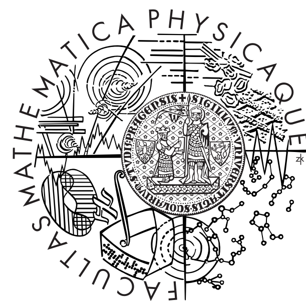
Faculty of Mathematics and Physics
Charles University, Prague

MASTER THESIS



Radovan Šesták

Suffix Array for Large Alphabet

Department of Software Engineering

Supervisor: Mgr. Jan Lánský
Computer Science

2007

Let the bridge stand long and proud. Thanks be to those who funded it, provided knowledge for design and helped with implementation.

I declare that I have worked out this thesis on my own, using only the resources stated. I agree that the thesis may be publicly available.

Prague, April 20, 2007 Radovan Šesták

Contents

1 Introduction **5**

2 Definitions and notation **6**
2.1 Alphabet, rotation, suffix, index array, suffix array 6
2.2 Complexity 8
2.3 Words and Syllables 10

3 BWT **12**
3.1 BWT – coding 12
3.1.1 Sort rotations 12
3.1.2 Take last characters of rotations 13
3.2 Reverse BWT - decoding 13
3.2.1 Find first characters of rotations 13
3.2.2 Build list of predecessor characters 13
3.2.3 Form output 16

4 BWT (Coding) Algorithms **17**
4.1 Karp-Miller-Rosenberg's (KMR) algorithm 17
4.2 Manber-Myears' (MM) algorithm 18
4.3 Sadakane's algorithm 18
4.4 Larsson's algorithm 19
4.4.1 Description 19
4.4.2 Memory requirements 20
4.4.3 Time complexity 21
4.5 Seward's algorithm 22
4.5.1 Description 22
4.5.2 Memory requirements 24
4.5.3 Time complexity 24
4.6 Our Seward based 1-level bucket algorithm 25
4.7 Itoh's algorithm 25
4.8 Kao's modification of Itoh's algorithm 25

4.9 Kärkkäinen and Sanders' algorithm 26
4.9.1 Description 26
4.9.2 Memory requirements 29
4.9.3 Time complexity 30

5 Reverse BWT for suffixes 31

6 Improving performance of BWT 36
6.1 Testing Environment 36
6.2 Corpus Summary 37
6.3 Notation in Results 37
6.4 Comparing rotations vs. comparing suffixes 38
6.5 Choosing q-sort 42
6.6 Reverse of Input String 43
6.7 Changing the block size 45
6.8 Influence of Alphabet 48
6.9 Choosing Algorithm 51

7 Conclusion 58

A Contents of Compact Disk 59

Title: Suffix Array for Large Alphabet
Author: Radovan Šesták
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský, zizelevak@gmail.com
Abstract: Burrows-Wheeler Transform (BWT) [3] is used as the major part in block compression which has good balance of speed and compression ratio. Suffix arrays are used in the coding phase of BWT and we focus on creating them for alphabet larger than 2^8 symbols. The motivation for this work has been software project XBW [4] – an application for compression of large XML files. The role of BWT is to reorder input before applying other algorithms. We describe and implement three families of algorithms for encoding. First is inspired by the work of Sadakane [10] and further improved by Larsson [8]. Second family includes algorithm by Seward [11] and algorithm by Itoh further improved by Kao [5]. Finally we present algorithm by Kärkkäinen and Sanders [6] for constructing suffix arrays in linear time.

As our main result we show that for textual data using syllables or words as alphabet improves both run time and compression ratio of block compression.
Keywords: suffix arrays, BWT, data compression, text compression

Název práce: Suffixové pole pro velkou abecedu
Autor: Radovan Šesták
Katedra: Katedra softwarového inženýrství
Vedoucí práce: Mgr. Jan Lánský, zizelevak@gmail.com
Abstrakt: Burrows-Wheelerova Transformace (BWT) [3] je používána jako hlavní část blokové komprese, která má dobrý kompresní poměr a přijatelný čas běhu. Suffixová pole jsou používána v kódovací fázi BWT a my se soustředíme na jejich tvorbu pro abecedu větší než 2^8 symbolů. Motivací pro tuhle práci byl softwarový projekt XBW [4] – aplikace pro kompresi velkých XML souborů. Úkolem BWT je přeuspořádat vstup před použitím jiných algoritmů. Popisujeme a implementujeme tři skupiny algoritmů pro kódování. První je inspirována prací Sadakana [10] a dále vylepšená Larssonem [8]. Druhá skupina obsahuje algoritmus od Sewarda [11] a algoritmus od Itoha vylepšený Kaoem [5]. Závěrem prezentujeme algoritmus od Kärkkäinena a Sanderse [6] pro konstrukci suffixových polí v lineárním čase.

Jako hlavní výsledek ukážeme, že pro textová data použití slabik nebo slov jako abecedy zlepšuje čas běhu i kompresní poměr.
Klíčová slova: suffixová pole, BWT, datová komprese, komprese textu

Chapter 1

Introduction

Burrows-Wheeler transform [3] (BWT) produces reversible permutation of input string. In the resulting string predecessors of common suffixes are clustered together and therefore long sequences of repeated characters occur. In order to obtain transformed string one has to sort lexicographically all rotations of the string. Very closely related to BWT is suffix sorting. If we perform the BWT on a string terminated by special character (smaller than all characters in the string), we obtain the same sorted order of strings as when creating suffix array. Hence we can use suffix sorting instead of BWT if we allow slight modification of string. General string comparison algorithms have time complexity $O(n^2 log\, n)$ where n is the length of the string. However several algorithms that run in $O(n\, log\, n)$ time are known and this is the case of Sadakane [10] type algorithms. Also algorithm working in linear time $O(n)$, due to Kärkkäinen and Senders [6], is known and we have implemented it as well. We also present algorithms with worse guaranteed running time which perform better on non repetitive inputs. The reverse – obtaining original string from transformed string can be done in linear time. The running time of BWT is critical for the speed of the block compression since it is the most time demanding part. The tradeoff between compression ratio and running time is determined by size of block transformed at once. The block size n determines the time complexity, which is usually superlinear, as well as memory complexity which is usually in range $5n$ to $11n$.

In next sections we define the terms used and present algorithms for BWT that serve as starting points for our optimization. First we present modifications that enable use of large alphabet and later optimizations to improve running time.

Chapter 2

Definitions and notation

2.1 Alphabet, rotation, suffix, index array, suffix array

Definition 2.1.1. Let a set Σ be an *alphabet*. Then we call its elements *characters* or *symbols* and by $|\Sigma|$ we denote size of the alphabet.

In most compression programs that use block compression, the size of alphabet is 2^8. By large alphabet we mean alphabet larger than 2^8. When we use words or syllables as alphabet, then for textual files of few megabytes the size of alphabet is usually around 2^{16}.

Definition 2.1.2. An ordered set of characters we call *string*. By

$$X \equiv x_0 x_1 ... x_{n-1},\ \forall i \in \{0, .., n-1\},\ x_i \in \Sigma$$

we denote string of length n over alphabet Σ.

We use a lot of notation taken from linguistics. We believe that they are intuitive and straightforward.

Definition 2.1.3. I-th *rotation* of a string $X = x_0 x_1 ... x_{n-1}$ is a string

$$R_i = x_i x_{i+1} .. x_{n-1} x_0 .. x_{i-1}\,.$$

A very closely related term is suffix.

Definition 2.1.4. I-th *suffix* of string $X = x_0 x_1 ... x_{n-1}$ is string

$$S_i = x_i x_{i+1} .. x_{n-1}\,.$$

Character after x_i in string is x_{i+i}. For rotation the character following x_n is x_0. If we want to use the same notation and to work easily with numbers in range $0..n-1$ we note

$$|i| = i \, mod \, n \, .$$

We also use array notation that is slightly unorthodox.

Definition 2.1.5.

$$[i..j] \equiv \begin{cases} \{i, i+1, .., j-1, j\} & i \leq j \\ \{i, .., n-1, 0, .., j\} & i > j \end{cases}$$

$$[i..j) \equiv \begin{cases} \{i, i+1, .., j-1\} & i \leq j \\ \{i, .., n-1, 0, .., j-1\} & i > j \end{cases}$$

and for strings or arrays

$$X[i] \equiv x_i$$

$$X[i..j] \equiv \begin{cases} x_i, x_{i+1}, .., x_{j-1}, x_j & i \leq j \\ x_i, .., x_{n-1}, x_0, .., x_j & i > j \end{cases}$$

$$X[i..j) \equiv \begin{cases} x_i, x_{i+1}, .., x_{j-1} & i \leq j \\ x_i, .., x_{n-1}, x_0, .., x_{j-1} & i > j \end{cases}$$

When comparing rotations or suffixes we use lexicographic order.

Definition 2.1.6. Rotation R_i is *smaller* than R_j iff

$$R_i < R_j \iff \exists k \in [0..n) : \forall l \in [0..k) \, x_{|i+l|} = x_{|j+l|} \, \& \, x_{|i+k|} < x_{|j+k|}$$

and similarly for suffixes

$$S_i < S_j \iff \exists k \in [0..n) : \forall l \in [0..k) \, x_{i+l} = x_{j+l} \, \& \, _{i+k} < x_{j+|} \vee \, i+l = n) \, .$$

We can choose any fixed order of characters and hence we use natural numbers for identifying the characters and comparing their order. We want to find lexicographical order of all rotations (suffixes) of string X that we will store in index array I. Index array can be either rotation array or suffix array.

Definition 2.1.7. *Rotation array* denoted by RA of a string X is an array holding starting positions of rotations in sorted order. Formally

$$RA \text{ is rotation array } \iff (\forall i, j \in \{0..n-1\}, \, i < j \, \rightarrow R_{RA[i]} \leq R_{RA[j]}) \, .$$

Analogically for *suffix array*:

$$SA \text{ is suffix array } \iff (\forall i, j \in \{0..n-1\}, \, i < j \, \rightarrow S_{SA[i]} \leq S_{SA[j]}) \, .$$

Note that two different rotations can be equal, but suffixes can not.

We use special symbol \$ as terminal symbol of string and $\circ$ to denote concatenation of strings.

Definition 2.1.8. The *terminal symbol* \$ is not from the alphabet and is smaller than all characters

$$\$ \notin \Sigma,\, \forall x \in \Sigma :\, \$ < x\,.$$

Then

$$X_\$ \equiv X \circ \$$$

denotes a string terminated with special character.

Definition 2.1.9. The result of BWT on string X is $\tilde{X}$. If we do not specify the string on which we perform the transformation we note the result by B.

$$B \equiv b_0..b_{n-1} \text{ where } b_i = x_{|I[i]-1|}$$

$$\tilde{X} \equiv B$$

Burrows and Wheeler used rotations, so in the previous definition RA instead of I was used. Later as we will show, SA can be used instead of RA. Hence we use I to denote index array, to imply that we can use both. For rotations $i-th$ character in transformed string B is the last character of i-th rotation.

In some algorithms we do not compare whole rotations at once. We use k-th order and increase k throughout the algorithm.

Definition 2.1.10. We note k-order of strings by $<_k$:

$$R_i <_k R_j \iff \exists l \in [0..k),\, R_i[0..l) = R_j[0..l) \,\&\, R_i[l] < R_j[l]$$

$$R_i =_k R_j \iff R_i[0...k) = X_j[0..k)$$

2.2 Complexity

We mention memory and time complexity of algorithms after describing the algorithm.

We expect representation of alphabet as whole numbers in standard data types. In specific unsigned types of size 1 byte for $|\Sigma| \leq 2^8$, 2 bytes for $2^8 < |\Sigma| \leq 2^{16}$ and 4 bytes for $2^{16} < |\Sigma| \leq 2^{32}$ and note this by $bytesize(|\Sigma|)$.

Definition 2.2.1. For positive value k we define $bytesize(k)$ as:

$$bytesize(k) \equiv \begin{cases} 2^i & 2^{8i} < k \leq 2^{8(i+1)} \end{cases}$$

for types we use this notes:

$$bytesize(k) \equiv \begin{cases} 1 & k \leq 2^8 \\ 2 & 2^8 < k \leq 2^{16} \\ 4 & 2^{16} < k \leq 2^{32} \end{cases}$$

We could have used $\lceil log_{2^8}|\Sigma| \cdot n \rceil$ as number of required bytes, but such implementation would lack clarity and suffer from lower performance due to non-standard pointer arithmetic.

For performance of direct comparison algorithms (Seward, Itoh) a critical factor is average match length *AML*. Match length *ML* is noted by some authors also as longest common prefix *lcp*.

Definition 2.2.2. *Match length* of strings

$$ML(R_i, R_j) = max\{k; R_i[0..k) = R_j[0..k)\}$$

We are interested in match length of neighboring rotations/suffixes. Hence we define

Definition 2.2.3. *Average match length*

$$AML = \frac{1}{n-1} \sum_{i=0}^{n-2} ML(R_{I[i]}, R_{I[i+1]})$$

Instead of rotations, suffixes can be used as in original definition by Sadakane. We will use both depending on whether we are constructing rotation array RA or suffix array SA.

For algorithms that multiply the order in each run (Sadakane, Kärkkäinen&Sanders) time complexity depends on maximal match length.

Definition 2.2.4. *Maximal match length*

$$MML = max_{i \in [0..n-2]}\{ML(R_{I[i]}, R_{[i+1]})\}$$

which is equivalent to $MML = max_{i,j \in [0..n)}\{ML(R_i, R_j)\}$.

2.3 Words and Syllables

In Section 6.8 we discuss how change of alphabet can significantly improve performance of BWT. The most common is interpretation of one byte as symbol and then alphabet consists of all such symbols. In program XBW we also use alphabet of syllables and words inspired by Lánský [7]. Words and syllables as we will define them should correspond to common uses of these terms. We consider *word* to be a single unit of language with meaning and *syllable* to be a single unit of speach. Now we present formal definitions.

Definition 2.3.1. *Letters* $\Sigma_L \subset \Sigma$ *are a subset of alphabet. Nonletters are the remaining characters* $\Sigma_N \equiv \Sigma \setminus \Sigma_L$*. Let* $\Sigma_D \subset \Sigma_N$ *be set of digits, then* $\Sigma_S \equiv \Sigma_N \setminus \Sigma_D$ *is a set of special characters.*

For forming words and syllables we distinguish types of letters.

Definition 2.3.2. Let $\Sigma_{Ls} \subset \Sigma_L$ be a set of *small letters* then $\Sigma_{Lc} \equiv \Sigma_L \setminus \Sigma_{Ls}$ is a set of *capital letters*.

Let $\Sigma_{Lv} \subset \Sigma_L$ be a set of *vowels* then $\Sigma_{Lc} \equiv \Sigma_L \setminus \Sigma_{Lv}$ is a set of *consonants*.

In some languages letter can act as either vowel or consonat depending on context. For simplicity and because the effect on compression is small, we fix for each letter if it is vowel or consonant.

Now we can define words.

Definition 2.3.3. Let Σ be a finite nonempty set of symbols and $\alpha \equiv \alpha_1, \ldots, \alpha_n$, $\alpha_i \in \Sigma$. Then we distinguish following types of words:

special $\alpha_i \in \Sigma_S \; \forall i \in [1..n]$

numeric $\alpha_i \in \Sigma_D$ for $\forall i \in [1..n]$

small $\alpha_i \in \Sigma_{Ls}$ for $\forall i \in [1..n]$

capital $\alpha_i \in \Sigma_{Lc}$ for $\forall i \in [1..n]$

mixed $\alpha_1 \in \Sigma_{Lc} \;\&\; \alpha_i \in \Sigma_{Ls} \; \forall i \in [2..n]$

For practical purposes when we parse input into words we also limit maximum length of words. We note special and numeric words by *words from non-letters* and the rest by *words from letters*.

Definition 2.3.4. *Syllable* is part of word. More precisely: let α be a word from letters. Then σ_i, $i \in [1..k]$ are syllables if each and every syllable contains exactly one vowel and $\alpha = \sigma_1 \circ \cdots \circ \sigma_k$. If word α contains no vowel then the whole word is syllable.

This definition is ambiguous, because we can get different sets of syllables which fulfill the conditions in definition. In fact in application XBW we use three types of partitioning into syllables. They assign different number of consonants from each side to vowels which form the core of syllable.

Also we have not specified yet which characters belong to which group. For english we consider letters to be characters `a..z` and `A..Z`. For other languages the sets of letters are different. Distinction of vowels and consonants also largely depends on language used.

Chapter 3

BWT

In this section we show how BWT (coding) [3] and the reverse BWT (decoding) [3, 12] work. We show it using two examples: string X = `abraca` and string X = `banana`.

3.1 BWT – coding

The result of BWT is string $\tilde{X}$ where $\tilde{X}[i] = X[|I[i]-1|]$ and index of original string in array I which we denote by *index* and for which $I[index] = 0$. So we need to sort rotations and take last characters of the rotations. See Figure 3.1 for all rotations.

R_0	*abraca*	R_0	*banana*
R_1	*bracaa*	R_1	*ananab*
R_2	*racaab*	R_2	*nanaba*
R_3	*acaabr*	R_3	*anaban*
R_4	*caabra*	R_4	*nabana*
R_5	*aabrac*	R_5	*abanan*

Figure 3.1: Rotations of *abraca* and *banana*.

3.1.1 Sort rotations

First we sort all rotations and get rotation array (index array). For our example *index* is 1 for `abraca` and 3 for `banana`. We will need index of original string for decoding. See Figure 3.2 for sorted rotations.

$R_{I[0]}$	$aabra$**c**		$R_{I[0]}$	$abana$**n**
$R_{I[1]}$	$abrac$**a**		$R_{I[1]}$	$anaba$**n**
$R_{I[2]}$	$acaab$**r**		$R_{I[2]}$	$anana$**b**
$R_{I[3]}$	$braca$**a**		$R_{I[3]}$	$banan$**a**
$R_{I[4]}$	$caabr$**a**		$R_{I[4]}$	$naban$**a**
$R_{I[5]}$	$racaa$**b**		$R_{I[5]}$	$nanab$**a**

Figure 3.2: Sorted rotations of *abraca* and *banana*.

3.1.2 Take last characters of rotations

Now we take the last characters of the rotations to get $\tilde{X}_{\texttt{abraca}} = \texttt{caraab}$ $\tilde{X}_{\texttt{banana}} = \texttt{nnbaaa}$. These are the last columns of the matrix holding all rotations Figure 3.2. The output of the transformation is pair $(\tilde{X}, \textit{index})$. For our example (`caraab, 1`) and (`nnbaaa, 3`) respectively.

3.2 Reverse BWT - decoding

We use output of previous algorithm as input. Hence our input is string $\tilde{X} =$ `caraab` and $\textit{index} = 1$ which we want to decode into `abraca`. For another example $X =$ `banana` the transformed string $\tilde{X} =$ `nnnbaaa` and $\textit{index} = 3$.

We know that last character of X is $\tilde{X}[\textit{index}]$. In the following paragraphs we show how we can output X from right to left. Although the decoding is longer and a bit tricky to explain, fast algorithm for decoding due to Seward [12] is short and simple. See Algorithm 1.

3.2.1 Find first characters of rotations

Let F be the first column of the matrix M. We can obtain it by sorting the string $\tilde{X}$. Observe that any column of M is permutation of X and therefore also of $\tilde{X}$. The rows of M are sorted, and F is the first column of M, hence the characters in F are also sorted. In our example $F_{abraca} =$ `aaabcr` and $F_{banana} =$ `aaabnn` respectively. See Figure 3.3.

3.2.2 Build list of predecessor characters

To explain this step we use the matrix M. However only arrays $\tilde{X}$ and F along with value $\textit{index}$ are available to the algorithm.

Let M' be matrix formed from M by shifting each row one character to the right $M'[i,j] = M[i, |j-1|]$. See Figure 3.4.

Algorithm 1 REVERSE BWT (decoding)

```
{C[0..Σ − 1] array for frequences of characters intialised to 0}

{first phase – find first characters of rotations}
for i = 0 to n − 1 do
  C[X̃[i]] ⇐ C[X̃[i]] + 1
end for
for i = 0 to |Σ| do
  j ⇐ C[i]
  C[i] ⇐ sum
  sum ⇐ sum + j
end for

{second phase – build predecessor list}
for i = 0 to n − 1 do
  T[i] ⇐ C[X̃[i]]
  C[X̃[i]] ⇐ C[X̃[i]] + 1
end for

{third phase – form output}
for i = n − 1 downto 0 do
  X[i] ⇐ X̃[index]
  index ⇐ T[index]
end for
```

row	M_{abraca}	M_{banana}
0	**a***abrac*	**a***banan*
1	**a***braca*	**a***naban*
2	**a***caabr*	**a***nanab*
3	**b***racaa*	**b***anana*
4	**c***aabra*	**n***abana*
5	**r***acaab*	**n***anaba*

Figure 3.3: Matrix of sorted rotations with first column F highlighted.

row	M_{abraca}	M'_{abraca}	M_{banana}	M'_{banana}
0	*aabrac*	*caabra*	*abanan*	*nabana*
1	*abraca*	*aabrac*	*anaban*	*nanaba*
2	*acaabr*	*racaab*	*ananab*	*banana*
3	*bracaa*	*abraca*	*banana*	*abanan*
4	*caabra*	*acaabr*	*nabana*	*anaban*
5	*racaab*	*bracaa*	*nanaba*	*ananab*

Figure 3.4: Matrix M of rotations and matrix M' of shifted rotations.

Each row of M' is a rotation of X, and for each row of M there is corresponding row in M'. The rows of M' are sorted lexicographically starting with their second character. If we take rows of M' starting with any fixed character c they are sorted lexicographically relative to one another – they are sorted lexicographically starting with second character and they have the same first character which does not affect the order. So strings in M' that begin with `c` appear in the same order as strings that begin with `c` in M.

Consider strings starting with character `a` from our example. The rows `aabrac, abraca, acaabr` are rows 0,1,2 in M_{abraca} and correspond to rows 1,3,4 in M'_{abraca}.

F is first column of M and last column of M'. $\tilde{X}$ is first column of $M'$and last column of M. Using F and $\tilde{X}$ we calculate vector T that indicates the correspondence between the rows of the two matrices, in the sence that for each $j \in [0..n)$, row j of M' corresponds to row $T[j]$ of M.

If $\tilde{X}[j]$ is the k-th instance of c in $\tilde{X}$, then $T[j] = i$ where $F[i]$ is the k-th instance of c in F. Note that T represents a one-to-one correspondence between elements of F and elements of $\tilde{X}$, and $F[T[j]] = \tilde{X}[j]$.

In our examples $T_{abraca} = (4, 0, 5, 1, 2, 3)$ and $T_{banana} = (4, 5, 3, 0, 1, 2)$.

row	M_{abraca}	M'_{abraca}	M_{banana}	M'_{banana}
0	**aabrac**	*cabrac*	**abanan**	*nabana*
1	**abraca**	**aabrac**	**anaban**	*nanaba*
2	**acaabr**	*racaab*	**ananab**	*banana*
3	*bracaa*	**abraca**	*banana*	**abanan**
4	*caabra*	**acaabr**	*nabana*	**anaban**
5	*racaab*	*bracaa*	*nanaba*	**ananab**

Figure 3.5: Rows starting with `a` in M and corresponding rows in M'.

3.2.3 Form output

Now, for each $i \in [0..n)$ the characters $\tilde{X}[i]$ and $F[i]$ are the last and first characters of the row i of M. Since each row is a rotation of X, the character $\tilde{X}[i]$ cyclically precedes the character $F[i]$ in X. From the construction of T, we have $F[T[j]] = \tilde{X}[j]$. Substituting $i = T[j]$, we have $\tilde{X}[T[j]]$ cyclically precedes $\tilde{X}[j]$ in X. ($\tilde{X}[T[j]]$ precedes $F[T[j]]$ and $F[T[j]] = \tilde{X}[j]$)

The index *index* is defined such that the row *index* of M is X. Thus, the last character of X is $\tilde{X}[index]$. We use the vector T to give the predecessors of each character. The original string is hence $X[n-1-i] = \tilde{X}[T^i[index]]\ i \in [0..n)$, where $T^0[x] = x$, $T^{i+1}[x] = T[T^i[x]]$. This yields X, the original input to the BWT coder.

Chapter 4

BWT (Coding) Algorithms

In this section we briefly describe several algorithms for BWT. Most of them were designed for constructing suffix arrays, but we formulate them as algorithms for BWT - so instead of suffixes we work with rotations and as result of the algorithm we consider index array I. First we start with algorithms that use so-called doubling technique and progress as the main idea is developed into fast algorithm. Then we continue with algorithms using the so-called copying of pointers as a method for minimising amount or rotations sorted using comparison based sort (qsort). After that we present algorithm for linear time suffix array construction that is not suitable for sorting rotations; it needs $ terminated input string.

4.1 Karp-Miller-Rosenberg's (KMR) algorithm

In first iteration KMR algorithm [10] sorts and splits all rotations according to their first character using radix sort. Two rotations are in the same group iff they start with the same character. We assign rotation R_i number $V[i]$. All rotations in the same group have been assigned the same number $v = V[i]$ and we represent this groups by this number v. Different groups are represented by different value. Since rotations in any given group start with the same character we can compare them starting with second character - to do this we use values assigned to rotations in the first iterations. If we compare R_i with R_j that are from the same group, using $V[|i+1|]$ and $V[|j+1|]$ as keys, we get groups sorted by the first two characters. If we repeat this approach again using $V[|i+2|]$ as key for R_i, we get groups sorted by first four characters since the keys now can distinguish rotations by first two characters. In each step we double the number of characters by which the rotations are sorted. If groups are split and ordered in alphabetical order, all rotations can be sorted

in lexicographical order.

The key idea is the so-called *doubling technique*. All rotations in the same group have the same first k characters $X[i..i+k)$ after $log_2 k$ iteration. Since the numbers $V[|i+k|]$ are already calculated in the last iteration, we can split the groups according to characters $X[|i+k|..|i+2k|)$ in the next iteration. Consequently, all rotations can be sorted in lexicographic order within $log\, n$ iterations.

4.2 Manber-Myears' (MM) algorithm

MM algorithm [10, 8] uses the doubling technique described in the algorithm by Karp, Miller and Rosenberg. First we sort and group all rotations by their first symbols using radix sort. Rotations are given numbers $V[i]$ like in the KMR algorithm. In array $I[0..n)$ indexes of rotations sorted by $<_k$-order of rotations. Next all groups are traversed from left in order of $I[i]$, note that rotations from the same group form continoues interval in I, to sort rotations in groups by the second symbols. If $I[0] = i$, R_{i-1} is the smallest rotation among the group $v = V[i-1]$. We can move rotations to their correct position according to first two symbols. The number of iterations is at most $log\, n$ and each iteration can be done in $O(n)$ time, therefore this algorithm works in $O(n\, log\, n)$ time.

4.3 Sadakane's algorithm

Sadakane algorithm [10, 8] uses comparison based algorithm to sort rotations in group in each iteration instead of radix sort like approach used by Manber and Myears. MM algorithm passes in each iteration whole array I to encounter all keys, even of already sorted rotations. On real life data most rotations get sorted in first few passes. Sadakane's algorithm is faster in most cases than MM, because it skips sequences of already sorted rotations. When comparison based qsort is used, Sadakane proposed using Bentley-Sedgewick's tripartite qsort algorithm, time of each pass depends on the number of remaining unsorted rotations.

After i iterations, rotations are sorted according to their first $k = 2^i$ characters. In array $I[0..n)$ rotation in $<_k$-order are stored and in array $V[0..n)$ we hold group numbers. At the end of algorithm I meats conditions of RA - rotation array. If there are no two equal rotations, at the end all values in V are distinct and V is inverse of I and vice versa.

Rotations which have the same prefix of length k form group and have

Algorithm 2 Sadakane

sort rotations according to their first character using radix sort
create groups and set their sizes and set $k = 1$
while $k < n$ **do**
  sort unsorted groups according to $<_\mathrm{k}$ order
  split groups
  combine consecutive sorted groups $k = 2k$
end while

equal $V[i]$ values after end of iteration. Since $V[i]$ and $I[i]$ are updated only when order of rotations could have possibly changed, we distinguish whether group is unsorted. A group is called unsorted if it contains two rotations that are equal according to $<_k$ order. Each sorted group starts as group containing one rotation. But later, because we want to skip sorted groups, we merge consecutive sorted groups. So when R_i gets into sorted group v, $v = V[i]$ and $I[j] = i$, then values of $V[i]$ and $I[j]$ remain the same and consistent with $<_k$ order in V and I in all following iterations.

Throughout whole algorithm array V has following properties:

$$R_i < R_j \rightarrow V[i] \leq V[j]$$

$$R_i \neq_k R_j \rightarrow V[i] \neq V[j]$$

where again k depends on the iteration. Note that $V[i]$ and $V[j]$ can have different values even if $R_i =_k R_j$. In addition to arrays I and V array S is used to store sizes of groups. $V[i]$ represents the number of rotations that are smaller than R_i according to $<_k$ order and $V[I[i]] = i$ if R_i is in sorted group.

First group starts at $v = 0$ and its size is $S[0]$. Qsort runs on rotations $I[0..S[0] - 1]$ using $V[i + k]$ as key for R_i. Next group starts at $v = S[v]$. If consecutive rotations $R_{I[i]}$ and $R_{I[i+1]}$ in the same group have different keys $V[I[i] + k] \neq V[I[i + 1] + k]$ we split the group. See Algorithm 2.

4.4 Larsson's algorithm

4.4.1 Description

Larsson in his paper [8] shows how to get rid of array S holding sizes of groups in Sadakane's algorithm. In addition he proves that using tripartite quick sort algorithm, for example the mentioned Bentley-Sedgewick's, the Sadakane's algorithm is guaranteed to run in $O(n \log n)$ time.

The algorithms are very similar, but Larsson proposes using V slightly differently. Instead of $V[i]$ representing the number of rotations smaller than R_i according to $<_k$ order as in Sadakane's algorithm, $V[i]$ is the number of rotations less of equal than R_i at the end of iteration. At all times for unsorted groups $V[i]$ is the end of the group - $R_{I[V[i]]}$ is the last rotation in the group. Consecutive single size groups (they are consecutive in array I) are handled as follows: if l is the first position ($R_{I[l]}$the first rotation) in the sequence and r the last, we interchange the values of $V[I[l]]$ and $V[I[r]]$; for all other rotations in the sequence $V[I[i]] = i$. Note that for all rotations in sorted groups, including the first and the last in sequence of sorted groups, $V[I[V[I[i]]]] = i$. So the Larsson's trick is that $V[i]$ points to the last rotation in group and $V[I[V[i]]]$ is the key for rotation i. This way we omit an array for storing sizes of groups.

To split groups, start at the left end of the group and move to the right comparing the key with the key of next element ($V[I[V[I[i] + k]]]$ and $V[I[V[I[i + 1] + k]]]$). Since we do this just after sorting the group by these keys, the key of $R_{I[i+1]}$ can be either equal or larger than key of $R_{I[i]}$. If the key of next element is larger we split the group. We need to remember where current group started and whether we are processing sequence of one element groups. This enables us to set correct values into V after whole group is passed and we encounter another one. If group is sorted and l is the first position in the group, in that case $l = V[I[V[I[l]]]] \leq V[I[l]]$, we can pass whole group in constant time because $V[I[l]]$ points to position of last rotation in the group. If $l < V[I[V[I[l]]]] = V[I[l]]$ we encountered unsorted array and use comparison based sort on range $[l..V[l]]$.

4.4.2 Memory requirements

The most memory demanding phase is the main one, where we need to hold in memory arrays $V[0..n)$ and $I[0..n)$. In the first phase we radix sort the input and for that we need original string $X[0..n)$, array for indexes $I[0..n)$ and array for storing frequencies $F[0..|\Sigma|)$. Or we can use comparison based algorithm, requiring only X and I, to sort the rotations according to first characters. Array X requires $n \cdot bytesize(|\Sigma|)$ bytes, while V and I require $n \cdot bytesize(n)$ bytes. Hence for first stage we need $n \cdot (bytesize(n) + bytesize(|\Sigma|)) + c$ and if use radix sort $|\Sigma| \cdot bytesize(n)$ bytes more. In main stage the memory required is $2n \cdot bytesize(n) + c$ plus $n \cdot bytesize(|\Sigma|)$ for X that we need till end of algorithm. So as long as $n \geq |\Sigma|$, the total memory requirements are $2n \cdot bytesize(n) + n \cdot bytesize(|\Sigma|) + c$.

4.4.3 Time complexity

Trivial bound for Larsson's algorithm is $O(n \cdot log^2 n)$. We do at most $log\, n$ iterations and in each iteration we qsort interval of length at most n. We show that the time complexity in worst case is $O(n \cdot log\, n)$ when using tripartite qsort with *perfect median* as the comparison based sort. Tripartite qsort partitions the input into elements smaller than pivot, elements equal to pivot and elements greater than pivot. Smaller and greater elements are sorted recursively.

Consider the sorting process in Sadakane's algorithm as implicit construction of ternary tree. Each node corresponds to one *partitioning* in tripartite qsort. Left child of the node corresponds to elements smaller than pivot, middle child to elements equal to pivot and right child to elements larger than pivot. A leaf represents end of partitioning when only one element is left.

We want to show that path from any leaf to root has length $O(log\, n)$. If a node v is middle child of its parent and rotations were sorted by first k symbols, than the rotations in children of v will be sorted by first $2k$ symbols. Hence there can be at most $log\, n$ middle nodes between root and leaf.

If a node v is left or right child of its parent, than the number of rotations in it is at most half of the number of rotations in its parent (perfect median). Again the number of nodes of this type between root and any leaf can be at most $log\, n$.

Now we see that this implicit tree has depth $O(log\, n)$. To see that the work done in each depth is $O(n)$ recall that partitioning in qsort requires work linear in number of elements (median can be found in linear time). When we partition rotations each rotation is assigned to exactly one group. Therefore no two nodes in the same level share a rotation, and consequently, sum of all rotations in nodes in any fixed level is at most n. Hence in each level at most $O(n)$ work is done yielding $O(n \cdot log\, n)$ time complexity for whole algorithm. This bound is sharp due to using comparison based algorithm.

In the trivial bound $n^2 \cdot log\, n$ we bounded number of iterations by $log\, n$. Note that we can finegrain this bound to $log\, MML$. The number of elements that we have to qsort at k-th iteration depends on the number of rotations that have ML with some other rotations at least 2^k and this value is strongly correlated with AML [8].

4.5 Seward's algorithm

4.5.1 Description

The main idea used in algorithm due to Seward called "copy" in his paper [11] is to use almost sorted 1-level buckets to omit sorting some 2-level buckets. K-level buckets are groups of rotations that share the same prefix of length k.

Bucket c is 1-level bucket of rotations starting with character c and bucket c_1c_2 is 2-level bucket consisting of rotations starting with prefix c_1c_2. We store in array $F[0..|\Sigma|^2]$ the start positions of 2-level buckets in index array $I[0..n)$. If we have sorted bucket c_2 we can determine order of buckets c_1c_2, $c_1 \neq c_2$ by passing bucket c_2. If rotations R_i and R_j start with the same letter then their relative order is given by relative order of R_{i+1} and R_{j+1}. So we know what is the first rotation in bucket c_1c_2, $c_1 \neq c_2$ once we know the first rotation in bucket c_2. We are quaranteed that when passing bucket c_2 we go through all rotations from bucket c_1c_2 and hence set order of all rotations from that bucket.

Another trick is the use of buckets c_2c_3, $c_2 \neq c_3$ to sort bucket c_2c_2. If R_i is the first rotation in bucket c_2c_2 then R_{i+1} is the first rotation in bucket c_2. But if the third letter in that rotation x_{i+2} is different from c_2 we already have sorted bucket c_2x_{i+2}. Hence we get the first rotation if we pass buckets c_2c_3, $c_3 < c_2$. (Given that third symbol of first rotation in c_2c_2 is c_3, $c_3 < c_2$. If $c_3 > c_2$ we will get to this rotation during passing buckets c_2c_3, $c_3 > c_2$ from right.) If there is rotation R_i that starts with longer sequence of equal symbols for instance $c_2c_2c_2$ then there is rotation that starts with sequence of equal symbols that is shorter by one symbol R_{i+1} (except when input string consists of just one distinct symbol). And we will hit the rotation R_{i+1}before hitting R_i, so we are guaranteed to know the order of any rotation in bucket c_2c_2 when we need it. We only need to check when passing bucket c_2 from left whether the current rotation is smaller than virtual rotation $c_2c_2..c_2$ consisting of infinite sequence of symbols c_2.

Again $X[0..n)$ is input string, $I[0..n)$ index array. Array $F[0..|\Sigma|^2]$ is used to store borders of 2-level buckets and arrays $S[0..|\Sigma|]$ and $E[0..|\Sigma|]$ for storing start and end respectively, of unfilled parts of buckets c_1c_2 for fixed c_2.

Different strategies for choosing the bucket c_2, note that we qsort buckets c_2c_3, can be used. The most simple approach is to use lexicographic order of symbols. Better approach is to choose the bucket in which total number of rotations in unsorted buckets c_2c_3 is the smallest.

Algorithm 3 Seward

perform 2-level bucket sort (F holding starting positions of 2-level buckets in I)
  while $\exists$ unsorted 1-level bucket c_2 **do**
    for all $c_3 \neq c_2$ **do**
      sort c_2c_3 buckets using qsort
    end for
    $S[c_1] \Leftarrow$ start of bucket c_1c_2, $E[c_1] \Leftarrow$ end of c_1c_2
    $j \Leftarrow$ start of bucket c_2
    while $j < S[c_2]$ **do** $\{S[c_2]$ underapproximates $c_2c_2..c_2$ boundary$\}$
      $c_1 \Leftarrow X[(I[j]-1) \mod n]$
      $I[S[c_1]] \Leftarrow (I[j]-1) \mod n$
      $j \Leftarrow j+1$
      $S[c_1] = S[c_1]+1$
    end while
    $j \Leftarrow$ end of bucket c_2
    while $j > E[c_2]$ **do** $\{E[c_2]$ overapproximates $c_2c_2..c_2$ boundary$\}$
      $c_1 \Leftarrow X[([I_j]-1) \mod n]$
      $I[E[c_1]] \Leftarrow (I[j]-1) \mod n$
      $j \Leftarrow j-1$
      $E[c_1] \Leftarrow E[c_1]-1$
    end while
  end while

4.5.2 Memory requirements

We need to hold in memory arrays X requiring $n \cdot bytesize(|\Sigma|)$ bytes, I taking up $n \cdot bytesize(n)$ bytes and F for which we need $|\Sigma|^2 \cdot bytesize(n)$ bytes. The memory required for F renders Seward's algorithm useless for large alphabets. Array F is needed due to use of 2-level buckets and the need to store where the buckets begin. And finally we use arrays S and E of size $|\Sigma| \cdot bytesize(n)$. In total we need $n \cdot (bytesize(n) + bytesize(|\Sigma|)) + |\Sigma|^2 \cdot bytesize(n)$.

4.5.3 Time complexity

Time complexity of Seward's algorithm, like that of any comparison based algorithm, is $O(AML \cdot n \cdot log\, n)$, neglecting the radix sort phase, where AML is the average match length of input. The most time demanding part of the algorithm is sorting 2-level buckets using qsort for small alphabets. Total time complexity is including radix sort $O(AML \cdot n \cdot log\, n\ +\ |\Sigma|^2)$.

Consider the binary tree implicitly constructed during qsort. Nodes are calls of qsort that partition rotations into two groups; smaller than pivot and greater than pivot. No rotations get into both groups. Hence the sets of rotations in nodes, for any fixed tree depth, are disjoint. In each partitioning we compare all rotations in the node with pivot. For every rotation $R_{I[i]}$ the longest match length is either with rotation $R_{I[i-1]}$ or with $R_{I[i+1]}$. So we can use match length of neighbor as upper bound on number of symbols compared when comparing rotation with pivot. At each depth of tree we did $O(n)$ comparisons of rotations with pivot of respective node. Now recall that AML is defined as average of match lengths of neighbors to see that at any fixed depth of tree we compared $O(AML \cdot n)$ symbols. Since the depth of the tree is $O(log\, n)$ we get that the algorithm runs in $O(AML \cdot n \cdot log\, n)$ time. The problem is that for highly repetive inputs AML can range up to n.

Consider $X =$`ababab...abc` as an example of input with $AML = \Theta(n)$. The Seward's algorithm on this input calls qsort on bucket `ab` or `ba`. Both contain approximately $n/2$ rotations and the first $n/4$ rotations $\{R_{2i+q};\, i \in [0..n/4)\}$ ($q = 0$ for bucket *ab* and $q = 1$ for bucket *ba*) have pairwise match length at least $n/4$. Hence we will need to compare $\Theta(n^2)$ symbols for each depth of tree resulting in total runtime $\Omega(n^2 log\, n)$ for this input. This shows that the upper bound $n^2\, log\, n)$ is sharp. For input $X =$`aa..ab` which has $AML = \Theta(n)$ Seward's algorithm will not qsort bucket `aa` containing almost all rotations. Note that in fact the algorithm will run in $O(n)$ for this input. This shows that AML can not be used as lower bound.

As we have shown total time complexity is $O(AML \cdot n \cdot log\, n + |\Sigma|^2)$.

4.6 Our Seward based 1-level bucket algorithm

The typical output of parser of program XML creates input with $2^{14} < |\Sigma| < 2^{18}$ and $2^{20} < n < 2^{26}$. Hence both time and memory requirements due to 2-level buckets creation would dominate over the rest of algorithm ($|\Sigma|^2 > n \cdot log\, n$). Therefore we looked for an algorithm that would only use 1-level buckets. The algorithm we came up with uses the same idea of copying pointers. We hold pointers to unsorted parts of 1-level buckets and since we call qsort on buckets from smallest to largest we can copy pointer for rotation R_i for rotation R_{i-1} if $x_i < x_{i-1}$. But since we do not write all rotations of given bucket at once, we need to put the replaced rotation where the other rotation has been in I. For this we need array C which is the inverse of index array $C[I[i]] = i$. Another disadvantage compared to Seward's algorithm is that buckets cc are not constructed in linear time, but are sorted using qsort.

Memory requirements differ from Seward's by the use of 1-level buckets and array $C[0..n]$. So we get $n \cdot (2 \cdot bytesize(n) + bytesize(|\Sigma|)) + 2 \cdot |\Sigma| \cdot bytesize(n)$.

Time complexity is again very similar. Again the difference is in use of different level of buckets. We get $O(AML \cdot n \cdot log\, n + |\Sigma|)$.

4.7 Itoh's algorithm

The main idea of the Itoh's algorithm is to divide rotations into two groups so that we only have to sort one group using comparison based algorithm. The order in the other group and of all rotations can be then constructed in linear time. Group A contains rotation R_i iff $x_i > x_{i+1}$ and group B contains all other rotations. Consider rotations divided into 1-level buckets. Rotations from group A are smaller than rotations from group B if they start with the same symbol. Now if we pass group B in ascending order we can determine the order of rotations from group A.

Memory requirements are $n \cdot (2 \cdot bytesize(n) + bytesize(|\Sigma|)) + 2|\Sigma| \cdot bytesize(n)$ and time complexity is $O(AML \cdot n \cdot log\, n + |\Sigma|)$.

4.8 Kao's modification of Itoh's algorithm

We again divide rotations into groups, but we do not qsort whole group at once. Rather we qsort unsorted part of group from one bucket. We always choose between the leftmost and rightmost bucket whichever is smaller. Rotation R_i belongs to:

- Type A: iff $x_i > x_{|i+1|}$
- Type B: iff $x_i = x_{|i+1|}$ and $R_{|i+1|}$ is Type A or Type B
- Type C: iff $x_i = x_{|i+1|}$ and $R_{|i+1|}$ is Type C or Type D
- Type D: iff $x_i < x_{|i+1|}$

We use arrays $BucketSizeA$, $BucketSizeB$, $BucketSizeC$, $BucketSizeD$, all of them have size $|\Sigma|$ for holding sizes of buckets. Arrays $BucketPtrA$, $BucketPtrB$, $BucketPtrC$, $BucketPtrD$ point to unsorted parts of buckets in $I[0..n)$. $BucketPtrA$, $BucketPtrB$ point to the left end of unsorted bucket while $BucketPtrC$, $BucketPtrD$ point to the right end of buckets. In addition we use array $C[0..n]$ inverse of I ($I[C[i]] = i$) for finding position in I for given rotation. Note that rotations starting with the same symbol of type A are smaler than all rotations of type B those than type C and finally all groups contain smaller rotations than group of type D. See Algorithm 4.8.

Note that the leftmost unprocessed type A bucket $l = left$ is sorted, because the second symbols are smaller than first. Hence when we get to l we have already processed all smaller symbols from left. So while processing rotations R_{i-1} we had to hit all from type A bucket starting with symbol l. (The only exception would be if input X would contain only one symbol.) If l is the smallest symbol then type A bucket is empty. On the other hand some rotations of type D could have been assigned their index, but not necessarily all because $r = right$ can at the moment be far from l. So when we have been assigning the indexes of rotations from this type D bucket, we had to switch places of rotations at that index. For switching rotation R_i with rotation R_j we need to know position of R_i in I - we have it stored in $C[i]$.

Memory requirements are $n \cdot (2 \cdot bytesize(n) + bytesize(|\Sigma|)) + 4 \cdot |\Sigma| \cdot bytesize(n)$ and time complexity is $O(AML \cdot n \cdot log\, n + |\Sigma|)$.

4.9 Kärkkäinen and Sanders' algorithm

4.9.1 Description

This algorithm is the first one to directly construct suffix array without constructing suffix tree. It recursively sorts suffixes at positions $i\, mod\, 3 \neq 0$. Then it sorts positions $i\, mod\, 3 = 0$ and finally merges those two groups. It implicitly uses the idea of *doubling technique*, but it generally multiplies the sorted length by higher number (in this case 3). We describe version of the algorithm DC3 that divides suffixes into three groups and is elegant.

Algorithm 4 Kao

{Bucket c is bucket of rotations starting with character c. We divide this bucket further into buckets of type A, B, C and D.}
radix sort I by first symbols
set $BucketSize(A, B, C, D)$ arrays
{$BucketSizeA[c]$ holds the number of rotations belonging to group A starting with character c}
set $BucketPtr(A, B, C, D)$ arrays
{$BucketPtrA[c]$ points to left end of rotations belonging to group A starting with character c}
$left \Leftarrow 0$
$right \Leftarrow |\Sigma| - 1$

while $\exists$ unsorted group **do**
  if unsorted part of bucket $left$ of type $D \leq$ unsorted part of bucket $right$ of type A **then**
    Pass whole sorted bucket of type A upto end of type B bucket from left setting indexes of rotations $R_{i-1} \geq R_i$.
    Qsort the unsorted part of type D bucket.
    Pass whole type D bucket from right upto start of type C bucket setting indexes of rotations R_{i-1} from type C bucket.
    Pass type C and type D from left setting indexes of rotations $R_{i-1} > R_i$.
    $left \Leftarrow left + 1$
  else
    Mirror of actions in the first case.
  end if
end while

In previous algorithms we have explained how to use them to construct index array I storing order of rotations for input string X. However for this algorithm, straightforward switch from sorting suffixes to sorting rotations in not possible. So we can only use it to construct suffix array. Recall that suffix array of $X\$$ is equivalent to index array of $X\$$.

We assign suffixes $S_i : i \, mod \, 3 = 1$ (that we call group 1 suffixes) numbers $rank(S_i) = R'_1[i/3]$ so that they correspond to $<_3$ order: $S_i <_3 S_j \leftrightarrow R'_1[i/3] < R'_1[j/3]$ and $S_i =_3 S_j \leftrightarrow R'_1[i/3] = R'_1[j/3]$. In the rank array R'_1, number corresponding to S_i is followed by number corresponding to S_{i+3}. So to compare S_i with S_j we can instead compare suffixes $R'_1[i..n/3]$ with $R'_1[j..n/3]$ from array R'_1. If there are no two equal numbers in R'_1 we can

construct suffix array SA_1 for R'_1 by single pass of R'_1. If not, we use recursion to find suffix array for R'_1. Note that we always triple the order by which the suffixes are sorted.

This itself would not suffice. But due to the fact that the string is \$ terminated we can sort suffixes $S_i : i \, mod \, 3 \neq 0$ at the same time. For suffixes $S_i : i \, mod \, 3 = 2$ (group 2 suffixes) we create $R'{}_2$ the same way as we created $R'{}_1$ for group 1 suffixes. Now we concatenate them to obtain $R'_{12} = R'_1 \circ R'_2$. Note that the numbers have to be chosen so that the $<_3$ condition is met also for comparing S_i and S_j from group. We can use this array due to the fact that the last numbers in R'_1 and R'_2 are unique. So for suffix from R'_1 the order is determined before we look for number from R'_2. This holds since our string is \$ terminated and would not work with rotations. For technical reasons consider string to be terminated by \$\$.

Consider example $X = mississippi\$\$$. The triplets for group 1 suffixes are $[iss, iss, ipp, i\$\$]$ and $[ssi, ssi, ppi]$ for group 2. Array $R'_{12} = [3, 3, 2, 1] \circ [5, 5, 4]$ meets the conditions for $<_3$ order of group 12 suffixes. In the next recursive call the triplets would be $[321, 554]$ and $[215, 54\$]$. Now all the triplets would be unique and no more recursion is required.

Following is the description of implementation of the idea described.

Step 0: Construct a sample.

For $k = 0, 1, 2$ $R_k = [i \in [0, n]; i \, mod \, 3 = k]$. Let $R_{12} = R_1 \circ R_2$ be the array of sample positions and S_R the set of sample suffixes.

Example. $R_1 = [1, 4, 7, 10]$, $R_2 = [2, 5, 8]$, $R_{12} = [1, 4, 7, 10, 2, 5, 8]$.

Step 1: Radix sort sample suffixes.

First we radix sort array R_{12} using first three symbols of corresponding suffix as keys.

(First we call stable radix sort using third symbol as key: $key(R[i]) = S_i[2] = x_{i+2}$ to get array SA'''_{12}.

Second call of radix sort on SA'''_{12} using $key(SA'[i]) = S_i[1] = x_{i+1}$ to get SA''_{12}.

Third call of radix sort on SA''_{12} using $key(SA'[i]) = S_i[1] = x_{i+1}$ to get SA'_{12}.)

Now we have array SA'_{12} similar to suffix array, but it just represents $<_3$ order of suffixes.

Pass the SA'_{12} and set ranks of suffixes into R'_{12}.

(Start with $rank = 1$ and if $S_{SA'''[i]} <_3 S_{SA'''[i+1]}$ increase $rank$ by one. If suffix $S_{SA'''[i]}$ is from group 1: $R'_{12}[SA'''[i]/3] = rank$. If the suffix is from group 2: $R'_{12}[SA'''[i]/3 + n/3] = rank$.)

After this step $R'_{12} = [3, 3, 2, 1] \circ [5, 5, 4]$.

If all ranks are different then array SA'_{12} is the suffix array for group 12. Otherwise *recursively call* algorithm DC3 with $R' \circ \$\$$ as input string.

After recursive call $R'_{12} = [4,3,2,1] \circ [6,5,4]$ and $SA'_{12} = [3,2,1,0,6,5,4]$.

Step 2: Sort nonsample suffixes.

To sort suffixes $S_i : i \, mod \, 3 = 0$ (group 0) we use ranks of already sorted suffixes. To compare two suffixes from group 0 use pair $(x_i, rank(S_{i+1}))$as key for S_i. Recall that for S_i from group 0, $rank(S_{i+1})$ is defined in R'_{12}. Since $S_i \leq S_j \leftrightarrow (x_i, rank(S_{i+1}) \leq (x_j, rank(S_{j+1}))$ it is enough to use radix sort.

Step 3: Merge.

When merging group 12 with group 0 we distinguish two cases. To compare S_i from group 0 with

- S_j from group 1: $S_i \leq S_j \leftrightarrow (x_i, rank(S_{i+1})) \leq (x_j, rank(S_{j+1}))$
- S_j from group 2: $S_i \leq S_j \leftrightarrow (x_i, x_{i+1}, rank(S_{i+2})) \leq (x_j, x_{j+1}, rank(S_{j+2}))$

4.9.2 Memory requirements

We need array X for storing input of size $n \cdot bytesize(|\Sigma|)$ throughout the whole algorithm. The phase of sorting sample requires storing input strings in recursive calls which requires $n \cdot bytesize(|\Sigma|)$ for array X in the top call which works with alphabet Σ plus $\sum_{i=1}^{\infty} (\frac{2}{3})^i \cdot bytesize(n)$ for recursive calls which work with alphabet of ranks. This is equal to $n \cdot bytesize(|\Sigma|) + \frac{7}{3} n \cdot bytesize(n)$.

Sorting non-sample suffixes requires original input array X, two arrays of size $\frac{n}{3} \cdot bytesize(n)$ and two arrays for already sorted group 12 of size $\frac{2}{3} n \cdot bytesize(n)$. Total of $n \cdot bytesize(|\Sigma|) + 2n \cdot bytesize(n)$ bytes.

The most memory demanding is the final merging. We need arrays X with input string, R_{12} with ranks of suffixes from group 12, SA_{12} - suffix array for group 12 suffixes, SA_0 - suffix array for group 0 suffixes and finally SA where we will store the resulting suffix array for input X. These arrays total to $n \cdot bytesize(|\Sigma|) + \frac{8}{3} n \cdot bytesize(n)$ bytes.

4.9.3 Time complexity

The algorithm devides suffixes into two groups and radix sorts them in $O(n)$. Then it recursively calls itself on two thirds of original data. Finally merging in linear time is performed. Hence the recursive equation is $T(n) = O(n) + T(\frac{2}{3}n)$ which has solution $T(n) = O(n)$. The number of recursive calls depends on MML.

Chapter 5

Reverse BWT for string transformed using suffix array

Burrows and Wheeler have invented the transform while working with suffix arrays [3]. The decoding as described in Section 3.2 uses the structure of rotations to find original string. For most inputs rotation array and suffix arrays are the same, but they can also be quite different. Recall that we form $\widetilde{X}$ from X by taking last characters of rotations in order in which they appear in index array I. We show how the rotation arrays and suffix arrays are related and propose algorithm for decoding transformed string $\widetilde{X}$ created using suffix array instead of rotation array. The preference of suffixes over rotations is due to better running times. In addition some algorithms, such as linear algorithm due to Kärkkainen and Sanders described in Section 4.9, need to work with suffixes. We show how to avoid using $X\$$ as input string. We use example string `dabraca` for illustration.

Lemma 5.0.1. *Rotation array and suffix array for string $X\$$ are equal.*

Proof. When we compare two rotations, their relative order is always determined when we encounter \$ or sooner since \$ is not in Σ. Hence we compare

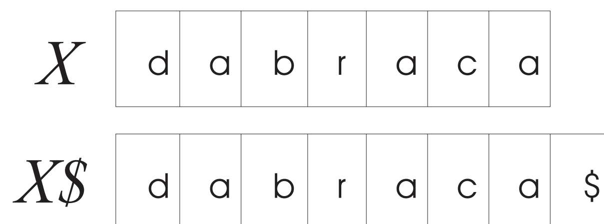


Figure 5.1: String X=`dabraca`.

RA_X		1	4	6	2	5	0	3
SA_X		6	1	4	2	5	0	3
$SA_{X\$}$	7	6	1	4	2	5	0	3

Figure 5.2: Rotation array and suffix array for `dabraca`.

rotations and suffixes the same way. □

We see that we can perform BWT using suffix array instead of rotation array if we \$ terminate the input string. The disadvantage is that we have to increase the length of the string. Consequently transformed string is longer than input string and after decoding we have to truncate the string to get rid of the terminal symbol \$. When we split input string into blocks, which is common to lower memory requirements and running time, we can not place the transformed string $\widetilde{X}$ back into its place in array holding the input string, since $\widetilde{X}$ is longer than X.

Lemma 5.0.2. *Let SA_X be suffix array for string X. Then suffix array $SA_{X\$}$ for string $X\$$ is given by:*

$$SA_{X\$}[0] = n,\ \forall i \in [1..n] :\ SA_{X\$}[i] = SA_X[i-1]$$

Let $SA_{X\$}$ be suffix array for terminated string $X\$$. Then suffix array SA_X for string X is given by:

$$\forall i \in [0..n) :\ SA_X[i] = SA_{X\$}[i+1]$$

Proof. First entry in $SA_{X\$}$ is the position of \$. Relative order of all suffixes from X is the same in $X\$$ since \$ placed at the end of string does not affect the order. □

Lemma 5.0.3. *Let $\widetilde{X\$}$ be the transformed string for $X\$$ and $\widetilde{X}$ transformed string for X. Then*

$$index_{X\$} = index_X + 1,\ \widetilde{X\$}[0] = x_{n-1},\ \widetilde{X\$}[index_{X\$}] = \$,\ \widetilde{X}[index_X] = x_{n-1}$$

and for the rest

$$\forall i \in [1..n] \setminus \{index_{X\$}\};\ \widetilde{X\$}[i] = \widetilde{X}[i-1]$$

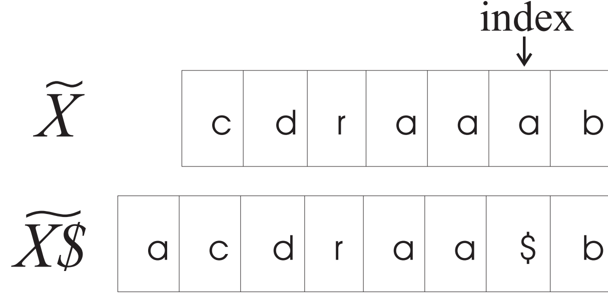


Figure 5.3: Transformed string for `dabraca`.

Proof. This is direct consequence of previous lemma. Note that the shift is due to suffix $S_n = \$$ which is smaller than all other suffixes. □

We use the previous lemmas to propose a way to decode transformed string $\widetilde{X}$ created using suffix array.

Theorem 5.0.4. *We can decode $\widetilde{X}$ created using suffix array into original string X.*

Proof. We have shown that $SA_{X\$} = RA_{X\$}$ and hence $\widetilde{X\$}$ is the same regardless if we use suffix array or rotation array. We have also shown how are $\widetilde{X}$ and $\widetilde{X\$}$ related when we use suffix arrays. Now we use this similarity to modify Algorithm 1 to perform decoding of $\widetilde{X}$ created using suffix array. For the resulting algorithm see Algorithm 5.

After we have counted the frequencies of characters and set array $C[0..|\Sigma|]$ so that $C[c]$ stores number of suffixes starting with character smaller than `c` the relation is following: $C_{\widetilde{X}}[c] = C_{\widetilde{X\$}}[c] - 1$. The second phase of decoding algorithm sets predecessors in array $T[0..n]$ by passing $\widetilde{X}$ from left. We set $T[i] = C[\widetilde{x_i}]$ and we increase $C[\widetilde{x_i}]$ by one - this is were we use the fact that if we shift all rotations to right, rotations starting with the same character appear in the same relative order. See Section 3.2.2.

For $\widetilde{X\$}$ we hit x_{n-1} as the very first character at position $i = 0$ ($\widetilde{X\$}[0] = x_{n-1}$) and we increase $C[x_{n-1}]$, while for $\widetilde{X}$ we hit x_{n-1} at $\widetilde{X}[index_X]$. If we increase $C[x_{n-1}]$ for $\widetilde{X}$ at the beginning of the pass and do not increase $C[x_{n-1}]$ at position $index_X$ we get the following relation of predecessor arrays: $T_{\widetilde{X}}[i] = T_{\widetilde{X\$}}[i+1] - 1$ for all i except $i = index_X$. Since x_{n-1} is the first character in $\widetilde{X\$}$ its predecessor will be equal to $indexC \equiv T_{\widetilde{X\$}}[0] = C[x_{n-1}]$ just after the first phase (before we have been increasing it by one when

Algorithm 5 Reverse BWT (decoding)

```
{C[0..Σ − 1] array for frequencies of characters initialized to 0}

{first phase – find first characters of rotations}
for i = 0 to n − 1 do
   C[X̃[i]] ⇐ C[X̃[i]] + 1
end for
for i = 0 to |Σ| do
   j ⇐ C[i]
   C[i] ⇐ sum
   sum ⇐ sum + j
end for

{second phase – build predecessor list}
index_C ⇐ C[X̃[index]]
C[X̃[index]] ⇐ C[X̃[index]] + 1
for i = 0 to index − 1 do
   T[i] ⇐ C[X̃[i]]
   C[X̃[i]] ⇐ C[X̃[i]] + 1
end for
T_index ⇐ index_C
for i = index + 1 to n − 1 do
   T[i] ⇐ C[X̃[i]]
   C[X̃[i]] ⇐ C[X̃[i]] + 1
end for

{third phase – form output}
for i = n − 1 downto 0 do
   X[i] ⇐ X̃[index]
   index ⇐ T[index]
end for
```

	$	a	b	c	d	r
C_X		0	3	4	5	6
$C_{X\$}$	0	1	4	5	6	7

Figure 5.4: Array C after first phase of Algorithm 5.

T_X		4	5	6	1	2	0	3
$T_{X\$}$	1	5	6	7	2	3	0	4

Figure 5.5: Array T after first phase of Algorithm 5.

assigning predecessors). So we set $T_{\widetilde{X}}[index_X] = indexC$ and the predecessor array $T_{\widetilde{X}}$ looks as follows:

$$\forall i \in [0..n) \setminus \{index_X\} : T_{\widetilde{X}}[i] = T_{\widetilde{X\$}}[i+1] - 1$$

$$T_{\widetilde{X}}[index_X] = T_{\widetilde{X\$}}[0]$$

Note that $T_{\widetilde{X\$}}[index_{X\$}] = 0$ and hence when decoding $\widetilde{X\$}$ we get to position 0 just after position $index_{X\$}$. The position 0 in $T_{\widetilde{X\$}}$ is correlated with position $index_X$ in $T_{\widetilde{X}}$ and hence from this point the decoding will follow the same pattern for both $\widetilde{X}$ and $\widetilde{X\$}$. □

T_X	4	5	6	0	1	2	3

Figure 5.6: Array T that we would obtain using Algorithm 1 for reverse BWT for rotations.

Chapter 6

Improving performance of BWT

In this chapter we discuss various aspects of implementation that influence the performance of presented algorithms. The factor that has biggest influence on performance of BWT is alphabet. We show that for large textual files using alphabet of words greatly improves the speed of BWT and in addition improves the compression ratio.

We have already shown that we can use suffixes instead of rotations. Now we are about to show that algorithms described are faster when they use suffixes and that compression ratio is almost the same for suffixes and rotations.

We have implemented several comparison based algorithms and tested them for performance. Results show that careful choice of appropriate sorting algorithm as subroutine for BWT coding algorithms can improve the running time by tens of percent.

Finally, and from first glance most importantly, we need to choose BWT coding algorithm. We do this choice as last, because we first optimize the algorithms before making decisions on their performance. Decisive factor for appropriate algorithm is AML of the input. For repetitive inputs, we should use algorithm with good asymptotic complexity such as Larsson's algorithm or Kärkkäinen and Sanders' algorithm.

6.1 Testing Environment

All presented results have been obtained using application XBW [4]. This program is highly modular and supports number of parameters. For measuring compression ratio, we used BWT over specified alphabet followed by move-to-front (MTF) and run-length-encoding (RLE). For RLE parameter –RLE=3 has been used. For details, see documentation of the program.

All algorithms have been implemented for various alphabet sizes using 1, 2 and 4 byte numbers in an attempt to minimize cache misses. We also implemented several subroutines that are used by the algorithms. Namely we implemented half a dozen quick sort algorithms, and three comparison functions for comparing strings. In addition all algorithms expect Kärkkainen and Sanders' have been implemented for both sorting rotations and suffixes.

Run time has been measured on Linux system and we list user time plus system time. Hence we exclude time for waiting for I/O devices. The code has been compiled using gcc version 4.1 with optimization parameter -O3. The personal computer on which we run the tests had processor Athlon 64 X2 3800+ and 2GB of RAM.

6.2 Corpus Summary

We use files from several corpora for testing. From Calgary corpus we use files *book2, paper1* and *progc*. Two of them contain English texts and the last one is program in C language. From Cantebury corpus we use file *E.coli* containing DNA sequence, file *bible.txt* with Bible in English and file *world192.txt* containing CIA World Factbook from 1992. We have taken some larger files from corpus Silesia. Namely file *dickens* with texts by Dickens' and file *webster* with Webster's dictionary. The largest file called *enwik8* contains pages from Wikipedia and is taken from compression contest. Finally we have two XML files containing hundreds of web pages. File *xml_ cz* contains web pages in Czech and file *xml_ en* web pages in English.

In Table 6.1 we show basic information about files. AML and MML are measures of repetitiveness of files (see Section 2.2.3 and Section 2.2.4). These two parameters together with size of the file are the most important factors that influence run time of BWT.

6.3 Notation in Results

We try to use consistent notation throughout the tables. For most tables we list the results for whole corpus. In some tables we list results for each file specifically while in others for whole corpus. When not specified, results are for whole corpus.

Results for compression ratio are in bits per byte. Results for run time are in seconds and represent sum of user time and system time on Linux system.

For algorithms we use following notation:

File Name	Size in Bytes	AML	MML
E.coli	4638690	17.4	2815
bible.txt	4047392	14.0	551
book2	610856	9.6	246
dickens	10192446	56.1	10263
enwik8	100000000	16.4	5317
paper1	53161	8.0	104
progc	39611	8.3	156
webster	41458703	34.3	11293
world192.txt	2473400	23.0	559
xml_cz	19100318	2701.0	54814
xml_en	14493168	1639.1	53402

Table 6.1: Corpus Summary

Basic – Algorithm that runs one or two level bucket sort. Then each bucket is sorted by comparison based sort. We show results for various qsorts.

Sada – Sadakane's algorithm improved by Larsson. See Section 4.4.

Sew1level – Our modification of Seward's algorithm that uses 1 level buckets. See Section 4.6.

Itoh – Itoh's algorithm improved by Kao. See Section 4.8.

Seward – Seward's algorithm. See Section 4.5.

KS – Kärkkäinen and Sanders' algorithm. See Section 4.9.

6.4 Comparing rotations vs. comparing suffixes

Suffix arrays and rotation arrays are very similar, but are usually not identical. Hence also the transformed strings differ slightly. In Table 6.2 you can see that the differences in compression ratio are really negligible. More careful analysis of separate files shows as well that the difference is irrelevant.

When comparing two rotations we have to keep track of how many characters we have compared and also to check that we did not get over the end of array when we have to continue from the beginning. Simple implementation of comparison function can be found in Algorithm 6. In each iteration we

Bits per Byte	Bytes	2 Bytes	Syllable	Word
rotation	**1.697233**	1.776992	1.644089	**1.663771**
suffix	1.697234	**1.776991**	**1.644088**	1.663780

rotation – result for transformed string using rotation array
suffix – result for transformed string using suffix array

Table 6.2: Compression Ratio of Rotations vs. Suffixes

Algorithm 6 Compare rotations - simple

```
int compare(int first, int second, VARTYPE X[], int n){
    if(first==second)
        return(0);
    for(i=0;i<n;i++){
        if(X[first++]!=X[second++])
            return(X[--first]-X[--second]);
        if(first>=n)
            first-=n;
        if(second>=n)
            second-=n;
    }
    return(0);
}
```

check three variables against n. Seward [11] proposed to extend the array X by small value k so that in each iteration we compare more characters from suffixes, but checking values $i, first, second$ only once. Seward measured up to 45% improvement when this trick has been used on direct comparison based algorithm. We have to set $X[n+i] = X[i \, mod \, n]$ for `i` $\in$ [`0..k`). See Algorithm 7 for details where we use $k = 2$ for illustration, but value around 10 yields better results. We have used $k = 8$ in our implementation.

Now we consider comparison of two suffixes which is faster than comparing rotations even in optimized version. We again use extension of array X where we place terminal value. Let all values from alphabet in X be positive. We set $X[n] = 0$ and we can use the function as in Algorithm 8 to compare two suffixes. Note that we only compare the characters from array and no extra variables.

In Table 6.3 we list results that clearly show that for all algorithms that

Algorithm 7 Compare rotations - unroll

```
int compare(int first, int second, VARTYPE X[], int n)
{
    if(first==second)
        return(0);
    if(X[first++]!=X[second++])
        return(X[--first]-X[--second]);
    if(X[first++]!=X[second++])
        return(X[--first]-X[--second]);
    if(first>=n)
        first-=n;
    if(second>=n)
        second-=n;
    while(i=0;i<n;i+=2){
        if(X[first++]!=X[second++])
            return(X[--first]-X[--second]);
        if(X[first++]!=X[second++])
            return(X[--first]-X[--second]);
        if(first>=n)
            first-=n;
        if(second>=n)
            second-=n;
    }
    return(0);
}
```

Algorithm 8 Compare suffixes

```
int compare(int first, int second, VARTYPE X[], int n){
    if(first==second)
        return(0);
    while(X[first++]==X[second++]);
    return(X[--first]-X[--second]);
}
```

Rot. / Suffix	BSORT	QSORT	Basic	Sada	Sew1level	Itoh	Seward	KS
R_S	1	3P	957.06	422.36	541.54	467.35	n/a	n/a
R_{UR}	1	3P	570.45	422.13	342.53	300.5	n/a	n/a
S	1	3P	**420.28**	427.54	**263.27**	**236.56**	n/a	407.3
R_S	2	3P	922.84	**355.34**	545.86	466.84	439.8	n/a
R_{UR}	2	3P	526.78	**355.56**	346.31	305.57	261.98	n/a
S	2	3P	**386.41**	361.11	**267.87**	**237.42**	**199.09**	407.3
R_S	1	SI	1180.81	**667.61**	657.58	561.56	n/a	n/a
R_{UR}	1	SI	803.23	**669.25**	461.44	398.93	n/a	n/a
S	1	SI	**511.77**	701.04	**311.28**	**275.76**	n/a	407.3
R_S	2	SI	1134.97	**490.62**	662.07	561.93	535.41	n/a
R_{UR}	2	SI	761.42	**488.31**	465.25	398.53	366.95	n/a
S	2	SI	**473.48**	514.39	**315.25**	**275.13**	**238.32**	407.3
R_{BS}	1	BS	801.54	422.38	459.3	398.45	n/a	n/a
S	1	BS	**623.33**	425.96	**254.05**	**225.56**	n/a	407.3
R_{BS}	2	BS	788.42	354.81	463.76	398.56	375.62	n/a
S	2	BS	**610.82**	360.09	**258.26**	**226.07**	**198.77**	407.3

Rot. / Suffix: R_S – rotation with simple cmp function, R_{UR} – rotation with unroll cmp function, R_{BS} – rotation with special cmp function for BS, S – suffix.
BSORT: 1 – one-level bucket sort, 2 – two-level.
QSORT: SI – simple q-sort, 3P – 3-part q-sort, BS – Bentley&Sedgewick's q-sort.

Table 6.3: Run Time of Rotations vs. Suffixes

```
// Suffix compare function 1
if(first==second)
  return(0);
while(X[first++]==X[second++]);
return(X[-first]-X[-second]);


// Suffix compare function 2
if(first==second)
  return(0);
while(X[first]==X[second]){
  first++;
  second++;
}
return(X[-first]-X[-second]);
```

Figure 6.1: Suffix Compare Variants

compare whole strings, all except Sadakane's and Kärkkäinen and Sanders', suffixes yield better results. Unroll version of comparison function for rotations really improves run time by around 40 % over with simple comparison function. Using suffixes we can decrease the run time by another 25 %.

We would like to warn that the effectivity of implementation of comparison functions is strongly compiler dependent. Since these functions are used very frequently it is important to inline them into qsort function in order to omit function calls. As curiosity and warning we give two versions of body for comparison functions for suffixes in Figure 6.1. Although semantically the same, difference in run time of whole BWT algorithm due to this difference has been over 20 %. [1] Even more interestingly one version is better for our implementation of tripartite qsort and the other for simple qsort. This emphasizes how much of the instructions are used on comparing the strings and that comparing long strings is bottle neck of algorithms.

6.5 Choosing q-sort

We have implemented several comparison-based sorting algorithms that are used in algorithms which we have presented in Chapter 4. Our goal has been to compare their performance for different BWT coding algorithms and to

[1] gcc 4.1

choose the appropriate ones to be used. Table 6.4 shows that this choice is important.

Method of choosing a pivot is important for all q-sort algorithms. Using median as pivot is costly, but on the other hand, choosing some predefined element as pivot often leads to bad partitioning. We have tested using predefined pivot, median of nine elements as pivot and method of switching among first, middle and last element in recursive calls due to Sedgewick. Another specific of our application is that comparison of keys is more costly than switching order of elements for which algorithm due to Bentley is well suited.

For Sadakane's algorithm improved by Larsson, that uses group number of rotation as key, 3-part q-sort due to Bentley and McIlroy [2] performs very well. This q-sort divides elements into three groups: elements smaller, equal, and bigger than pivot. It recursively sorts elements smaller and bigger than pivot, avoiding sorting again elements equal to pivot. For other algorithms, 3-part q-sort also works better compared to basic quick sort implementation we call simple q-sort. However this difference is not that big and is due to better choice of pivot and using insert sort when few elements are left.

We have also implemented algorithm due to Bentley and Sedgewick [1] which is aimed at sorting strings. Its core is the 3-part q-sort, but for strings it also recursively calls itself on elements with first character equal to pivot. In recursive call for elements that start with the same character it uses their second character as key. We refer to it as BS q-sort.

We can see that BS q-sort performs the best for rotation with simple comparison function. When we compare long blocks of string quickly as for suffixes, 3-part q-sort is the fastest among q-sorts we list in Table 6.4.

Note that Kärkkäinen and Sanders' algorithm is not using any of the qsorts. For Sadakane's algorithm 3-part qsort and BS qsort are equivalent since length of key is one.

6.6 Reverse of Input String

In decoding process, as described in 3.2, original string is decoded from end. For this reason it is common to perform BWT on input string that we reverse. This way the decompression can start from front. In Table 6.5 we show that this slightly worsens the compression ratio. The exception is when we use 2 bytes as alphabet. This suggests that for textual data we can slightly better predict what was before some suffix than to predict what follows after prefix. Or to be more precise: BWT better groups common prefixes of suffixes than vice versa.

Rot. / Suffix	BSORT	QSORT	Basic	Sada	Sew1level	Itoh	Seward	KS
R_S	1	SI	1180.81	667.61	657.58	561.56	n/a	n/a
R_S	1	3P	957.06	**422.36**	541.54	467.35	n/a	n/a
R_{BS}	1	BS	**801.54**	**422.38**	**459.3**	**398.4**5	n/a	n/a
R_S	2	SI	1134.97	490.62	662.07	561.93	535.41	n/a
R_S	2	3P	922.84	**355.34**	545.86	466.84	439.8	n/a
R_{BS}	2	BS	**788.42**	**354.81**	**463.76**	**398.56**	**375.62**	n/a
R_{UR}	1	SI	803.23	669.25	461.44	398.93	n/a	n/a
R_{UR}	1	3P	**570.45**	**422.13**	**342.53**	**300.5**	n/a	n/a
R_{BS}	1	BS	801.54	**422.38**	459.3	398.45	n/a	n/a
R_{UR}	2	SI	761.42	488.31	465.25	398.53	366.95	n/a
R_{UR}	2	3P	526.78	**355.56**	**346.31**	**305.57**	**261.98**	n/a
R_{BS}	2	BS	788.42	**354.81**	463.76	398.56	375.62	n/a
S	1	SI	511.77	701.04	311.28	275.76	n/a	407.3
S	1	3P	**420.28**	**427.54**	263.27	236.56	n/a	407.3
S	1	BS	623.33	**425.96**	**254.05**	**225.56**	n/a	407.3
S	2	SI	473.48	514.39	315.25	275.13	238.32	407.3
S	2	3P	**386.41**	**361.11**	267.87	237.42	199.09	407.3
S	2	BS	610.82	**360.09**	**258.26**	**226.07**	**198.77**	407.3

Rot. / Suffix: R_S – rotation with simple cmp function, R_{UR} – rotation with unroll cmp function, R_{BS} – rotation with special cmp function for BS, S – suffix.
BSORT: 1 – one-level bucket sort, 2 – two-level.
QSORT: SI – simple q-sort, 3P – 3-part q-sort, BS – Bentley&Sedgewick's q-sort.

Table 6.4: Comparison of Run Time of QSORT Functions

Bits per Byte	Bytes	2 Bytes	Syllable	Word
normal	**1.698**	1.777	**1.644**	**1.664**
reverse	1.716	**1.751**	1.680	1.680

normal - standard BWT for input string
reverse - input string has been reversed before BWT

Table 6.5: Influence of Reverse String on Compression Ratio

Seconds	Basic	Sada	Sew1level	Itoh	Seward	KS
normal	610.82	360.09	**258.26**	**226.07**	**198.77**	**407.3**
reversed	**609.51**	**358.29**	269.54	231.7	209.59	410.29

normal – standard BWT for input string
reverse – input string has been reversed before BWT

Table 6.6: Influence of Reverse String on Run Time

Note that reverse string has the same AML and MML as original string. Consider any two strings and their longest common prefix. Now this common prefix is their longest common suffix for reversed string. Any matching substring for two strings is also matching substring if we consider those two strings reversed.

We have tested also the run time and as you can see in Table 6.6 there are no big differences in run time for any algorithm. Slight difference in run time of algorithms is due to different sizes of two level buckets resulting in different number of strings compared using qsort. Algorithm by Kärkkäinen and Senders' does not recurse on suffixes S_i, $i \equiv 0 \ mod \ 3$ while for reversed string this changes to reversed suffixes S_i, $i \equiv (file_size + 1 \ mod \ 3) \ mod \ 3$. Note that suffixes from different modulo group can have different MML which influences number of recursive calls for Kärkkäinen and Senders' algorithm.

6.7 Changing the block size

In most implementations of block compression that we have seen each byte is taken to be a symbol from alphabet. Large files are split into blocks and the BWT is carried on each block. For larger blocks the compression ratio is better. Usually the blocks have to be at least 100KB large so that block compression – BWT followed by move to front (MTF) and run length encoding (RLE) has better compression ratio than dictionary LZ type algorithms.

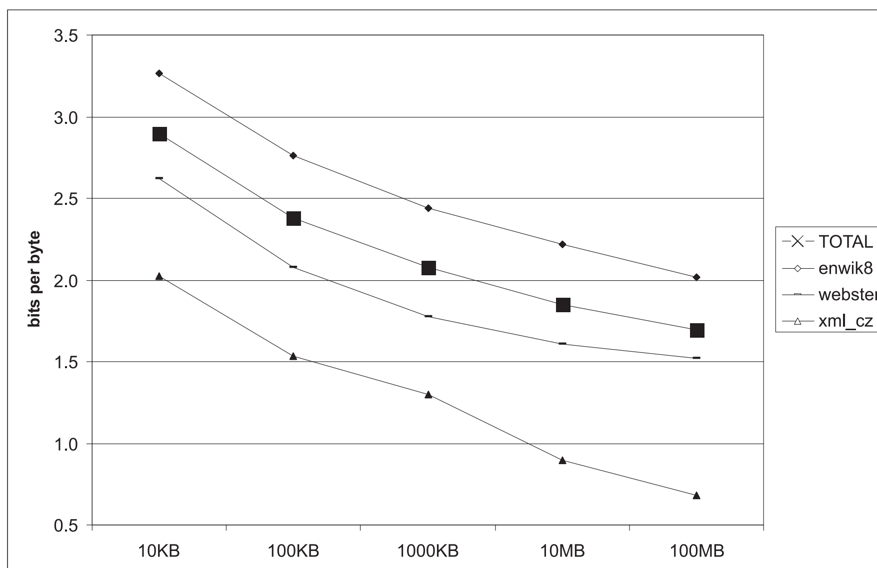


Figure 6.2: Block Size vs. Compression Ratio

Table 6.7 shows the influence of block size on compression ratio. We see that increasing the size of block improves compression ratio significantly. Also the results suggest that there is no upper bound on block size, above which compression ratio does not improve with increase of block size.

For the textual files from our corpus, compression ratio improves approximately with logarithm of block size. Although it is not possible to make conclusions from testing few files, the fact that we selected the textual data quite randomly without considering how well they can be compressed or how repetitive they are suggests that the result we got should be applicable for most textual files. See Figure 6.2.

The disadvantages of large blocks are memory requirements and running time. For alphabet of bytes the memory requirements for most algorithms are around $10n$ where n is the size of input in bytes. For all algorithms we have tested, the dependency is linear. For details about memory and time complexity of specific algorithm see its description in Chapter 4.

In Table 6.8 we show how does run time of BWT change depending on block size. The time shown is sum of times for all files from corpus with specified block size. We can see that the dependency of run time on block size is different for various algorithms. Run time of algorithms with better asymptotic complexity (KS, Sada) grows slower with increasing block size than for algorithms that compare whole strings. This is consequence of larger AML and MML in larger blocks.

		10KB	100KB	1000KB	10MB	100MB
E.coli	4639 KB	2.0983	2.0940	2.0914	**2.0873**	**2.0873**
bible.txt	4047 KB	2.6622	2.0953	1.7911	**1.7134**	**1.7134**
book2	611 KB	3.1567	2.5234	**2.2127**	**2.2127**	**2.2127**
dickens	10192 KB	3.2734	2.7258	2.3567	2.1795	**2.1679**
enwik8	100000 KB	3.2676	2.7587	2.4410	2.2161	**2.0180**
paper1	53 KB	3.1924	**2.6186**	**2.6186**	**2.6186**	**2.6186**
progc	40 KB	3.0123	**2.6443**	**2.6443**	**2.6443**	**2.6443**
webster	41459 KB	2.6179	2.0758	1.7779	1.6074	**1.5233**
world192.txt	2473 KB	3.3766	2.2352	1.6502	**1.4758**	**1.4758**
xml_cz	19100 KB	2.0263	1.5368	1.2976	0.8940	**0.6814**
xml_en	14493 KB	2.2034	1.6971	1.4132	1.0187	**0.8702**
TOTAL	197108 KB	2.8937	2.3803	2.0787	1.8476	**1.6972**

Table 6.7: Influence of Block Size on Compression Ratio

	Basic	Sada	Sew1level	Itoh	Seward	KS
10KB	**61.6**	**53.6**	**40.6**	**29.0**	54.0	**38.5**
100KB	77.9	68.2	51.5	43.0	**38.5**	68.7
1000KB	106.0	144.1	84.0	76.6	54.6	239.0
10MB	344.9	245.3	167.0	148.1	125.9	327.6
100MB	608.9	358.9	257.8	225.6	197.7	407.9

Table 6.8: Influence of Block Size on Run Time

	Bytes	2 Bytes	Syllable	Word
E.coli	2.087	**1.999**	2.454	3.050
bible.txt	1.714	1.804	**1.586**	1.583
book2	2.213	2.566	**2.170**	2.225
dickens	2.168	2.186	**2.072**	2.083
enwik8	2.018	2.079	1.933	**1.926**
paper1	**2.618**	4.414	2.753	2.865
progc	**2.644**	4.932	2.902	2.894
webster	1.523	1.570	**1.507**	1.552
world192.txt	1.476	1.674	**1.464**	1.475
xml_cz	0.682	0.884	0.641	**0.636**
xml_en	0.870	1.057	0.824	**0.815**
TOTAL	1.698	1.777	**1.644**	1.664

Table 6.9: Influence of Alphabet on Compression Ratio

6.8 Influence of Alphabet

In this section we will discuss the influence of alphabet on compression ratio of block compression and system requirements for performing BWT. In Table 6.9 we show compression ratio for different alphabets and in Table 6.10 we show run times obtained.

	Basic	Sada	Sew1level	Itoh	Seward	KS
byte	623.3	426.0	254.1	225.6	n/a	407.3
2 bytes	93.4	151.2	69.1	69.3	n/a	209.2
syllable	108.9	160.0	74.7	70.8	n/a	196.0
word	77.4	120.7	52.2	49.1	n/a	146.8

Table 6.10: Influence of Alphabet on Run Time

In most implementations of compression programs using BWT the size of alphabet used is 2^8. Since all values of the byte can be used, there is no value left that could be used as terminal symbol \$. Hence we need to work with rotations which as we have shown is slower than working with suffixes. We can overcome this by using more bytes to represent one symbol, but at a cost of requiring more memory and producing more cache misses.

If we shorten the input that will be transformed, the BWT will be quicker. The most straightforward approach is to use two bytes instead of one byte of input as alphabet. For most algorithms that we have described, the memory

and time complexity dependency on size of alphabet is linear and hence we can use alphabet of size 2^{16} for real life use without problem. On the other hand for Seward's algorithm (Section 4.5.3) and related algorithm by Manzinni and Ferragina [9], which require $|\Sigma|^2$ bytes of memory for storing starting positions of level 2 buckets, we would require 2^{32} bytes for an array which is not possible on 32 bit architectures and wasteful for all systems.

If we create dictionary, we can easily number symbols from alphabet starting with number one and leave zero for terminal symbol $. For terminated string we can use suffixes which improves the run time.

If we can find substrings which are frequent in input string and use them as alphabet, then we can improve both compression ratio and run time. Long substrings mean that we shorten the input for BWT and if they are frequent, we can compress dictionary effectively. The drawback is that finding frequent substrings in general is nor easy nor fast. Hence simple heuristics are the first to try.

In application XBW [4] we have implemented parser that enables us to use characters, syllables or words as alphabet. For definition of words and syllables see Section 2.3. This approach proves to be very effective for textual files.

Creating dictionary for non-textual data remains a challenge. We would like to test several approaches similar to idea in LZ algorithm in future work. Another approach we plan to test is counting frequencies of substrings of limited length.

We have tested in program XBW use of syllables and words as alphabet for textual data. Our results show that for text files over 500KB syllables as alphabet have the best compression ratio. For large textual files, namely *enwik8, xml_cz, xml_en*, with some XML markup words have the best compression ratio. We also tested using 2 bytes as alphabet, but this method degrades the compression. Results also show that syllables and words for large files produce very similar results.

For files larger than a megabyte, use of words and syllables as alphabet is equivalent in terms of compression ratio.

Now we present results for dependency of run time of algorithms on alphabet used. Using any of the mentioned alphabets we improve the run time dramatically. The reason is very simple – large alphabet shortens the input array for BWT. But there are also another factors at which we will look closer; namely AML and MML. In Tables 6.11 and 6.13 you can see how AML and MML change for different alphabet.

Specially look at change of run time of algorithms from first group. For algorithm *Itoh*, the run time for alphabet of words is less than quarter compared with byte alphabet. This massive difference is not just due to reduction

	bytes	2 bytes	syllable	word
E.coli	4638690	2319345	829617	18687
bible.txt	4047392	2023696	2011488	1686466
book2	610856	305428	303006	231323
dickens	10192446	5096223	4933442	4040241
enwik8	100000000	50000000	46999581	35405034
paper1	53161	26581	26940	20869
progc	39611	19806	20535	17390
webster	41458703	20729352	20924847	17970536
world192.txt	2473400	1236700	1247553	940040
xml_cz	19100318	9550159	9339300	7788878
xml_en	14493168	7246584	6981183	5793929
TOTAL	197107745	98553873	93617492	73913393

Table 6.11: Influence of Alphabet on Array Size

	bytes	2 bytes	syllable	word
E.coli	256	65536	121319	18685
bible.txt	256	65536	5734	13727
book2	256	65536	4685	8482
dickens	256	65536	12099	35457
enwik8	256	65536	128598	403326
paper1	256	65536	1612	1992
progc	256	65536	1395	1446
webster	256	65536	37477	206145
world192.txt	256	65536	12776	23137
xml_cz	256	65536	28667	61516
xml_en	256	65536	24842	45160

Table 6.12: Influence of Alphabet Type on Alphabet Size

AML	bytes	2 bytes	syllable	word
E.coli	17.4	7.2	1.5	0.0
bible.txt	14.0	5.9	7.2	6.3
book2	9.6	4.0	5.3	4.8
dickens	56.1	22.4	23.6	16.0
enwik8	16.4	7.1	8.2	6.7
paper1	8.0	3.0	4.5	4.2
progc	8.3	3.0	4.3	3.9
webster	34.3	15.5	15.4	13.0
world192.txt	23.0	9.8	11.7	9.3
xml_cz	2701.0	910.4	665.9	515.6
xml_en	1639.1	590.0	386.6	303.1

Table 6.13: Influence of Alphabet on AML

of length of input string to 40%. Note that word alphabet reduced AML which is critical factor for performance of algorithms from first group. AML decreases by over 80% for files with the highest AML.

Now we look at run time of algorithms for each file in Tables 6.14, 6.15, 6.16. We choose one algorithm from each group; namely *Itoh*, *Sada* and *KS*. In Figure 6.3 you can see comparison of their run time for whole corpus. In Figures 6.4 and 6.5 we list speed of algorithms in millions of symbols per second. For Figure 6.4 we take size of original whole corpus as base size and for Figure 6.5 the base size is length of the input string over specified alphabet. The length of string over alphabet of bytes is the same as size of corpus in bytes. For 2 bytes the length is half of the size of corpus. For syllables and words see Table 6.11.

Figure 6.5 shows that increase of speed of algorithm *Itoh* is not just due to shortening of the input string. Note that in Figure 6.5 we have speeds relative to length of input string. This is consequence of decreasing AML of input string with change of alphabet.

6.9 Choosing Algorithm

In this section we will compare optimized versions of coding algorithms. First we can leave out of our choice algorithm by Seward since it's memory requirements are $\Theta(|\Sigma|^2)$ and we want to work with large alphabet. In Table 6.12 you can see that for large files we get size of alphabet well over 2^{16}. For example for file *enwik8* we would need 186GB for storing starting positions

	bytes	2 bytes	syllable	word
E.coli	2.2	1.0	0.3	0.2
bible.txt	1.9	0.8	0.8	0.7
book2	0.2	0.1	0.1	0.1
dickens	7.1	2.5	2.6	1.9
enwik8	70.4	30.0	30.2	19.0
paper1	0.0	0.0	0.0	0.1
progc	0.0	0.0	0.0	0.1
webster	33.6	12.0	14.3	11.0
world192.txt	1.2	0.4	0.5	0.4
xml_cz	77.7	15.5	15.4	10.8
xml_en	31.4	7.0	6.6	4.7
TOTAL	225.6	69.3	70.8	49.1

Table 6.14: *Itoh* Algorithm: Run Time vs Alphabet

	bytes	2 bytes	syllable	word
E.coli	7.6	3.1	0.8	0.0
bible.txt	5.4	1.9	2.2	1.8
book2	0.5	0.2	0.2	0.1
dickens	15.5	5.4	6.3	5.1
enwik8	235.7	83.4	88.3	63.0
paper1	0.0	0.0	0.0	0.0
progc	0.0	0.0	0.0	0.0
webster	81.0	27.6	32.1	26.9
world192.txt	3.1	1.0	1.2	0.9
xml_cz	46.2	17.1	17.5	13.8
xml_en	30.9	11.5	11.5	9.1
TOTAL	426.0	151.2	160.0	120.7

Table 6.15: *Sada* Algorithm: Run Time vs Alphabet

	bytes	2 bytes	syllable	word
E.coli.4	7.6	3.8	1.4	0.0
bible.txt.4	6.3	3.4	3.2	2.6
book2.4	0.6	0.3	0.3	0.2
dickens.4	18.3	10.0	9.3	7.7
enwik8.4	236.5	118.2	111.2	77.8
paper1.4	0.0	0.0	0.0	0.0
progc.4	0.0	0.0	0.0	0.0
webster.4	78.9	41.4	40.9	34.7
world192.txt.4	3.6	1.9	1.7	1.3
xml_cz.4	31.5	17.1	16.0	12.9
xml_en.4	24.0	13.1	11.9	9.6
TOTAL	407.3	209.2	196.0	146.8

Table 6.16: *KS* Algorithm: Run Time vs Alphabet

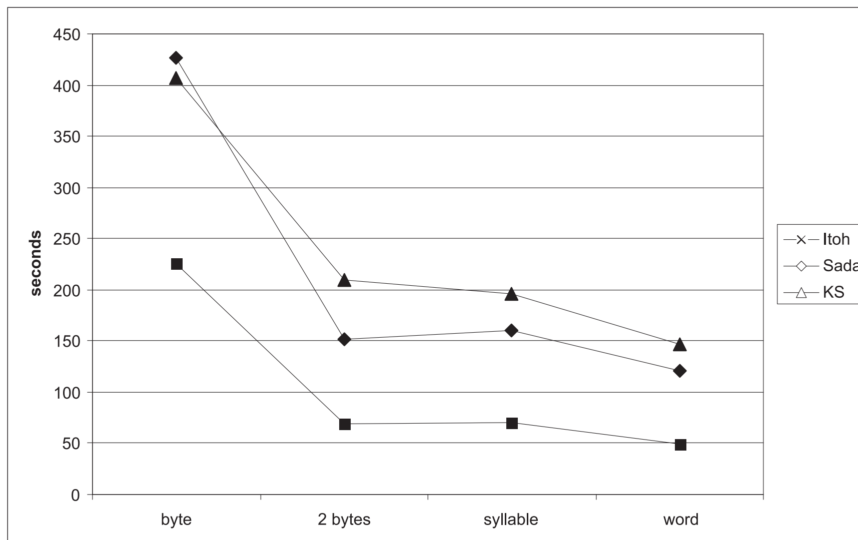


Figure 6.3: Run Time vs. Alphabet

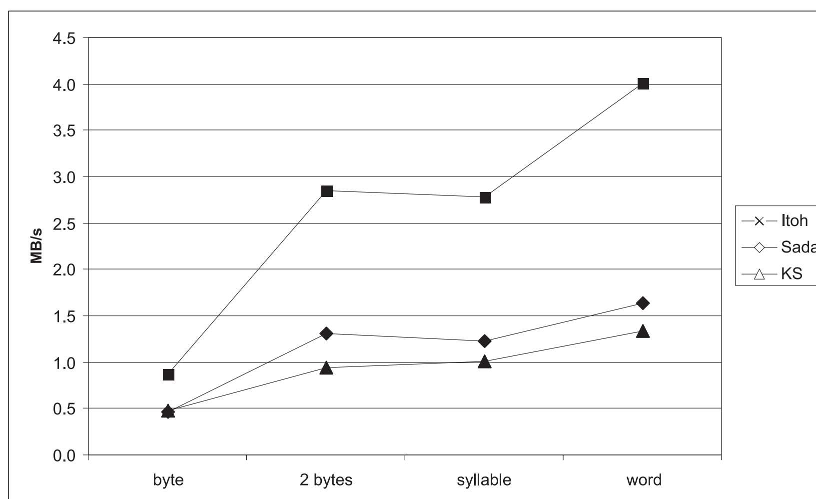


Figure 6.4: Coding Speed vs. Alphabet in MB/s

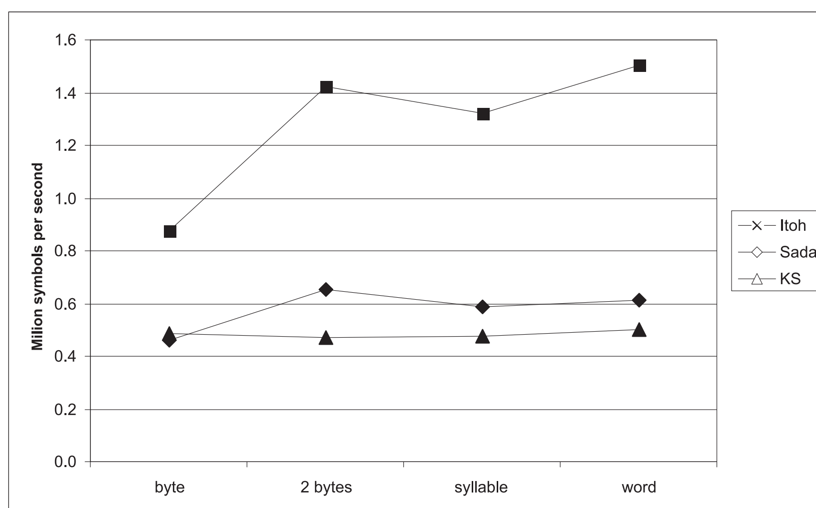


Figure 6.5: Coding Speed vs. Alphabet in Symbols per Second

of 2 level buckets.

In Tables 6.17, 6.18 and 6.19 we list run times of algorithms for each file. Data have been measured using suffixes with *BS* qsort and 1 level bucket sort. The first table is for alphabet of bytes, the second for syllable alphabet and the third for alphabet of words. From the remaining three algorithms from the first group (*Basic*, *Sew1level*, *Itoh*) we choose *Itoh* because it is the fastest. Only for two files for syllables our *Sew1level* is slightly faster than *Itoh*. Hence we conclude that *Itoh* supersedes *Basic* and *Sew1level*.

Now we have three algorithms each with different asymptotic complexity to consider. The choice of the best algorithm depends on AML of the file. For most files *Itoh* runs twice faster than the other two. However for file *xml_cz* which has the highest AML among all files *Itoh* runs more than twice as long as *KS*. In general AML can reach up to n where n is length of string. For such inputs *Itoh* algorithm can not be used. Hence for very repetitive inputs we need one of the algorithms *Sada* or *KS* which have good asymptotic complexity. Here we have to make trade off between run time and memory complexity since *KS* is faster, but requires more memory. Recall that for byte alphabet *Sada* requires $9n$ bytes while *KS* $11\frac{2}{3}n$. We suggest using *KS*, because it can be used also when not whole data fit into RAM. Recall that it does not as the only one use qsort which jumps in data quite randomly. *KS* uses radix sort to find ranks and in final phase, it uses merge sort. Hence it accesses data in arrays sequentially.

For robust implementation we suggest first trying to use *Itoh* and if the input file is too repetitive, fall back to algorithm *KS*. We can count total length of comparisons we made in *Itoh* and if it exceeds threshold value, end it and use the other algorithm. We see that *Itoh* starts to be slower *KS* when the AML exceeds 1000. Again note that by use of large alphabet we decrease the AML of input.

	Basic	Sew1level	Itoh	Sada	KS
E.coli	3.66	2.59	2.20	7.61	7.61
bible.txt	2.43	1.84	1.90	5.39	6.25
book2	0.23	0.19	0.19	0.53	0.64
dickens	12.20	7.20	7.05	15.51	18.31
enwik8	93.29	73.87	70.36	235.71	236.47
paper1	0.01	0.01	0.01	0.02	0.01
progc	0.01	0.01	0.01	0.01	0.01
webster	55.72	35.39	33.57	80.99	78.92
world192.txt	1.72	1.24	1.16	3.07	3.64
xml_cz	313.82	93.63	77.74	46.22	31.45
xml_en	140.24	38.08	31.37	30.90	23.99
TOTAL	623.33	254.05	225.56	425.96	407.30

Table 6.17: Run Time of Algrorithms for Bytes

	Basic	Sew1level	Itoh	Sada	KS
E.coli	0.47	0.29	0.32	0.78	1.40
bible.txt	1.02	0.74	0.79	2.20	3.21
book2	0.10	0.08	0.09	0.18	0.29
dickens	3.66	2.46	2.58	6.30	9.33
enwik8	38.37	29.40	30.25	88.26	111.17
paper1	0.00	0.00	0.02	0.01	0.01
progc	0.00	0.01	0.02	0.01	0.01
webster	19.54	13.60	14.26	32.11	40.93
world192.txt	0.65	0.52	0.49	1.24	1.74
xml_cz	32.75	19.48	15.39	17.46	16.02
xml_en	12.36	8.14	6.58	11.46	11.93
TOTAL	108.93	74.70	70.77	160.00	196.05

Table 6.18: Run Time of Algorithms for Syllables

	Basic	Sew1level	Itoh	Sada	KS
E.coli	0.00	0.02	0.24	0.01	0.02
bible.txt	0.85	0.60	0.71	1.79	2.57
book2	0.07	0.06	0.11	0.13	0.21
dickens	2.68	1.73	1.93	5.10	7.71
enwik8	26.50	18.61	19.03	63.02	77.80
paper1	0.01	0.01	0.12	0.01	0.01
progc	0.00	0.01	0.12	0.01	0.01
webster	15.45	10.38	11.01	26.91	34.67
world192.txt	0.46	0.37	0.40	0.89	1.26
xml_cz	22.52	14.26	10.77	13.75	12.92
xml_en	8.84	6.15	4.70	9.10	9.57
TOTAL	77.37	52.21	49.14	120.72	146.75

Table 6.19: Run Time of Algorithms for Words

Chapter 7

Conclusion

We have described Burrows-Wheeler Transform and role of suffix arrays in this transform. Since our motivation is the use of BWT in block compression, we show how several modifications can improve compression ratio and run time of suffix arrays creation.

We have implemented several algorithms for one, two and four byte alphabet. One the algorithms is our own inspired by work of Seward [11]. For optimization we have compared several qsort algorithms and comparison functions. Then we have compared optimized coding algorithms for various data and have proposed which should be used.

We have presented our proof that suffix arrays can be used instead of rotation arrays in BWT when the string is not terminated by special symbol.

Our main positive result is that using alphabet of words for textual data can improve compression ratio of block compression by ten percent and increase speed of coding more than four times.

We see as perspective using large alphabet also for binary data. For this purpose we plan to look for algorithm that would find frequent substrings in input and create dictionary of them. Then we would perform BWT over alphabet of words from this dictionary.

Appendix A

Contents of Compact Disk

We include on compact disk sources of program XBW. Code related to this thesis is in directory `bwt`. We include also documentation of program XBW and corpus used for testing performance.

Bibliography

[1] Bentley and Sedgewick. Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[2] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[4] Radovan Šesták et al. Xbw project, 2007.

[5] Tsai-Hsing Kao. Improving suffix-array construction algorithms with applications.

[6] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003.

[7] Jan Lánský. Slabiková komprese, 2005.

[8] N. Jesper Larsson. Notes on suffix sorting. Technical Report LU-CS-TR:98-199, Department of Computer Science, Lund University, Sweden, jun 1998.

[9] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm (extended abstract), 2004.

[10] K. Sadakane. A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. In *Proceedings of IEEE Data Compression Conference (DCC'98)*, pages 129–138, 1998.

[11] Julian Seward. On the performance of bwt sorting algorithms. In *DCC '00: Proceedings of the Conference on Data Compression*, page 173, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Julian Seward. Space-time tradeoffs in the inverse b-w transform. In *Data Compression Conference*, pages 439–448, 2001.

[13] Pavel Žoha. Algoritmy konstrukce sufixového pole, 2006.