different from its corresponding operation update since parameter δ does not apply here:

- $\text{flip}(i)$: sets $A[i] \leftarrow \begin{cases} 0 & \text{if } A[i] = 1, \\ 1 & \text{if } A[i] = 0. \end{cases}$

The *Dynamic Bit Vector with Indels problem* extends the Dynamic Bit Vector problem with the operations insert and delete . For both operations the corresponding definitions of the Searchable Partial Sums with Indels problem with x being restricted to the values 0 and 1 can be used.

The *Dynamic Sequence with Indels problem* [MN06] is an extension of the Dynamic Bit Vector with Indels problem with an alphabet $\{0, \dots, \sigma - 1\}$ of size σ . The supported operations are:

- $\text{rank}_c(i)$: returns number of occurrences of symbol c in A up to position i
- $\text{select}_c(i)$: returns $\min\{j \mid \text{rank}_c(j) = i\}$
- $\text{insert}(c, i)$: inserts c between $A[i - 1]$ and $A[i]$, where $c \in \{0, \dots, \sigma - 1\}$
- $\text{delete}(i)$: deletes $A[i]$ from the sequence A

Dietz [Die89] gives a solution to the Partial Sums problem that needs $\mathcal{O}(\log n / \log \log n)$ time for the two operations sum and update . The space usage of his solution is $\Theta(n \log n)$ bits in addition to the sequence, where $k = \Theta(\log n)$. Raman et al. [RRR01] give a solution which is based on Dietz's solution. They reduced the space usage to $kn + o(kn)$ bits and generalized the result to $k = \mathcal{O}(\log n)$. They gave a trade-off result with $\mathcal{O}(\log_b n)$ time for sum and $\mathcal{O}(b)$ time for update for any $b \geq \log n / \log \log n$. Furthermore, for the Searchable Partial Sums problem, they added the select operation and gave a solution where all operations take $\mathcal{O}(\log n / \log \log n)$ worst-case time. Their solution also uses $kn + o(kn)$ bits of space. A tradeoff as above, they give only

for $k = 1$ using $o(n)$ bits in addition to the bit vector. The update time reduces to amortized $\mathcal{O}(b)$.

For the static Bit Vector problem two constant time solutions are presented in this work (see 2.7.2), one using $n + o(n)$ bits and the other using $nH_0 + o(n)$ bits of space, where H_0 is the 0-th order entropy.

Hon et al. [WKHS03] further improve the results of Raman et al. for the Searchable Partial Sums problem, allowing the tradeoff rank and select in $\mathcal{O}(\log_b(n))$ and update in $\mathcal{O}(b)$ time for any positive integer $k = \mathcal{O}(1)$. They also give a solution to the Searchable Partial Sum with Indels problem by providing the operations delete and insert in $\mathcal{O}(b)$ amortized time. Mäkinen and Navarro [MN06] give a solution with the same space usage but $\mathcal{O}(\log n)$ worst-case time for all operations and for any $k = \mathcal{O}(\log n)$.

The first entropy-bound solution of the Dynamic Bit Vector with Indels problem is given by Blandford and Blelloch [BB04]. Their solution uses $\mathcal{O}(nH_0)$ bits of space. Mäkinen and Navarro [MN06] further reduce the constant factor to one, achieving a $nH_0 + o(n)$ bits solution.

Dynamic Sequence with Indels

In 2000 Ferragina and Manzini [FM00] proposed an index for a dynamic collection of z texts that uses $\mathcal{O}(nH_k + z \log n) + o(n)$ bits of space, for any fixed $k \geq 0$. This index can find *occ* occurrences of some pattern p in $\mathcal{O}(|p| \log^3 n + occ \log n)$ worst-case time. The insertion of some text t_{new} takes $\mathcal{O}(|t_{\text{new}}|n)$ amortized time and the deletion of some text t_i takes $\mathcal{O}(|t_i| \log^2 n)$ time, also amortized.

Chan et al. [CHL04] remove the $z \log n$ term and achieve an $\mathcal{O}(n\sigma)$ bits representation. Insertion or deletion of some text t_i takes $\mathcal{O}(|t_i| \log n)$ time, while searching for a pattern p takes $\mathcal{O}(|p| \log n + occ \log^2 n)$ time. $\mathcal{O}(|p| \log n)$ is the time to count occurrences of a pattern.

Mäkinen and Navarro [MN06]² extend the results of Hon et al. [WKHS03] and present a dynamic data structure that requires $nH_0 + o(n)$ bits which performs rank, select, and insert and delete of bits in $\mathcal{O}(\log n)$ worst-case time.

They use these results to show how a dynamic full-text self-index for a collection of z texts can be built. Their index uses the wavelet tree (Section 2.8.3) and achieves a space usage of $nH_h + o(n \log \sigma)$ bits for any $h \leq \alpha \log_\sigma n$, constant $0 < \alpha < 1$ and with weaker condition $\log n = \mathcal{O}(w)$. All character insertion and deletion operations take $\mathcal{O}(\log n \log \sigma)$ time. Counting a pattern takes $\mathcal{O}(|p| \log n \log \sigma)$ time and locating

²Results are taken from a preprint journal version of this paper – personal communication with Veli Mäkinen.

each occurrence takes $\mathcal{O}(\log^2 \log n)$ time. Retrieving a substring of length l takes $\mathcal{O}(\log n(l \log \sigma + \log n \log \log n))$ time.

They provide a tradeoff which allows to query in $\mathcal{O}(1/\epsilon \log n \lceil \log \sigma / \log \log n \rceil)$ time. The update time then increases to $\mathcal{O}(1/\epsilon \log^{1+\epsilon} n \log \log n)$.

2 Prerequisites and Basic Concepts

In this work we assume a standard *word* random access model (RAM). The computer wordsize w is assumed to be such that $w = \Theta(\log n)$ ¹, where n is the maximum size of the problem instance. The standard operations (bit-shifts, additions) on $\mathcal{O}(w)$ -bit integers are assumed to take constant time in this model.

2.1 Notation

Throughout this work we will use the following notations. $\Sigma = \{c_1, \dots, c_\sigma\}$ is a finite alphabet over a set of symbols, where σ denotes the size of Σ , $\sigma = |\Sigma|$. For example, DNA consists of the alphabet $\{A, C, G, T\}$ of size four. The symbols $\{c_1, \dots, c_\sigma\}$ are ordered lexicographically, $c_1 < \dots < c_\sigma$. Further notations are:

- $\$$ is a symbol not occurring in Σ , $\$ \notin \Sigma$.
- $\$$ is lexicographically smaller than any symbol in Σ , $\forall c \in \Sigma : \$ < c$.
- $t \in \Sigma^n$ is a text² of length $n = |t|$, $n \in \mathbb{N}$.
- ϵ denotes the empty text with length 0.
- $t[i]$ is the symbol at position i in t , $1 \leq i \leq n$.
- $t[a, b] := t[a]t[a+1]\dots t[b]$ denotes the *substring* (or *pattern*) of t that begins at position a and ends at position b , $1 \leq a \leq b \leq n$.
- p is a *prefix* of the string $t \quad :\iff \quad p \in \{t[1, i] \mid i \in [1..n]\} \cup \{\epsilon\}$
- p is a *suffix* of the string $t \quad :\iff \quad p \in \{t[i, n] \mid i \in [1..n]\} \cup \{\epsilon\}$
- The *longest common prefix* (short: *lcp*) of a set of k strings $\{t_1, \dots, t_k\}$ is defined as follows:

$$\text{lcp}(t_1, \dots, t_k) := t_1[1, i], \text{ where } i = \max\{m \in \mathbb{N}_0 \mid t_1[1, m] = \dots = t_k[1, m]\}.$$

- If not explicitly stated otherwise, \log always denotes \log_2 .

¹See [NM06a] for the more general case $\log n = \mathcal{O}(w)$.

²The terms *text*, *string*, and *sequence* are used interchangeably.

2.2 Entropy

Based on previous ideas of information by Hartley [Har28], Shannon [Sha48] defined the information content (also called *self-information*) of an event E to be

$$I(E) = -\log(p_E), \quad (2.1)$$

where p_E is the probability of the event E . The unit of this measure is bit³ when the base of the logarithm is 2. This definition rates less probable events with a higher information content.

Also based on Shannon's notion of entropy, the *zero-order entropy* of a text t over an alphabet $\Sigma = \{c_1, \dots, c_\sigma\}$ of size σ with uncorrelated symbols and with probabilities $p(c_i)$ for each symbol $c_i \in \Sigma$ is

$$H_0(t) = -\sum_{i=1}^{\sigma} p_i \log(p_i). \quad (2.2)$$

The entropy gives a lower bound for the average code length. Thus, $nH_0(t)$ denotes a lower bound for the compressed length of a text t , when the compressor is required to use a fixed code table to encode the symbols of t .

The conditional *k-th order entropy*, where the probability of a symbol c_i depends on the k previous symbols in a given text t , the so called *k-kontext* W , is defined as

$$H_k(t) = -\sum_{i=1}^{\sigma} \sum_{W \in \Sigma^k} p(Wc_i) \log(p(c_i|W)). \quad (2.3)$$

nH_k is the lower bound for the compressed length of a text t , when the compressor is required to use a fixed code table for each different context of length k . The k -th order entropy is a more realistic lower-bound for texts. For example, considering the three-context in English texts it is much more likely for the symbol e to occur after the sequence “th” than after “the”.

2.3 Queries

Indices are used for different search queries. We consider three queries, *occurrence*, *count* and *locate*, which can be differentiated. *Occurrence* asks if the string p is a substring of t . *Count* tells how often p occurs in t . *Locate* gives the set of all positions in t , where p occurs. Although, *locate* contains the answers for the other queries, some indices need more time to answer this query than for *occurrence* and *count*.

³Short for *binary digit*. Supposedly leads back to J. W. Tukey [Sha48].

2.4 Red-Black Tree

A red-black tree is a balanced binary search tree with coloured nodes. It satisfies the following five red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

This definition is taken from [CLRS01]. Leaves are meant to be additional nodes that don't carry satellite information as the other nodes. Furthermore they are all replaced by a special node, the sentinel. Instead of using the NULL pointer for relatives (parent, left-, and right-child) where the relatives don't exist, we point to the sentinel node.

Maintaining the above five red-black properties guarantees that the height of tree always remains $\mathcal{O}(\log n)$, where n is the number of nodes in the tree. Thus, all algorithms on a tree that traverse a path of the tree in a bottom-up or top-down fashion and use only constant time operations at each node on the path are also guaranteed to have a worst-case run time of $\mathcal{O}(\log n)$.

Given a red-black tree, it is possible that the insertion or deletion of a node destroys some of the red-black properties. A constant number of rotations, which are local shifts of nodes including their subtrees in constant time, restore the red-black property of the whole tree. Whenever we describe algorithms which work on red-black trees, we assume that rotations are performed implicitly after insertion and deletion of nodes when necessary.

2.5 Suffix Tree

An explicit index of all substrings contained in t would be quite space-inefficient, since t can contain up to n^2 different substrings. A more space-efficient approach is the *suffix tree*. Here the fact is used that each substring of t is a prefix of a suffix of t . To make sure that no suffix is a prefix of another suffix, we append the unique character $\$$ to t . Each substring is now implicitly represented by the suffix starting at the same position in $t\$$.

Each edge of the suffix tree (Figure 2.1) is labeled with a substring of $t\$$ and each path from the root to a leaf represents a suffix of $t\$$, which can be obtained by concatenating

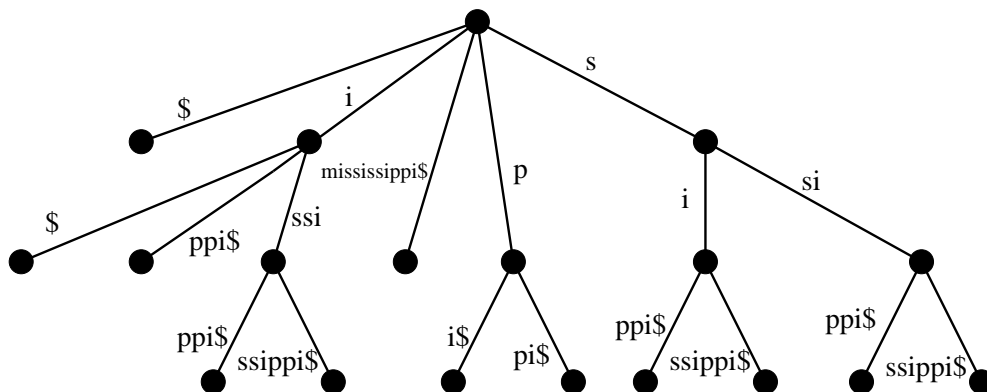


Figure 2.1: Suffix tree for “mississippi\$” with edge labels.

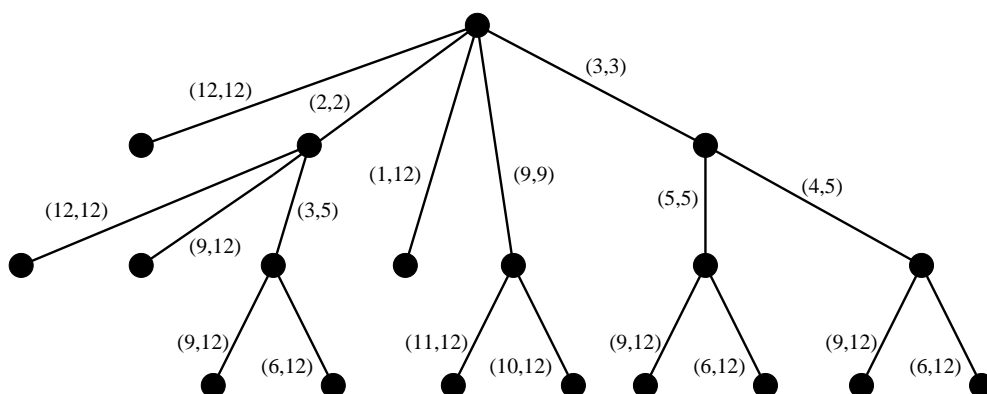


Figure 2.2: Suffix tree for “mississippi\$” with indices.

the labels of all edges on this path. We denote this the path-label. Outgoing edges of the same node start with different labels. Furthermore, each suffix of $t\$$ is represented in this tree. Thus, using $\$$, there is a bijection between leaves of the suffix tree and the suffixes of $t\$$. The suffix tree we consider here is a compact tree, that is, each edge leads to a branching internal node or to a leaf.

Notice that not the substring itself is stored, instead two indices, pointing to the beginning and end of an occurrence in $t\$$ (Figure 2.2).

Suffix links (Figure 2.3) are needed to achieve linear construction time and are used in some applications mentioned below. For each non-root node with path label ax , where a is a single symbol and x some string, a suffix link points to the node with path label x . If x is empty, the suffix link points to the root node. The root node itself doesn't have a suffix link.

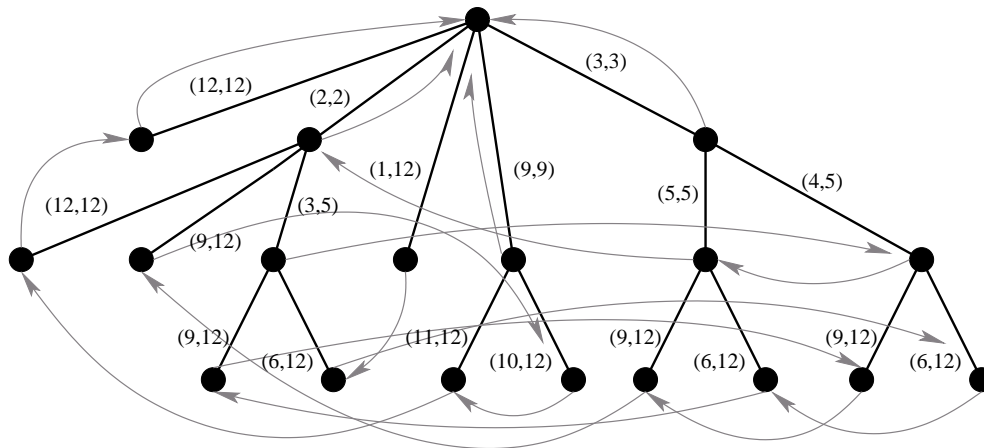


Figure 2.3: Suffix tree for “mississippi\$” with suffix links (grey).

2.5.1 Using the Suffix Tree

Querying for the occurrence of a pattern p in $t\$$ works as follows: We check if the root node has an outgoing edge whose label equals a prefix of p . If there is no such edge, there is no occurrence. Otherwise, we go to the node the edge is pointing to. We repeat the same procedure as for the root node, this time comparing the symbols in p following those already compared. This is done until the end of p or $\$$ is reached. With this procedure querying for occurrences takes $\mathcal{O}(m)$ time with a fixed-size alphabet, where m is the length of p . Counting the number of occurrences is done by counting the number of leaves of the subtree below the match. This takes $\mathcal{O}(m + occ)$ time, where occ is the number of occurrences. For locate, one takes the last pointer of each path to these leaves, and subtracts the length of the path from the root to the leaf.

Further applications of suffix trees are finding e.g. *maximal exact matches* (MEM’s), *maximal unique matches* (MUM’s), *minimal unique substrings*, *maximal repeats*, and *tandem repeats* [Apo85].

2.5.2 Asymptotics

The space needed by the suffix tree is $\Theta(n \log n)$ bits, plus $\mathcal{O}(n \log |\Sigma|)$ bits for the text. Today, there exist several kinds of suffix tree construction algorithms that differ in time and space usage. Some incorporate special types of construction schemes like online- or lazy-construction. The algorithms with best worst-case running time asymptotics for constructing suffix trees are optimal linear-time algorithms [Wei73, McC76, Ukk95, FCFM00]. Other algorithms exist with e.g. $\mathcal{O}(n \log_{|\Sigma|} n)$ expected running time that perform well in practice, sometimes even better than the linear-time algorithms due to their good locality behavior and the possibility of parallelization and lazy construction [GKS99].

Although linear in the length of the text, the space needed for suffix trees is quite large. They require about 20 [Kur99] to 30 bytes per input character in the worst case, depending on the implementation. Some implementations achieve 10 bytes per input character on average [Kur99].

2.6 Suffix Array

The *suffix array* was developed by Manber and Myers [MM93]. It uses $\Theta(n \log n)$ bits for the index, plus $\mathcal{O}(n \log |\Sigma|)$ for the indexed text like the suffix tree, but with smaller constant factors. In the typical case of word size $w = 4$ bytes and $\log n \leq w$, the suffix array requires only $4n$ bytes.

2.6.1 Concept

A suffix array is an array that contains all suffixes, represented by the starting positions of the suffixes, of a sequence t in lexicographically sorted order. For example, the string “mississippi” contains eleven⁴ suffixes. These eleven suffixes are shown in Figure 2.4 in sorted order, together with their starting positions (left column).

11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sissippi
6:	ssippi
3:	ssissippi

Figure 2.4: Suffixes of *mississippi* in sorted order.

The starting positions in Figure 2.4 read top-down give the suffix array [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] of “mississippi”.

2.6.2 Using the Suffix Array

Occurrences of a pattern p can be found by a binary search on the suffix array using lexicographical comparisons. Notice that if one occurrence of p has been found, the

⁴The twelfth suffix ϵ can be ignored since it is lexicographically always the smallest suffix.

other occurrences, if any, are direct neighbors. This rather simple approach takes $\mathcal{O}(m \log n)$ time since each comparison takes $\mathcal{O}(m)$ time.

This can be improved by keeping track of the length of the *longest common prefix* (see Section 2.1) of three strings, the suffixes at the current lower and upper border of the binary search and the pattern. Then the lexical comparisons of p and the suffixes can be accelerated by starting the comparisons of p and a suffix by this new offset, thus avoiding unnecessary character comparisons. This improves the average-case search time to $\mathcal{O}(m + \log n)$. Storing the lcp values in a preprocessing step during the construction of the suffix array further improves the worst-case time for the search to $\mathcal{O}(m + \log n)$ [MM93].

2.6.3 Enhanced Suffix Array

Another recent development is the *enhanced suffix array* [AKO04] with additional tables (called *lcp-*, *child-*, and *suffix link table*), which can be implemented to require little more than $4n$ bytes [AKO04]⁵. For example, finding all *occ* occurrences of a pattern p can be performed in optimal time $\mathcal{O}(m + occ)$. Moreover, because the enhanced suffix array simulates the traversal⁶ of suffix trees all the suffix tree applications can be performed in the same time complexity on the enhanced suffix array. Depending on the application, not all tables are always needed.

2.6.4 Construction

The suffix array can be constructed in $\mathcal{O}(n)$ time, e.g. by first constructing a suffix tree [Gus97]. Direct construction has been shown to be possible, too [KSPP03, KS03, KA03]. The same holds for the enhanced suffix array since each additional table can be constructed in $\mathcal{O}(n)$ time.

2.7 Rank and Select on a Static Bit Vector in Constant Time

The interest in a data structure that represents a bit vector and provides the operations rank and select is mainly motivated by the space-efficient simulation of tree traversals on bit vectors [Zak80, Jac89]. A linked data structure, like the tree, with n nodes needs space for $\mathcal{O}(n)$ pointers. Each of them needs $\mathcal{O}(\log n)$ bits of space, thus the total space needed just to store the pointers is $\mathcal{O}(n \log n)$ bits.

⁵The authors recognize that this implementation of the tables may affect the worst-case time complexities of the algorithms, whereas their first implementation, without this constraint, uses four times more space.

⁶Bottom-up traversal of the complete suffix tree, top-down traversal of a subtree, and traversal of the suffix tree using suffix links [AKO04].

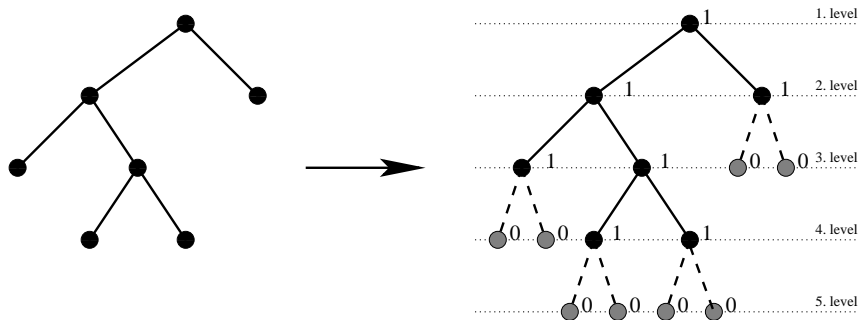


Figure 2.5: Assignment of bits to the nodes of the tree and its new leaves.

2.7.1 Zaks' Sequence

The idea is to store the structure of a binary⁷ tree in a bit vector, called *Zaks' Sequence*: Each node of the tree is marked with a 1. Then we append new leaves to each leaf and mark these new leaves with 0 (see Figure 2.5).

Now, we read the bits in level-wise (top-down) order, in each level, e.g., from left to right. For a tree with n nodes, we get a bit sequence of length $2n + 1$, the bit vector.

The resulting bit vector for the tree in the example (Figure 2.5) is “1111100001100000”. This is the concatenation of the level bit vectors “1” (root node), “11”, “1100”, “0011”, and “0000”.

Let the bit vector B be some text over the alphabet $\{0, 1\}$ representing a tree. To simulate the traversal of the tree using B , the operations *rank* and *select* can formally be defined as:

- $\text{rank}_b(B, i) := |\{j \mid B[j] = b \wedge j \in [1..i]\}|$
- $\text{select}_b(B, i) := \min\{j \mid \text{rank}_b(B, j) = i\}$

$\text{Rank}_b(B, i)$ tells how many characters b there are up to position i in B , where $b \in \{0, 1\}$. $\text{Select}_b(B, i)$ tells which position contains the i -th b .

Using some constant time representation of rank and select, the traversal from a node i to its parent or children can be performed in constant time:

- $\text{left}(i) = 2 \text{rank}(B, i)$
- $\text{right}(i) = 2 \text{rank}(B, i) + 1$
- $\text{parent}(i) = \text{select}(B, \lfloor i/2 \rfloor)$

⁷This can easily be generalized to k -ary trees.

We have represented the hierarchy of the tree using $2n + o(n)$ bits. Storing the data that are associated with the edges or vertices of the tree demands further $\mathcal{O}(n \log k)$ bits of space, where k is the number of possible different values (see Figure 2.6 for an example).

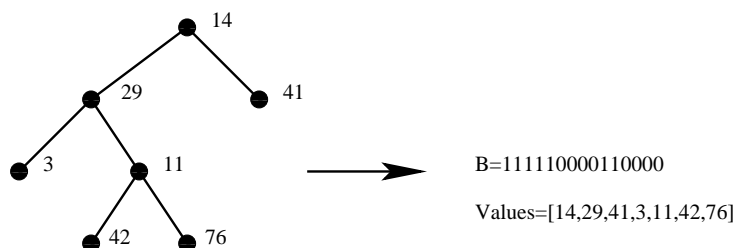


Figure 2.6: Representation of traditional tree (left) with B and array of values (right).

The problem of finding a data structure that represents B , allows constant time access to each of its bits⁸ and performs the operations rank and select in constant time is also called the *indexable dictionary problem* [RRR02].

The trivial solution for computing $\text{rank}_b(B, i)$ (the same holds for $\text{select}_b(B, i)$) given B in constant time is to store $\text{rank}_b(B, i)$ for each position of B in a preprocessing step. This would take $\mathcal{O}(n \log n)$ bits plus n bits for representing B , where $n = |B|$.

2.7.2 Static Bit Vector with Rank

We assume a *(Static) Bit Vector* B to be an abstract data structure that stores a sequence of n bits and supports the operations $\text{look-up}(i)$, $\text{rank}(i)$, and $\text{select}(i)$. Since for our purpose $\text{select}(i)$ is not needed, we will focus on $\text{rank}(i)$. Later we will consider the *Dynamic Bit Vector*, that has two additional operations, $\text{insert}(bit, i)$ and $\text{delete}(i)$, which allow to insert or delete a bit at position i . The next two sections introduce data structures for the representation of a sequence of bits that support constant time rank queries.

Static Bit Vector with $n + o(n)$ bits

For constant time $\text{rank}_b(B, i)$, the idea here is to store a dictionary of size $o(n)$ in addition to the bit vector B . This dictionary consists of three tables, *superblockrank*, *blockrank*, and *smallrank* [Jac88, Mun96, Cla96, NM06b].

Each table stores partial solutions. B is divided into superblocks and these are again divided into blocks. Absolute rank values for each superblock are stored in *superblockrank*, and relative rank values of the blocks within these superblocks are stored in

⁸Also called *membership query*, see e.g. [BM94].

blockrank. Choosing appropriate sizes for the superblocks and the blocks, the blocks are of size $\mathcal{O}(w)$ and thus can be read in constant time. Given table *smallrank*, local rank values of each position inside each possible block can be computed in constant time, too. Summing up over these partial solutions yields $\text{rank}(B, i) = \text{rank}_1(B, i)$. Notice that $\text{rank}_0(B, i) = i - \text{rank}_1(B, i)$ holds. A more detailed explanation considering the space consumption of each table follows:

Superblockrank Let $l = \lceil \log n \rceil^2$. Then store each l -th $\text{rank}(B, i)$ in the array *superblockrank*. For $n/l = n/\log^2 n$ entries⁹, with $\log n$ bits for each, the total space usage of *superblockrank* is $n/\log n$ bits.

Blockrank Let $t = \lceil \log n \rceil$. For every t -th position, store the number of bits between the actual position and the last predecessor position that is stored in the array *superblockrank*. For $n/t = n/\log n$ entries with $\log l = \log \lceil \log n \rceil^2$ for each entry, the total space usage of *blockrank* is $(n/\log n) \cdot \log \lceil \log n \rceil^2 = \mathcal{O}((n \log \log n)/\log n)$ bits.

Given the tables *superblockrank* and *blockrank* we can answer $\text{rank}(B, i)$ in $\mathcal{O}(\log n)$ time, that is $\text{rank}(B, i) = \text{superblockrank}[i/l] + \text{blockrank}[i/t] + \text{rank}_1(B, t \cdot (i/t), i)$, where $\text{rank}_b(B, i', i)$ computes the number of b -bits in the range $B[i', i]$.

Smallrank Instead of counting the number of bits from position $t \cdot (i/t)$ up to position i in a t -sized block, one could store the local $\text{rank}(B, t \cdot (i/t), i)$ values for each possible t -sized interval and each of the t positions in a table. Assuming a computer word size of $w = \Omega(\log n)$, the rank could then be computed in constant time. But for t of size $\log n$, we would have to store values for n possible configurations, which is too much. The solution, called the *four-Russians* technique [ADKz70], is to store rank values for bit vectors of size $t/2 = \log(n)/2$, having only $2^{\log(n)/2} = \sqrt{n}$ possible different bit vectors. Thus, for all \sqrt{n} bit vectors and each $t/2 = \log(n)/2$ positions we have to store rank values that need $\log(t) = \log(\log n)$ bits of space. The total space usage for the table *smallrank* is $O(\sqrt{n} \log(n) \log(\log n))$ bits.

To compute $\text{rank}_1(B, t \cdot (i/t), i)$ efficiently, we need to access table *smallrank* twice, once for the first half, bit vector c_i , and once for the second half, bit vector d_i , where $c_i d_i$ is the t -sized bit vector that contains position i .

$$\begin{aligned} \text{rank}_1(B, i) &= \text{superblockrank}[i/l] + \text{blockrank}[i/t] \\ &\quad + \text{smallrank}[c_i][\min\{i \bmod t, t/2 - 1\}] \\ &\quad + \text{smallrank}[d_i][\max\{i \bmod t - t/2, -1\}], \end{aligned}$$

where $\text{smallrank}[d_i][-1] = 0$.

⁹We assume all divisions x/y to yield $\lfloor x/y \rfloor$.

Static Bit Vector with $nH_0 + o(n)$ bits

A more space-efficient solution is given by Raman et al. [RRR02]. The tables *superblockrank* and *blockrank* are almost the same as in the section before. This time we choose $t = \lceil \log n/2 \rceil$, which doesn't change the asymptotic space usage of *blockrank*. The bit vector B is not directly stored. Instead, each block is represented by a pair (κ, r) , where κ denotes the class identifier with $0 \leq \kappa \leq t$ and r an index within class κ . B is represented by a sequence B' of these pairs. Table *smallrank* is replaced by a set of t tables $\{\text{smallrank}_i \mid i \in [1..t]\}$.

Representation B' of B We have a partition of B into blocks of size t . Each block belongs to a class κ that is determined by the number κ_i of bits that are set. Storing a class identifier takes $\log(t)$ bits. Each class represents $\binom{t}{\kappa_i}$ blocks, thus storing index r takes $\lceil \log \binom{t}{\kappa_i} \rceil$ bits. The class identifiers in B' amount to $\mathcal{O}(n/t \cdot \log t) = \mathcal{O}(n \log \log n / \log n)$ bits. The indexes in B' amount to

$$\sum_{i=1}^{\lceil n/t \rceil} \left\lceil \log \binom{t}{\kappa_i} \right\rceil$$

bits. It can be shown that this is bounded [NM06b]:

$$\begin{aligned} \sum_{i=1}^{\lceil n/t \rceil} \left\lceil \log \binom{t}{\kappa_i} \right\rceil &\leq \log \left(\prod_{i=1}^{\lceil n/t \rceil} \left\lceil \binom{t}{\kappa_i} \right\rceil \right) + n/t \leq \log \binom{n}{\kappa_1 + \dots + \kappa_{\lceil n/t \rceil}} + n/t \\ &= \log \binom{n}{k} + n/t \leq nH_0(B) + \mathcal{O}(n/\log n), \end{aligned}$$

where $k = \kappa_1 + \dots + \kappa_{\lceil n/t \rceil}$. Therefore, the total space usage of B' is $nH_0(B) + \mathcal{O}(n \log \log n / \log n)$.

Smallrank_i The *smallrank* tables store the local rank answers for each block. The space usage is the same as in the section before. We have to consider \sqrt{n} bit vectors and $t = \lceil \log n/2 \rceil$ positions. Each rank value takes $\log(t) = \log(\log(n)/2)$ bits. This accumulates to $\mathcal{O}(\sqrt{n} \log n \log \log n)$ bits of space.

2.8 Rank and Select on General Sequences

Similar to the bit vector we define an abstract data structure for general sequences. This data structure provides the operations $\text{look-up}(i)$, $\text{rank}(i)$, and $\text{select}(i)$ on a text t of length n over the alphabet Σ . The definition of generalized *rank* and *select* for texts with an alphabet size greater than two is straightforward:

- $\text{rank}_c(t, i) := |\{j \in [1..i] \mid t[j] = c\}|$
- $\text{select}_c(t, i) := \min\{j \mid \text{rank}_c(t, j) = i\}$

where $c \in \Sigma$.

For the general sequence with indels we further need the operations $\text{insert}(c, i)$ and $\text{delete}(i)$.

2.8.1 Linear Representation

A trivial constant time solution is to use bit vectors B_c for each $c \in \Sigma$, such that $B_c[i] = 1$ if and only if $t[i] = c$. Then $\text{rank}_c(t, i) = \text{rank}_1(B_c, i)$ and $\text{select}_c(t, j) = \text{select}_1(B_c, j)$ holds. Given the $nH_0 + o(n)$ -bit solution for bit vectors, the space needed for the bit vectors B_c is:

$$\begin{aligned}
 \sum_{c \in \Sigma} (nH_0(B_c) + o(n)) &= \sum_{c \in \Sigma} (n_c \log(n/n_c) + (n - n_c) \log(n/(n - n_c)) + o(n)) \\
 &= \sum_{c \in \Sigma} (n_c \log(n/n_c) + (n - n_c) \log(1 + n_c/(n - n_c)) + o(n)) \\
 &\leq \sum_{c \in \Sigma} (n_c \log(n/n_c) + (n - n_c)(n_c/(n - n_c)) + o(n)) \\
 &= \sum_{c \in \Sigma} (n_c \log(n/n_c) + \mathcal{O}(n_c) + o(n)) \\
 &= nH_0(t) + \mathcal{O}(n) + o(\sigma n)
 \end{aligned}$$

Moreover, Ferragina et al. [FMMN06] have shown that the $\mathcal{O}(n)$ term can be avoided.

2.8.2 Balanced Wavelet Tree

The *balanced wavelet tree* [GGV03] is a binary tree of height $h = \lceil \log(\sigma) \rceil$, in which all leaves are at depth h or $h - 1$. Each node x represents a subset Σ_x of size σ_x of alphabet Σ . The root represents all symbols in Σ , that is $\Sigma_{root} = \Sigma$. Each left child of a node x represents the first $\lceil \sigma_x/2 \rceil$ symbols of Σ_x , the right child of x represents the second half in Σ_x . Given this property, each leaf represents a single distinct alphabet symbol and each symbol is represented by one leaf.

We associate a bit vector B_x to each node x except the leaves (see Figure 2.7). If the corresponding leaf for symbol $t[i]$ belongs to the left subtree of x , $B_x[i]$ is set to 0, 1 otherwise. Note, that for general $\sigma \in \mathbb{N}$ it is not certain that one can always divide a set of symbols into two subsets of the same size. Thus, leaves may be located at different depths, h and $h - 1$. To compute select it will be necessary to know the

location of the leaves. These may be computed each time when needed in $\mathcal{O}(\log n)$ time by a top-down traversal in the tree or the pointers can be explicitly stored, using $\mathcal{O}(\sigma \log n)$ additional bits.

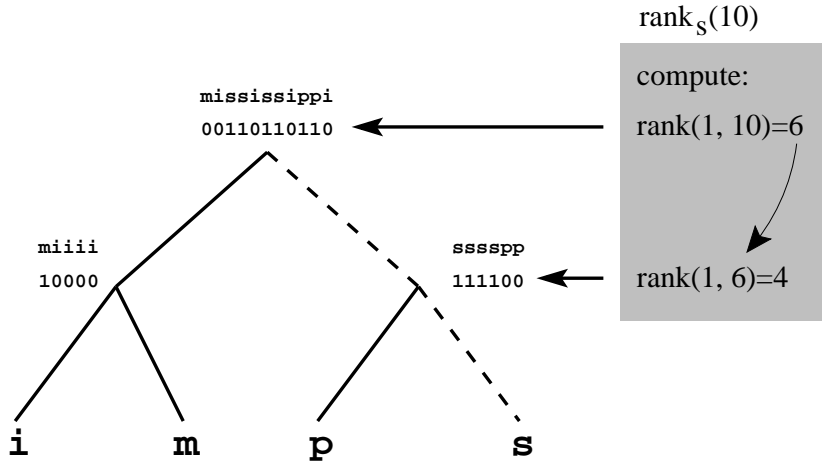


Figure 2.7: Computation of $\text{rank}_s(10)$ in the Wavelet Tree for “mississippi”.

Look-up To look-up the symbol at position i in the sequence, we start by performing a look-up of the bit at position i in the bit vector associated with the root. We get some bit b and compute $i \leftarrow \text{rank}(b, i)$. If $b = 0$, we go to the left child, otherwise to the right child. Given the new value for i , we perform look-up again like we did for the root and repeat the whole procedure until a leaf is reached. The symbol that is associated with the leaf is the result.

Rank To compute $\text{rank}_c(t, i)$, we walk from the root to the leaf that corresponds to c . E.g. assuming that the leaves are ordered according to the lexicographical weight of their symbol, the nodes on the path down to the leaf can be computed in constant time by a simple lexicographical comparison. For each node on the path down to the leaf, we compute $i \leftarrow \text{rank}(b, i)$, where $b = 0$ if the next child is a left child, $b = 1$ otherwise. When the leaf is reached, $i \leftarrow \text{rank}(b, i)$ gives $\text{rank}_c(t, i)$.

Select Computing $\text{select}_c(t, j)$ is a bottom-up approach. Start at the leaf l_c that corresponds to $t[j]$. The leaf can be found by a simple table look-up. If l_c is a left child, we compute $s \leftarrow \text{select}_0(B_x[j])$, $\text{select}_1(B_x[j])$ otherwise, where x is the parent of l_c . Then copy $x \leftarrow \text{parent}(x)$ and $j \leftarrow s$ and recompute $s \leftarrow \text{select}_0(B_x[j])$. Repeat this until the root is reached.

Each of these three operations take $\mathcal{O}(\log(\sigma))$ time, because one has to traverse the tree once and the computation at each node can be performed in constant time.

The interesting feature of the *wavelet tree* is its space usage. For example, using the $n + o(n)$ bits-implementation of bit vectors, each level of the tree uses $n + o(n)$ bits. Since there are $\log \sigma$ levels, overall space usage is $n \log \sigma + o(n \log \sigma)$. Using the $nH_0 + o(n)$ bits-implementation, overall space usage reduces to $nH_0(t) + o(n \log \sigma)$ bits [GGV03].

2.8.3 Huffman-shaped Wavelet Tree

Another alternative is to use a *Huffman-shaped Wavelet Tree* [MN05], that is the tree is formed according to the Huffman tree hierarchy.

Huffman encoding Huffman [Huf52] assigns a unique binary code to each symbol of the alphabet. The important feature is that the length of a code varies and depends on the frequency of its symbol. More common symbols are represented by shorter codes and less common symbols by longer codes. The average code length is H_0 instead of $\log \sigma$. Using this coding it is possible to represent a text of length n with at most $(H_0 + 1)n$ bits.

The algorithm to compute these codes works as follows: Given a forest of σ trees each consisting of one node. For each symbol there is one tree. Assigned to these trees is a probability, which is the relative frequency of the corresponding symbol. Then the two trees with the smallest relative frequency are merged. The resulting tree has a probability assigned, that is the sum of the probabilities of the two subtrees. This merging of trees is repeated until there is only one tree left, whose assigned probability is 1. The merging process creates a binary tree, where each leaf represents one symbol, internal nodes represent subsets of the alphabet and the root node represents the whole alphabet. Labeling each edge pointing to a left child with a “0”, and edges pointing to a right child with a “1”. The concatenated edge labels on a path from the root to a leaf determine the code for the corresponding symbol.

The average length of a root-to-leaf path is at most $H_0 + 1$ [Huf52]. Gallager [Gal78, BMW94] has proven a tighter bound: The average code length per symbol is at most $H + p + 1 - \log e + \log \log e \approx H + p + 0.086$, where H is the entropy of the model and p the probability of the most likely symbol.

A further property of the codes is that they are *prefix-free*, no code word is a prefix of another code word. This can easily be shown via the binary tree hierarchy: The last edge of a path corresponding to some code leads to a single leaf and thus will not share this edge with some other code. An example where this property is necessary is an incoming encoded data stream, where one wants to know after each bit read whether the current code word is complete or not.

Wavelet tree Using this shape for the wavelet tree is straightforward. The only difference is that the lexicographic order of characters will not necessarily correspond to the order of their appearance in the tree. A lexicographical comparison is not possible and thus the algorithm for rank lacks information about the correctly chosen child, when performing the top-down walk. An additional data structure is required to know which path leads to the correct character. For example, using the Huffman tree, one can create a code table that contains the binary Huffman code for each character. The bits of each code give the path from the root down to the corresponding leaf. Of course, one could also avoid such a structure by using the pointers that refer directly to the correct leaf in the wavelet tree for each character, as they are used for select. Then one would walk upwards to the root and would know the path needed for rank. This, of course, is a bit slower than using the code table since the path would have to be traversed twice. Concerning the time complexity, the average running time for a vertical traversal of the tree is now $\mathcal{O}(H_0)$, instead of $\mathcal{O}(\log \sigma)$ as in the balanced wavelet tree.

2.8.4 k -ary Wavelet Tree

Another idea is to use the *k-ary wavelet tree* [FMMN06], where $k \leq \sigma$. Here we consider a k -ary balanced tree, where the set of symbols associated to a node n is divided into k subsets, one subset for each child of n . Both queries, rank and select, take $\mathcal{O}(\log_k \sigma)$ time.

2.9 Burrows-Wheeler Transformation

The *Burrows-Wheeler* transformation (BWT) [BW94] permutes a text in such a way that the permutation is reversible. To construct the BWT, $\text{bwt}(t)$, for some text t of length n , think of a conceptual matrix M (Figure 2.8), where the rows consist of all n distinct cyclic permutations of t . We sort the rows in lexicographic order. Then the symbols in the last column L , read top-down, denote the BWT of t .

A direct implementation of the conceptual matrix is quite inefficient. Instead of using rows, one can use pointers that tell at which position in t the row starts. A common approach to construct the $\text{bwt}(t)$ is to construct the suffix array SA for t first. Then $\text{bwt}(t)[i] := t[\text{SA}^*[i]]$, where $\text{SA}^*[i] := \text{SA}[i] - 1$ if $\text{SA}^*[i] \neq 0$, else $\text{SA}^*[i] := n$. But this only holds if the last character of t is a special end character, e.g. “\$”, which is lexicographically smaller than all the other characters in Σ and appears nowhere else in the text. The reason for this is that sorting of suffixes and sorting of cyclic permutations doesn’t always give the same order. When comparing suffixes, it is possible that the shorter suffix is identical to some prefix of the longer suffix. Then the shorter suffix is considered to be lexicographically smaller. Cyclic rotations always

	F	L
12:	\$	mississippi
11:	i	\$mississipp
8:	ippi	\$mississ
5:	issippi	\$miss
2:	ississippi	\$m
1:	mississippi	\$
10:	pi	\$mississip
9:	ppi	\$mississi
7:	sippi	\$missis
4:	sissippi	\$mis
6:	ssippi	\$missi
3:	ssissippi	\$mi

Figure 2.8: Conceptual BWT matrix $M(t)$, where $t = \text{“mississippi$”}$.

have the same length. Taking the two rotations that correspond to these suffixes, the permutation that corresponds to the shorter suffix may have a character at the position where the shorter suffix ends, which is lexicographically greater than the character at the same position in the permutation that corresponds to the longer suffix.

2.9.1 Reverse Transformation

The most important feature of this transformation is that it is reversible. For this, we need to know which row of the conceptual matrix contains the original text. This can be done by using a pointer p . Another way is to append the special character “\$” to t . Then “\$” would indicate the position of t in the matrix. We don’t use “\$”, thus we consider the number of occurrences of “\$” to be zero in this example.

Compute the array $C[1, \sigma]$, where $C[c_s]$ is the number of occurrences of characters $\{\$, c_1, \dots, c_{s-1}\}$ in t . $C[c] + 1$ is the position of the first occurrence of c in the first column F of the matrix. We define $\text{map}_c(L, i) := C[c] + \text{rank}_c(L, i)$ and the so called *LF-mapping*, $\text{LF}(L, i) := \text{map}_{L[i]}(L, i)$ [FM00]. Using p and the LF-mapping, it is possible to reconstruct t backwards:

Algorithm 1 Reverse Burrows-Wheeler transformation

```

 $i \leftarrow p$ 
for  $j \leftarrow n$  to 1 do
   $t[j] \leftarrow L[i]$ 
   $i \leftarrow \text{LF}(L, i)$ 
end for
return  $t$ 

```

For each position i in the suffix array SA, the LF-mapping returns the position $\text{LF}(i)$ of the suffix $\text{SA}[\text{LF}(i)]$ that is one symbol shorter than $\text{SA}[i]$. That this holds can be shown by the definition of the LF-mapping: Given some position i , the $\text{C}[\text{L}[i]]$ summand of the LF-mapping points to the suffix that is located in the suffix array one position before the first suffix that starts with $\text{L}[i]$. This is because the suffixes in the suffix array are sorted lexicographically and $\text{C}[\text{L}[i]]$ is the number of characters in L that are smaller than $\text{L}[i]$. All suffixes that start with an $\text{L}[i]$ define the $\text{L}[i]$ -context in the suffix array. These suffixes are sorted within their context depending on characters that follow their first characters. The suffixes that are obtained by removing the first character $\text{L}[i]$ have the same relative order in the suffix array as those in the $\text{L}[i]$ -context. Furthermore, the corresponding rows in the BWT all share the same last character $\text{L}[i]$. Since both sets of suffixes, $\text{L}[i]$ -context and the corresponding shorter suffixes, share the same relative lexicographic order, rank applied to position i of some last character $\text{L}[i]$ in L gives the relative position of the corresponding longer suffix within its $\text{L}[i]$ -context.

Note the similarity between the LF-mapping and suffix links at the leaves in a suffix tree. In the suffix tree, the suffix link for the last suffix of length one will always point to the root node instead of pointing to the node with path label t . LF-mapping for some bwt always defines a single-cycle permutation $(i \text{ LF}^1(i) \dots \text{LF}^{n-1}(i))$, where LF^n denotes the n -th iterate of LF. The cycle that starts with p constructs t backwards, $t = \text{LF}^{n-1}(p) \dots \text{LF}^1(p)p$.

2.9.2 Compression

The main purpose of the Burrows-Wheeler transformation is to use it in data compression. The Burrows-Wheeler transformation consists of the last column of the conceptual matrix, thus its characters precede those of the first column. Since the rows are sorted lexicographically, characters are grouped together in the Burrows-Wheeler transformation according to their context. When probabilities for characters depend on their context, which holds for example for natural languages or DNA, it is likely that these groups contain longer repeats of characters. Using for example the *move-to-front* technique [BSTW86, Eli87] in addition higher compression rates can be achieved when compressed with an entropy encoder.

2.10 Backwards Search

Another application of the Burrows-Wheeler transformation, using the *map* function, is the *backwards search* [FM00]. Given $L = \text{bwt}(t)$ of some text t , it is possible to count all occurrences of a pattern p in t in $\mathcal{O}(m)$ steps if function *map* uses a constant time implementation of rank.

Algorithm 2 BackwardsSearchCount($p[1..m], L[1..n]$)

```
 $j \leftarrow m$ 
 $sp \leftarrow 1$ 
 $ep \leftarrow n$ 
while ( $sp \leq ep$ ) and ( $j \geq 1$ ) do
   $s \leftarrow p[j]$ 
   $sp \leftarrow \text{map}_s(L, sp - 1) + 1$ 
   $ep \leftarrow \text{map}_s(L, ep)$ 
   $j \leftarrow j - 1$ 
end while
if  $ep < sp$  then
  return Found no occurrences.
else
  return Found  $ep - sp + 1$  occurrences.
end if
```

The *backwards search* provides the bounds of an interval in the suffix array that contains the position of the first character of each occurrence of p in t . This algorithm only needs L and the *rank* dictionary, where L may be compressed depending on the implementation of *rank*. The main advantage of such an index using the *backwards search* is that it doesn't need the suffix array anymore, thus it uses less space. However, the operation *locate* is not directly supported.

Indeed, the LF-Mapping also allows to reconstruct the suffix array, similar to the reverse *Burrows-Wheeler* transformation. Then, one has to start with the pointer p and construct the whole suffix array up to the interval containing the matches. Of course, this is quite inefficient. A better solution that doesn't construct the whole suffix array is used in the *FM-index* (Section 2.12). It uses the sampled suffix array, which we explain next.

2.11 Sampled Suffix Array

The backwards search does not directly give the location of a pattern. The interval that has been found by the backwards search has the same interval bounds as the interval in the suffix array that contains the locations of the search pattern. Since storing the whole suffix array takes too much space, a common technique is to store only a subset of the suffix array values, also called *sampling*, and to compute successive intermediate values when they are needed. A sample rate s of e.g. $s = 1/d = 1/\log n$ means that the values of all suffix array positions are stored that are equal to some multiple of $\log n$. These values are pointers to characters of the text pointing to the first position and then each d -th position of the text. For a static BWT the sampled

values are usually stored in an array where the order of the samples is the same as in the suffix array.

Given some position i within an interval of the backwards search in the BWT, we would like to know the position in the text where the match occurs. We first assume that position i corresponds to some suffix array position that has been sampled. This is indicated by an additional bit vector I with rank support that has the same length as the BWT. Each bit in the bit vector indicates whether the suffix array position i has been sampled or not. We know that position i in the BWT has been sampled and we get $I[i] = 1$. Now we need to find the correct position in the array of sampled values. Since the sampled values have the same order as they occur in the suffix array, $\text{rank}(i)$ applied to I gives the position of the sampled value and we know the position of the match within the text. Assuming i does not correspond to some sampled position, we get $I[i] = 0$. We then start the LF-mapping and traverse the text backwards until we find some position i' with $I[i'] = 1$. Since there is a sample for each d -th position in the text, the LF-mapping is performed at most $d - 1$ times. The number of LF-mappings needed to find the next sampled position plus the value of that sample is the location of the match. For example, given some constant time algorithm for rank and a sampled suffix array with $d = \log n$, the time to locate all occurrences of a pattern p of length m is $\mathcal{O}(m + \text{occ} \log n)$, where occ is the number of occurrences of p in t . Additional space needed for this sampling is $nH_0(I) + o(n)$ (see Section 2.7.2) bits for the bit vector I and $(n/d) \log n = n$ bytes for the sampled suffix array. Of course, adjusting d makes other time/space tradeoffs possible.

2.12 FM-Index

The *FM-index* of Ferragina and Manzini [FM00] is the first index using the backwards search. To be able to locate, they mark each $\log^{1+\epsilon}$ -th position of t , where $\epsilon > 0$, and store each corresponding entry of the suffix array in a new array. Using $\mathcal{O}(n/\log^\epsilon n)$ bits of space, their technique allows to locate occurrences in $\mathcal{O}(\text{occ} \log^{1+\epsilon} n)$ time, where occ is the number of occurrences of p in t . For a detailed description of the algorithm see [FM00, NM06b].

The index is based on a linearized representation (see Section 2.8.1) of the BWT using an implementation of *rank* similar to the techniques mentioned in Section 2.7.2. With $5nH_k + o(n \log \sigma)$ bits of space usage, this index is the first to encode the index size with respect to the high-order empirical entropy [GGV03]. This is achieved by compressing $\text{bwt}(t)$ first using a move-to-front coder, then run-length encoding and finally a variable-length prefix code which is similar to the Elias γ -encoding [Eli75].

In this work we will use the term FM-index for any index of some t that is based on $\text{bwt}(t)$ and the backwards search.

3 Direct Construction of the Burrows-Wheeler Transform

An application of the Dynamic Sequence with Indels is the direct construction of the Burrows-Wheeler Transform. The term “direct” is a description for a method where the solution $f(ax)$ for some function f and some sequence ax , where a is a symbol, is computed using the solution $f(x)$. Practically this means that we can construct the BWT while reading the text backwards. This is similar to the online construction, where the data is read in forward direction. We will first show how the direct construction for a single text works, then how this can be extended to a collection of texts by introducing our concept of the pseudo BWT.

3.1 Single Text

For the BWT-construction of a single text, we will first only consider the case when the text ends with a special character “\$” that occurs nowhere else. Assume we have the BWT for the suffix “ississippi\$” which is “ipss\$piissii”. We can create the BWT for the next larger suffix “mississippi\$” by two modifications on the first BWT. The new character (here “m”) gets inserted at the previous position of “\$”, say i , and “\$” itself is inserted at position $LF(i)$ [CHL04].

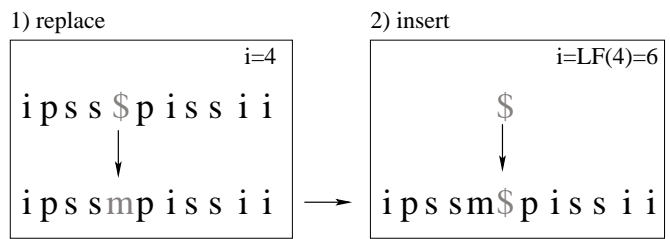


Figure 3.1: Transformation of $bwt("ississippi\$")$ into $bwt("mississippi\$")$

In Figure 3.2 are all suffixes, sorted by length, of “mississippi\$” listed (left) together with their corresponding BWT (right). Notice that there are always two differences between two consecutive BWTs.

i	$t[n-i, n]$:	$\text{bwt}(t[n-i, n])$
0	\$:	\$
1	i\$:	i\$
2	pi\$:	ip\$
3	ppi\$:	ipp\$
4	ippi\$:	ip\$pi
5	sippi\$:	ipspi\$
6	ssippi\$:	ipspis\$
7	issippi\$:	ips\$pi\$
8	sissippi\$:	ipsspi\$
9	ssissippi\$:	ipsspi\$
10	ississippi\$:	ipss\$pi\$
11	mississippi\$:	ipssm\$pi\$

Figure 3.2: Suffixes of “mississippis\$” (left) and the corresponding BWT(right).

The BWT can be seen as a kind of compressed suffix array as it consists only of characters instead of pointers. For some text t , the i 'th character c in the $\text{bwt}(t)$ represents the i 'th suffix in the suffix array $sa(t)$. Then c is always the character that precedes the corresponding suffix in the original text. Furthermore, notice the special case of the longest suffix, where the preceding character c is the last character of the text. In our example this is always the special character “\$”. Assume we have some text t of length n and its $\text{bwt}(t)$. If we want to construct the BWT for $t' = at$, where a is some character, we have to modify $\text{bwt}(t)$. The suffix t of t' was represented by “\$” because this was the preceding character. Now, after reading the new character a , a is the preceding character of t . This is why we have to replace the old “\$” with the new character a . The last thing to do is to insert the new suffix t' into the BWT.

It is not quite obvious that the LF-mapping is able to give us the correct insertion position for the new suffix. In general, we expect the LF-mapping to give us the position of the character that represents the previous suffix. But this suffix is not inserted yet. After replacing “\$” with the new character, the current sequence is no longer a proper BWT. Assuming the last suffix has already been inserted we would have a complete BWT, $\text{bwt}(t')$, and LF-mapping would correctly point to this last suffix, represented by “\$”. Remember that LF-mapping $\text{LF}(i)$ consists of the sum of $C[L[i]]$ and $\text{rank}_{L[i]}(i)$. In which way does the “\$” have influence on the LF-mapping? We only have to consider the case $L[i] = c \neq \text{“$”}$. “\$” does have no impact on $\text{rank}_c(i)$. But, if “\$” exists, it always contributes 1 to $C[L[i]]$ because it is lexicographically smaller than c . The idea is to simulate the sequence to be a correct $\text{bwt}(t')$. We compensate for the missing “\$” by adding 1 to $\text{LF}(i)$. This $\text{LF}(i) + 1$ points to the correct insertion point for “\$” then.

One can directly use this technique on a dynamic sequence to get a direct construction algorithm for the BWT of t . We begin with a BWT of length 0 and read t backwards.

Then each character is inserted as above by LF-mapping. In fact, it is not necessary to insert “\$” before the text is completely read. It is enough to insert “\$” once after the last character is read. When we apply the LF-mapping, “\$” has been removed and not yet inserted. Instead of inserting “\$” at position $\text{LF}(i)$ and then replacing it with the new character, we can also insert the new character immediately at position $\text{LF}(i)$ without ever inserting “\$”. Only after all characters are read we insert “\$”. Of course, the intermediate sequences $\text{bwt}'(t)$ are not proper BWTs, but they can always be converted into a proper one by inserting “\$”.

3.2 Collection of Texts and the Pseudo BWT

We introduce the concept of the Pseudo BWT that allows to maintain a set of texts with less effort in the management of the collection compared to the BWT.

The idea of inserting several texts into the BWT is related to the idea of inserting the suffixes of all texts into one suffix array. The most obvious way to construct a BWT for a collection of z texts is to concatenate the texts and construct the BWT for this new text. Using a single special end character for the whole text guarantees that the starting point for the reverse construction always is at position 1. Moreover, one could use unique special end characters $\$j$, where $\$j \notin \Sigma$, for each text t_j , that are lexicographically smaller than the other characters in Σ and pairwise lexicographically different. Then the starting position for each text t_j would directly correspond to the lexicographical order of its end character $\$j$ among the other end characters.

Introducing a new special character for each new text, especially when there is a large number of texts, is not feasible because this increases the size of the alphabet. The usage of the same special end character $\$$ for each text at least guarantees the starting points to be somewhere within the first z positions of the BWT. We know that the starting points must directly depend on the lexicographical order of the texts among each other.

The main advantage of using the special character is that, when a new text is inserted, the starting positions of the texts that are already in the BWT can change by at most one position. In contrast, using no special end character, one would have to keep track of each starting position within the BWT each time a single character is inserted into the BWT, which happens $|t_j|$ times when inserting text t_j .

The problem of using only one special end character, that also occurs somewhere else in the text, is that the direct construction algorithm via LF-mapping as described above is not able to always produce a correct BWT (see Figure 3.3).

Since both texts in Figure 3.3 share the suffix “\$”, the insertion of the new character “a” can modify the current lexicographic order. But the direct construction based on the LF-mapping does not move rotations to their correct lexicographic position.

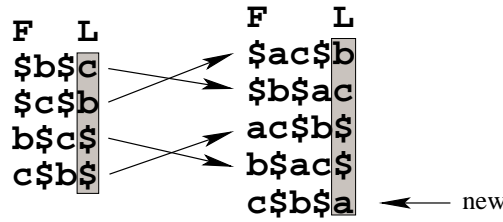


Figure 3.3: Difference between $\text{bwt}(\text{"c$b$"})$ (left) and $\text{bwt}(\text{"ac$b$"})$ (right).

This is the reason why we need the special end character for the direct construction of the BWT of a single text. Although the result of the direct construction is not guaranteed to be a correct BWT, we will show that the LF-mapping on this *pseudo BWT* (PBWT) works and can be used the same way as a correct BWT.

Here is an example of the direct construction with 5 texts (“dddzx\$”, “xz\$”, “cxz\$”, “dxz\$”, and “dxz\$”) where a “b” is appended to the front of the text “xz\$”. Some rows change their position within the conceptual matrix:

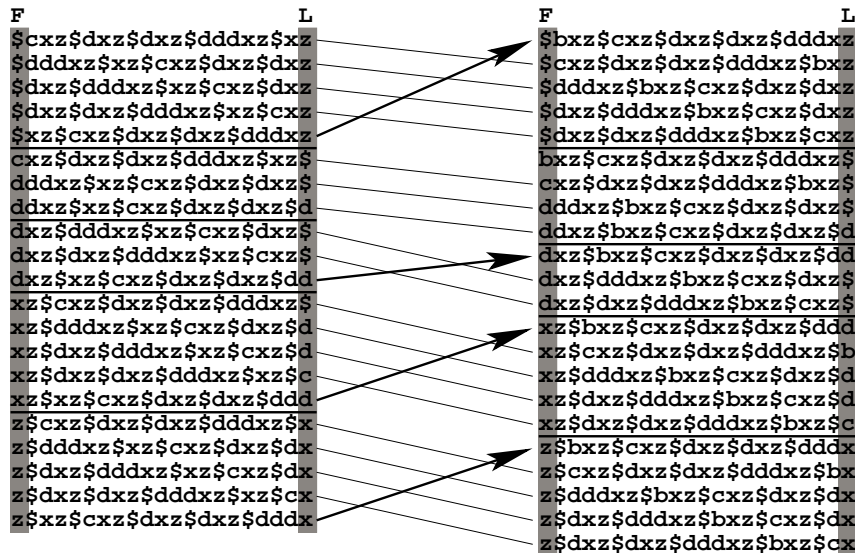


Figure 3.4: Differences between $M(\text{"dddzx$xzcxzdxzdxz"})$ and $M(\text{"dddzx$bzxcxzdxzdxz"})$.

A row in a matrix can change its position when several texts share a suffix u , a prefix of the row is equal to a suffix v of u , and the insertion point of the new character is located directly after this prefix. Then another row must exist that shares the same prefix v . Since both (or more) rows share this prefix, the important characters for the lexicographic order among the rows are now those behind the prefixes whereas the newly inserted character can change the rows’ lexicographic order among the rows

with same prefix v . For the BWT it means that a character can change its position within its v -context.

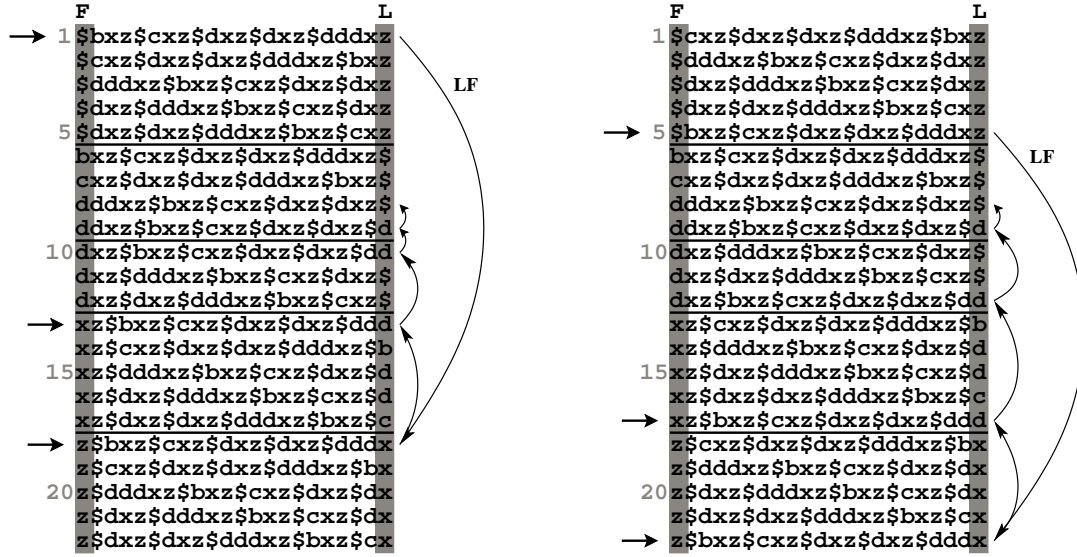


Figure 3.5: Reconstruction of “dddzx\$” using correct BWT (left) and pseudo BWT (right).

Assume we have a row in a proper BWT at position i with prefix v . We denote the row’s last character c . The current context of v is the last column of the submatrix that consists of all rows starting with v . It contains p c ’s. We know that the next context cv , to which $LF(i)$ will point, consists of p rows. Due to the rank summand in the LF-mapping we know that LF-mapping applied to the j -th row in v -context that end with a c will point to the j -th row in the cv -context.

Assume the shared suffix u is of maximal length k . Then for each suffix of u , for example v , there exists an interval of rows whose prefix equals v . Iterative LF-mapping started with any row i whose first character equals “\$” and its suffix equals $u[1, k - 1]$ will consecutively visit the rows $LF^q(i)$, each within the v_q -context, where v_q denotes a suffix of u of length $q \leq k$. Thus, a row can move to another position within its context, as long as all the other rows that belong to u also move within their context to the position, which ensures that the rank part of LF-mapping in the chain of contexts is correct. After insertion of a new character it is possible to keep rows at their position, even when the lexicographic order is disturbed, since the LF-mapping remains correct.

Remember that LF-mapping on a single text is a single-cycle permutation (Section 3.1). For a BWT with z texts and end character “\$”, the LF-mapping is a z -cycle permutation because the LF applied to the i ’th “\$” always points to i , and thus closes the cycle that corresponds to the text at position i . If the lexicographic order is not

guaranteed, we have a PBWT and the cycles are not always closed. LF applied to the end of some text t_i can point to the beginning of some other text t_j . Thus we have a z' -cycle permutation, where $1 \leq z' \leq z$. This deviation from the proper BWT is no problem for our application since we always reconstruct only a single text with a known starting point and don't need to care which path the LF-mapping follows after this text.

Furthermore, to choose an insertion point for a new text which respects the lexicographic order of the other texts can be quite time-consuming, e.g. worst-case $\mathcal{O}(n \log n \log H_0)$ for the implementation used in this work. It would need a backwards search to find the correct insertion point of the new text.

Counting within the PBWT is the same as in the BWT (Section 2.10). The size of the search interval defines the number of matches. In the PBWT the order of rows in the current search interval may be different from the order of the same interval in the BWT, but since all rows within the interval share the same prefix v and no other row outside of the interval can have this prefix v , backwards search on the PBWT always gives us a correct interval.

Asymptotics The asymptotics depend on the underlying data structures used to store the PBWT and to compute the LF-mapping. For the asymptotics of our implementation see Section 4.3.

4 Implementation of a Dynamic FM-Index

For the implementation of the dynamic FM-index we begin with the implementation of the Dynamic Bit Vector with Indels. This is the underlying data structure of the Dynamic Sequence with Indels which essentially consists of a wavelet tree. The last step consists of adding structures that are needed for counting, locating, and handling of the collection of texts.

4.1 Dynamic Bit Vector with Indels

A *Dynamic Bit Vector with Indels* is an abstract data structure that stores a sequence of n bits and supports the operations $\text{look-up}(i)$, $\text{rank}(i)$, $\text{select}(i)$, $\text{insert}(\text{bit}, i)$, and $\text{delete}(i)$.

4.1.1 Data Structure

The implementation is mainly based on the data structure COUNT_c [CHL04] and the bit vector described in [MN06]. It uses the standard red-black tree [Bay72, GS78, CLRS01] and achieves $\mathcal{O}(\log n)$ time for the operations. It uses $\mathcal{O}(n)$ bits, where n is the length of the bit sequence. The main idea is to split the sequence of bits into shorter blocks, which are stored at the nodes of the balanced tree. This is a slight difference to [MN06], where the blocks are stored only at the leaves. Further information is stored at the nodes to compute rank and select in $\mathcal{O}(\log n)$ time.

More precisely, each node v stores one block $\text{block}(v)$ of $(\log n)/2$ up to $2 \log n$ bits of the original sequence. Although $\text{block}(v)$ may contain a variable number of bits, space usage of a block is always $2 \log n$ bits to allow insertion of bits in constant time under the assumption $\log n \in \Theta(w)$. Concatenating the bits of all blocks collected by an in-order traversal of the tree thus reconstructs the original sequence.

Each node v of the tree contains four counters: p_sub gives the number of bits that are located within the subtree rooted at v , p_loc gives the number of bits that are stored in $\text{block}(v)$, r_sub gives the number of bits that are set within this subtree and r_loc gives the number of bits that are set in $\text{block}(v)$.

Instead of using a null-pointer for children that do not exist, we point to a single generic NIL node (Section 2.4) where all the counters are set to 0. This guarantees that the algorithm safely skips non-existing children.

4.1.2 Algorithms

Look-up To find the node that contains the i -th bit, we do a top-down walk beginning at the root down to the node containing position i . At each node v we check whether position i is located in the left subtree, in the current node or the right subtree. Since each subtree represents only a substring of the bit vector, finding an absolute position i corresponds to finding some relative position j within this subtree. The top-down walk starts with $j \leftarrow i$ and node $v \leftarrow \text{root}$:

If counter $\text{p_sub}(\text{left}(v))$ of the left child of v is bigger than j , position i has to be located in this subtree, v is set to the left child of the current v and the loop begins again at the new v . If this is not the case, we set $j \leftarrow j - \text{p_sub}(\text{left}(v))$ and check whether j is smaller than $\text{p_loc}(v)$. If this is the case, bit at position i is located at the current node. Since $\log n \in \Theta(w)$, this last look-up $\text{block}(v)[j]$ can be performed in $\mathcal{O}(1)$ time, the algorithm terminates and returns the bit at position i . If i is also not located in v then it must be located in the right subtree. We have to adapt $j \leftarrow j - \text{p_loc}(v)$ and repeat the procedure with $v \leftarrow \text{right}(v)$.

Rank To perform $\text{rank}(i)$ we walk from the root to the node that contains the i -th bit the same way as before. In addition to this procedure, every time when we choose the right subtree $\text{right}(v)$ for the next loop, the rank value for the “missed” left subtree $\text{r_sub}(\text{left}(v))$ and the rank value of v , $\text{r_loc}(v)$, are summed up. When the node is reached that contains position i , the rank value of the last “missed” left subtree is also added to the collected rank value. The last step is to determine the rank value of position j within its block. This value can be retrieved in $\mathcal{O}(w/|\text{byte}|) = \mathcal{O}(\log n)$ time by using a small table which stores rank results for all possible bit vectors of length $|\text{byte}|$, where $|\text{byte}|$ is the number of bits a byte consists of. The sum of these partial rank answers is the result of the algorithm.

Select The algorithm for computing $\text{select}(i)$ is in principle the same as for the rank operation. Instead of using the p -counters to decide which subtree to take next, the corresponding r -counters are used. Instead of summing up rank values, corresponding position values are summed up. If the node is found that contains the i -th set bit, a last query is needed to find the exact position of the j -th set bit. This is done in $\mathcal{O}(\log n)$ time. The result of this query is added to the collected position values and this final sum is the result of the algorithm.

Insert/delete The position to insert or delete a bit can be found in the same manner as in look-up. A right-shift of the bits right from the actual insertion point, including the bit at the insertion point, allows to insert the bit at this position. When a node contains $2 \log n$ bits after insertion, it will be split into two nodes both containing one half of the bits. When a bit is deleted, the bits right of the deleted bit are shifted one position to the left. If after deletion the node contains $(\log n)/2$ or less bits, we try to merge it with its predecessor or successor. This is only done if the merging produced a node that contained less than $2 \log n$ bits. After splitting and merging we have to rebalance the tree by restoring the red-black tree property and to update the counters `p_sub` and `r_sub` on the path from inserted nodes and parents of deleted nodes up to the root.

4.1.3 Asymptotics

For our implementation of the dynamic bit vector, we chose using a constant size for the blocks at the nodes which are used for storing the bits. This avoids time and space consuming adaptations (two techniques are explained in [MN06], Section 4.4) of the index whenever n changes. To make clear that this is a constant, we will call this parameter `LOGN`, which can be changed to other values (see Section 4.3.4) allowing different time-space tradeoffs. The default value of this parameter is `LOGN = 64`.

All the operations need $\mathcal{O}(\log n)$ time for the traversal of the tree because the red-black tree ensures the height of the tree to stay $\mathcal{O}(\log n)$. The operations within the last node take $\mathcal{O}(\log n)$ (rank and select) and $\mathcal{O}(1)$ time (look-up, insert, and delete), respectively. Rebalancing of the tree after insertion or deletion can be performed in $\mathcal{O}(1)$ time. Thus, $\mathcal{O}(\log n)$ is the running time for all of the operations. Now we take a look at the space usage (on a 32-bit machine and `LOGN = 64`): For a sequence of n bits, there are at most $n/(\text{LOGN}/2) = n/32$ and at least $n/(2 \cdot \text{LOGN}) = n/128$ nodes. Each node consists of 5 pointers (including a pointer to the *VTable* [Sta05]), 4 counters, an enum structure and a block with reserved space ($2 \cdot \text{LOGN} = 128$ bits = 16 bytes in this example) for the actual bits. Thus, the space usage for each node is 56 bytes (448 bits) on a 32-bit machine. The total space usage for a sequence of n bits is $14n + \mathcal{O}(1)$ bits in the worst case. In contrast, considering the best case where the whole capacity of the blocks is used, the total space usage for a sequence of n bits is $3.5n + \mathcal{O}(1)$ bits.

4.2 Dynamic Sequence with Indels

A dynamic sequence can be implemented by using the same generalization methods as described for the static case. The only difference is that the underlying bit vectors are dynamic. Insertion and deletion in this data structure are easy to implement. We

can use a modified version of the look-up algorithm where the corresponding bits at each bit vector are inserted or deleted on the way down to the leaf of the wavelet tree.

Since the paths inside the wavelet tree from the root to the leaves correspond to the Huffman codes and thus the path lengths are on average at most $H_0 + 1$, each of the n symbols in t is coded by at most $\mathcal{O}(H_0 + 1)$ bits on average. We have $\mathcal{O}(H_0)$ instead of H_0 because the underlying bit vector needs $\mathcal{O}(1)$ bits to represent one bit. Using the combination of Huffman-shaped wavelet tree and dynamic bit vector as described above we achieve $\mathcal{O}(n(H_0 + 1))$ bits of space usage. For the character operations, we have an average running time of $\mathcal{O}((H_0 + 1) \log n)$, where the $\mathcal{O}(\log n)$ comes from the operations on the bit vectors and $\mathcal{O}(H_0 + 1)$ from the average time to traverse a path (see Table 4.1). Time to construct the Dynamic Sequence is $\mathcal{O}((H_0 + 1)n \log(n))$, where n is the length of sequence.

CHARACTER OPERATION	TIME
Insert	$\mathcal{O}((H_0 + 1) \log(n))$
Retrieve	$\mathcal{O}((H_0 + 1) \log(n))$
Delete	$\mathcal{O}((H_0 + 1) \log(n))$
Rank	$\mathcal{O}((H_0 + 1) \log(n))$

Table 4.1: Time usage of character operations in the Dynamic Sequence with Indels.

Shape of the wavelet tree To achieve an encoding of $\mathcal{O}(H_0)$ bits on average for each character, it is necessary that the shape of the wavelet tree has a Huffman shape, which depends on the character distribution of the text. The problem of dynamic sequence is that the insertion or deletion of a character may change the character distribution of the text in such a way that the current shape of the wavelet tree is not a Huffman shape anymore. An adjustment of the wavelet tree to the correct shape would be quite time consuming since all n encoded characters in the dynamic sequence would have to be adapted. A practical solution to this problem, especially when working on a collection of texts, is the assumption that all texts that are inserted share nearly the same character distribution. For example, when we know that we insert only English texts, we can expect that they have a similar character distribution. This means that we need to know the character distribution of the texts in advance when constructing the wavelet tree.

4.3 Dynamic FM-Index for a Collection of Texts

The purpose of this self-index which uses the basic techniques of the original static FM-index is threefold: First, increased applicability by allowing to index a collection of texts instead of a single text. Second, the index can be modified (insertion and deletion

of texts) at runtime without reconstructing the whole index. This can improve speed especially when the text to be deleted or inserted is significantly smaller than all the other texts of the collection together. Third, improved space usage by compressing the collection of texts close to 0-th order entropy.

Let $T = \{t_1, \dots, t_z\}$ be a collection of z texts, where each text ends with a special character $\$$ that does not occur anywhere else in the text and is lexicographically smaller than all the other characters. $n = \sum_{1 \leq j \leq z} (|t_j|)$ denotes the total length of all texts in the collection.

The core data structure of this index consists of the Dynamic Sequence with Indels (Section 4.2) based on the Huffman-shaped wavelet tree (Section 2.8.3) and the Dynamic Bit Vectors (Section 4.1) at the nodes of the wavelet tree. Further data structures are HANDLE, POS, and DSSA which are needed for the management of the collection and to report where matches are located.

The character distribution which is needed for the construction of the Huffman-shaped wavelet tree has to be read from a file that contains a text with a representative character distribution. It has to be ensured that each character which can occur later on is found at least once in this text, otherwise the character is not represented in the wavelet tree shape and can not be encoded. Note that the separation character “\$” has to be encoded, too. This is why an estimation of the average number of texts has to be given in advance. This number and a sample text have to be given when constructing the index. Note that here the term H_0 refers to the estimated character distribution including the special character and can thus deviate from $H_0(t_1 \dots t_z)$.

The separation character “\$” in this implementation is represented by an ASCII NUL character. To avoid insertion of characters within the first z positions of the PBWT, it is not allowed to use this character somewhere else in the texts.

Insertion of a new text Texts are inserted as described in the direct construction of the PBWT (Section 3.2). The insertion point for a new text is always $z + 1$, where z is the number of texts that are currently stored in the PBWT. The user gets a handle for this text, which can later be used to identify the text.

Retrieving a text To find the correct starting point of texts, two data structures, HANDLE and POS, are used. Collecting the characters at the positions given by the iterative LF-mapping reconstructs the text.

Deletion of a text First, we need to know the starting position of the text t_i to be deleted. This is done the same way as when retrieving the text. Deletion of a text in backwards direction is not possible since the deletion of the smallest suffix in the corresponding suffix array destroys the suffix array and hence the PBWT.

Furthermore, the LF-mapping results might not be correct. Thus, direct deletion of a text in backwards direction is not possible. Without additional data structures it is also not possible to walk (and delete) the text in forward direction. Our approach is to perform a backwards walk of the text and to collect all positions. This list of positions is sorted in increasing order using introsort [Mus97] and then all characters, beginning with those at the highest positions, are deleted. During the deletion process the PBWT may be corrupt, but after the whole text is deleted, it is a correct PBWT. Parallel to the deletion of characters in the PBWT, the corresponding bits in the indicator bit vector (defined later in Section 4.3.3) have to be deleted, too. The worst-case running time for this is $\mathcal{O}(|t_i| \log |t_i|)$ due to the sorting of positions using $\mathcal{O}(|t_i| \log n)$ space for the $|t_i|$ positions (of $\log n$ bits each).

Count pattern Counting occurrences of a pattern is implemented with the same backwards search algorithm as described in Section 2.10, only with the difference that rank and C are dynamic. Rank is computed with the Dynamic Sequence and C is implemented with a binary heap (Section 4.3.1).

Locate pattern Section 4.3.3 explains how this is done using a dynamic sampled suffix array.

4.3.1 Table C

Remember that $C[c]$ is the number of characters that are smaller than c . If C was implemented as a simple array of counters (each counter using w bytes) of length $2^{|\text{byte}|}$ as in the static version, updating it would take $\mathcal{O}(2^{|\text{byte}|})$ time, where $|\text{byte}|$ denotes the number of bits a byte holds.

We use a complete binary tree where each leaf corresponds to a character. Each leaf holds the number of occurrences of its character in the index. Each internal node of the tree stores the sum of occurrences of characters that belong to leaves located at this subtree. Starting at the leaf that corresponds to c , we traverse the tree up to the root. On the way up, we count all subtree sums on the left side of the current position. This whole sum is the number of characters currently in the index that are smaller than c . Deletion or insertion of character c can be performed by decreasing or increasing the stored sums at the nodes on the path from leaf to root.

Furthermore, function C is implemented as a binary heap based on an array data structure that consists of $2 \cdot 2^{|\text{byte}|}$ counters. This array simulates the tree as described above. The values of the tree are read in level-wise order and stored in the same order in the array beginning with index 1 and the root node [Wil64]. Note that this is similar to the special case of a Zaks' Sequence (Section 2.7.1) which would consist of a trivial indicator bit vector with a run of $2 \cdot 2^{|\text{byte}|}$ 1's and a second run with the

same number of 0's. Instead of performing rank on this bit vector we can directly compute $\text{parent}(i) = \lfloor i/2 \rfloor$, $\text{left}(i) = 2i$, and $\text{right}(i) = 2i + 1$, where i is the current position in the array.

For simplicity, in this implementation, table C is designed independently of the actual alphabet size σ . It always works on the alphabet size $\sigma_{\max} = 2^{|\text{byte}|}$, including the special character “\$”. Thus, the time for accessing or updating C is always $\mathcal{O}(\log(\sigma_{\max})) = \mathcal{O}(|\text{byte}|) = \mathcal{O}(1)$.

4.3.2 HANDLE and POS

HANDLE and POS are responsible for the management of texts within the PBWT and relieve the user of this task. Each text that has been inserted gets associated with a unique handle that will not change as long as the text is in the index. The handle is an identification number in the range $1 \dots 2^w - 1$ that allows the user to retrieve and delete the text that is associated with the handle or to locate matches. The advantage of this concept is that the user doesn't have to care about changes of starting positions of texts within the index, especially after deletions. Furthermore, when a text is deleted, its handle can be reused later for a new text. All operations, including the search for a free handle, take worst-case $\mathcal{O}(\log m)$ time, where m is the number of texts in the collection.

Data Structure

The data structure for the management of the collection consists of two substructures, HANDLE and POS. HANDLE is a red-black tree that allows access, insertion, and deletion of handles. Each node of the tree is associated with a handle number, the search key of the tree. Furthermore, each node has stored the size of its subtree and the maximal value of all handles within its subtree. This information allows to find the smallest free handle available in asymptotically the same time as the other operations.

POS is a red-black tree where each node corresponds to a text. Whereas in HANDLE the nodes are sorted according to their handle value, here they are sorted according to the starting positions of the texts in the PBWT. Given a node within this tree, we can determine the text's starting position by a traversal where we sum up all the subtree sizes of left subtrees on the way up to the root. The nodes in trees HANDLE and POS have pointers that point to the corresponding node in the other structure. As nodes in HANDLE, nodes in POS contain the subtree sizes. Furthermore, in POS they store the size of the texts they belong to.

Algorithms

Finding a text position in the PBWT For example, we assume that the collection consists of five texts, with handles 5, 6, 11, 21, and 47 (Figure 4.1). The values of the handles do not correspond to the order of their texts in the PBWT. For example, if the user wants to retrieve the text associated with handle 6, the program performs a binary search (grey path in HANDLE) on HANDLE to find the node that corresponds to this handle. A pointer (grey dotted line) at this node points to the corresponding node in POS. Then the traversal (grey path in POS) to the root gives the information about the starting point of the text within the PBWT. The size of subtree left of the traversed path denotes the number of texts with a smaller starting position than the text with handle 6. In this example this number is 2, thus its starting position must be at position 3.

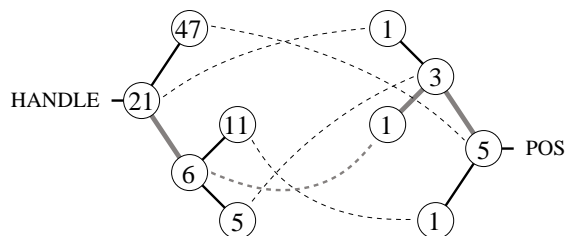


Figure 4.1: HANDLE and POS with links (dotted lines) between their nodes. The values shown in HANDLE are the handle values. The values shown in POS are the subtree sizes of the nodes.

Finding a new handle Reusing handles after texts have been removed avoids running out of handles as long as the number of texts in the collection at the same time does not exceed the number of all possible handles. For example in our implementation we use handles with 32 bits, which theoretically allows us to keep up to 2^{32} texts at the same time in the collection. The algorithm for finding a free handle in HANDLE makes use of two counters, one for the subtree size and one for the maximal key. They are stored at the nodes of HANDLE. The algorithm for finding new handles is described in Algorithm 3.

Asymptotics

Access and update of HANDLE and POS can be performed in $\mathcal{O}(\log m)$ time. Each tree has m nodes, which use $\mathcal{O}(\log m)$ bits of space. The total space usage is $\mathcal{O}(m \log m)$.

Algorithm 3 Find new handle in HANDLE

1. The algorithm starts with $v \leftarrow \text{root}$ and $\text{smaller_handles} \leftarrow 0$.
2. If the current subtree size of v plus smaller_handles is greater than the maximal handle number within the subtree of v , there must be a free handle available in that subtree. If not, go to step 6.
3. If the same property holds for the left child $\text{left}[v]$ we set $v \leftarrow \text{left}[v]$ and continue with step 3 of the algorithm, else continue.
4. We add the subtree size of $\text{left}[v]$ to smaller_handles and check the property of $\text{right}[v]$. If the property holds for $\text{right}[v]$ we set $v \leftarrow \text{right}[v]$ and go on with the second step. Otherwise, continue.
5. Since v does contain a free handle, but none of its children do, there must be a free handle available between the right-most node of the left subtree of v and v or between the left-most node of the right subtree of v and v and we insert a new node for the new handle. Leave algorithm.
6. If the property didn't hold for the root, it means that there is no smaller key available. The next free handle is 1 plus the value of the right-most handle in the tree and a new right-child node is appended to the right-most node. Leave algorithm.

4.3.3 DSSA – Dynamic Sampled Suffix Array

Given a collection of texts and a search pattern, the user wants to know for each match which text in the collection contains the match and the position of the match relative to the first character of the text. Here we also use the sampled suffix array approach (Section 2.11), but as the PBWT itself, the set of sampled values and the indicator bit vector have to be dynamic.

Indicator bit vector For the indicator bit vector we use the same dynamic bit vector as described in Section 4.1. Every time we insert a character into the PBWT, say c , we also insert a bit at the same position to the bit vector. This bit is 1 if the position of c in t_{new} is some multiple of d , otherwise it is 0.

Samples All the sampled values are stored in a red-black search tree. Each node contains one sample. Note that we don't take the values from the suffix array of the concatenated sequence of texts. Instead, we sample values from the suffix arrays of each text in the collection. These values are given by the position of the suffixes in the new text. The samples in the red-black tree have the same order as their corresponding indicator bits. Thus, the insertion position for a new sample is the same as the relative position of its indicator 1-bit at some position i among the other 1-bits. The relative position of this 1-bit within the bit vector can be determined with $\text{rank}(i)$. The nodes

of the red-black tree have stored the sizes of their subtrees. These subtree sizes can be used to find the $\text{rank}(i)$ -th node.

Algorithm This algorithm is very similar to the one for the static FM-index. Given some position i in the PBWT that corresponds to a match (for example, found by count), we have to find the next sampled position i' via LF-mapping. $\text{Rank}(i')$ applied to the dynamic bit vector gives the position of the sample node within the sequence of ordered nodes in the tree. The correct node can easily be found by a top-down walk using the subtree values at the nodes of the tree. The node contains the information about the location of the sample, which is defined by the handle of the text and the position of the sample within the text. Adding the number of LF-mappings for finding the sampled position to the position of the sample gives the position of the match in the text.

Asymptotics Each LF-mapping step takes $\mathcal{O}((H_0 + 1) \log n)$ time, and a query to the indicator bit vector takes $\mathcal{O}(\log n)$ time. Using a sample rate of $s = 1/\log^2 n$, it takes $\mathcal{O}((H_0 + 1)(\log n)/s) = \mathcal{O}((H_0 + 1) \log^3 n)$ time to find the next sampled position. The search in the tree for the sample node takes $\mathcal{O}(\log(n/s)) = \mathcal{O}(\log n)$ time. The tree uses $\mathcal{O}((n/s) \log n) = \mathcal{O}(n \log^3 n)$ bytes of space.

4.3.4 Implementation Remarks

The program is written in C++. Three compile-time constants, `SAMPLE`, `LOGN`, and `BUFFER` are supported and can be set via the preprocessor option “-D”. “-DSAMPLE=d” can be used to set the sample rate $s = 1/d$. Default is $d = 1024$ which refers to a sample rate of $1/\log^2(n)$ on a 32-bit machine. “-DSAMPLE=0” compiles the program without the data structures (indicator bit vector, DSSA) which are used for storing the sampled suffix array values. This can be useful if someone only wants to construct the Burrows-Wheeler Transformation and doesn't need the locate operation of the index. The second macro “-DLOGN=size” is used to change the size of the blocks in the Dynamic Bit Vector. Default is 64. Both parameters can be adjusted to achieve different time-space tradeoffs.

To improve space usage it is possible to read a sequence from disk and to store the BWT to disk without additional use of the main memory. As the sequences have to be read in reverse direction when inserted into the dynamic FM-index, using a buffer reduces the number of accesses to the disk. The size of the buffer can be set via “-DBUFFER=bytes”. The default size is 1 MB. “-DBUFFER=0” sets the size of the buffer equal to the size of the sequence and the sequence can be read in one step.

4.3.5 Asymptotics of the Dynamic FM-Index

The asymptotics of our dynamic FM-index are listed below (Tables 4.2 and 4.3). The first table lists all data structures together with their space usage. Since the size of the tree containing the sampled suffix array values depends on parameter s , the sampling rate, the right-most column assumes $s = \log^{-2} n$ as an example.

Given a collection T with z texts, t_i denotes the i -th text of this collection, where $1 \leq i \leq z$. t_{new} denotes a new text which will be added to the collection. The construction of the index consists of reading a sample file for determining a character distribution and using this distribution for the construction of the empty wavelet tree. Crucial for the construction time of this index is only the insertion time for the new texts in the collection.

STRUCTURE	SPACE	SPACE WITH $s = \log^{-2} n$
Wavelet tree	$\mathcal{O}(n(H_0 + 1))$	
Table C	$ \text{byte} \cdot 2 \cdot 2^{ \text{byte} } \cdot w$	
HANDLE	$\mathcal{O}(z \log z)$	
POS	$\mathcal{O}(z \log z)$	
DSSA/vector	$\mathcal{O}(n)$	
DSSA/samples	$\mathcal{O}((n \log n)/s)$	$\mathcal{O}(n \log^3 n)$

Table 4.2: Space usage of the dynamic FM-index.

Note that additional space, $\mathcal{O}(|t_i| \log(n))$, is used while deletion of some text t_i from the collection.

The operations insert, delete, and retrieve are always applied to complete texts, for example t_i or t_{new} . Query time for locate is given with sampling rate s (middle column) and with a concrete sampling rate $s = \log^{-2} n$ (right column) (Table 4.3).

OPERATION	TIME	TIME WITH $s = \log^{-2} n$
Insert t_{new}	$\mathcal{O}(t_{\text{new}} (H_0 + 1) \log n)$	
Retrieve i	$\mathcal{O}(t_i (H_0 + 1) \log n)$	
Delete i	$\mathcal{O}(t_i \log t_i)$	
Occurrence p	$\mathcal{O}(p (H_0 + 1) \log n)$	
Count p	$\mathcal{O}(p (H_0 + 1) \log n)$	
Locate p	$\mathcal{O}(p (H_0 + 1) \log n)$ $+\mathcal{O}(\text{occ}(H_0 + 1)(\log n)/s)$ $+\mathcal{O}(\text{occ} \log(n/s))$	$\mathcal{O}(p (H_0 + 1) \log n)$ $+\mathcal{O}(\text{occ}(H_0 + 1) \log^3 n)$ $+\mathcal{O}(\text{occ} \log n)$

Table 4.3: Time usage of the dynamic FM-index.

5 Experiments

We have done three type of experiments. The first experiment is to study the effects of choosing different values of `LOGN` (and thus different sizes of the bit blocks) in the Dynamic Bit Vector concerning both, time and space consumption of the index. The second experiment consists of a comparison between our dynamic FM-index and the program “bwt” [Kär06] concerning the efficiency of the BWT-construction. The third experiment compares the time of our dynamic index with the time of a static index, that is needed to add a sequence to an already existing index.

The tests were run on a PC with a 1.86 Ghz Intel Pentium M processor and 1 GB of main memory running Linux. We used the GNU g++ compiler version 4.1.1 with compiler option “-O3”. For time measurements we used the unix `time` command (sum of user and sys time) and for space measurements we used `memusage` (heap peak), a tool that can be downloaded from the Pizza&Chili Corpus site¹.

5.1 Experiment 1: Different Block sizes

The first experiment consists of the construction of the BWT with varying block size in the Dynamic Bit Vector. For the tests we have chosen three types of sequences: DNA, proteins, and English texts. Each sequence is stored in a 10 MB ASCII file. The sequences have been taken from the Pizza&Chili Corpus where we used the first 10 MB of each 50 MB file. We constructed the BWT for all three sequences with varying values of `LOGN`, namely all powers of 2 in the range 8 to 2048. Note that the axes for the `LOGN` values use logarithmic scale. The first graph (Figure 5.1) plots `LOGN` against time usage. In this experiment the space usage includes explicit storage of sequences in the main memory. The second graph (Figure 5.2) plots `LOGN` against the space usage. The third graph (Figure 5.3) plots the time usage against the space usage. For clarification, the fourth graph (Figure 5.4) is a zoom into the third graph. In addition, the third dimension (`LOGN`) is explicitly stated for some runs.

¹<http://pizzachili.dcc.uchile.cl/>

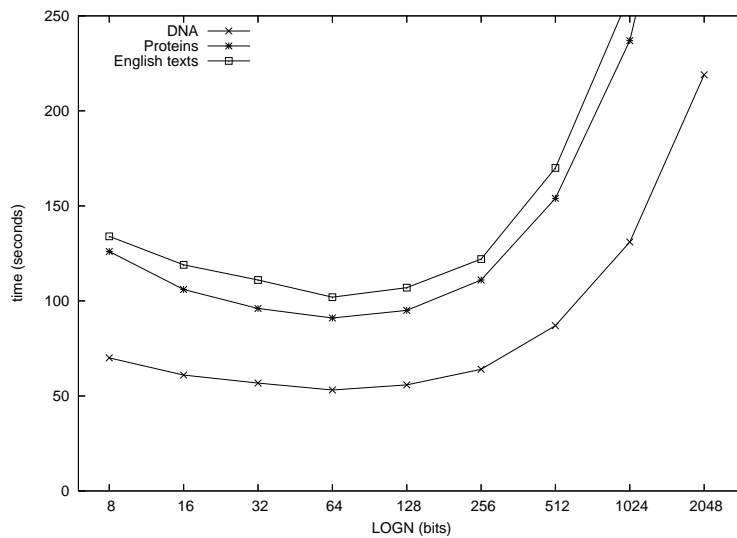


Figure 5.1: Effects of different LOGN values on the time usage.

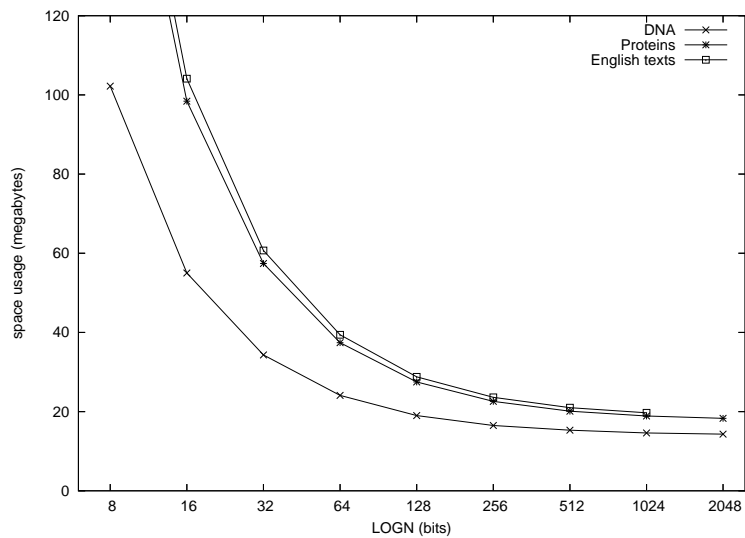


Figure 5.2: Effects of different LOGN values on the space usage.

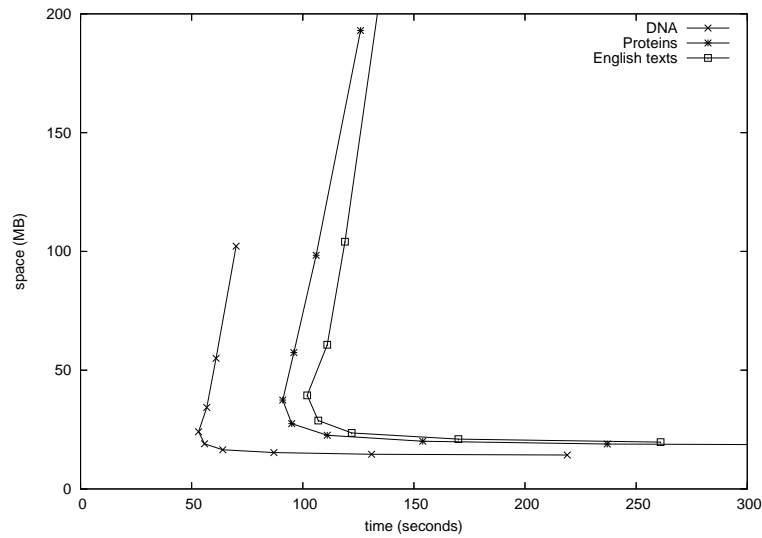


Figure 5.3: Several time-space tradeoffs with varying LOGN.

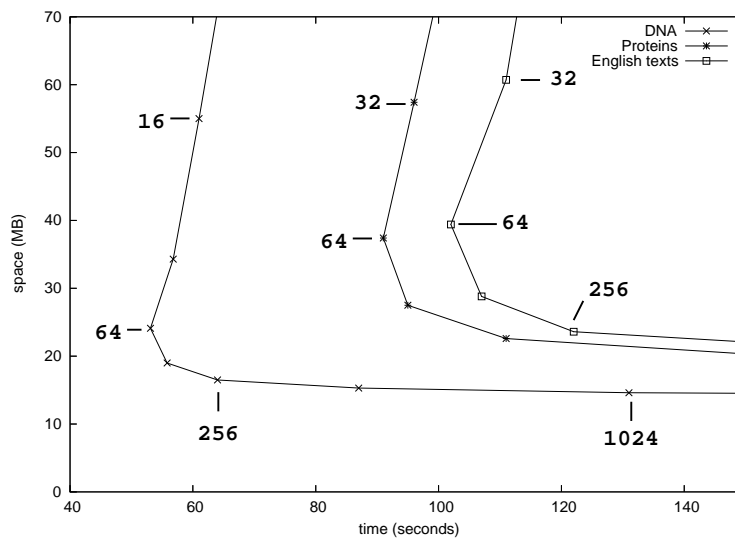


Figure 5.4: Zoomed in: Several time-space tradeoffs with varying LOGN.

The first graph shows how the time is effected by LOGN. For all sequences the minimum is located at $\text{LOGN} = 64$. This refers to a block size of 128 bits and 32 up to 128 bits that are actually stored in each block. This seems to be the best balance between tree height (traversals) and block size (bit operations) concerning time. The second

graph shows the space usage. It gets smaller with increasing LOGN, but the rate of minimization decreases. This is to be expected, as with increasing LOGN the size of the tree decreases.

The last two graphs (Figures 5.3 and 5.4) visualize these time-space tradeoffs. The horizontal axis represents the time and the vertical axis represents the space usage. As there is no axis for LOGN, the values of this third dimension are explicitly given for some runs in the last graph.

Which value for LOGN should be used if time and space are both important? For $\text{LOGN} < 64$ both, time and space usage, increase. It is quite obvious that this would be a bad choice. $\text{LOGN} \geq 64$ allows different tradeoffs. For example, choosing $\text{LOGN} = 128$ may be interesting since we can improve space usage quite significantly compared to $\text{LOGN} = 64$ and lose only little time. With still higher values of LOGN time usage increases much faster than space usage improves.

5.2 Experiment 2: Comparison of the BWT construction

As the dynamic FM-index is based on the Burrows-Wheeler Transformation, one can use it as a simple construction tool for the BWT. After insertion of a text, reading all Dynamic Bit Vector trees in an in-order traversal yields the BWT. This experiment compares time and space usage of our implementation with a recent algorithm by Juha Kärkkäinen [Kär06]. This algorithm has an $\mathcal{O}(n \log n + vn)$ worst-case and $\mathcal{O}(n \log n + \sqrt{vn})$ average-case runtime, where $v \in [3, n^{2/3}]$. It uses $\mathcal{O}(n \log n / \sqrt{v})$ space in addition to the text and the BWT. Kärkkäinen offers two implementations “`bwt`” and “`dnabwt`”. `bwt` uses $v = 128$ and `dnabwt`, which is a more space efficient implementation for DNA, uses $v = 256$.

All programs read the sequences from a file and store them in main memory. Afterwards, the uncompressed BWT is written directly to disk, without storing it in main memory. `dnabwt` uses a 2-bit encoding for DNA internally.

We compiled our program without support for sampled suffix array values and with $\text{LOGN} = 128$. `bwt` and `dnabwt` have been compiled with default values. For the experiments we constructed the BWT of 10, 20, 30, 40, and 50 megabyte ASCII files of DNA, proteins, and English texts. The first graph (Figure 5.5) shows the time usage and the second graph (Figure 5.6) shows the space usage of all runs. DFMI denotes the dynamic FM-index.

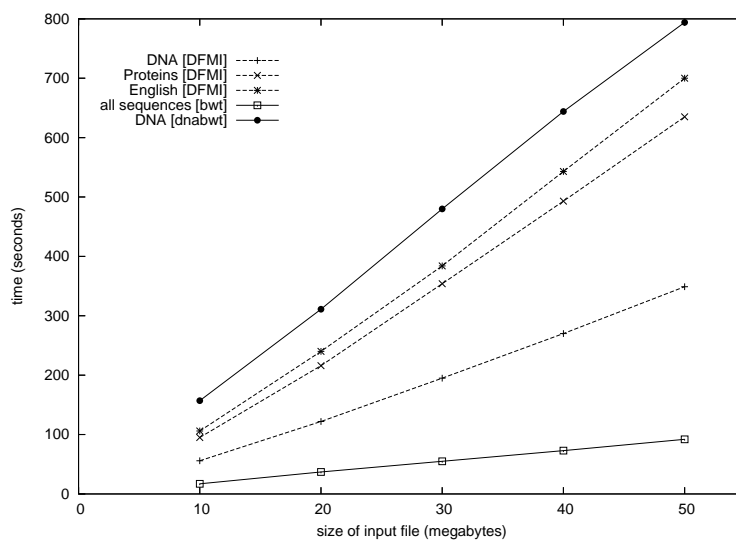


Figure 5.5: Time comparison of DFMI, bwt, and dnabwt.

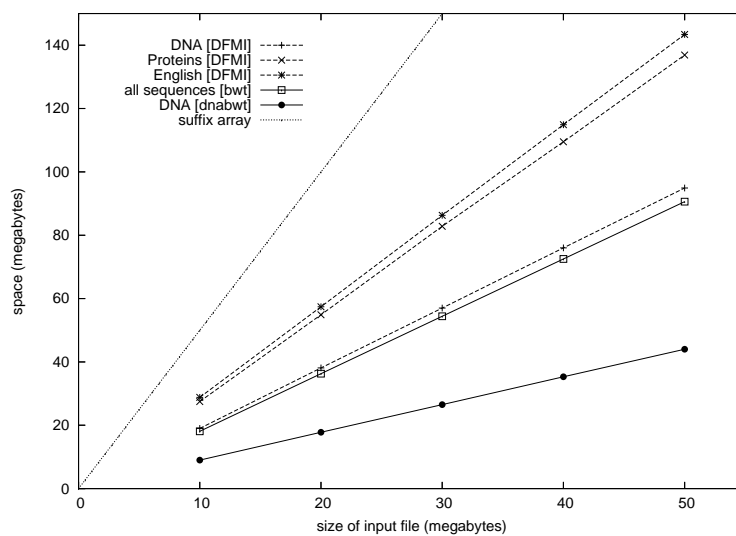


Figure 5.6: Space comparison of DFMI, bwt, and dnabwt.

Runs of DFMI are visualized with dashed lines. DFMI uses for proteins and English

texts similar time and space. For DNA it uses less time and space compared to proteins and English texts. `bwt` uses the same time and space for all three sequences. The dotted line in the second graph visualises the theoretical space usage of a suffix array which uses $4n$ bytes plus n bytes for the text itself. This does not include the space usage at construction time of the suffix array.

Comparing DFMI and `bwt`, one can see that only for DNA both programs approximately use the same space. For proteins and English texts, DFMI uses about 1.5 times the space that `bwt` uses. Considering time, DFMI is about 3-4 times slower on DNA and on the other sequences 7-8 times slower than `bwt`. `dnabwt` uses least space for DNA, but takes the most time. All programs need significantly less space than the classic BWT-construction approach via the suffix array.

Thus, for efficient construction of the BWT, the program `bwt` seems to be the better choice, concerning both time and space. For DNA, `dnabwt` constructs the BWT in even less space than `bwt`.

5.3 Experiment 3: Dynamic versus Static FM-Index

This experiment is designed to figure out in which situation the dynamic FM-index has advantages over a static index. While the dynamic index can simply add a new text to an already existing collection of texts, the static index needs to be completely reconstructed in order to include the new text. We expect that for texts that are much smaller than the texts that are already contained in the index, our dynamic FM-index should be faster than a static index. In this experiment we measure how small a sequence that we add to the index has to be, such that the dynamic FM-index is faster than the static index.

Our program has been compiled with default values, which includes the sampled suffix array values. For comparison, we chose the static FM-index Version 2 by Ferragina and Manzini [FM00]². It is based on the deep-shallow suffix array and BWT construction algorithm [MF02]. We modified the provided program `fm_build` such that it terminates after construction of index and doesn't store the index to disk.

We indexed 100 MB sequence files, DNA, proteins and English texts with the dynamic FM-index. Then we added sequences of length 1 up to 8 megabytes. We measured the time that it takes to add the sequences to the index. For the static index, we concatenated the 100 MB file and the smaller files and constructed the static index for each of these files of size 101 MB, 102 MB, ... and so on (Figures 5.7, 5.8, and 5.9).

²<http://roquefort.di.unipi.it/~ferrax/fmindexV2/>

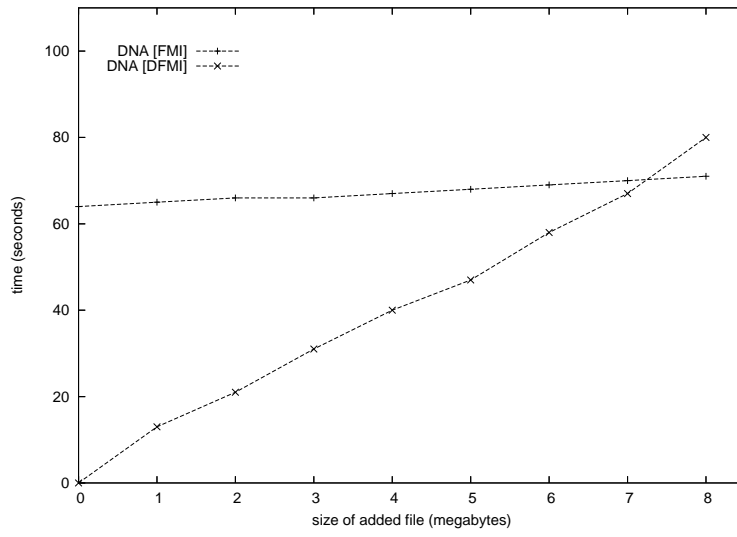


Figure 5.7: DFMI versus FMI: DNA

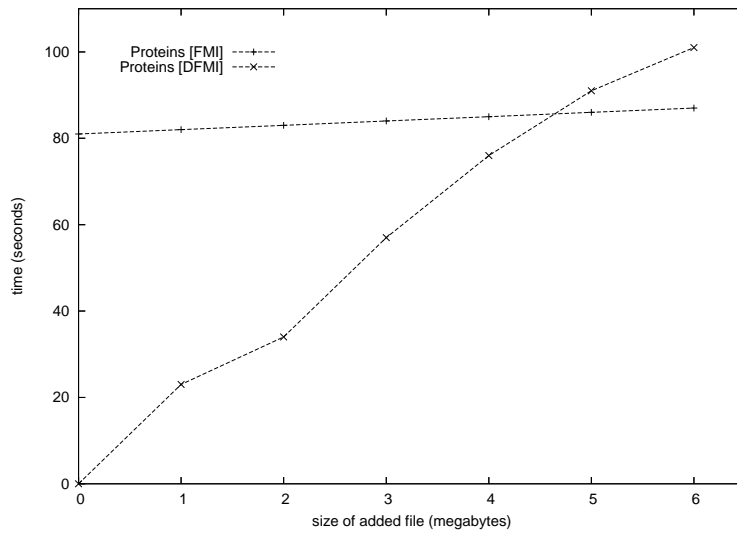


Figure 5.8: DFMI versus FMI: Proteins

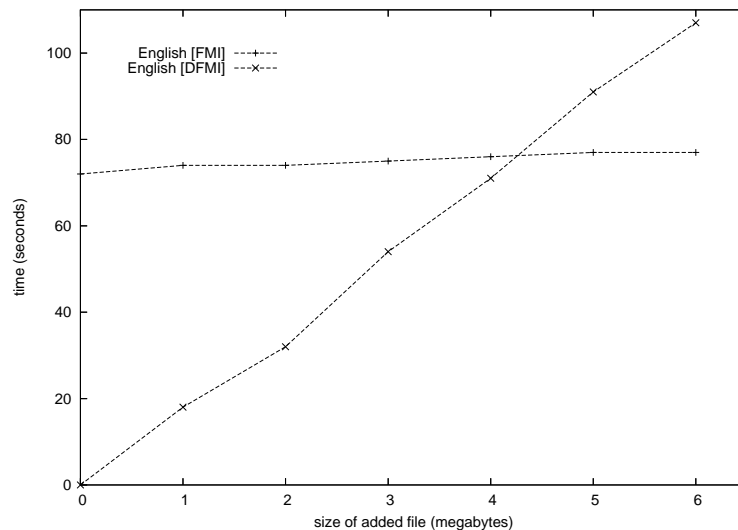


Figure 5.9: DFMI versus FMI: English texts

As we have expected, the dynamic FM-index can insert sequences faster than the static index if the sequences are much shorter than the sequences that are already in the index. For the proteins and English sequences, the dynamic FM-index inserts sequences that are shorter than 4 megabytes into an 100 MB sequence in less time than the static FM-index takes for its construction. For DNA the dynamic FM-index is faster for sequences that are shorter than 7 megabytes. For longer sequences the static FM-index is faster.

This experiment has only considered 100 MB sequences and the results can not automatically be generalized to sequences of other lengths because the construction time of the dynamic FM-index does not develop linearly in the size of the input sequences. For more explanatory power more experiments including smaller and bigger sequences have to be done.

Furthermore, the times to construct the dynamic indices for the 100 MB sequences are 14:34 minutes for DNA, 25:47 minutes for proteins, and 26:29 minutes for English texts. The maximal space usage at construction time is about 307 MB for DNA, 438 MB for proteins, and 461 MB for English texts. Independent of the sequence type, the static FM-index uses 612 MB for the construction. For the final index that is stored to disk, it uses about 38 MB for DNA, 64 MB for proteins, and 47 MB for English texts.

6 Summary and Prospects

This is the first implementation of a *dynamic FM-index for a collection of texts* whose space usage is bound by the 0-order entropy. It provides an easy-to-use interface with its concept of handles. We have shown that the index uses less space for the construction of the BWT than the classic BWT-construction with the suffix array. However, our implementation can not keep up with other programs that are designed to construct the BWT efficiently, concerning time and space usage, as the experiments have shown. An advantage of our implementation is that it already provides the LF-mapping. This allows, for example, to compute sampled suffix array values without an actual suffix array. We have used this in the implementation of a compressed suffix tree [NVM07] which is based on a compressed suffix array. Using our implementation of the dynamic FM-index, we have been able to improve both, time and space usage of the construction of the compressed suffix arrays including the BWT, the sampled suffix array, and its inverse. The latter is needed to support retrieval of any substring of the text in the compressed suffix tree.

The main advantage of this index is that it allows to insert or delete texts of the collection at runtime, which avoids reconstruction of the whole index. At least for 100 MB sequences we have shown that our implementation can insert a sequence of size less than 4-7 megabytes faster than the static FM-index can construct the same concatenated sequence. This might be of interest for applications where only small modifications to a collection of texts are performed but reconstruction of a static index would take too much time.

Character Distribution The current implementation counts the characters of some exemplary sequence to get a character distribution. This could be improved by storing the character distribution for each sequence type separately, for example for proteins and English texts. Such information should also ideally include the expected average number of texts in the collection.

Persistence Furthermore, the current implementation does not allow to store the index to disk. One could for example store the index without any tree structures, thus converting the index into a static one. The information which is needed to restore the index into main memory contains the Huffman-compressed BWT, sampled suffixes, indicator bit vector and the handle–text associations.

Secondary Memory If some data is too big to be completely stored in main memory, secondary memory will be used. This can lead to a significant slow-down since access to secondary memory is slower than the access to the main memory. Secondary memory implementations take care of this problem by reducing the number of accesses to disk and reading whole blocks of data from secondary memory into the main memory. The algorithm can then operate on this block. This is only possible if the data on which the algorithm operates can be found in such a contiguous block. This is not the case for the backwards search or reverse construction in the BWT, for example. Access to the BWT follows an unpredictable pattern which is not limited to any block that is much smaller than the whole BWT. Thus, a future extension of this implementation towards efficient secondary memory usage seems to be unlikely.

Acknowledgments

I thank my supervisors Dr. Veli Mäkinen, Dipl.-Inform. Klaus-Bernd Schürmann, and Prof. Jens Stoye for making possible this diploma thesis and for their advice during my work on this thesis. I also thank Romina Drees, Christian Höner zu Siederdisen, Nils Hoffmann, Gonzalo Navarro, and especially Marcel Martin for several helpful discussions and proofreading my thesis.

Bibliography

- [ADKz70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.
- [AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.
- [Apo85] A. Apostolico. The Myriad Virtues of Subword Trees. *NATO Advanced Science Institutes, Series F*, 12:85–96, 1985.
- [Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.
- [BB04] Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 11–19, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
- [BM94] Andrej Brodnik and J. Ian Munro. Membership in constant time and minimum space. In *European Symposium on Algorithms*, pages 72–81, 1994.
- [BMW94] Timothy C. Bell, Alistair Moffat, and Ian H. Witten. Compressing the digital library. In *Proceedings of the First Annual Conference on the Theory and Practice of Digital Libraries*, 1994.
- [BSTW86] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.
- [BW94] Michael Burrows and David John Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital System Research Center, May 1994.
- [BYCC03] Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors. *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, volume 2676 of *Lecture Notes in Computer Science*. Springer, 2003.

- [CHL04] H. L. Chan, W. K. Hon, and T. W. Lam. Compressed index for a dynamic collection of texts. In *In Proceedings of Symposium on Combinatorial Pattern Matching*, pages 445–456, 2004.
- [Cla96] David Richard Clark. *Compact PAT trees*. PhD thesis, David R. Cheriton School of Computer Science, Department of Computer Science, University of Waterloo, Canada, 1996.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
- [Die89] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 39–46, London, UK, 1989. Springer-Verlag.
- [Eli75] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.
- [Eli87] Peter Elias. Interval and recency rank source coding: Two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory*, 33(1):3–10, 1987.
- [FCFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
- [FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society.
- [FMMN06] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 2006. To appear.
- [Gal78] R.G. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, 1978.
- [GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
- [GKS99] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. In *WAE '99: Proceedings of the 3rd International*

- Workshop on Algorithm Engineering*, pages 30–42, London, UK, 1999. Springer-Verlag.
- [GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE, IEEE, 1978.
- [Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.
- [Har28] R.V.L. Hartley. Transmission of information. *Bell System Technical Journal*, 7(4):535–563, 1928.
- [Huf52] David A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.
- [Jac88] Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, School of Computer Science Carnegie Mellon University, 1988.
- [Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE, IEEE, 1989.
- [KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Baeza-Yates et al. [BYCC03], pages 200–210.
- [Kär06] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 2006. to appear.
- [KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C.M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, languages and programming : 30th International Colloquium, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, Eindhoven, The Netherlands, 2003. Springer.
- [KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In Baeza-Yates et al. [BYCC03], pages 186–199.
- [Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999.
- [McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
- [MF02] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. In *ESA '02: Proceedings of the 10th Annual*

- European Symposium on Algorithms*, pages 698–710, London, UK, 2002. Springer-Verlag.
- [MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- [MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
- [MN06] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, LNCS 4009, pages 307–318, 2006.
- [Mun96] J. Ian Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, London, UK, 1996. Springer-Verlag.
- [Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software - Practice and Experience*, 27(8):983–993, 1997.
- [NM06a] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2006. To appear.
- [NM06b] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes (survey, 2nd revised version). Technical Report TR/DCC-2006-6, Department of Computer Science, University of Chile, April 2006.
- [NVM07] Kashyap Dixit Niko Välimäki, Wolfgang Gerlach and Veli Mäkinen. Compressed suffix tree - a basis for genome-scale sequence analysis. *Bioinformatics*, 2007. Accepted to Bioinformatics.
- [RRR01] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *WADS '01: Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 426–437, London, UK, 2001. Springer-Verlag.
- [RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
- [Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.
- [Sta05] R. Stallman. *Using the GNU Compiler Collection (GCC Version 4.1.1)*. Free Software Foundation, Inc., 2005. <http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc.pdf>.

- [Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.
- [Wil64] John W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.
- [WKHS03] K. Sadakane W. K. Hon and W. K. Sung. Succinct data structures for searchable partial sums. In *In Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 505–516, 2003.
- [Zak80] S. Zaks. Lexicographic generation of ordered trees. *j-THEOR-COMP-SCI*, 10(1):63–82, January 1980.

Declaration

I hereby declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Bielefeld, 28th February 2007

.....
Wolfgang Gerlach